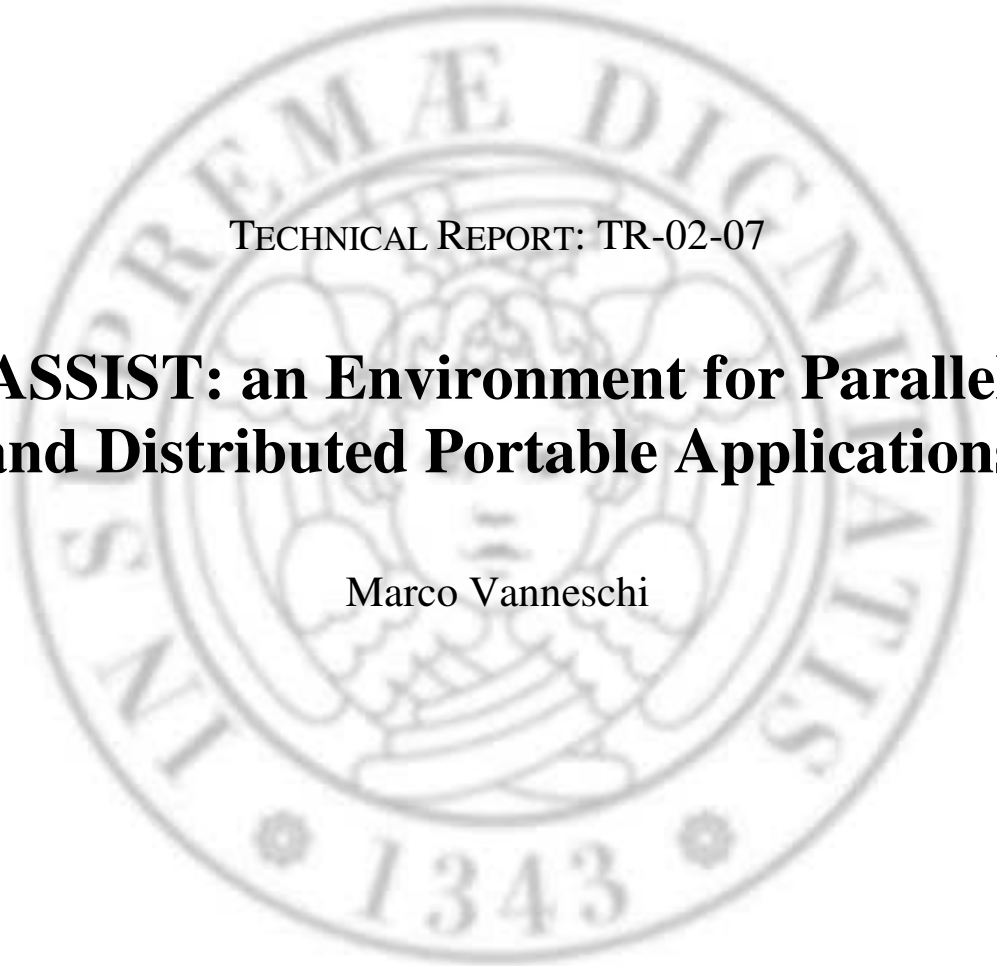
UNIVERSITÀ DI PISA

DIPARTIMENTO DI INFORMATICA



TECHNICAL REPORT: TR-02-07

ASSIST: an Environment for Parallel and Distributed Portable Applications

Marco Vanneschi

May 17, 2002

ADDRESS: Corso Italia 40, 56125 Pisa, Italy. TEL: +39 050 2212700, FAX: +39 050 2212726

This report is an extended version of a paper to be published in *Parallel Computing*, 2002

ASSIST:
an Environment for Parallel and Distributed Portable Applications

Marco Vanneschi

Department of Computer Science, University of Pisa

Corso Italia 40, 56125 Pisa, Italy - vannesch@di.unipi.it

Abstract

ASSIST (A Software development System based upon Integrated Skeleton Technology) is a proposal of a new programming environment oriented to the development of parallel and distributed high-performance applications according to a unified approach. The main goals are: high-level programmability and software productivity for complex multidisciplinary applications, including data-intensive and interactive software; performance portability across different platforms, including homogenous parallel machines and cluster/Beowulf systems, heterogeneous clusters and network computers, and computational Grids; effective reuse of parallel software; efficient evolution of applications through versions scalable according to the underlying technologies.

The proposal is based on the evolution and joining of the software *component* technology and of the *structured parallel programming* technology. Component technology must be extended in the direction of parallel and distributed high-performance applications for emerging large-scale heterogeneous platforms, including Grids. According to our experience in structured parallel programming, in ASSIST we intend to overcome some limitations of the classical *skeletons* approach to improve generality and flexibility, expressive power and efficiency for irregular, dynamic and interactive applications, as well as for complex combinations of task and data parallelism. A new paradigm, called "parallel module" (*parmod*), is defined which, in addition to expressing the semantics of several skeletons as particular cases, is able to express more general parallel and distributed program structures. ASSIST allows the programmer to design the applications in the form of generic *graphs* of parallel components, including both data-flow and nondeterministic reactive computation. Another distinguishing feature is that ASSIST modules are able to utilize *external objects*, including shared data structures and abstract objects (e.g. CORBA), with standard interfacing mechanisms. In turn, an ASSIST application can be *reused and exported as a component* for other applications, possibly expressed in different formalisms.

A substantial part of our research project is devoted to the architecture and implementation of the support to the new environment. The purpose of this paper is to show the principles of the proposed approach in terms of the programming model (successive papers will deal with the environment implementation and with performance evaluation). Initially the motivations and the rationales for the proposal are discussed in the context of software component technology, large-scale platforms and structured parallel programming. Then the features and the characteristics of the ASSIST programming model are described according to an operational semantics style and using several examples to drive the presentation, to show the expressive power and to discuss the research issues.

Keywords: Parallel and distributed programming environments; Structured parallel programming; Skeletons model; Shared Objects; Large-scale platforms, Software component technology.

This work has been supported by the *Italian Space Agency*: ASI-PQE2000 Programme on "Development of Earth Observation Applications by means of Systems and Tools for High-performance Computing", and by the *National Research Council*: Agenzia 2000 CNR Programme on "Development Environment for Multiplatform and Multilanguage High-performance Applications, Based upon the Objects Model and Structured Parallel Programming".

Contents

1. A unified approach to parallel and distributed high-performance programming 4
1.1. Software component technology and high-performance computing 4
1.2. High-performance applications on emerging large-scale platforms 5
1.3. Structured parallel programming: beyond the "classical" skeletons model 6
1.4. Features of ASSIST 7

2. Structure of ASSIST programs 8
2.1. Component-based structuring: graph composition 8
2.2. Declaration of ASSIST programs 9

3. Parallel module 10
3.1. Virtual processors 11
3.2. Topology 11
3.3. Internal state 12
3.4. Input streams 13
3.5. Virtual processors section 14
3.6. Output streams 16
3.7. Pipeline semantics of the *parmod* sections 16

4. External objects 17
4.1. Shared variables 18
4.2. Integration of Distributed Shared Memory libraries 19
4.3. Remote objects 19

5. Forms of parallelism in ASSIST-CL 19
5.1. Emulation of "classical" skeletons and their composition 20
5.2. Efficient combinations of stream-parallel + data-parallel structures 21
5.3. A dynamic computation: C4.5 algorithm for data mining 22

6. Conclusion 23

1. A unified approach to parallel and distributed high-performance programming

We discuss the motivations and the rationales for the proposal of a new programming environment oriented to the development of parallel and distributed high-performance applications according to a unified approach.

We start from the requirement of extending the software component technology in the direction of parallel and distributed high-performance applications. Then we discuss how this requirements agrees with the trends in large-scale heterogeneous platforms, including Grids, and that a unified approach to high-performance programming is needed. At this point, we review the concept of structured parallel programming, and in particular the skeletons model, which, in the last years, has contributed to significant advances in models and technologies for high-level parallel programming. We argue that, for our goals, the pure skeleton model has some limitations that must be overcome in order to define a solid and flexible technology for component-based high-performance environments. At the end of this section we define the features of ASSIST, the proposed programming environment, whose programming model will be described in sections 2, 3 and 4.

1.1. Software component technology and high-performance computing

Software component technology is playing a central role in the development of complex and multidisciplinary applications, especially in distributed and loosely coupled systems. As known, some current component models with commercial support are Microsoft DCOM (Distributed Component Object Model), OMG CORBA (Common Object Request Broker Architecture), and Sun Enterprise JavaBeans (FBJ). The key ideas of the component technology [11], that can be considered an evolutionary step beyond object-oriented methodology, are: *i)* the definition of a component as "physical packaging of executable software with a well-defined and published interface", *ii)* the clear distinction between definition and implementation, *iii)* the design of complex systems though the composition of smaller, possibly existing, components according to the principles of modular software development. The main, strongly interrelated, goals are:

- *portability* across different hardware-software platforms,
- *reuse* of existing components to create different, more complex systems,
- easy *evolution* of applications, since modifications to existing software should be "localized" inside some components and have little or no effect on the other components of the application. Similar considerations apply to the *specificity* of components, and to the capability of designing several *versions* of the same application, e.g. differing in performance and/or optimized for specific platforms.

These goals are fundamental for the high-performance computing field too, both in scientific and in industrial applications. However, the current component models do not stress the utilization of component technology on high-performance computing platforms. Generally speaking, all the above requirements are more difficult to achieve in a high-performance context, because they must be matched with the performance and scalability requirements, notably [33, 34]:

- portability means *performance portability*, i.e. how to exploit the features of different platforms to achieve the best level of performance;
- *reuse of parallel programs* may be seriously constrained by the parallel methodology adopted, especially if it is platform-dependent;
- *evolution of high performance software*, and the design of several *versions* of the same application, must seriously take into account the evolution of the underlying platform technology.

For these reasons, some interesting projects [1, 9, 10, 25, 26, 28] are focused on the issue of characterizing the component technology for high-performance computing applications, and this is also one of the main goals of our proposal. We need to extend the current concept and technology of components towards the following interrelated goals:

a) concept and technology for *parallel components*,

b) *unified approach* to the development of parallel and distributed high-performance applications.

1.2. High-performance applications on emerging large-scale platforms

To understand our approach, it is useful to point out what is the impact of the trends in *high-performance large-scale platforms* on the methodologies and tools for developing high-performance applications.

The trends in ICT platforms are clearly focused in the direction of integrating computing resources into *large-scale platforms* [17]: the high-performance computing resources to be integrated include parallel machines such as SMP shared memory systems, previous-generation supercomputers, homogeneous and heterogeneous Beowulf clusters of PCs and/or workstations, as well as Beowulf clusters of parallel (SMP) nodes. The interconnection of such resources, according to a client-server or a peer-to-peer approach [12], is not limited to state-of-the art distributed systems at the Internet level: high-speed communication technologies are emerging, not only at the cluster and LAN level, but also at the Intranet and MAN level, e.g. very high-bandwidth optical networks. In this context, the *Grid Computing* paradigm [4, 18, 19, 20] is emerging as an outstanding approach to realize new large-scale platforms, especially to support scalable Virtual Organizations efficiently and in a secure and controlled way.

A high-performance component-based application for large-scale platforms should be developed in a very flexible and portable way, with the following goals in mind:

a) each component can be expressed with the proper *degree of parallelism and granularity* to exploit the characteristics of the systems on which it will be executed and/or to meet the specificity goals;

b) according to the evolution of the underlying platform and/or applications requirements, the structure of a component, and in particular the degree of granularity and parallelism, can be *changed easily* by operating limited transformations without modifying its interfaces. At the current level of knowledge, it is common that such transformations are done "manually", however the experienced features of compiler technology for structured parallel programming suggest that, at least partially, automatic transformations could be performed: in both cases the *programming model* plays an outstanding role;

c) the whole organization of a high-performance application can reflect the availability of new high-bandwidth and low-latency communication technologies able to make the degree of coupling stronger:

  i) distributed platforms at Intranet/MAN level could be exploited in a similar way to current Beowulf clusters or parallel machines,

  ii) fine-to-medium grain parallelism could be exploited also at global level;

d) the *run-time support of the programming model* must be optimized *wrt* the different underlying technologies, i.e. specialized and personalized supports to components/objects have to be realized for systems where fine-medium grain parallelism can be efficiently exploited. Such supports may be functionally equivalent to current component/object distributed supports, but much more efficient.

In general, the simplified approach, consisting in encapsulating current-technology parallel codes into current-technology components, is not sufficient to achieve the above goals.

By now it is widely recognized that, at the programming model and environment level, the high-performance requirement for large-scale/grid platforms imposes new approaches able to take into account both the aspects related to the distribution of computations and resources, and the aspects related to parallelism [9, 17, 21, 28, 30]. The development of grid/network computing applications requires, in general, capabilities and properties beyond those needed in both sequential programming and in "classical" parallel/distributed programming, as it requires the management of computations and environments that are typically *dynamic* and *heterogeneous* in their nature and that include resource hierarchies with different features (e.g. memory and network).

The programming environment for large-scale/grid platforms should make usage of *cost models* specifically developed to allow applications to be migrated between different execution environments preserving their efficiency and their performance. The cost models will also be used to handle those situations related to dynamic features that are typical of Grid environments [21], e.g. to drive the choices between different possible alternatives in the usage of hardware/software resources.

1.3. Structured parallel programming: beyond the "classical" skeletons model

Models and tools for structured parallel programming [13] have been developed in the last years to meet some of the requirements and goals discussed in the previous sections. In particular, the *skeletons* model [13, 16, 22, 31, 33, 34, 35] has some very interesting features:

- a parallel program is developed by composing instances of specific forms of parallelism (skeletons) that are met frequently in parallel structures, like stream-parallel pipeline and farm, or data-parallel map, reduce and stencil/based, or dynamic divide-and-conquer. Some skeletons models give a elegant solutions to the aged problem of integrating task and data parallelism [3];
- a coordination language, defined by the chosen skeletons considered as programming constructs, acts as a metalanguage for expressing complex program structures;
- a cost model and an optimized implementation scheme is known for each skeleton and, in some cases, for their composition.

In our experience on the SkIE environment [31, 33, 34, 35], skeletons resemble some features of components in that*: i)* skeletons are composed according to well-defined interfaces, separating the implementation from the definition of the parallel program, *ii)* the sequential parts of skeleton instances can be expresses in any "host" language, exploiting the compiler of such language without modifications, *iii)* existing sequential codes can be reused in the sequential parts of skeletons, *iv)* performance portability is satisfactory for homogenous platforms and non distributed applications, as, in these cases, cost models are known to allow the programs to be recompiled efficiently.

Other structured parallel programming approaches with much higher abstraction level, such as Linda or some concurrent object-oriented formalisms [32], are not able to meet the requirement of performance portability. At the other extreme, message-passing (MPI) or shared memory libraries, as well as HPF and similar data-parallel languages [27, 23], have not sufficient expressive power to support high-level development of complex applications and performance portability. Approaches that tend to adopt parallel *design patterns* have interesting capabilities [15] similar to skeletons, though with some notable differences which, however, are not essential for the goals of the present discussion.

Despite several advantages of skeletons or parallel design patterns, a clear evolution of structured parallel programming beyond such models is needed, at least for the following reasons:

a) in addition to the capability of expressing some typical parallel schemes, we need a capability of *structuring component-based applications*. A large degree of *flexibility* in expressing parallel and distributed program structures is essential: though in some cases typical skeletons structures are sufficient, in general *generic graph structures* are required for complex compositions in multidisciplinary applications;

b) basically, skeleton-based programming models have functional and deterministic semantics that can be a serious obstacle in many complex applications. The concepts of *internal state* of parallel components and of *nondeterminism* in communications/interactions between components are fundamental, as well as *dynamic interactivity* in a client-server or in a peer-to-peer environment. In general, all such features are not stressed in skeletons models;

c) though in many skeletons programs the integration of task and data parallelism is expressed efficiently, there are other cases in which the composition of the available skeletons is not natural or it is inefficient. In our experience with SkIE, this occurs in *some* irregular and/or dynamic problems for which the *integration of (task) stream parallelism and data parallelism* is a source of inefficiency and produces codes that are more complex than the equivalent codes exploiting only data parallelism;

d) to develop a complex, multidisciplinary application we need to be able to utilize *predefined objects* in a modular and invisible way: they can be *abstract objects* according to commercial standards (e.g. CORBA), as well as abstractions of *systems resources* (devices, file, severs, and so on);

e) in the same way, *several kinds of libraries* (scientific, image processing, data mining, and so on) *should be integrated in the environment* in a flexible and efficient way;

f) in many applications the adoption of a *shared space of objects*, or a Distributed Shared Memory space (independent of the distributed nature of the underlying platform), is fundamental to efficiently manipulate very large data sets, to simplify the programming of irregular and/or dynamic problems and, sometimes, to mask communication overheads. In our experience [8, 14] the *integration of shared objects into a structured parallel programming formalism* increases the expressive power and the efficiency of parallel programs significantly;

g) at our knowledge, the skeletons model is not suitable to fully support *the reuse of parallel applications* written in different formalisms (i.e. sometimes a good degree of reuse is achieved for *sequential* codes only). This goal has to be achieved in the new context of parallel components.

1.4. Features of ASSIST

Our proposal is a new programming environment for parallel and distributed high-performance applications, called ASSIST (A Software development System based upon Integrated Skeletons Technology). The design of applications is done by means of a *coordination language*, called ASSIST-CL, whose programming model has the following features:

1) parallel/distributed programs can be expressed by *generic graphs*, without renouncing the possibility of structuring specific (skeletons-like) graphs for which some interesting cost models already exist;

2) *the components can be parallel modules or sequential modules*, with high flexibility of replacing components in order to modify the degree of granularity according to the application requirements of and/or to the underlying platform evolution;

3) a construct, which is more flexible and powerful than the "classical" skeletons, is introduced. It could be considered a sort of "generic" skeleton which can be programmed in such a way to emulate the most common "specific" skeletons, but which is also able to easily express new forms of parallelism (e.g. optimized forms of task + data parallelism, nondeterminism, interactivity), as well as their variants and personalizations. When necessary, the parallelism forms that are expressed could be at a lower abstraction level with respect to the "classical" skeletons. We call this new construct *parallel module*, or *parmod;*

4) generality and flexibility does not mean to abandon the idea of structured parallel programming. On the contrary, the proposal of a "generic" skeleton and of applications as graphs of parallel and sequential modules aims to reinforce such idea and, at the same time, to achieve a more satisfactory trade-off between high-level structuring, expressive power, reuse and efficiency. In particular, one of the basic features of structured parallel programming that we wish to preserve is related to the efficiency of implementation:

- on the one hand, compared to "classical" skeletons, it is more difficult to define a simple cost model for a generic construct like *parmod*, and this can render more difficult some possibilities of introducing *optimizations at compile time*;
- on the other hand, a basic feature of structured parallel programming, which will characterize *parmod* too, is the ability *to optimize the run-time support* once for all. At least with the current knowledge of software development for large-scale platforms/grids, we believe that run-time support optimizations are much more important than compile-time optimizations (which, anyway, remain a significant research issue);

5) the parallel and the sequential modules have an *internal state*. The parallel modules can have a *nondeterministic behaviour*;

6) composition of parallel and sequential modules is expressed, primarily, by means of the very general mechanisms of *streams*, by which we can represent powerful *interfaces* simply and effectively. In addition, modules can *share objects* implemented by forms of Distributed Shared Memory, invoked through their original APIs or methods. While the stream-based composition is the basic mechanism for structuring applications and defining the component interfaces, shared memory objects are an additional mechanism for solving problems of memory space, programmability of highly dynamic structures, and communication of heavy data sets;

7) the modules of a parallel application can refer to any kind of existing *external objects*, like CORBA and other commercial standards objects, as well to system objects, though their interfaces. In the same way, *an* ASSIST *parallel application can be exported as a component or as an object* to other applications: the stream-based interface mechanisms is transformed at compile-time in a mechanism compliant to existing standards, i.e. CORBA IDL interfaces in the first version of ASSIST;

8) finally, the methodology by which the environment is implemented is itself component- and object-based, in order to be able to flexibly modifying its implementation according to significant changes in the underlying technologies and/or in applications requirements.

2. Structure of ASSIST programs

2.1. Component-based structuring: graph composition

The structure of an ASSIST program is a *graph*, whose nodes are *components* and the arcs are abstract interfaces that support *streams*, i.e. ordered sequences, possibly of unlimited length, of typed values. Any graph structure, possibly including cycles and merge (many-to-one) and broadcast (one-to-many) streams, is permitted.

Components are expressed by ASSIST *modules*, which may be *parallel modules* (*parmod*, defined in the next section) or *sequential modules*. A sequential module is the simplest component expressed in ASSIST: it has an internal state and is activated by the input stream values according to a deterministic data-flow behaviour (the nondeterministic behaviour can be expressed only by parallel modules).

Fig. 1 shows the example graph of a simple ASSIST program composed by modules M_1, M_2, M_3, M_4, M_5.

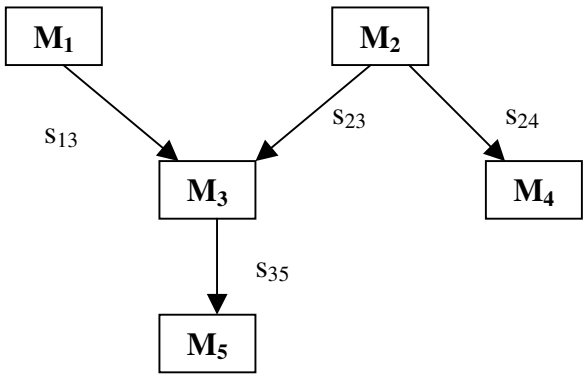


Fig. 1 – An ASSIST graph

M_1 and M_2 have no input stream and, in fact, are in charge of generating streams s_{13}, s_{23} and s_{24}. M_2 has two output streams s_{23} and s_{24}; in general, during an activation, M_2 may send values onto both s_{23} and s_{24}, or only onto one of them, or none of them. M_3 may have a deterministic or, if it is a parallel module, a nondeterministic behaviour on the input streams s_{13} and s_{23}. M_4 and M_5 have no output stream, thus are in charge of generating the result data structures of the program.

A composition of modules, expressed by a graph P, may be, in turn, *reused as a component* of a more complex program Q, e.g. Q is composed of P and a composition of modules R. In general, the designer of Q does not wish to have visibility of the internal structure and implementation of P; the composition is legal provided that P is correctly *interfaced* to R, i.e. provided that the types of input and output streams, defining any interface between P and R, are compatible. In the example of fig. 1, any subgraph could have been defined, and implemented, separately and then reused to build the whole application.

Streams are the structured way to compose modules into an application. In addition, components can also interact by means of *external (shared) objects*, in general not expressed in ASSIST-CL.

2.2. Declaration of ASSIST programs

Formally, a composition (denoted by the keyword **generic**), is defined by: name of the composition, name and type of input streams and output streams for each module, and names of the component modules. Particular structures of graphs, considered primitives in ASSIST-CL, are the stream-parallel skeletons *pipe, farm, loop* [31]. Although such structures can be expressed directly by means of composition of parallel and sequential modules, they are also provided as primitives to more easily design application that can be effectively expressed by classical skeletons.

The composition denoted by the keyword **main** describes a complete ASSIST program; it has no input and no output stream. The composition of fig. 1 could be declared as:

```
generic  main (argc, argv)
             {
               // declaration of streams
                    stream  int s13;
                    stream  int [N1][N1] s23;
                    stream  int [N2][N2] s24;
                    stream  int s35;
               // declaration of modules and their interfaces
                    M1 (ouput_stream s13);
                    M2 (output_stream s23, s24);
                    M3 (input_stream s13, s23; output_stream s35);
                    M4 (input_stream s24);
                    M5 (input_stream s35);
             }
```

An ASSIST-CL program consists of a *main* part, and of a *declaration* part that includes: definition of new data types, definition or interface of external objects, definition of sequential modules, definition of parallel modules, definition of the composition graph (generic or SkIE-like skeletons), definition of sequential pieces of codes invoked by the sequential and parallel modules.

The syntax of ASSIST-CL is basically that of C, enriched by constructs and data types characterizing the parallel programming model. Some ASSIST data types are similar or compliant to those of CORBA IDL. In the first version of ASSIST-CL, the *host languages*, by which to express sequential pieces of code, are C, C++ and Fortran.

For example, let us suppose to have defined the following sequential piece of code:

```
proc   fun (in int a, int b[N][N]; out int c)
       $C{
              < body >
       }C$
```

Suppose that module M_3 of the composition in fig. 1 is sequential with data-flow behaviour (at each activation, M_3 waits for a pair of values from the input streams s_{13} and s_{23}, and produces a value onto the output stream s_{35}). M_3 could be defined as:

```
M3  (input_stream s13 (int x), s23 (int A[N][N]); output_stream s35 (int result))
       $C{  < local declarations >
            result = fun (x, A);
            assist_out (s35, result);
       }C$
```

Since no a-priori rule exists for the management of the output streams (which streams to select and when to transmit values onto each of them), an explicit command, called ASSIST**-out** (*output_stream_name*, *value*), is utilized, in sequential and in parallel modules, to transmit a value onto an output stream.

3. Parallel module

The construct called *parallel module* (**parmod**) is the proposed solution for many problems introduced in section 1, and in particular to meet the requirements of section 1.4.

The idea of *parmod* is shown graphically in fig. 2.

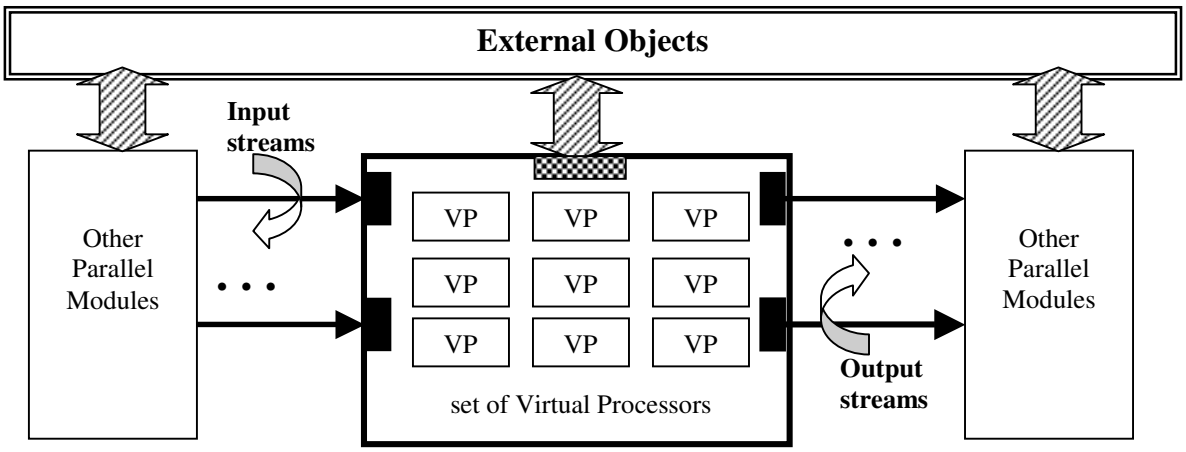


Fig. 2 – Graphical scheme of a parallel module

A *parmod* is defined by the following elements:

1. Virtual processors
2. Topology
3. Internal state
4. Input streams
5. Output streams
6. External objects.

They will be discussed in the next sections, where an *informal* view of the syntax and semantics of ASSIST-CL 1.0 will also be given (the full definition of the coordination language is out of the scope of this paper and can be found in the detailed documentation). The following simple example will be used to guide the presentation and to exemplify the syntax and semantics:

Example

Let us consider the following algorithm:

```
var A: array [1 .. N, 1 .. N] of int; x: int;
function F (int, int, int): int; < definition of function F >
for h := 1 to N do
        for i : = 1 to N do
                for j := 1 to N do
                        A[i, j] := F(x, A[i, h], A[h, j])
```

This computation can be expressed by the following abstract data-parallel program structure:

```
for h := 1 to N do
        forall i, j : = 1 to N do
                A[i, j] := F(x, A[i, h], A[h, j])
```

which exploits N^2 virtual processors, where each virtual processor *VP[i, j]* is assigned the value *x* and the value of *A[i, j]*. Though simple, this parallel program is not trivial to implement efficiently because the communication *stencil* varies at each iteration step (for each value of *h*).

Let us define the interface of a *parmod*, named *Demo*, which implements the parallel version of this problem:

```
parmod  Demo (input_stream in_scalar (int x), in_matrix (int A [N][N]);
              output_stream out_matrix (int B [N][N]))
{. . .}
```

It has two typed input streams, named *in_scalar* and *in_matrix*, and a typed output stream, named *out_matrix*.

We will study two different versions:

- *Data-flow:* at each activation Demo operates on a *new* pair of values *(x, A)*, thus waiting for a new value on both the input streams,
- *Nondeterministic:* a new execution is activated when a new value of *x* is received, using the old value of A, *or* when a new value of *A* is received, using the old value of *x*. A selection strategy is applied when new values of *x* and *A* are simultaneously detected on both input streams.

Both versions of the ASSIST program are collected in **Appendix A**.

3.1. Virtual processors

A set of *virtual processors* (VPs), i.e. independent and cooperating entities delegated to perform the parallel computation, is defined as the "calculation engine" of a *parmod*.

In the example of Demo, we have just N^2 VPs. Of course, the support tools will map the set of VPs onto a suitable set of physical processing nodes, statically or dynamically.

In this example, all VPs operate collectively in a *data-parallel* fashion, where data are exchanged between VPs at each iteration according to the communication stencil expressed by the abstract definition. Notice that *not* all programs expressed by a *parmod* are data-parallel: on the contrary, this is just a (simple) particular case inside a much larger space of possible structuring models that can do exploited in ASSIST-CL This issue will be discussed in successive sections.

3.2. Topology

This feature is related to the *naming* scheme of VPs. Each VP has a unique identifier, which often is useful be expressed in a parametric way in order to implement collective forms of parallelism, e.g. data-parallelism. As a consequence, the naming scheme is often related to the way in which the internal state (section 3.3) is assigned and distributed to VPs and to the way in which it is referred.

The topology is in no way related to the cooperation scheme, or the communication stencil, of VPs.

The following topologies are defined in ASSIST 1.0:

- *multidimensional array*, to denote every VPs by the values of one or more indexes. This is the topology for our example Demo:

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

  The generic virtual processor has name VP[i][j];

- *none*: in this case the name of VPs is not significant for the computation to be expressed, e.g. in several farm-like structures composed of fully independent workers;
- *one*: the *parmod* has only one VP. This is different from the sequential module, because a *parmod-one* exploits many of the general features of a *parmod*, in particular nondeterminism and stream control features (sections 3.4, 3.6, 3.7), while a sequential module has a deterministic data-flow behaviour.

Notice that *more than one topology is possible for the same problem*, according to the parallelization strategy and/or the granularity that the programmer wishes to express. The component-based structuring of an ASSIST program allows the designer to change easily the internal implementation, e.g. the granularity, according to new requirements (when a cost model is known, such modifications could also be done by delegating the most intensive job to the compiler, as it occurs in some skeletons programs). As a special case, the topology *one* is adopted when the programmer wishes to force a sequential (though nondeterministic) execution, being aware that a parallel implementation could be inefficient on the currently available platform; however, such a component is predisposed to be modified into a parallel one when the platform will be modified and/or more efficient parallelization strategies will be possibly defined.

3.3. Internal state

A *parmod* has a state that is logically *partitioned* and/or *replicated* into VPs. Some state variables will be also utilized to control the communication from the input streams and to the output streams.

In our example Demo, we will assign the input values of *A* and *x* to state variables: respectively *S,* partitioned (each element of the array *S* is assigned to a distinct VP), and *L*, replicated (each VP has its own copy of this value; possible consistency problems, not present in this and many other examples, are at full responsibility of the programmer).

The declaration and initialization section of Demo is:

```
attribute    int S[N][N] scatter onto VP;
attribute    int L replicated;
init { // empty in this example }
```

Many partitioning schemes, including all the most popular data-parallel ones, are possible using “free variables” [31]. For example, the previous declaration of *S* is an abbreviation for:

```
attribute    int S[N][N] scatter S[*i][*j] onto VP[i][j];
```

In the *nondeterministic version*, the state consists of *S* and *L*, as before, and also of some *control variables* that will be used in the input section to drive the nondeterminism control:

```
attribute bool accept_x;
attribute int priority_A, priority_x; init
{ L = <initial value of x, replicated in all VPs>;
  accept_x = false;
  priority_A = 1;
  priority_x = 0;
}
```

The boolean variable *accept_x* is used to control when a new value, present on the input stream *in_scalar*, can be accepted and then distributed to the VPs. For example, we state that a new value *x* is not accepted until at least one value from the input stream *in_matrix* will be received (*L* is initialized explicitly in all VPs).

The integer variables *priority_A* and *priority_x,* initialized as shown, will be used in the input section to drive a priority-based nondeterministic selection when both input channels contain at least one value.

3.4. Input streams

As discussed in section 2, streams are the basic mechanism for the composition and interfacing of component modules. Operating on streams means that, *for each activation*, a module receives values from one or more input streams, execute its function on such values and on the current state values, in general modifying the state variables, and then it possibly transmits the result and/or some state values onto one or more output streams.

The input streams may be used according to a *data-flow* scheme when, for each activation, the module waits for values from all the input streams. In the most general, case a subset of streams is selected at each activation: this expresses a *nondeterministic* behaviour. The semantics of nondeterminism is the one of the CSP model [24] based on *guarded commands*, and in particular the semantics of ECPS concurrent language [7]: every guard contains at most three elements, each of which may be absent: an *input guard*, a *local guard* (boolean predicate on control state variables), and a *priority* which can be varied by program.

Each input stream is associated an independent *distribution strategy*, i.e. a strategy to transmit the received value to the VPs. The following strategies are defined:

- *on demand*: a received value is transmitted to a "ready" VP, chosen nondeterministically,
- *scatter*: a received, structured value is *partitioned* to the VPs according to a rule expressed by program,
- *multicast*: a received value is sent to all the VPs belonging to a certain subset expressed by program,
- *broadcast*: a received value is sent to all VPs.

Again, a large variety of scatter and multicast distributions can be expressed using free variables [31].

In the Demo example, the *data-flow version* has the following input section:

```
input_section
{  on in_scalar and in_matrix
              distribution A scatter to S;
              distribution x broadcast to L;
}
```

We have just one input guard (**on**) that is a *multiple* input guard (waits a value from both *in_scalar* **and** *in_matrix*). When the guard is verified, *A* is scattered to *S* and *x* broadcasted to all the copies of *L.*

In the *nondeterministic version* of Demo, the input section can be the following:

```
input_section
{  on priority_A, , in_matrix
        distribution A scatter to S;
        operation {accept_x = true;
                   priority_A = 0;
                   priority_x = 1;
                  }
  on priority_x, accept_x, in_scalar
        distribution x broadcast to L;
        operation { priority_A = 1;
                    priority_x = 0;
                  }
}
```

The guarded command has two guards (**on**). The first one is verified when a new *A* value is received, the second one when a new *x* value is received *provided that accept_x* is true. When the evaluation starts, if both guards are verified the highest priority guard is selected. After the execution of a guard, the distribution of the received value occurs.

Another piece of code (keyword **operation**) is executed *in the input section* to properly modify the value of some state control variables (in our case, to modify *accept_x,* and the priorities according to a simple pseudo-random strategy).

In general, each input value may be distributed:

- *to the state variables* in VPs, as in the Demo example,
- *to the formal input parameters* of the functions executed by the VPs.

If useful, before to be distributed, an input value may be *pre-processed* by program.

3.5. Virtual processors section

In the Demo example with *data-flow* behaviour , each VP starts a new activation as soon as new input values are distributed from the input section (in this example, new values assigned to the state variables *S* and *L*). In the virtual processors section, *the condition for which the activation occurs* is expressed through the *input list* (**in**) of the *action* to be executed, where we declare the *names of the input streams* from which the input parameters must be received (*in_scalar* and *in_matrix* in the example):

```
virtual_processors
  {Compute_with_stencil(in in_scalar, in_matrix; out out_matrix)
       {< body >
       }
  }
```

In the *body* part, the function to be executed by the VPs is defined, also taking into account how the input data are distributed to the VPs and how the results are to be distributed onto the output streams.

In this simple example, exactly the same command sequence is executed by all the VPs. This is just a particular case: in general, *disjoint subsets of VPs can execute different command sequences*; in the extreme case all the VPs can have a distinct behaviour. In the nondeterministic version an even more general structuring of the virtual processors section will be shown. This is another notable feature that distinguishes ASSIST-CL from any other parallel formalism, in particular data parallel and SPMD formalisms [32].

In a different example, we would have wished to express that a certain command sequence C_1 is executed by only the VPs on the borders of a matrix topology, while the internal VPs execute a different command sequence C_2. The body can be written as follows (**VP** and **or** are keywords):

```
VP i=0, j=0..N-1 or i=N-1, j=0..N-1 or i=1..N-2, j=0 or i=1..N-2, j=N-1
    { C1 }
VP i=1..N-2, j=1..N-2
    { C2 }
```

The subset of VPs associated to a certain command sequence C is called the *enabling subset* of C.

The VP body in the *data-flow version* of Demo is:

```
VP i=1..N, j=1..N
   {for (h=0; h<N; h++)
            F(in L, S[i][h], S[h][j]; out S[i][j]);
      assist_out (out_matrix.go, 1);
   }
```

where F is defined as a **proc**. In this example, the ASSIST**_out** command sends just a "synchronization signal" to the controller of the output stream *out_matrix*: as we shall see, the value of the result array (B) will be got directly from the internal state *S* of VPs, instead of performing a double copy of the array.

Synchronization in data-parallel program structures

The command **for** is a construct of ASSIST-CL to denote the *parallel* execution of a command sequence C (invocation of F in our case) on all the VPs belonging to the enabling subset of C. The parallel execution of a command sequence C can also be controlled by the **while** construct, whose guard may also contain the parallel predicate *reduce* applied to the internal state of the enabling subset of C.

The semantics of *for* and *while* guarantees the consistency of the internal state of VPs belonging to the enabling subset, i.e. *at the beginning of an iteration the VPs internal state is the one updated at the end of the previous iteration*. Notice that a barrier is not implied by this semantics, as several synchronization schemes can be adopted at the implementation level to optimize the program execution.

In data-parallel command sequences expressed by *for* and *while*, the well-known "*owner computes rule*" is assumed as a constraint to manipulate the internal state: every VP can modify only the state variables assigned (partitioned or replicated) to it, while can read the state variables assigned to the other VPs. Other more powerful, relaxed schemes will be studied and experimented in the next version of ASSIST-CL.

The command **sync** is introduced to explicitly synchronize the sequentialization of parallel actions.

Let us now consider the virtual processors section of Demo in the *nondeterministic version*. We have now a distinct *action* for each guard defined in the input section: the first action (*Compute_with_stencil_A*) operates on the new value of *S* (and on the current value of *L*); the second action (*Compute_with_stencil_x*) operates on the new value of *L* (and on the current value of *S*). *According to the outcome of the evaluation of guards in the input section, the respective action in the virtual processors section is executed.*

The virtual processors section is expressed by (notice the input lists of the two distinct actions):

```
virtual_processors
  { Compute_with_stencil_A (in in_matrix; out out_matrix)
        { VP i=1..N, j=1..N
            { for (h=0; h<N; h++)
                  F(in L, S[i][h], S[h][j]; out S[i][j]);
              assist_out (out_matrix.go, 1);
        }
    Compute_with_stencil_x (in in_scalar; out out_matrix)
        {VP i=1..N, j=1..N
            { for (h=0; h<N; h++)
                  F(in L, S[i][h], S[h][j]; out S[i][j]);
              assist_out (out_matrix.go, 1);
            }
        }
  }
```

In this particular case, the body of both actions is the same, *in general distinct bodies are associated to different actions.*

In conclusion, about the possibilities of performing distinct actions by distinct VPs:

- *the virtual processors section contains as many distinct actions as are the guards in the input section that cause the distribution of values to input parameters or state variables of the VPs,*
- *every action can be executed on distinct enabling subsets of VPs, where each subset is associated its own body*.

3.6. Output streams

As seen, the results of an activation may be sent onto one or more output stream, or not sent at all. This choice is controlled explicitly by program.

In the output section of a module, the values to be sent onto each output stream are *collected* from the VPs according to one of the following strategies:

- **from ANY**: nondeterministic selection of one of the VPs,
- **from ALL** the VPs.

Before to be sent, an output value may be *post-processed* by program; moreover the state control variables, which possibly are shared with the input section, can be modified also in the output section.

The output section of Demo, both in the data-flow and the nondeterministic version, is:

```
output_section
{  output  out_matrix (int go)
     {  collects go from ALL VP[i][j];
        return B as S;
     }
}
```

An **output** *action* is specified independently for each output stream (just one in our case).

The **return** command causes the effective transmission of values onto the selected output stream. In our example, this command is executed when the synchronization signal *go* has been received *from all* the VPs. The clause **as** indicates that the value to be transmitted is the current value of a state variable (*S*) of VPs.

3.7. Pipeline semantics of the *parmod* sections

As seen, the functionalities of a *parmod* are decomposed into three blocks:

- **Input Section** (IS): control of nondeterminism, possible pre-processing and modification of state control variables, and distribution to VPs;
- **Virtual Processors** section (VPS): execution of the main function delegated to the module, and transmission of values to the Output Section;
- **Output Section** (OS), one *for each output stream*: collection of output parameters and/or state values from VPS, possible post-processing and modification of state control variables, and transmission onto the output channel.

These blocks cooperate according to a *pipeline scheme*, as shown in fig. 3:

The overlapping among the activities of the three sections is often very important for optimizing the performance and scalability, in particular to mask external communication *wrt* calculation time in VPs, and

also to mask the delay of those possible pre/post processing operations that may not be convenient to parallelize in the VP section. Furthermore, this organization allows the compiling tools to optimize the implementation of *parmod*s compositions, i.e. IS and OS of interfaced modules can often be merged.

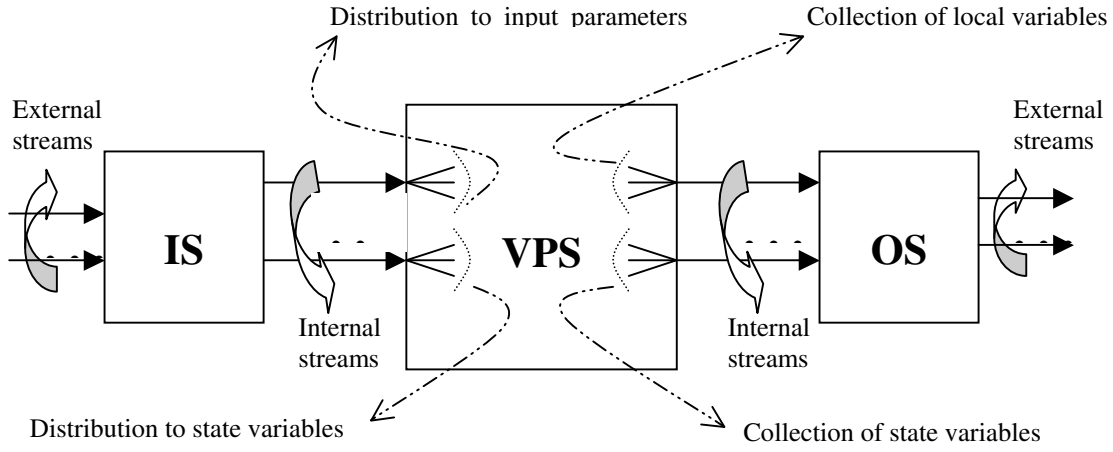


Fig. 3 – Pipeline semantics of the sections of a *parmod*

Internal streams exist between IS and VPS, and between VPS and OS, carrying both state values and input/output parameters. All the communication and consistency issues about the internal streams are entirely managed by the run-time support without any intervention of the programmer. The following rules:

i) the internal streams preserve the order,

ii) the values distributed from IS to VPS, and from VPS to OS, are consumed at each activation,

iii) new input values are distributed to state variables in VPS at the end of the current activation,

iv) if the *as* clause is used in the *return* command of some output stream controller, the state values are transmitted from VPS to OS at the end of the current activation,

are sufficient to ensure that the pipeline semantics is equivalent to the sequential semantics of IS-VPS-OS.

The programmer must be aware of the pipeline semantics of IS-VPS-OS only as far as the state control variables are concerned when, for synchronization or consistency reasons, a sequentialization has to be forced among the sections. This can simply be done by acting on the state control variables which are *shared* by IS, VPS and OS: for example, IS renders a guard false, and OS renders the same guard true again, i.e. OS "re-enables" some input streams (that were "disabled" by IS) and the consequent distribution of data to VPs. Of course, if the programmer is not interested in the pipelined behaviour, this is just the mechanism to force a sequential behaviour of IS, VPS and OS.

In the Demo example, no special synchronization among the sections is needed: we let IS, VPS and OS work in pipeline, thus overlapping all the activities of IS and OS to the parallel function calculation in VPS.

4. External objects

A module (sequential or parallel) of an ASSIST program can refer to external objects according to the interfaces/methods or APIs of such object. As seen, this is a mechanism to exploit (import) the functionalities of possibly existing objects defined outside the application.

External objects are also a mechanism to cooperate with other modules of the same application, in addition to the stream mechanism. While the streams can be referred only at the beginning and at the end of an activation (i.e. it cannot be referred during the activation of VPs), an external object can be referred by the *parmod* sections in any phase (input, VP processing, output).

An ASSIST *module* itself, or in general *a composition of modules*, *can be defined and exported*, i.e. *used as an external object/component* by (the same application or by) any other application, possibly written in

another formalism. In ASSIST 1.0 the CORBA IDL [29] interface mechanism is used for this purpose. The transformation of the interface mechanism (from streams to IDL invocations) is done automatically and is integrated in the compilation suite of ASSIST. The mechanism is based on a proper *IDL interface*, acting as a wrapper, and on a proper implementation of its methods to permit the interaction and/or the execution of a parallel program by remote clients.

In general, the goals of external objects are the following:

a) to provide a powerful mechanism to import/export abstract objects in commercial standards,

b) to provide a standard modality to interact with system functionalities (servers),

c) to optimize scalability when the by-reference communication is more efficient than the by-value one;

d) to overcome the limitations of the single node memory capacity in distributed architectures,

e) to make the management of dynamic and/or irregular program/data structures easier.

Three kinds of external objects are distinguished:

1) Shared variables

2) Distributed Shared Memory libraries

3) Remote objects

4.1. Shared variables

A first kind of external object is defined just in terms of the *same data types of* ASSIST-CL. Any variable can be defined as *shared* by modules of an ASSIST application. This can be interpreted as an extension of the concept of internal state of modules: now *the state can also be shared between distinct modules*.

In the Demo example, as shown in fig. 4, arrays A and B could be shared by Demo with the modules M_A, M_B by using the declarations

```
shared int A[N][N];
shared int B[N][N];
```

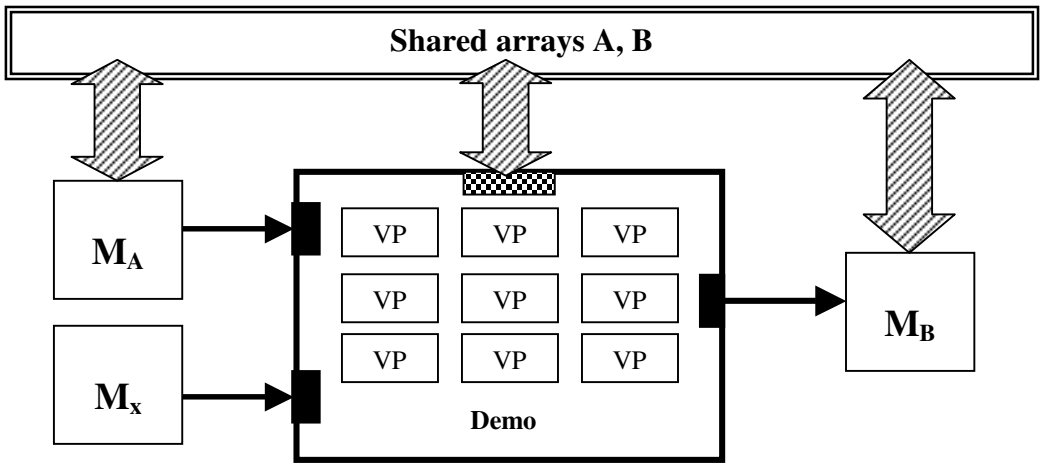


Fig. 4 – Utilization of shared variables

Defining *A* and *B* as shared variables, the streams from M_A to Demo and from Demo to M_B do nor carry the values of the arrays, but simply *synchronization signals* informing that a new copy has been produced in the shared space. Since no primitive synchronization mechanism is associated to the shared variables (they are "normal" ASSIST-CL types), it is responsibility of the programmer to provide the proper synchronization

using streams. Often, this very simple mechanism is sufficient to achieve a significant gain in performance by reducing the copying overhead of large data structures.

As with the internal state, shared variables can be partitioned or replicated to VPs, as well as distributed/collected to/from VPs, for example:

```
external_object A scatter onto VP
```

In the previous example, when a synchronization signal from M_A is received, the input section of Demo provides to access the external object *A* and to distribute it to VPs according to the same scatter strategy adopted in the first version (section 3.4).

4.2. Integration of Distributed Shared Memory libraries

In many problems, the goals *c), d), e)* mentioned above can be met by means of external objects expressed by abstractions of shared memory. In particular, we consider the integration of libraries for

i) Distributed Shared Memory (DSM)

ii) abstract objects implemented on top of some DSM.

While on shared variables we can only execute the operations corresponding to the ASSIST-CL types, on the shared memory objects the proper set of operations is defined for expressing powerful strategies of allocation, manipulation and synchronization.

Currently, we are using *DVSA* [6] for *i),* and *Shared_Tree* [8] and *SHOB* [6] libraries for *ii),* all such tools having been developed at the Computer Science Department of Pisa. However, any existing library (e.g. JIAJIA) can be integrated and utilized by the ASSIST. Of course, it is responsibility of the programmer to correctly utilize the library according to its semantics (interfaces, consistency model, data types).

A declaration like

```
using_lib mylib;
```

causes the integration of the environment of the *mylib* library into the ASSIST environment. If some *mylib* types are compliant with the ASSIST-CL types, they can also be used in the stream definitions. A module can declare the utilization of a library in the following way:

```
external_lib mylib type_mylib var_name;
```

4.3. Remote objects

The utilization of remote objects through CORBA (in the next versions of ASSIST, other commercial standards) is done according to modalities similar to the ones described for DSM libraries.

ASSIST-CL defines an ORB interface, called ASSIST**_orb**, which contains APIs to connect and utilize remote objects in ASSIST applications. No constraints are imposed on the access and utilization of a CORBA object by an ASSIST program, except that the registration modality of the external object (Single Thread or Multi Thread) must be verified according to the implementation of the object itself.

5. Forms of parallelism in ASSIST-CL

In this section we utilize some examples to show the range of utilization of ASSIST-CL in parallel programming. For the sake of brevity, the analysis will be done informally, in terms of modeling and structuring of the parallel problems, without writing detailed ASSIST-CL programs. All the examples have in common the possibility of using some unique features of ASSIST-CL:

- composition in the form of, possibly structured, graphs,
- flexibility in state assignment strategies,
- flexibility in input/output stream distribution strategies,

- combinations of nondeterministic and data-flow behaviours,
- flexibility in exploiting the VPs of a *parmod* (enabling sets),
- utilization of external objects,

In the general definition of ASSIST, no specific form of parallelism is associated to the VPs of a *parmod*:

a) at one extreme, we have an extension of structured parallel programming: all the VPs can participate collectively to parallel computations typical of the skeletons models, for example pipeline, farm, data parallel map, data parallel with communication stencil (fixed or variable or dynamic), data-flow loop, divide & conquer. Moreover, all these schemes can also be implemented as extensions, or variants, or combinations that are not usual in classical skeletons. Interesting examples, to be mentioned explicitly, are:

  i) *efficient combinations of task parallelism and data parallelism* (see sections 5.2, 5.3),

  ii) decomposing the sets of VPs of the same *parmod* into disjoint cooperating subsets realized according to different structured programming schemes,

  iii) the same set of VPs of a *parmod* exploits *different forms of parallelism during different activations* (see section 5.3);

b) at the other extreme, each VP could be defined independently from the others as in a generic concurrent MPMD program, including the possibility of dynamic activation of VPs;

c) between these two extreme cases, we can flexibly select parallel program structures that exploit the structured parallel programming principles as much as possible, without giving up the capability of introducing specific parallel patterns to deal with irregular and/or dynamic situations (see section 5.3).

Classes *a)* and *c)* are semantically well defined. In these schemes we exploit all the features of ASSIST-CL listed above, to express complex program structures which are supported in a reliable and efficient way already in ASSIST 1.0; examples will be discussed in the following subsections.

The schemes of class *b)* constitute a very heterogeneous set (including also lower abstraction level schemes), and currently it is not possible to give general structuring rules to predict the behaviour of any possible scheme inside this class: in fact, we are interested in program structures that can be analyzed at compile-time and for which the properties of the run-time support libraries are known once for all (correctness and efficiency properties). This does not exclude that many interesting programs of class *b)* can work, but *currently* correctness and/or efficiency issues are mainly at the responsibility of the programmer.

Some constraints about the structures expressed by a *parmod* depend upon the solution to the state consistency problem. In ASSIST 1.0 the basic constraint is the adoption of the "*owner computes rule*", as seen in the Demo example. Moreover ASSIST 1.0 contains the following constraints:

  i) no dynamic activation of VPs in a *parmod*,

  ii) no nesting of forms of parallelism inside a *parmod*, i.e. VPs are sequential.

In the next version of ASSIST the relaxation of these constraints will be studied, aiming at recognizing how several interesting cases, currently belonging to class *b),* can be sufficiently characterized to pass into class *c).* In particular more powerful schemes for state consistency in structured parallel programming will be experimented [2].

5.1. Emulation of "classical" skeletons and their composition

All the data-parallel skeletons can be expressed, easily and efficiently, by a *parmod*:

- Computations *map-like*
- Computations with *communication stencils*
  - *Static stencils*: stencils that are recognizable at compile time and that remain fixed during all the iterations (including *reduce* computations),

- *Variable stencils*: stencils that are recognizable at compile time and that vary passing from one iteration to the next one (including *multiprefix* computations),
- *Dynamic stencils:* stencils that cannot be recognized at compile time (index values are results of functions applied to the state of the computation).

All the simple stream-parallel skeletons are available as primitive ones. However, they can be implemented directly in ASSIST-CL, especially when a certain *problem-dependent personalization* is required:

- *Pipelines*, and in general pipelined executions of parallel structures, are expressed as compositions of sequential and/or parallel modules. Communications can be expressed by streams and/or by shared variables;
- *Farms*, and in general farms of parallel structures, can be expressed in two ways:
  - as *compositions of modules*, where some of them (e.g. emitter and collector in SkIE) are delegated to implement (possibly complex and problem-dependent) strategies of load balancing and stream ordering. Interactions can be expressed by streams and/or shared variables, or distributed shared memory objects;
  - as a *single parmod* with topology of kind *none* and with input stream distribution strategy of kind *on demand*. Read-only state variables may be replicated among the farm workers and/or shared. The load balancing strategy cannot be modified *wrt* the one implemented by the *on demand* distribution, while several stream ordering strategies can be easily expressed in the output section of *parmod*.

5.2. Efficient combinations of stream-parallel + data-parallel structures

The skeletons models allow programs to be structured as a combination of task parallel (stream-parallel) and data-parallel skeletons. However, in some cases this combination is a source of inefficiency because it introduces a substantial amount of synchronizations [35], typically:

- sequences of stream generations starting from data structures (e.g. as result of a data-parallel skeleton, or when a stream is "packed" for load balancing or consistency requirements) or vice versa,
- implementation of barriers for the initializations and termination of data-parallel skeletons.

In many problems, a larger degree of asynchrony and nondeterminism could be very useful, drastically increasing performance and scalability. This is possible with *parmod* in ASSIST-CL.

For example, consider the following simple computation

```
var A, B, C: array [1 .. N, 1 .. N] of T;
function F (T, T): T; ...
        forall i, j : = 1 to N do   C[i, j] := F(A[i, j], B[i, j])
```

with the constraint (imposed by the whole structure of the application) that array *A* is produced as a *stream of rows*, *B* as a *stream of columns*, and *C* as a *stream of single elements*. As known, this kind of computation is applied in many problems (e.g. simulation of waves in a volume of material).

A solution using a data-parallel *map* skeleton can be expressed simply, however it is quite inefficient because of the excessive overhead of the transformations from streams to partitioned data structures (*A, B*) and vice versa (*C*), as well as because of the sequence of initialisations and terminations of map.

Using a *parmod*, a fully asynchronous *systolic* computation with N^2 virtual processors can be easily programmed. Owing to the possibility of using *more than one stream* and *nondeterminism* control, a pipelined execution is achieved with minimum overhead: input scattering (*A* rows onto rows of VPs, *B* columns onto columns of VPs), calculation, and output gathering. Nondeterminism control is very useful in order to approximate the ideal temporal evolution in which the reception of an *A* row is associated to the reception of a *B* column. If this behaviour were achieved in a synchronous way, again this could introduce

additional overheads: in ASSIST-CL the desired behaviour is not forced rigidly, since temporary variations with respect the ideal situation are efficiently amortized by the asynchronous nature of *parmod*.

5.3. A dynamic computation: C4.5 algorithm for data mining

Many irregular/dynamic problems (e.g. N-body, adaptive multigrid) can be expressed effectively in ASSIST-CL using a suitable combination of *parmod*s and *shared objects* to solve the complex problems of load balancing and locality of references (typical schemes of class *c)*).

In this section we show the example of C4.5, a well-known data mining classifier according to the approach of *decision trees* [36, 14].
A decision tree recursively partitions the input set until the partitions consist (mostly) of data from a same class. The root of the tree corresponds to all the input, and each decision node splits the *cases* (records containing the list of attributes and a class identifier) of the training set according to some test on their attributes. The leaves are class-homogeneous, disjoint subsets of the input. A path from the root to any given leaf thus defines a series of tests. All cases in the leaf satisfy these tests, and they belong to a certain class. The tree describes the structure of the classes, and it is easy to use to predict the class of new, unseen data. Apart from some implementation issues, the general form of the tree induction process is mainly a *divide-and-conquer* (D&C) computation, proceeding from the root and recursively classifying smaller and smaller partitions of the input.

Though in principle the nature of the problem suggests a direct parallelization of the D&C scheme, it is difficult to achieve a good scalability by such approach. The problem is characterized by high degree of load unbalance, large memory space requirement, and heavy communication load. In [5, 8, 14] it has been shown that scalability can be sensibly improved by a hybrid parallelization strategy, in which D&C is merged with a data-parallel scheme applied to the *cases* of the training set. In the hybrid approach, we can recognize an initial sequence of computational steps during which a data-parallel approach is the most efficient since the issues of load balancing are not prevalent. This initial sequence is followed by another one in which load balancing is progressively becoming a serious problems: a farming scheme is now much more efficient than a data-parallel one. We can generalize this behaviour by recognizing a class of parallel problems for which different parallelization strategies can be alternated during successive phases of the computation.

Moreover, the dynamic nature of the data structures involved in a decision tree problem suggests that shared objects are a very important mechanism for solving the problems of programmability, memory space and communication load. This scheme has been emulated in [8, 14] by a mixed model "skeletons + shared objects" using the abstraction of the Shared_Tree library.

An ASSIST-CL solution is represented in Fig. 5. The detailed ASSIST program is reported in **Appendix B**.

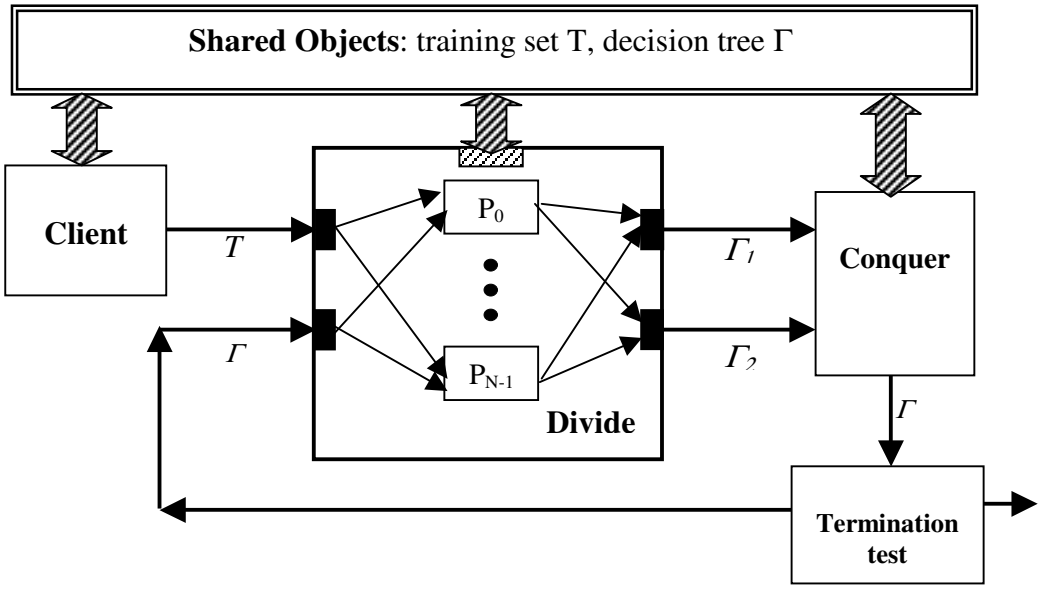


Fig. 5- Scheme of parallel C4.5 in ASSIST-CL

ASSIST-CL possesses the features that are required in parallelizing a problem according to the approach above: *graph composition of modules, nondeterminism control on multiple streams, efficient combination of stream parallelism and data parallelism, shared objects*, and the possibility of *distinct behaviour in different phases of the parallel computation.*
The application modules are composed in a cyclic graph, in which the *Divide* module is a *parmod* with one-dimensional array topology. To better exploit the external objects facility, streams are of type "reference to shared objects (of type decision tree or of type training set)". The input streams (two in the figure, however this number can be increased in the generalization of the problem) are managed according to the forms of parallelism to be adopted during the various phases: stream T is distributed with a *scatter* strategy, in order to drive a data-parallel computation in VPs during the first phase, while stream Γ is distributed with an *on demand* strategy in order to use the VPs as a farm during successive phases. Correspondingly, more output streams are defined for the *Divide* module in order to clearly distinguish the parallelism form to be selected. *Conquer* and *Termination_test* are ASSIST modules, possibly of the kind *parmod-one*.

All this computation can be exported as a component to clients, possibly expressed in a different formalism.

6. Conclusion

We have presented a new programming environment oriented to the development of parallel and distributed high-performance multidisciplinary applications according to a unified approach. The proposal is based on the evolution and joining of the software component technology and of the structured parallel programming technology: component technology is characterized in the direction of parallel and distributed high-performance applications for emerging large-scale heterogeneous platforms, including Grids; structured parallel programming is extended, with respect to the pure skeletons model, for supporting more generality and flexibility, to render reuse of parallel software truly feasible, to exploit external objects, and to achieve better efficiently in irregular, dynamic and interactive applications, as well in complex combinations of task and data parallelism.

At the light of the very complex set of problems to be dealt with, our research in organized in successive phases. The first version of the coordination language will contain some constraints *wrt* to final goal, anyway it will be used in a rich set of applications developed in some current projects supported by the Italian Space Agency, the National Research Council, and the Ministry of University and Research. The main applications belong to the following areas: Earth Observation Systems and Environment Control; Scientific Simulation and Computational Chemistry; Knowledge Discovery in Data Bases, Question Answering and Search Engines; Image Processing and Geographical Information Systems.

As usually, increasing the expressive power of a formalism leads to an increase of the implementation difficulty. Some current constraints of ASSIST-CL are also due to the willingness to ensure efficiency of implementation. The current implementation is based on a flexible abstract machine and run-time support, which exploit the underlying mechanisms of ACE and Distributed Shared Memory libraries. The compiler, realized according to the object-oriented technology, initially makes use of a set of pragmas for the sake of experimentation. The first version of the implementation will run on homogenous parallel machines and clusters (Linux), and it will contain also basic interfaces for experimenting ASSIST in heterogeneous Grids. Work is in progress to define and to realize the next version in such a way that, in addition to progressively removing the language constraints, working with heterogeneous large-scale platforms and Grid programming will become outstanding features of ASSIST.

The architecture of the run-time supports and tools, and the capability of efficiently adapting the cooperation of parallel and distributed components to the features of the various underlying technologies, is fundamental for demonstrating the power and the efficacy of the new approach to the development of high-performance applications on large-scale platforms. *A substantial part of our research is devoted to this task.* The purpose of this initial paper has been to show the principles and features of the proposed approach in terms of the programming model.

Successive papers will describe the architecture and implementation of run-time support and development tools in detail, as well as will report on performance evaluations for significant examples and applications. First experiences and benchmarks, as well as companion experiences done in the past with formalisms emulating ASSIST-CL (e.g. SKIE + shared objects [14]), are very encouraging.

Acknowledgments

The paper describes a part of the work done by the ASSIST Group at the Department of Computer Science, University of Pisa. Special thanks are due to: Marco Danelutto, coordinator of the programming environment activities; Davide Guerri (Synapsis) and Paolo Pesciullesi, who played a very significant role in the definition of ASSIST and are working enthusiastically to the implementation together with Pier Paolo Ciullo, Marco Lettere (Synapsis), Roberto Ravazzolo and Massimo Torquati (HuginSoft) and Corrado Zoccolo. Leonardo Vaglini participated actively to the definition of ASSIST-CL. Andrea Controzzi and Sonia Campo (PhD student) worked in various phases of the project. I wish also to thank Marco Aldinucci and Massimo Coppola, who demonstrated the feasibility of the combination of structured parallel programming and external objects on data-mining algorithms, for continuous help and useful discussions.

References

[1] R.C. Armstrong, A. Chung, "POET (parallel object-oriented environment and toolkit) and frameworks for scientific distributed computing", *Hawaii Int. Conf. on System Sciences*, 1997.

[2] M. Aldinucci, *Smart Memory Parallel Architectures,* PhD Thesis, Dept. Comp. Science, Univ. of Pisa, 2002.

[3] H.E Bal, M. Heines, "Approaches for Integrating Task and Data Parallelism", *IEEE Concurrency*, IEEE Computer Society, vol. 6, n. 3, 1998, 74-84.

[4] M. Baker, R. Buyya, and D. Laforenza, "The Grid: International Efforts in Global Computing", *Proc. of SSGRR 2000 Computer & eBusiness Conference*, L'Aquila, July 31 – August 6, 2000

[5] P. Becuzzi, M. Coppola, S. Ruggieri, M. Vanneschi, "Parallelisation of C4.5 as a Particular Divide & Conquer Computation", *3rd Workshop on High Performance Data Mining*, May 2000, Cancun, Mexico.

[6] F. Baiardi, D. Guerri, P. Mori, L. Moroni, L. Ricci, "Two Layers Distributed Shared Memory", *Proc. HPCN*, 2001.

[7] F. Baiardi, M. Vanneschi, *Concurrent Programming Languages*, F. Angeli.

[8] G. Carletti, M. Coppola, "Structured Parallel Programming and Shared Objects: Experience in Data Mining Classifiers", *Proc. ParCo 2001 Int. Conf.*

[9] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski, "Toward a common component architecture for high performance scientific computing. *Proc. 8th High Performance Distributed Computing (HPDC'99),* 1999

[10] *Common Component Architecture Forum*, www.acl.lan1.gov/cca-forum.

[11] G. Larsen (ed), Component-based Enterprise Frameworks, Special Issue of *Communications of the ACM*, October 2000, vol. 43, n. 10.

[12] Clark D., "Face-to-face with Peer-to-Peer Networking", *IEEE Internet Computing*, January 2001, 18-21.

[13] M. Cole, *Algorithmic skeletons: structured management of parallel computation*, MIT Press, Cambridge, Mass., 1989.

[14] M.Coppola and M. Vanneschi, "High Performance Data Mining with Skeleton-based Structured Programming", to appear in *Parallel Computing*, special issue on Parallel Data Intensive Algorithms, Elsevier Science.

[15] M. Danelutto, "On Skeletons and Design Patterns" *Proc. ParCo 2001 Int. Conf.*

[16] J. Darlington, Y. Guo, H.W. To, Y. Jing, "Skeletons for Structured Parallel Composition", in *Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.

[17] *Report on the Programme consultation meeting "Large-scale Distributed Systems and Platforms, including GRID-based Systems",* Brussels, 10-11 April 2001, European Commission, Directorate-General Information Technology

[18] I. Foster, C. Kesselman, *The Grid: blueprint for a new computing infrastructure*, Morgan Kaufmann, 1999.

[19] I. Foster, C. Kesselman, S. Tuecke, *The Anatomy of the Grid*, January 2001

[20] I. Foster, C. Kesselman, J.M. Nick, S. Tuecke, *The Physiology of the Grid*, January 2002

[21] C. Lee, S. Matsuoka, D. Talia, A. Sussman, N. Karonis, G. Allen, J. Saltz, "A Grid Programming Primer", *Global Grid Forum 2*, Washington D.C., July 2001.

[22] D. Goswami, A. Singh, B. Preiss, “Using object-oriented techniques for realizing parallel architectural skeletons”, *LNCS* 1732, 1999, 130-141.

[23] High Performance Fortran Forum, *High Performance Fortran Language specification version 2.0*, 1997.

[24] C.A.R. Hoare, *A Model for CSP*, Oxford University Report.

[25] K. Keahey, P. Beckman, J. Ahrens, “Ligature: component architecture for high performance applications”, *The International Journal of High Performance Computing Applications*, vol. 14, n. 4, Winter 2000, 347-356.

[26] K. Keahey, D. Gannon, “PARDIS. a parallel approach to CORBA”, *6th IEEE Int. Symp. on High Performance Distributed Computing,* 1997, 31-39.

[27] C. Koelbel, D. Lovemann, R. Schreiber, G. Steele, M. Zosel, *The High Performance Fortran Handbook*, MIT Press, 1994.

[28] Newhouse S., Mayer A., and Darlington J., “A Software Architecture for HPC Grid Applications”, A. Bode et al. (Eds.), *EuroPar 2000, LNCS* 1900, pp. 686–689, 2000, Springer-Verlag

[29] Object Management Group, *The Common Object Request Broker: architecture and specification*, 1995.

[30] B. Smolinski, S. Kohn, N. Elliott, N. Dykman, “Language interoperability for high-performance parallel scientific components“, *LNCS* 1732, 1999, 61-71

[31] B. Bacci, M. Danelutto, S. Pelagatti, M. Vanneschi, "SkIE: A heterogeneous environment for HPC applications", *Parallel Computing*, 25:1827-1852, Dec. 1999

[32] D.B. Skillicorn, D. Talia, “Models and languages for parallel computation”, *ACM Computing Surveys*. 30 (2) 123-169, 1998.

[33] M. Vanneschi, “Heterogeneous HPC environments”, invited paper, *4th Int. Euro-Par Conference*, Southampton, Sept. 1998, in D. Pritchard and J. Reeve (Eds.), *Lecture Notes in Computer Science*, vol. 1470, pp. 21-34.

[34] M. Vanneschi, “PQE2000: HPC tools for industrial applications”. *IEEE Concurrency*, IEEE Computer Society, vol. 6, n. 4, 1998, 68 – 73.

[35] M. Vanneschi, “Parallel Paradigms for Scientific Computing”, *Lecture Notes in Chemistry,* vol. 75, 170-183, 2000.

[36] M.J. Zaki, “Parallel and Distributed Data Mining: an Introduction”. In: M.J. Zaki & C.T. Ho, *LNAI* 1759, 1999.

Appendix A

Example of *parmod* for a variable stencil data-parallel program

(section 3)

a) Data-flow version

```
parmod  Demo_Dataflow (input_stream in_scalar (int x), in_matrix (int A [N][N]);
                       output_stream out_matrix (int B [N][N]))
{

topology  array [i:N][j:N] VP;

attribute    int S[N][N] scatter onto VP;
attribute    int L replicated;
init { // empty in this example }

input_section
{  on in_scalar and in_matrix
             distribution A scatter to S;
             distribution x broadcast to L;
}

virtual_processors
  {Compute_with_stencil(in in_scalar, in_matrix; out out_matrix)
       {VP i=1..N, j=1..N
          {for (h=0; h<N; h++)
                    F(in L, S[i][h], S[h][j]; out S[i][j]);
             assist_out (out_matrix.go, 1);
          }
       }
  }

output_section
{  output  out_matrix (int go)
     {  collects go from ALL VP[i][j];
        return B as S;
     }
}

} // end of data-flow version
```

b) Nondeterministic version

```
parmod  Demo_Nondet (input_stream in_scalar (int x), in_matrix (int A [N][N]);
                     output_stream out_matrix (int B [N][N]))
{

topology  array [i:N][j:N] VP;

attribute bool accept_x;
attribute int priority_A, priority_x; init
{ L = <initial value of x, replicated in all VPs>;
  accept_x = false;
  priority_A = 1;
  priority_x = 0;
}

input_section
{  on priority_A, , in_matrix
        distribution A scatter to S;
        operation {accept_x = true;
                   priority_A = 0;
                   priority_x = 1;
                  }
  on priority_x, accept_x, in_scalar
        distribution x broadcast to L;
        operation { priority_A = 1;
                    priority_x = 0;
                  }
}

virtual_processors
  { Compute_with_stencil_A (in in_matrix; out out_matrix)
        { VP i=1..N, j=1..N
            { for (h=0; h<N; h++)
                  F(in L, S[i][h], S[h][j]; out S[i][j]);
              assist_out (out_matrix.go, 1);
        }
    Compute_with_stencil_x (in in_scalar; out out_matrix)
        {VP i=1..N, j=1..N
            { for (h=0; h<N; h++)
                  F(in L, S[i][h], S[h][j]; out S[i][j]);
              assist_out (out_matrix.go, 1);
            }
        }
  }
```

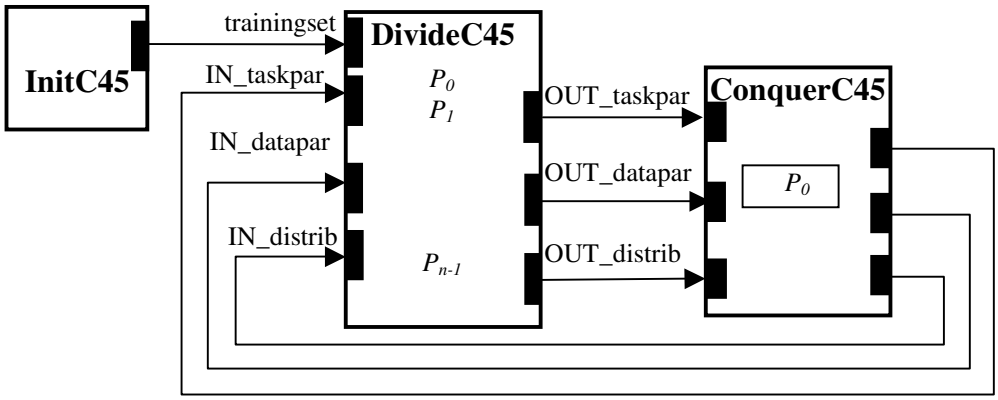
```
output_section
{  output  out_matrix (int go)
     {  collects go from ALL VP[i][j];
        return B as S;
     }
}

} // end of nondeterministic version
```

Appendix B

ASSIST implementation of C4.5 algorithm for Data Mining

This Appendix contains the ASSIST implementation of C4.5, which has been introduced, in a simplified version, in section 5.3. The program is due to Massimo Coppola and the ASSIST group.

The ASSIST program contains a sequential module **InitC45** and two parallel modules, **DivideC45** and **ConquerC45** (*one*). The data structures *Training Set (TS)* and *Decision Tree (DT)* are maintained in the Distributed Shared Memory (external objects) for memory space and programmability reasons.



InitC45 task is to generate and initialize *TS* and the reference to *DT* root. Both references are passed to *Divide C45* onto the stream "trainingset".

DivideC45 is the core of the program. It shows the *parmod* power and flexibility to solve irregular parallel problems. During a first phase, the split attribute search and data redistribution is executed according to a *data-parallel* style: each VP works on a horizontal partition of TS. In a second phase, when the task granularity is fine enough, *Divide C45* works according to a *task parallel* style: each VP computes a whole subtree autonomously.

Also *Conquer C45* works differently in different phases. When *DivideC45* works in a data-parallel way, *Conquer C45* collects the results about the split decision, updates DT, sends to *Divide C45* a table to drive the data redistribution, and receives a redistribution summary from *Divide C45*. One the data are partitioned, Conquer C45 is able to state whether the task parallel phase has to be started.

Three distinct streams are used to identify the computations to be performed during the different phases.

```
generic main() {
       stream TS_ref_t IN_trainingset;
       stream DT_ref_t IN_datapar;
       stream DT_ref_t OUT_datapar;
       stream DT_ref_t IN_taskpar;
       stream DT_ref_t OUT_taskpar;
       stream distrib_t IN_distrib;
       stream distrib_res_t OUT_distrib;

       InitC45(output_stream IN_datapar, IN_trainingset);
       DivideC45(input_stream IN_datapar, IN_taskpar, IN_distrib, IN_trainingset
                output_stream OUT_datapar, OUT_taskpar, OUT_distrib);
       ConquerC45(input_stream OUT_datapar, OUT_taskpar, OUT_distrib
                  output_stream IN_datapar, IN_taskpar, IN_distrib);
}
```

```
#define N ...


InitC45(output_stream IN_datapar(DT_ref_t DT), IN_trainingset(TS_ref_t TS))
$C{
	//...
	//C code to generate and to initialize Training Set and Decision Tree.
	//...

	assist_out(IN_datapar, DT);
	assist_out(IN_trainingset, TS);
}C$


parmod DivideC45(input_stream IN_datapar(DT_ref_t dpDT),
                              IN_taskpar(DT_ref_t tpDT),
                              IN_distrib(distrib_t dist),
                              IN_trainingset(TS_ref_t TS)
                 output_stream OUT_datapar(result_t dpRes),
                               OUT_taskpar(result_t tpRes),
                               OUT_distrib(distrib_res_t distRes))
{

	topology array[i:N] PV;

	attribute TSref_t TSref;
	attribute bool lg_distrib;
	attribute bool lg_TS;
	attribute bool lg_DT;

	init{
		lg_TS  = TRUE;
		lg_distrib = FALSE;
		lg_DT = FALSE;
	}


	input_section {
		on lg_TS, ,IN_trainingset{
			operation{
				lg_DT  = TRUE;
				lg_TS = FALSE;
				Tsref = TS;
			}

		on lg_DT, ,IN_datapar{
			distribution dpDT broadcast to PV;
			operation{
				lg_DT = FALSE;
				lg_distrib = TRUE;
			}

		on lg_DT, ,IN_taskpar{
			distribution tpDT on demand to PV;
		}
```

```
        on , ,IN_distrib{
                distribution dist broadcast to PV;
                operation{
                        lg_DT = TRUE;
                        lg_distrib = FALSE;
                }
}

virtual_processors{
        elabDataPar(in IN_datapar out OUT_datapar){
                VP i{
                        seqElabDataPar(in i, Tsref, IN_datapar.dpDT
                                        out OUT_datapar.dpRes);
                }
        }

        elabtaskPar(in IN_taskpar out OUT_taskpar){
                VP i{
                        seqElabDataPar(in i, Tsref, IN_taskpar.tpDT
                                        out OUT_taskpar.tpRes);
                }
        }

        elabDistrib(in IN_distrib out OUT_distrib){
                VP i{
                        seqElabDistrib(in i, Tsref, IN_distrib.dist
                                        out OUT_distrib.distRes);
                }
        }
}

output_section {
        output OUT_datapar(in result_t dpRes){
                local result_t res;

                collects dpRes from ALL PV;
                {
                        //collect the partial results of all the data-parallel
                        // computations, and finds the maximum of the gain function
                        // over the whole TS; fill the filds of local variable res
                        // with the values of the current decisions

                }
                // then sends the results of a data-parallel computation
               // to Conquer C45.

                return res;
        }

        output OUT_taskpar(in result_t tpRes){

                // sends a result of the task-parallel computation to Conquer C45
                collects tpRes from ANY PV;
                return tpRes;
        }
```

```
            output OUT_distrib(in distrib_result_t distRes){

                  //each VP participates to data redistribution and fills a table
                  // with the obtained results

                  collects distRes from ALL PV;

                        // executes the merging of data redistribution results.
                        // The produced table, distPes, is sent to Conquer C45
                  return distRes;
            }
      }
}


parmod ConquerC45(input_stream OUT_datapar(result_t dpRes),
                                 OUT_taskpar(result_t tpRes),
                                 OUT_distrib(distrib_result_t distRes)
                  output_stream IN_datapar(DT_ref_t dpDT),
                                  IN_taskpar(DT_ref_t tpDT),
                                  IN_distrib(distrib_t dist))
{
      topology one;
      attribute result_t res;
      attribute DT_ref_t DT;
      attribute lg_DT;
      attribute lg_distrib;

      init{
            lg_DT = TRUE;
            lg_distrib = FALSE;
      }


      input_section {
            on lg_data, OUT_datapar{
                  operation{
                        lg_DT = FALSE;
                        lg_distrib = TRUE;
                        res = dpRes;
                  }

            on lg_task, OUT_taskpar{
                  operation{
                        res = tpRes;
                  }
            on lg_distrib, OUT_distrib{
                  operation{
                        lg_DT = TRUE;
                        lg_distrib = FALSE;
                  }
            }
      }
```

```
	virtual_processors{
		conquerDataPar(in OUT_datapar out IN_distrib){
			seqConquerDataPar(in res out DT, IN_distrib.dist);
		}

		conquerTaskPar(in OUT_taskpar out IN_taskpar){
			seqConquerTaskPar(in res out IN_taskpar.tpDT);
		}

		conquerDistrib(in OUT_distrib out IN_datapar, IN_taskpar){
			seqConquerDistrib(in OUT_distrib.distRes
					out IN_datapar.dpDT, IN_taskpar.tpDT);
		}
	}


	output_section {
		output IN_datapar(in DT_ref_t dpDT) {
			collects dpDT from ALL PV[i][j];
			return dpDT;
		}

		output IN_taskpar(in DT_ref_t tpDT) {
			collects tpDT from ALL PV[i][j];
			{
				if (lg_data == true){
					lg_data = false;
					lg_task = true;
				}
			}
			return tpDT;
		}

		output IN_distrib(in dist_t dist) {
			collects dist from ALL PV[i][j];
			return dist;
		}

	}
}


proc seqElabDataPar(in int i, TS_ref_t TS, DT_ref_t DT out result_t Res)
$C{
	//...
	// sequential code for the data-parallel phase (maximum of gain function)
	//...
}C$

proc seqElabTaskPar(in int i, TS_ref_t TS, DT_ref_t DT out result_t Res)
$C{
	//...
	// sequential code for the task-parallel phase (load balanced computation
	// on subtrees according to the farm paradigm)
	//...
}C$
```

```
proc seqElabDistrib(in int i, TS_ref_t Tsref, distrib_t dist
                    out distrib_res_t distRes)
$C{
       //...
       // sequential code for the parallel redistribution of TS n-tuples.
       //...
}C$

proc conquerDataPar(in result_t res out DT_ref_t DT, distrib_t dist)
$C{
       // updates TD for the data-parallel phase
       //...
}C$

proc conquerTaskPar(in result_t res out DT_ref_t tpDT)
$C{
       // update TD for the task-parallel phase
       //...
       assist_out(IN_taskpar, DT);
}C$

proc seqConquerDistrib(in distrib_result_t distRes
                       out DT_ref_t dpDT, DT_ref_t tpDT)
$C{
       //...
       // process results obtained by a parallel redistribution.
       // Tasks are distributed onto the proper output stream
       //...
       If (.../*condizione => datapar*/... ){
              //prepara un'altra elaborazione data parallel
              //...
              assist_out(IN_datapar, dpDT);
       }
       else{
              /* condizione => taskpar */
              //prepara un'elaborazione task parallel
              //...
              assist_out(IN_taskpar, tpDT);
       }
}C$
```