UNIVERSIT¿ DI PISA

DIPARTIMENTO DI INFORMATICA

TECHNICAL REPORT: TR-04-09

ASSIST as a Research Framework for High-performance Grid Programming Environments

Marco Aldinucci, Massimo Coppola,
Marco Danelutto, Marco Vanneschi, Corrado Zoccolo

February 16, 2004

ADDRESS: via F. Buonarroti 2, 56127 Pisa, Italy. TEL: +39 050 2212700, FAX: +39 050 2212726

ASSIST as a Research Framework for High-performance Grid Programming Environments

Marco Aldinucci [(+*)], Massimo Coppola [(+*)], Marco Danelutto [(*)], Marco Vanneschi [(*)], Corrado Zoccolo [(*)]

[(*)] Dipartimento di Informatica, Universit‡ di Pisa

[(+)] Istituto di Scienza e Tecnologie dellíInformazione, CNR, Pisa

Abstract

ASSIST (A Software development System based upon Integrated Skeleton Technology) is a programming environment oriented to the development of parallel and distributed high-performance applications according to a unified approach. The language and implementation features of ASSIST are a result of our long-term research in parallel programming models and tools. ASSIST is evolving towards programming environments for high-performance complex enabling platforms, especially Grids. In this paper, we show how ASSIST can act as a valid research vehicle to study, experiment and realize Grid-aware programming environments for high-performance applications. Special emphasis is put on the innovative methodologies, strategies and tools for dynamically adaptive applications, that represent the necessary step for the success of Grid platforms.

First we discuss the conceptual framework for Grid-aware programming environments, based upon structured parallel programming and components technology, anticipating how ASSIST possesses the essential features required by such framework. Then we summarize the ASSIST programming model, showing its evolution, along the line of structured parallel programming, to solve critical problems of expressive power, flexibility, interoperability and efficiency; some examples, both of kernels and of complex applications, are used to point out the ASSIST features. The modular compiler model and the current implementation for heterogeneous platforms and Globus-based Grids are illustrated. We show the features that allow ASSIST programs to be used in CORBA infrastructures, that represents our basic starting point towards interoperability in Grid applications. Finally, the presentation of all the previous issues is used to derive an ASSIST-based model for supporting dynamically adaptive applications.

This work has been supported by the Italian MIUR FIRB Grid.it project (RBNE01KNFP) on High-performance Grid Platforms and Tools, and by the MIUR CNR Strategic Project L 499/97-2000 on High-performance Distributed Enabling Platforms.

1. Introduction: high-performance application development and Grids

The research activity of our group at the Department of Computer Science, University of Pisa, is focused on programming models and environments for the development of high-performance multidisciplinary applications. The enabling computing platforms we are considering are complex distributed architectures, whose nodes are parallel machines of any kind, including PC/workstation clusters. In general such platforms are characterized by heterogeneity of nodes, and by dynamicity in resource management and allocation. In this context, Grid platforms at various levels of integration [23], are of main interest, including complex distributed structures of general and dedicated subsystems, private heterogeneous networks, and systems for pervasive and ubiquitous computing. In the following, we shall speak of Grids to refer to such architectural scenario.

A *Grid-aware application* must be able to deal with heterogeneity and dynamicity in the most effective way (*adaptive* applications), in order to guarantee the *specified* level of performance in spite of the variety of run-time events causing modifications in resource availability (load unbalancing, node/network faults, administration issues, emergencies, and so on). With respect to traditional platforms, now it is much more important to rely on application development environments and tools that guarantee *high-level programmability* and *application compositionality*, software *interoperability* and *reuse*, and, at the same time, to be able to achieve *high-performance* and *capability to adapt to the evolution of the underlying technologies* (networks, nodes, clusters, operating systems, middleware, and so on) [7, 10, 22, 31, 35, 36, 40]. Achieving this high-level view of Grid application development is the basic goal of our research, notably in the *Grid.it* national project [29] and associated initiatives at the national and European level.

Though the programming environment and infrastructure must take into account outstanding issues in domain administration and security too, in this document we concentrate on the very fundamental issue of the programming model and its implementation. Our view of Grid application development is summarized by the level structure shown in Fig.1.

The Programming Environment is centered on the existence of a high-level, high-performance programming model and related development tools. A high-level view of compositionality, interoperability, reuse, performance and application adaptivity characterizes the Programming Environment we advocate. Applications are expressed entirely on top of this level. The level denoted by *Grid Abstract Machine* includes all the functionalities to support the preparation, loading and execution of the applications expressed in the formalism of the programming environment and transformed by the compiling tools. The Grid Abstract Machine *includes* the functionalities that, in the current view of Grids, are provided by the *Middleware tools and services*, e.g. moving bottom-up: the Connectivity (micro-kernel), Resource (resource management services) and Collective (collective and dynamic resource control and allocation) levels [24]. This Middleware may be one of the current/standard products (Globus Toolkit and its evolutions [23, 24, 25, 26]), or a subset of the services performed by them.

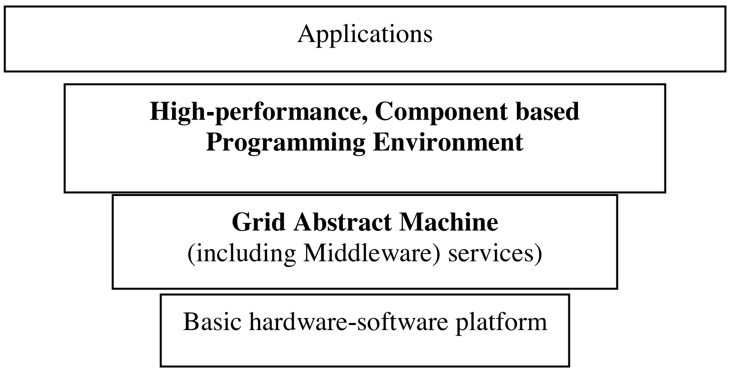

Applications
High-performance, Component based Programming Environment
Grid Abstract Machine
(including Middleware) services)
Basic hardware-software platform

Fig.1 ñ The role of Programming Environment in Grid application development

The Grid Abstract Machine exploits a subset of the Middleware services and adds very critical functionalities that *support the programming model and the development tools*, including all the strategies for resource management and scheduling and re-scheduling, allocation and re-allocation, as well as all the actions concerning the application structuring and re-structuring. By replacing the old-fashion OS-like view - according to which the application development occurs directly on top of the Middleware - by the view centered upon the Programming Environment and the Grid Abstract Machine, we wish to stress the programming-model based approach to system design, and, at the same time, to minimize the amount and variety of functionalities that are present in the underlying levels: i.e. these functionalities must be limited just to the support to the programming model and tools used to build Grid-aware, adaptive applications. Potentially, this approach leads to achieve a much better trade-off between programmability and interoperability, on one side, and performance, on the other side.

From the discussion above, it follows that the fundamental research issues, to design innovative platforms for Grid-aware applications, are the programming model and its implementation strategies. Other notable research projects [7, 10, 11, 17, 18, 22, 31, 35, 36, 40] propose to follow a similar approach, each one with its own characterization.

In this paper, we illustrate how ASSIST [41, 1, 2, 3] is a valid research vehicle to pursue this programming-model driven line. ASSIST (A Software development System based upon Integrated Skeleton Technology) is a programming environment oriented to the development of parallel and distributed high-performance applications according to a unified approach. It is supported by projects funded by the Ministry of University and Research, National Research Council, and Italian Space Agency.

In Sec. 2 we discuss the conceptual framework for Grid-aware programming environments, based upon structured parallel programming and components technology, anticipating how ASSIST possesses the essential features required by such framework. In Sec. 3 we summarize the ASSIST programming model, showing its evolution, along the line of structured parallel programming, to solve critical problems of expressive power, flexibility, interoperability and efficiency. Sec. 4 illustrates the modular compiler model and the current implementation for heterogeneous platforms and Globus-based Grids. In Sec. 5 we show the features that allow ASSIST programs to be used in CORBA infrastructures, that represents our basic starting point towards interoperability in Grid applications. Sec. 6 extends the presentation of Sec. 3 about the ASSIST-based frameworks for dynamically adaptive applications. Sec. 7 summarizes the current status of our research and its fundamental steps in the next two years.

2. Programming model: distribution, parallelism, interoperability and adaptivity

Currently, Grid applications are often designed according to a low-level approach (i.e., by relying on the Middleware services directly, possibly through a Grid portal) and, in many cases, they consist in single jobs or in limited forms of job composition (e.g. DAGs). Parallelism, where present, is limited inside single jobs, in a way that does not affect the external structure of the application (e.g. a job may be a MPI program). The result is that rarely Grid applications are Grid-aware and high-performance.

As discussed in the previous section, our point of view is radically different. It is based on the definition and realization of a programming model with the following features:

1. applications are expressed as compositions of high performance components,
2. a uniform approach is followed for distributed and parallel programming: in general components exploits internal parallelism and are executed in parallel with each other,
3. the strategies to drive the dynamic adaptation of applications are expressed in the same high-level formalism of the programming model.

Fig. 2 summarizes their interrelationships, which form the conceptual frameworks on which we found our research approach.

We will show that the ASSIST model, in the current version and according to its foreseen evolution, is a research vehicle which is consistent with the requirements of such frameworks.

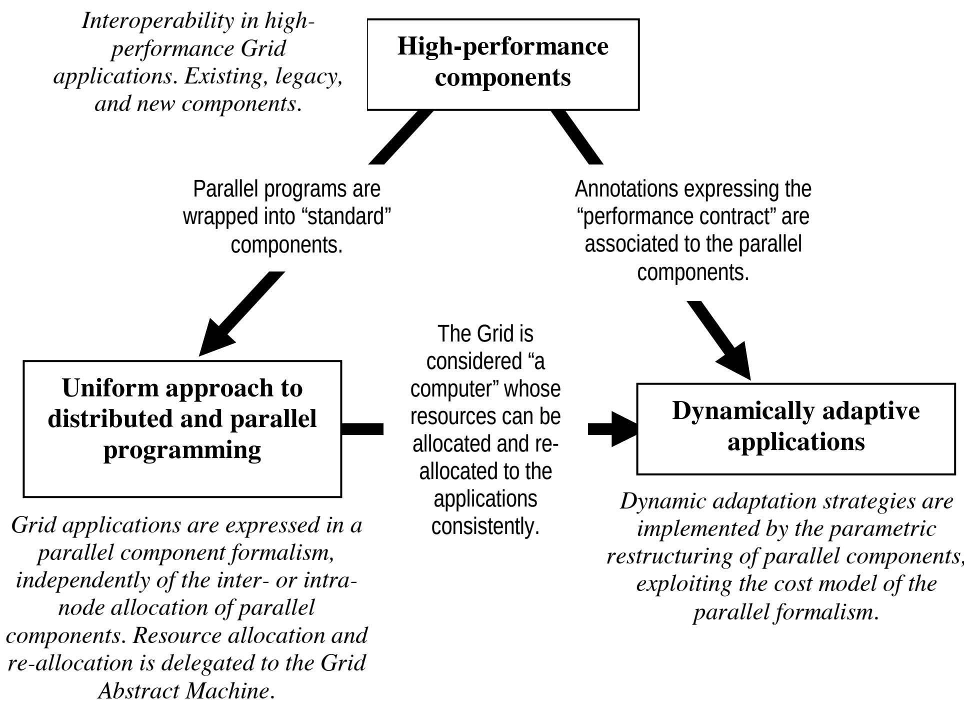

Interoperability in high-performance Grid applications. Existing, legacy, and new components.
High-performance components
Parallel programs are wrapped into "standard" components.
Annotations expressing the "performance contract" are associated to the parallel components.
Uniform approach to distributed and parallel programming
The Grid is considered "a computer" whose resources can be allocated and re-allocated to the applications consistently.
Dynamically adaptive applications
Grid applications are expressed in a parallel component formalism, independently of the inter- or intra-node allocation of parallel components. Resource allocation and re-allocation is delegated to the Grid Abstract Machine.
Dynamic adaptation strategies are implemented by the parametric restructuring of parallel components, exploiting the cost model of the parallel formalism.

Fig. 2 ñ The conceptual framework for Grid-aware programming environments

2.1. Grid-aware applications as compositions of high-performance components

Feature 1 is based on the proper exploitation of the *component* technology [9]. In our view, components are the basic mechanism to achieve compositionality by guaranteeing software interoperability and reuse. Here, we assume that the basic features of this software technology are known to the reader.

Achieving high-performance in component technology is currently an important research issue [5, 12, 30, 34, 35, 36]. Currently, we are evaluating how the existing standards (CCA [12, 28,], Java Beans [39], CCM [15], Web Services [16, 26, 43]) can be assumed as starting points to define and realize a robust component-based high-performance programming model, that can be widely accepted and that is able to interoperate in many application areas.

As we shall see, ASSIST provides the *abstraction* of high-performance components and high-performance composition of components, independently of any commercial standard. This allows us to understand *the basic features that high-performance components should possess*, in particular from the point of view of computation structuring, parallelism exploitation and modularity. These features will be properly merged with one or more commercial standard, or their future high-performance versions, in order to achieve extensive interoperability and reuse. The merging of high-performance programming and component technology must allow the designer to structure the application as the proper *composition of components*, i.e. some of which may be already existing / legay (possibly in binary form), and other ones are programmed from scratch (e.g. written in ASSIST) or as the combination of existing software into new parallel structures.

The current version of ASSIST (ASSIST 1.2) supports heterogeneity and the interoperability with several currant standards, in particular the *CORBA interoperability* [15, 33]: that is, not only an ASSIST program can act as a client of a CORBA server, but ASSIST programs can be easily defined as, and automatically transformed into, CORBA servers invoked by any CORBA client. The performance penalties introduced by the CORBA infrastructure, with respect to pure ASSIST programs, are quite acceptable for many applications with reasonable granularity. Though referred to an object-oriented approach, this experience proves that interoperability features can be merged into the ASSIST model, in order to design applications as composition of components, some of which are possibly parallel.

2.2. Uniform approach to distributed and parallel programming for Grid-aware applications

Despite the current limitations in Grid application development, Grid applications have to be *distributed* in the real meaning of the word, as known in theory since many years. With Feature 2 of the conceptual framework we further characterize this concept: we design a Grid application as a *parallel program* described by the parallel composition of parallel components (and possibly existing components). *No distinction is made a priori between parallelism and distribution*, i.e. between modules to be executed in the same (possibly parallel) Grid node or in distinct Grid nodes. In the same way, we do not restrict the application to be a single (sequential or internally parallel) job or a DAG of jobs. In general, the structure of the application can be any graph whose nodes are (parallel) components and the arcs are the mechanisms for their composition and interaction. The programming model of ASSIST is based on this concept.

At this point, it is important to clarify that modeling a Grid application as a parallel program does not necessarily mean that we are considering a Grid merely as a parallel machine, though in some cases this is a meaningful and effective view. There may be applications in which we could not be interested in inter-component parallelism or in optimizing such potential parallelism, possibly exploiting the parallelism at the intra-component level and forcing distinct components to be allocated onto distinct Grid nodes.

However, there are strong reasons in support to a *uniform view of distributed programming and parallel programming*, notably:

a) Provided that the module granularity is determined properly, there are many applications that can greatly benefit from inter-node parallelism, while additional performance is gained at the intra-node level. We point out that, because of the heterogeneous and dynamic nature of Grid platforms, the a-priori distinction between inter- and intra-node parallelism may be difficult or, by forcing it in some way, it may cause a sensible degradation in performance or fault-tolerance. Instead, we believe that the distinction between inter- and intra-node parallelism must be *delegated to the programming tools*, both at compile- and at run-time: as said in Sec. 1, it is for this reason that the resource management, scheduling and allocation functionalities must belong to the Programming Environment support (the Grid Abstract Machine), and in particular to the support of the programming formalism.

b) As a consequence, an approach that does not limit the parallelism opportunities is characterized by much more flexibility and performance: notably, it must be possible to adapt applications, without sensible or no modifications, to changes and evolutions in the underlying platform, such as in node architecture and multiplicity, communication latency / bandwidth, processor power, operating system facilities, and so on. This aspect is consistent with the trends in component technology, e.g. *application versioning* according to different requirements of users and/or availability of system resources.

2.3. Dynamically adaptive applications

The considerations of the previous points *a)* and *b)* are generalized to the possibility of developing dynamically adaptive applications, i.e. applications whose resource allocation varies at run-time to guarantee a desired level of performance. Re-scheduling and re-allocation of resources should occur because of node unavailability or node unbalancing, or because an increase in performance is required in response to an emergency (e.g. in an Earth Observation application for landslip detection, the response to some events may require a very large increase in computing power that can be rendered available by a large collection of Grid-connected machines).

Currently, despite some interesting ongoing projects [7, 10, 11, 17, 18, 22, 29, 31, 38, 40, 42, 44] try to find innovative solutions, when this problem is addressed only partial solutions are mentioned: notably, dynamic code/data movement.

In general, the problem does not consist merely in finding a better allocation of the same code and data, instead we need to take into account other more complex actions that imply a *transformation* of the executable version of the program, such as

- a different degree of parallelism,
- different data distribution and partitioning,
- and also alternative versions of the program implementing the same functionality, i.e. a different implementation of the same component or composition of components.

A rigorous approach to the adaptivity problem can be based upon the following points:

a) several *modalities of expressing the structuring and restructuring of a computation* must be available in the programming formalism,

b) these modalities must be characterized by a *cost model* (performance model) that can drive the structuring and restructuring phases with reasonable complexity and overhead.

These modalities can correspond to the usage of different (combinations of) *parallelism forms*, or *parallelism paradigms*, as it normally happens in *structured parallel programming* models based upon the *skeletons* concept [13, 6, 20, 4, 41]. In such models a consistent set of parallelism forms is provided to the programmer to structure/restructure the application at hand: for example, pipelines, farms or divide&conquer are typical task-parallel (stream-parallel) paradigms, while map, reduce, prefix, scan, stencil are typical data-parallel paradigms. In structured parallel programming, a *coordination language* is adopted that acts as a meta-language used to compose codes expressed in any standard language (C, C++, Java, Fortran). These codes may be already existing: for example they may be existing programs, libraries, or components themselves.

Skeletons have associated a semantic model and a cost model, that make this approach very promising also for Grid programming: because of the existence of the cost model, the static and dynamic implementation of each skeleton is *parametric* with respect to few parameters. For example, the actual degree of parallelism or the actual number of data partitions can be varied dynamically without affecting the code of the run-time support.

ASSIST is based on the structured parallel programming approach [41]. Beyond the ìclassicalî skeletons, the ASSIST programming model contains several features (graphs, parallel modules, external objects) that sensibly increase flexibility and expressive power, including the possibility to design *adaptive program structures* (see the previous consideration about the need for alternative versions of the same computation). The ASSIST features will be presented in Sec. 3.

In other words, we have seen that parallelism (and structured parallelism in particular) is not only useful *per se* (i.e. to exploit higher performance of a certain code, possibly allocated onto the same Grid node), but also it has an utilization which is much more consistent with the dynamically adaptive nature of Grid-aware applications: in fact, in our model structured parallel programming is *a way to specify the strategy for structuring and for restructuring a component or a composition of components.* Components are internally expressed in ASSIST, with the addition of proper scripting *annotations* for specifying the ìperformance contractî of the component (e.g. performance metrics, critical events, and so on). Notice that, in general, processes of the same component could be rescheduled onto *different* Grid nodes. These issues will be discussed in Sec. 6.

2.4. Interrelationships of programming model features

As shown in Fig. 2, *features 1, 2 and 3, that we advocate for the definition of a Grid programming model, are strongly interrelated.* Feature 1, that implies interoperability, is fundamental for being able to structure complex application that include existing and /or predefined software components, and their ìglueî is made possible and easy by the structured parallel programming approach. This feature is also fundamental to allow ìlegacy codeî usage in Grid programs.

Feature 3 requires that components of an application can be rescheduled and restructured dynamically: in turn, *this requires feature 2 (uniform approach to distributed and parallel programming)* because processes of the same parallel component could be restructured and reallocated onto different and distinct nodes, even in the case that at launch time this component has been allocated onto the same node in a sequential or differently parallelized fashion. The parametric feature of structured parallel programming makes the realization of a performance model for the dynamic restructuring of applications feasible.

Summing up:

- a Grid-aware application can be designed as a parallel program, properly ìwrappedî into a components structure (together with some possibly pre-existing components), without distinguishing between inter- or intra-node parallelism at the implementation level. Provided that an initial allocation of the components is done at launch time, the allocation of parts of the same components can be modified at run-time (both in identities of nodes and in amount of nodes) to deal with the dynamic adaptation strategies expressed in the same parallel formalism.

Finally, we observe that dealing with the complexity of the Grid programming model has beneficial effects on the same parallel programming principles *per se*. In fact, the possibility to express dynamically adaptive computations also contributes to the solution of *irregular and dynamic problems in parallel programming*, i.e. computations that cannot efficiently be expressed according to predefined paradigms and/or that need substantial modifications according to some data values known at run-time (e.g. parallel Barnes-Hut algorithm), including some *interactive* applications. ASSIST aims to be a solution to these problems too, since it goes beyond the typical limitations of ìclassicalî skeletons in dealing with irregularity, dynamicity and interactivity.

3. The ASSIST programming model

Structured parallel programming, based on the skeletons model, is an approach to deal with the complexity in the design of high-performance applications. The validity of this approach has been proved, in the last years, for homogeneous parallel machines (MPP, Clusters, Clusters of SMP). As discussed in the previous section, the basic features of structured parallel programming are much more valid for complex heterogeneous, dynamic platforms, like Grids.

The validity of the skeletons model can be summarized by the following elements:

a) parallel programs are written, by means of a *coordination language*, as compositions of predefined parallel paradigms, called *skeletons*. In the past experiences of our group (parallel languages P^{3}L and SkIE [6], Lithium [4]), they are the stream parallel skeletons pipeline, farm (also known as master-worker or parameter sweeping) and data-flow loop, and the data-parallel skeletons map (independent virtual processors), reduce and comp (sequential composition of map-reduce combinations). In other skeletal language, similar constructs are adopted. The ìhostî languages, i.e. the languages with which the sequential parts are written, are all the standard ones (C, C++; Java, Fortran) with their compilers;

b) owing to its clear semantics, for each skeleton it is known a set of *implementation templates* and a *cost model*, both parametric, that can be used to optimize the compilation and the run-time support taking into account the composition characteristics. The optimization of the composition is largely dependent upon the knowledge of the possible compositions, and also on some equivalence properties of the composition themselves;

c) from points *a)* e *b)* it follows that, often, writing structured parallel programs is rather easy and fast, since the most complex decisions related with the parallel implementation are delegated to the compiling and run-time tools.

The intensive experiences with skeletal languages have shown that, in many parallel programs, the performance measures are satisfactory, and comparable with those of the same programs expressed in lower level formalism, such as MPI, or in other parallel languages when possible (HPF for data-parallel computations only). For example this result is achieved by the parallel program depicted graphically in Fig. 3, that expresses a parallel version of the Data Mining A-priori algorithm [14] as a pipeline composition of six stages, two of which are farms and the other four are sequential modules (mainly performing file transfer).

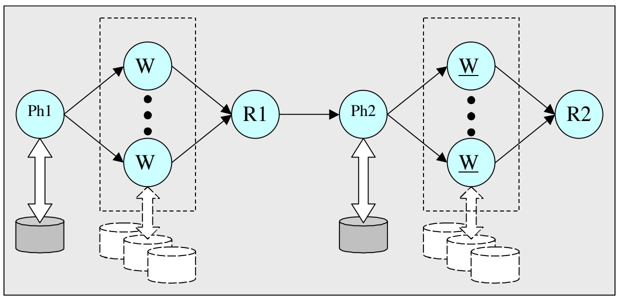


Fig. 3 - Skeleton structure of partitioned Apriori algorithm

The partitioned parallel Apriori requires two scans of the input to compute the set of frequent sets. The two phases have the same skeleton representation. Sequential module Ph1 reads in the input and sends data partitions to the workers W. Module R1 is a sequential reduction (union of the result sets). The same schema is repeated with different sequential modules (Ph2, $\underline{W}$, R2) to produce the result. The application scales well with database size; however, modules Ph1 and Ph2 are I/O bound, and the efficiency of the structure can be improved by forms of shared data access from within the workers.

Despite many advantages, the classical skeleton approach has several drawbacks in terms of expressive power and efficiency for complex applications, flexibility in adapting to a variety of combinations of parallel paradigms, as well as in terms of interoperability and suitability to component-based programming for Grid applications.

ASSIST is an evolution of the classical skeleton model aiming to act as a research vehicle in this complex scenario. In the following, we summarize the main features of the ASSIST programming model (ASSIST-CL: ASSIST Coordination Language) through their relationship with significant problems to be solved. For an extensive presentation of the ASSIST-CL syntax, semantics and implementation, the reader is referred to [41, 1, 2, 3].

3.1. High-performance programs as graphs

In several applications, the computation structures that can be expressed by the classical skeletons are not adequate. Notably:

a) many applications can be conceived in terms of the composition of independently developed components, without a predefined structure;

b) the composition structure may follow a data-flow or, alternatively, a nondeterministic model (event driven computations) in which the components, in general, can have an internal state. Many Work Flow models and their extensions have these features;

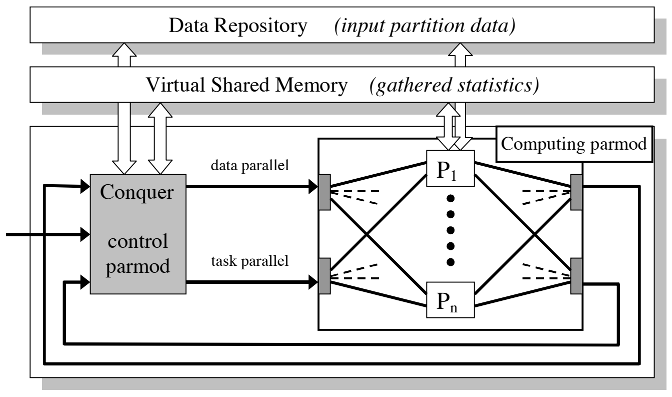
c) components can interact by means of several communication patterns, notably *streams* (sequences of typed objects), events, or single invocations, possibly in a RPC fashion;

d) many parallel algorithms can be expressed in a Divide&Conquer (D&C) style. Though a D&C specific skeleton can be defined [13, 4], its performance may be low in general, because of the large variety of configurations and situations in which this paradigms occurs. The solution, adopted in SkIE, making use of parallel compositions (e.g. pipelines, farms and map-reduce) inside a data-flow loop, has a similar drawback because a general implementation of the data-flow loop cannot be optimized for all the situations. Many irregular and dynamic parallel algorithms have similar features.

In ASSIST a parallel program can be structured as a *generic graph*, whose nodes correspond to parallel or sequential modules, and arcs correspond to communication channels over which *streams* of typed values are transmitted. Thus, in our case streams are the general *composition mechanism* for components. Other communication patterns (single invocation, RPC, and so on) can easily be expressed as particular cases of streams or implemented in a stream-based formalism. A computation, applied to the values of an input stream, terminates when the stream ends. Modules can have a *state* that is initialized at the beginning of the computation, and that is significant until the computation terminates. Each module can have any number of different input and output streams. In general, input streams are selected *nondeterministically* in a CSP-like fashion. Alternatively, a node may have a *data-flow* behaviour driven by all the input streams, or, in the most general case, a mixed nondeterministic + data-flow behaviour.

The choice of graphs and streams in ASSIST is not only more general, but it also eliminates the performance penalties mentioned in point *d)*. In fact, streams are explicitly controlled by program, and it is much more feasible to achieve an efficient implementation of simple communication channels and guarded commands, with respect to the complex communication patterns which are needed by D&C and irregular/dynamic computations.

Fig. 4 shows the structure of an ASSIST program for a parallel implementation of the Data Mining C4.5 algorithm according to a D&C solution. Conquer and Computing (the Divide functionality, i.e. the core of the computation) are ASSIST *parmod*s (see Sec. 3.2); the streams convey (references to) training set and decision tree objects. The description will be detailed in Sec. 3.5. It can be shown [14] that this parallel program has a performance comparable to the solution written directly in MPI or other lower-level libraries; this proves the validity of the implementation of streams and parallel modules also in computations structured as complex cyclic graphs.



Data Repository (input partition data)
Virtual Shared Memory (gathered statistics)
Computing parmod
data parallel
task parallel
Conquer
control parmod
P_1
P_n

Fig. 4 - Structure of a classifier (sequential code based on C4.5) made up of two interconnected parmods exploiting external objects

In the *component version* of ASSIST, single nodes of the program graph, or entire *subgraphs*, correspond to components, whose interfaces are expressed in terms of streams activating internal operations of the modules. For example, ìprovidesî and ìusesî ports, as well as events or other kind of interfacing mechanism [12, 15, 16, 28, 39, 43], can be mapped onto the ASSIST composition mechanisms. This issue is currently investigated and the precise definition of the ports-streams mapping will depend on the specific component model that will be adopted. However, structuring ASSIST programs in terms of graphs and streams (and in terms of external objects, see Sec. 3.3) is sufficient to capture the essential semantics of high-performance components, in a way that is largely independent of the specific component technology.

3.2. A generic skeleton: parmod

Every classical skeleton correspond to a *specialized* paradigm to express parallelism. Though in many algorithms they are exactly what the programmer needs, there are some applications which need much more flexible structures for expressive power and/or efficiency reasons. Notable examples are:

a) stream-parallel farms with an *ad-hoc* task scheduling strategy (master), workers having some form of internal state or additional communications, or specific task ordering strategy;

b) datañparallel algorithms with various kind of *stencils* that can be statically recognized (fixed or variable stencils), or dynamic stencils;

c) specific *data distribution strategies*, ranging from on-demand scheduling to scatter, multicast, broadcast;

d) *no limitation in the number of input/output streams*, controlled according to a *nondeterministic* or *data-flow* style, or both;

e) existence of *internal state*, considered significant for the duration of the computation, which is usually prohibited by the classical completely functional semantics of data-flow skeletons,

f) proper combinations of stream- and data-parallel paradigms *in the same module* (e.g. in systolic computations),

g) *modules that are able to behave according to different paradigms in different phases of run-time behaviour*. E.g. the Divide module in the C4.5 implementation of Fig. 4 may have a data-parallel behaviour or a farm-like behaviour in order to optimize load balancing during phases that, because of the multiplicity of data, have significantly different requirements.

All these situations can be emulated by means of specialized skeletons, but at the expense of code complexity and/or efficiency. For example, often the solution could consist in combining specific skeletons in parallel data-flow loop structures; however, besides the loop implementation, serious overhead sources are represented by the operations providing data transformation and redistribution necessary for the correct combination of skeletons (e.g. a pipeline combination of farm and data-parallel skeleton).

The ASSIST solution is a *generic* skeleton, that is a general-purpose construct that can be tailored, for each application instance, in order to deal with all the features needed to solve problems *a ñ g).* This construct, called *parallel module* or *parmod*, must possess the feature that, when it describes the computations equivalent to specialized skeletons, it does not incur in sensible overhead due to its generality.

A parallel module is defined as follows (see Fig. 5):

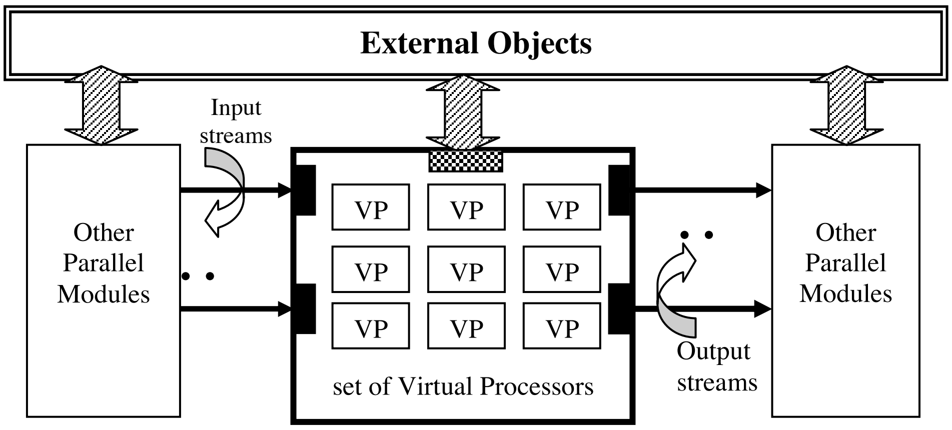


Fig. 5 ñ Elements of a parallel module (parmod)

- a set of *virtual processors* with internal state. The set has declared a certain *topology* to provide names to virtual processors, when needed, e.g. an *array* topology, or a *none* topology (no name: in this case all the virtual processors are anonymous);
- a set of typed *input streams* controlled in a data-flow and/or nondeterministic fashion. Once selected, the data of an input stream may be *pre-processed*, and are sent to the virtual processors according to the scatter, multicast, broadcast, or on-demand strategies. The chosen *distribution strategy* is expressed by program through a proper declaration;
- a set of *output streams* controlled independently by means of collective operations on the state value of the virtual processors. *Post-processing* on the data to be sent to other modules can be performed collectively.

It is easy to see that all the known specialized stream- and data-parallel skeletons can be expressed directly in ASSIST. More important, *no performance penalty* is paid with respect to the equivalent specialized skeletons. Several kernels have been tested in ASSIST to verify the comparison of parmod performance with respect to the specialized skeletons. The result [1, 2] is that the parmod performance is comparable to, or better than, the performance of specialized skeletons or programs written in data-parallel languages. For example, the ASSIST program corresponding to the A-priori computation of Fig. 3 is expressed by a graph, where the two farm nodes are expressed by parmods with none topology and on-demand distribution, while the other nodes are sequential modules (e.g. parmods with only *one* virtual processor, but with the all the parmod features about the control of input / output streams). The speed-up of the ASSIST program [1, 2, 14] is equal or better than the corresponding SkIE program (using specialized skeletons) and comparable to the MPI version.

Analogous considerations apply to the evaluation of data-parallel skeletons. In this case, the comparison is done, with respect to data-parallel languages (HPF), for fixed and variable stencils. Among all, the cases, in which ASSIST achieves higher performance, are the computations where we need a sophisticated combination of various parallel paradigms in the same module. Notable examples are the adaptive version of C4.5 and systolic algorithms. Other significant cases derive from the combined utilization of parmods and external objects (see next section).

An important observation concerns the *cost model*. In the specialized skeletons approach, analytical cost models are derived for some skeletons, especially stream parallel or simple data parallel ones. What is the position of ASSIST from this point of view? First of all, the implementation of a parmod can be optimized in the same way that it happens for simple skeletons: *the capability to optimize the run-time support is the true strength of structured parallel programming*. From this point of view, ASSIST is not different from the approaches based on specific skeletons.

Moreover, especially in complex applications and Grid platforms, we have to consider the issue of cost model in a much more general way, that is according to *the capability to express a cost of a parallel component or any combinations of parallel components in a context that is strongly variable and dynamic*. Dynamically adaptive Grid applications must have associated a *performance contract*, specifying information to evaluate the application performance and, possibly, to drive the application restructuring (see Sec. 2.3.and Sec. 6). In this context, we have to know a *library of cost models* for various structures, both elementary and composed; such library could be enriched according to the past *history* of the system or application field. For ASSIST, this library is known, at least in the same way that is was known with specific skeletons (e.g. in SkIE): we know the cost model of a parmod when it is equivalent to a farm or to a map/reduce, or similar simple skeletons; in addition, we know or can derive the cost models of a rich variety of data-parallel structures with fixed and variable stencils, as well as of some notable combinations of stream- and data-parallel (e.g. systolic), or with graphs with specific structures. All such information will be inserted in the *annotation* of the components of a Grid-aware application (Sec. 2.3), and exploited by the Grid Abstract Machine at launch time and at run time. In conclusion, ASSIST has (at least) all the features of a structured parallel programming model from the point of view of the cost model and its utilization.

3.3. External objects

Another feature to increase both flexibility and efficiency of ASSIST programs are the so called 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 objects. 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 Virtual Processors), an external object can be referred by a parmod in any phase (input, virtual processors processing, output).

In general, the goals of including / supporting external objects are the following:

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

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

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

iv) to overcome the limitations of the single node resources (notably, memory capacity) in distributed architectures,

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

Three kinds of external objects are distinguished:

a) 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.

b) Integration of Distributed Shared Memory libraries: in many problems, the goals *iii-v)* mentioned above can be met by means of external objects expressed by abstractions of shared memory. In particular, we integrated libraries for Distributed Shared Memory (DSM) and for 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.

c) Remote objects: they enable an ASSIST program to interact with external services through their specific interfaces (e.g. parallel file systems, databases, GUIs, other - possibly parallel - programs or libraries), using an object oriented formalism, provided for example by a distributed object middleware like

CORBA. The interoperability with CORBA has been introduced in Sec. 2.1 and will be described in Sec. 5.

The use of shared variables and shared objects is a further mechanism to increase the expressive power and efficiency of ASSIST programs. The preliminary experiences with the addition of shared objects to specialized skeletons [14] have proved the feasibility and effectiveness of this mechanism. A first example has been shown about the C4.5 algorithm (Fig. 4), where the heaviest data structures are shared by the virtual processors of the Divide parmod, while the streams convey references to such data structures.

Another significant example is shown in Fig. 6: it is an ASSIST version of the Barnes-Hut N-Body algorithm, using a proper combination of parmods and shared objects (vector of bodies and other control structures).

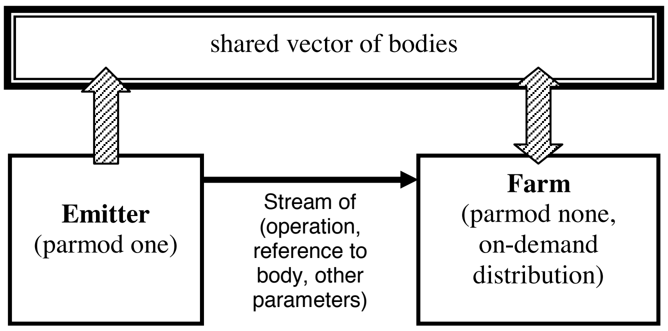

shared vector of bodies
Emitter
(parmod one)
Stream of (operation, reference to body, other parameters)
Farm
(parmod none, on-demand distribution)

Fig. 6 ñ ASSIST version of Barnes-Hut algorithm using proper combinations of parmods and shared objects

Owing to the utilization of shared objects, the modules are combined in a non-cyclic structure, exploiting both nondeterminism and data-flow behaviour. Streams convey just reference to the shared objects, that are created and scheduled by the Emitter module and processed by the Farm module.

While the performance of this algorithm expressed by specialized skeletons is rather low, the performance of this ASSIST version is quite satisfactory: the efficiency lies between 70 and 80% with 8-16 processors for rather small data sets (100K, 1000 K bodies). The achieved result is comparable with a dedicated MPI version. In particular, the overhead introduced by the ASSIST support in communication, nondeterminism and sharing is negligible with respect to the corresponding primitive underlying mechanisms. This and other benchmarks prove that ASSIST is able to express efficiently many irregular and dynamic/adaptable computations, though by adopting a high-level structured approach to parallel programming.

3.4. ASSIST applications

Till now we have shown the ASSIST performance in benchmarks represented by single computational kernels, belonging to a rather large collection of regular and irregular benchmarks and tests developed during the project. However, ASSIST has been also used successfully in the design of complex applications, notably:

- Image Processing: isosurfaces extraction, decimation, compression (IMATI-CNR, Genova),
- Earth Observation: SAR, Interferometry (Italian Space Agency),
- Computational Chemistry: molecular dynamics (ISTM-CNR, Perugia),
- MPEG2 parallel encoder (our Department),
- Knowledge Discovery in semistructured data sets (parallel query execution engine), Question Answering, Search Engines (our Department),
- a Data Mining framework for Customer Relationship Management platforms: this complex application, developed by our group in the SAIB Project (MIUR and Schlumberger SEMA SpA), will be illustrated in the next section.

These applications allow us to test the validity of the ASSIST approach, which has to be *evaluated* not only in terms of choice of the coordination language constructs, but *also in terms of a complex, interrelated set of outstanding requirements concerning modularity, software reuse, interoperability with many standard tools, feasibility to define efficient external objects, as well as in terms of the global efficiency of the implementation*. The next section is an illustration of these ASSIST features.

3.5. Example of complex application in ASSIST: a User Modeler Server for Customer Relationship Management platforms

SAIB (System for Internet Banking Applications) is a large research project which brings together some Italian academic institutions and industrial partners in the effort of producing an open-source based *Customer Relationship Manage*ment (CRM) solution for financial applications. We describe the *User Modeler Server* (UMS), a parallel Knowledge Discovery in Databases (KDD) system designed to be integrated within the SAIB system. Among the requirements there are flexibility and programmability, as well as high-performance customer profiling functionalities.

Our KDD system provides to the marketing analysts the Data Mining (DM) services needed to build customer profiles from a customer database. Knowledge models can then be deployed to the CRM main core, where they can be used to customize user interaction at different levels. The UMS performs both *batch* operations (heavyweight Data Mining algorithms, out of the main business flow) and *on-line* ones (per-user knowledge queries), which are subject to near-real-time constraints.

In developing a parallel KDD system we are faced with different issues:

- to deal with databases of several Gigabytes in size;
- to develop efficient parallel mining algorithms, managing both sustained and irregular demands of I/O and computation, scalable with available computing resources to higher performance and throughput;
- to ensure a high internal degree of integration of the system, as well as simplicity and performance of the interaction between the parallel mining algorithms, data management and KDD support modules for knowledge management and visualization
- to ensure system interoperability and ease of cooperation with different software technologies. Industry standard languages and technologies like XML, Java/EJB and ODBC have to be exploited to integrate advanced parallel modules within larger applications.

In the architecture of the UMS, these goals were accomplished with a portable, high-level software design thanks to the programming approach of ASSIST, to the features of its coordination language and of its supporting environment. The architecture of the UMS, shown in Fig. 7, is made up of a few basic elements, namely: (1) Activity Scheduler, i.e. control and visualization interface (**UI**, the User Interface), (2) a set of mining algorithms (**MA**), (3) data management module (**DR**, the *Data Reposito*ry) and (4) knowledge and meta-data management module (**KR**, the *Knowledge Repository*).

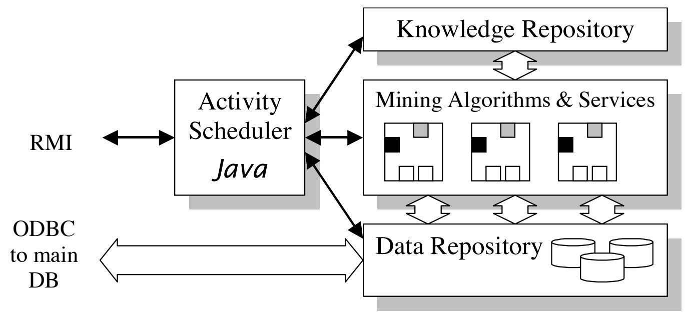


Fig. 7 - Overall architecture of the UMS module

The **UI** in our case is a gateway to the rest of the SAIB system, with no visualization and low level user interaction tasks. Instead, it exports to the CRM system a Java RMI interface for all of the supported KDD operations, and schedules them to the parallel mining engine. The UI is not performance critical, thus to speed up the implementation process we exploited the interoperability of ASSIST with Java, implementing the UI directly with this language. The UI cooperates with the rest of the UMS by means of Java native methods, file system I/O and calls to the ASSIST program loader.

The set of **MAs** contains ASSIST parallel programs, some of them described later on, that perform actual mining tasks, and simpler ones used to manage the DR data (e.g. selection and sorting). We reused parallel mining applications designed in our previous research [14], which were based on skeleton structures and data-flow stream cooperation, and we extended them with new functionalities. Taking advantage of the modular structure of ASSIST programs, we have evolved them so that they (1) interface to the DR module for most of the I/O, and (2) benefit from more sophisticated features like external objects, or control of nondeterminism.

The **DR** module is implemented as an *external object* of ASSIST. It provides high performance I/O support for large files with a simple record structure of typed fields (the kind of regular data tables we have to host in a mining warehouse), and a mechanism of block-oriented views to allow large grain, parallel operation on the same file. The DR is built on top of the PVFS parallel file system, and it is integrated within the ASSIST programs as an external object. Each code module can thus access the data, with the implementation layers of the DR coordinating their efforts, and the PVFS implementation layer moves data blocks in and out of each process memory, wrapping the actual file system layer. This design exploits parallel file system performance and bandwidth from within portable ASSIST programs.

The **KR** module manages the knowledge extracted by the mining algorithms, represented using the standard PMML 2.0 language. The KR is designed to allow storing, retrieving, refining and tracking the history of knowledge models produced by the mining algorithms. To manage the different states and attributes of models, that condition their use inside the SAIB system, we adopted a composite structure. A standard CVS server is used to store the XML textual data, access being mediated by a custom server process which performs additional controls and attribute caching, and is used from within ASSIST code. A client-side linked library provides XML parsing and serialization on models, as well as interface to the servers.

In composing the UMS modules we have exploited the features of interoperability of ASSIST, mixing Java, C and C++ code, stream and RMI cooperation. We describe some of the parallel modules performing the Data Mining functions. High performance and dynamic adaptiveness of these parallel applications rely on the parallel structure that is described at high level, supported by the ASSIST run-time.

Classification is performed with a decision tree induction algorithm, shown in Fig. 4. After a first, data parallel phase of the execution, we should switch to a task-parallel behavior, and below a certain node size to sequential computation. We have evolved the structure reported in [14] following the idea outlined before and in [41]. The same *parmod* implements all the different functionalities related to the Divide part of the algorithm. Another *parmod* with *one* topology acts as a controller and performs the Conquer steps. Dynamic load balancing in the task-parallel case is granted by the *on-demand* distribution. Stream guards controlled by shared variables make it clear and manageable the transition among the different behaviors. Access to the data is supported by the external object paradigm, both in distributed shared memory (node statistics are kept there) and in the DR module (which provides the input data).

Association Rules are computed by an Apriori-like algorithm. Its parallelization [14] is based on the partitioning method, which requires two phases of anonymous workers with load-balancing. Its structure has been shown in Fig. 3, expressed with traditional skeletons: actually, the farm and sequential skeletons are replaced by corresponding parmods. On the other hand, by exploiting the DR External Object, the input dataset is accessible to all the processes, removing the file transfer bottleneck, with relatively minor changes to the code.

The *Clustering* application is built around the association rules module. Following [32], we group together records that satisfy the most popular association rule of a dataset. Rules are mined again and again on unclustered records until a given threshold is reached. The parallel implementation fully reuses the association rule ASSIST modules. The program graph becomes cyclic, a controller module is in charge of recycling data (also applying a rearranging strategy to enhance locality and reduce the amount of I/O), nevertheless the change can be kept hidden to the contained modules.

4. ASSIST implementation

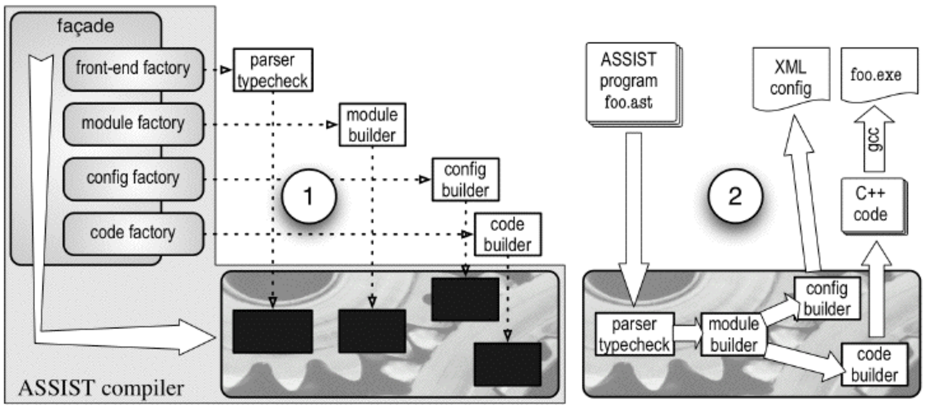
4.1. Modular design of ASSIST compiler

The ASSIST environment has been designed both to produce fast object code and to be modifiable on the need. It has been designed and implemented exploiting well-known software engineering techniques that have enabled the easy extension of programming environment features along several ASSIST versions.

Overall, the ASSIST compiler (namely `astcc`) produces code for plain POSIX/TCP workstation networks/clusters. We briefly outline the ASSIST implementation structure. The whole ASSIST-CL compiling tools have a three tier design:

- *front-end* (the top tier) parses ASSIST-CL syntax and produces an internal representation of the program;
- *core* (the middle tier) that is the compiler core. It translates the internal representation of a program into the *task code*. The task code represents a sort of C++ template-based, high-level and parallel assembly language. The step transforming internal representation into task code is completely implemented exploiting design pattern technology [27]. A faÁade pattern decouples compiler internals from the compiler engine; a builder is being used to actually produce the task code;
- *back-end* (the bottom tier) compiles task code down to the ASSIST abstract machine (CLAM, the *Coordination Language Abstract Machine*) object code. The CLAM is built on top of POSIX processes/threads and communication (SysV and TCP/IP sockets) primitives. All those primitives are used via the ACE (*Adaptive Communication Environment*) library [37].

The whole compilation process is illustrated in Fig. 8.



1) ASSIST builders are generated through proper factories and arranged in a work-flow. Compiler features may be easily extended adding new builders (e.g. a F90 code builder producing Fortan code). **2)** The ASSIST program is processed by all builders producing a XML configuration file and a number of source and object files. Eventually they are compiled through suitable standard compilers.

Fig. 8 - ASSIST compiler at work.

The result of compilation consists of two distinct items:

comp1) a set of *object code/DLLs* including the code needed to execute all application modules and run-time support. Each code may be produced in different version, each of them targeting a different platform;

comp2) a *configuration file* including module and run-time ìcoordinationî information. Coordination information are represented in XML format; they include two main classes of information: *static* and *dynamic*. Static information include parallel activities description, the description of (initial) modules graph, external libraries description and their bindings to modules. Dynamic information includes machine names, parallel degree of parallel modules, and modules-machines mapping.

A CLAM master process scans the XML configuration file produced by `astcc` compiler and arranges things in such a way that the CLAM slave processes, running on the target architecture processing nodes, load and execute the suitable code (either coming from ASSIST-CL source code or belonging to external libraries properly named by the programmer in the ASSIST-CL source code). A detailed description of the ASSIST implementation can be found in [1, 3]. As CLAM (and the object code itself) access POSIX features via ACE wrappers, and as ACE is available on different operating systems (such as Linux, MacOSX and Windows), CLAM actually behaves as the fourth tier of the compile/run process and guarantees a high degree of portability of the whole programming environment.

The compiler design allows efficient code to be generated, as each tier may take the most appropriate and efficient choices related to object code production. Furthermore, the heavy usage of well known software engineering techniques, such as the *design* *patterns*, insulate all the individual parts of the compiler in such a way that *modification*s in one compiler part neither affect the whole compilation process (but for the new features introduced/modified) nor require changes in other compiler parts. Eventually, ASSIST-CL compiled code is run by means of a dedicated loader (the `assistrun` command), that in turn activates CLAM run-time support.

A first example of modifiability of the ASSIST support has been represented by the integration of *mathematical MPI librarie*s belonging to the ScaLAPACK, FFTW and PAMIHR packages, directly in ASSIST, i.e. they are available to the programmer as ASSIST constructs [19]. The whole library code has then been wrapped in such a way that it looks like a normal parmod code to the programmer.

4.2. Supporting heterogeneity

The first design of ASSIST was targeted to homogeneous clusters of Linux machines. The items needed to produce code for heterogeneous architectures were basically two: the inclusion of some kind of XDR (*external data representati*on) wrapping messages flowing between heterogeneous processing elements, and the generation of proper *makefiles* to compile final object code. Both these problems have been solved exploiting the ASSIST-CL compiling tools structure. The `astcc` compiler uses a builder pattern both to generate the actual task code and to generate the makefiles needed to compile task code to object code. Thanks to this modularity, the only interventions needed to support Operating System and Hardware heterogeneity was to modify these builders in the following way:

- on the one side, communication routines are produced that either process memory communication buffers with XDR routines during marshaling and unmarshaling, or do not process them with XDR. The former routines will be used in case processing elements using different data representations (e.g. little/big endian machines) are involved in the communication. The latter routines instead will be used in those cases when homogeneous processing elements are involved in the communications. Proper makefiles are generated consequently;
- on the other side, the XML configuration file is arranged in such a way that XDR communication libraries are used when different architectures are involved and non-XDR routines are used in all the other cases.

Currently, heterogeneous clusters of Intel Linux and MacOSX platforms are supported. However, the technique we rely on is fully general: in order to add yet another architecture, it is sufficient to add the builder that produces code suitable for the new architecture. In the case this architecture includes a POSIX API, the proper builder may be produced with minor revisions of existing ones.

From the performance viewpoint, we experienced a fairly limited impact of marshaling routines on overall application code. Current benchmarks experience a slowdown of normalized performances, ranging from 1% (short messages) to 15% (long messages), for heterogeneous architectures with respect to homogenous ones.

4.3. ASSISTconf and the Grid

As seen, the ASSIST compiler decouples data needed to run compiled applications in two main classes, denoted by *comp1* and *comp2* in Sec. 4.1. Exploiting this feature, ASSIST-CL programs can be run on a Grid performing the following three steps:

1. the XML configuration file (comp 2) is analyzed and the resources needed to execute the program are individuated;
2. the resources needed to execute the program are gathered, and reserved, using the normal Grid middleware tools (e.g. those provided by the Globus toolkit);
3. the XML file is modified in such a way that the resources gathered are used to run the ASSIST code.

In order to demonstrate the feasibility of the approach, the *ASSISTconf* tool [8] has been developed in collaboration with the group of ISTI-CNR. ASSISTconf supports such kind of manipulation of the original XML configuration file. Actually, the tool only supports decisions taken directly by the programmer via a GUI: starting from information gathered from the Grid, the tool proposes to the programmer a set of choices. Afterward, the tool produces a new XML configuration file describing the new mapping of program entities onto Grid resources.

ASSISTconf just represents an intermediate step toward full Grid support in ASSIST. In order to *automatically* and effectively targeting the Grid, many factors have to be taken into account, which are traditionally handled by expert programmers: resource co-allocation, code and data staging, task scheduling, and the alike.

The extension to the general automatized solution is allowed by the global approach to the Grid programming environment stressed in this document, and by the design methodology of the ASSIST definition and implementation modularity. The structure of the existing ASSIST programming environment is exploited as follows:

1) resource co-allocation is decided on the basis of the contents of the XML configuration file produced by the ASSIST-CL compiling tools. In particular, the compiler already devises the number and the kind of resources needed to execute the code, mostly exploiting user provided parameters. A CLAM version targeting the Grid can easily process the XML configuration file in such a way that the proper resources are selected;

2) code and data staging are also managed by the CLAM setup process. Also on clusters, the first phase in the execution of an ASSIST-CL program consists in deploying the proper object/library code to the interested processing nodes;

3) task scheduling is completely under the control of CLAM and follows the directives taken from the XML configuration file.

5. ASSIST-CORBA interoperability

ASSIST provides various opportunities for interoperability, owing to its definition and implementation. For example, in Sec. 3.5 we saw the possibility of effectively mixing Java, C and C++ code, stream and RMI cooperation in complex parallel applications.

ASSIST provides full interoperability with CORBA objects. In this way we have achieved a first concrete result that combines the benefits of software reuse and interoperability with those of high-level parallel programming. CORBA, a distributed object-based middleware, is a well established commercial standard, and offers interoperability with several sequential languages. Recent studies recognized that CORBA technology could be leveraged to support the development of advanced Grid applications [35, 36]: the CORBA CoG kit [34], for example, provides CORBA applications with access to the Grid services provided by the Globus Toolkit.

ASSIST can easily import (or use) other pieces of software encapsulated in external CORBA objects, as in a traditional sequential language (i.e. C++). Most important for the sake of this paper, ASSIST can export parallel algorithms and applications as well. For example, a computation intensive component of a large application can be conveniently expressed in ASSIST to exploit parallelism and achieve good performance; it can be automatically encapsulated in a CORBA object, so that it can interact (e.g. be invoked) with the rest of the application (coded in any language supported by CORBA) by means of standard CORBA invocation mechanisms. This enables the construction of complex, scalable applications in an easy way.

We devised two possible ways of interconnecting an ASSIST subprogram to a distributed application, to address two different classes of problems:

a) *RMI-like synchronous invocation* of a subprogram (with arguments and return values), when the task that should be carried in parallel is well defined and insulated;

b) *stream-like asynchronous data passing* (realized using the standard CORBA event channel mechanism), when the production of data can be overlapped with the elaboration, or when we want to deal with partial data or results (for example when rendering images or videos).

An ASSIST subprogram, in order to be exported to the CORBA world, must be a composition of ASSIST modules (at the extreme, it can be a single module) in which one input stream and one output stream are left unconnected and are elected to be the input and output of the entire exported component. In the RMI-like case, a further constraint to be satisfied (that cannot be statically checked) is that, for every datum received from the input source, the program must produce one and only one datum on the output stream.

The process of exporting an ASSIST program in the CORBA world has been *automatized*: the ASSIST program undergoes an initial phase in which it is analyzed to verify that the conditions stated earlier are met. It is transformed (ìwrappedî), according to the option RMI-synchronous (a) *vs* stream-asynchronous (b) interaction chosen by the programmer, adding support code to interact with the CORBA runtime and services:

case a) The program is exported as a CORBA object with a single method: a module that acts as a *bridge* towards CORBA is attached to the program input and output stream; it implements the method invocation by delivering every request to the program input stream and then waiting for a response from the output stream, that will be returned to the caller;

case b) The program interacts through CORBA event channels: two modules are added, one that intercepts ingoing events and sends them to the input stream of the program, and the other that receives messages from the output stream and forwards them as outgoing events.

As a final step, suitable CORBA IDL interfaces are generated. This process is simplified by the fact that, in the module interfaces, ASSIST recognizes the standard CORBA types, which are mapped to native types following the standard CORBA type mapping. The generated interfaces, then, can be processed by CORBA compliant IDL compilers, to produce suitable stub code to invoke the ASSIST CORBA object (RMI-like case), or to publish or subscribe the events that are recognized as inputs or produced as outputs by the ASSIST component (stream-like case).

Two basic experiments demonstrate that the methodology is a viable solution to parallel software integration:

i) we have compared the service time of a data-parallel algorithm expressed in ASSIST and exported as a CORBA object (invoked from an external C++ application) to an equivalent solution expressed in the ASSIST-CL language. The ASSIST program is a synthetic data-parallel algorithm, operating on large matrices (the argument and result are 700 x 700 floats) in order to estimate the overhead introduced by CORBA communication mechanisms. The algorithm performs a $O(n^3)$ computation involving transcendental function evaluation;

ii) to evaluate the event-based asynchronous interaction, we have compared a stream-parallel algorithm expressed in ASSIST and exported via CORBA with an ASSIST implementation that uses only ASSIST native streams. The ASSIST program is a classical Mandelbrot set computation as a task farm, in which each task is a set of contiguous points that must be evaluated.

In both cases, the overhead introduced by the program transformation is comparable in absolute value to the one in the sequential case; this means that the implementation doesnít introduce inefficiencies. Moreover, the overhead affects the service time for a small percentage, allowing a good speed-up up to 16 processors (< 5% degradation).

6. Dynamically adaptive applications

Let us now come back to the discussion of Sec. 2.3 and 2.4 about the possibility of using ASSIST not only for parallelizing single components, but also, more in general, for expressing the global dynamic adaptation strategies in Grid-aware applications.

The research project *Grid.it* will develop, on top of available standard services (GT), a Grid Abstract Machine (see Sec. 1) based on an *Application Manager* (AM) for parallel and distributed applications according to the Grid-awareness principles. AM, that logically is a centralized entity whose implementation may be decentralized, will exploit the functionalities made available by

- Performance Model
- Monitoring
- Resource Discovery
- Scheduling strategies, both local to single nodes and global to Grid
- Allocation strategies of codes and data.

The current implementation of ASSIST for heterogeneous networks and Grids, described in Sec. 4, will be extended in order to support the *dynamic* allocation of ASSIST programs [20, 21]: this affects the run-time support of parmod and of shared objects, so that parts of the same parallel components can be re-allocated dynamically to different nodes according to the decisions of AM.

As discussed in Sec. 2, subgraphs of ASSIST programs will be wrapped into standard components and, in general, made interoperable with other non-ASSIST components in order to build Grid applications. Moreover, each component will provide a scripting annotation about the ìperformance contractî to be established with the Grid Abstract Machine.

The following example could serve to clarify the ASSIST-based approach to the design of dynamically adaptive applications. The application consists of the component composition shown in Fig. 9.

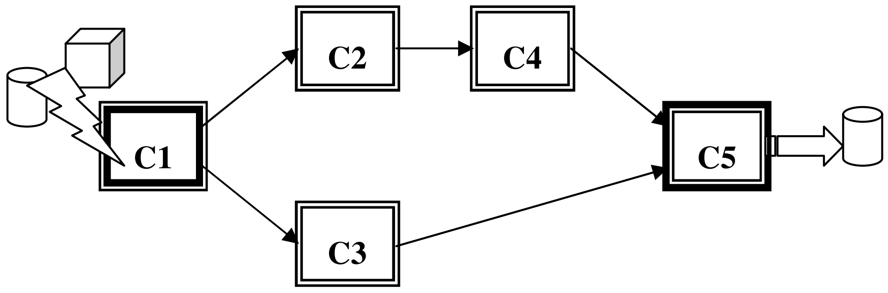


Fig. 9 ñ Example of an adaptive application expressed by parallel components

a) Component C1 is an interface towards a Grid memory hierarchy, that virtualizes and transforms data sets available on the Grid into two streams of objects, the one (whose elements have an elementary type) is sent to C2, and the other (whose elements have array type) is sent to C3. C1 may be an existing component available on the Grid, virtualized by an ASSIST program, for example by using external objects and/or the CORBA facility.

b) C2 is a component encapsulating an ASSIST program. The ìperformance contractî of C2 specifies that

- *by default* C2 is a sequential module (parmod with topology *one*) executing a certain function F;
- when the Monitoring and Performance Model services generate the event that signals the need or opportunity to adjusting the current performance level (*on restructuring*), C2 is transformed into a farm computation (parmod with topology none and data distribution on-demand) whose workers execute the same function F. AM of the Grid Abstract Machine determines the actual number of workers and their allocation to Grid resources: these may belong to the same Grid node (cluster) or to different Grid nodes. This is consistent with our conceptual framework (Sec. 2, Fig. 2), according to which the high-level version of the application is expressed by the structured parallel formalism with annotations, and all the allocation strategies are delegated to the Grid Abstract Machine.

c) C3 is a component encapsulating an ASSIST data-parallel program operating on each stream element of array type (parmod with topology array, proper distribution for the specific data parallel algorithm, and possibly a stencil expressed in the virtual processors section). Similarly to the approach described for C2, the ìperformance contractî of C3 specifies that, *by default*, the ASSIST program has to be executed on a single Grid node with cluster internal architecture, while *on restructuring* it can modify (increase) the parallelism degree (amount of real processors onto which the virtual processors are mapped). The re-allocation may exploit resources belonging to one Grid node or to distinct Grid nodes.

d) C4 is a component encapsulating an ASSIST program which, *by default*, is a sequential module, while *on restructuring* it is transformed into a parmod operating on the input stream according to a data-parallel or a farm style, depending on the values of the parmod state and on the input values themselves. In this case the adaptation principle is applied at two levels: at the program level and at the allocation level.

e) C5 is a component encapsulating an ASSIST program operating nondeterministically on the input values received from C3 or C4, and transforming the two streams into a data set. The ìperformance contractî of C5 specifies that C5 can be allocated and executed only on a certain Grid node and that no reconfiguration can occur. This may be due to security, or privacy, reasons, or to requirements related to the specific resource kinds needed to operate on the data set.

Let us assume that at a certain time C2 is becoming a bottleneck that causes a substantial degradation of performance of the whole application. AM provides to transform C2 into a version with the proper parallelism degree and to re-schedule and re-allocate this new version, assumed that, interacting with the Grid Resource

Management services, the necessary resources can be found. In case of restructuring of data-parallel components, the AM strategy must be applied also to the re-distribution of the data constituting the internal state of parmod.

7. Current and future work

We have shown how ASSIST can be used as a valid research framework in the definition and realization of innovative programming environments for high-performance Grid platforms. Our current research activity includes:

i) experiments on standard component frameworks, notably CCM and Web Services. Several benchmarks will be tested in order to verify the performance features especially when components are used to wrap ASSIST programs;

ii) realization of the full ASSIST implementation on top of the Globus toolkit, as discussed in Sec. 4.3.

The result of these short-time experiments will be exploited in the main tracks of our research:

a) *dynamic ASSIST*; i.e. definition and realization of ASSIST support to achieve the dynamic reallocation and restructuring of ASSIST programs on the Grid nodes. Recent experiences have proven the feasibility of this track [20, 21];

b) *performance model for Grid-aware applications*: performance models that, relying upon the cost model of ASSIST structured parallel programs, evaluate the performance of the Grid application. For this purpose, basic Middleware mechanisms, such as monitoring, will be exploited;

c) *Application Manager*: definition and realization of this fundamental component of the Grid Abstract Machine, according to the ASSIST-based approach introduced in this paper;

d) the results of activities *a), b), c)* will be exploited to design, according to the principles stated in Sec. 1 and 2, the first version of the *Grid.it programming environment*, whose release is expected by the end of 2004 / beginning of 2005.

Acknowledgements

We wish to thank the people of ASSIST group working on design, implementation and benchmarking of the environment tools: S. Campa, R. Ravazzolo, P. Pesciullesi, M. Torquati, G. Giaccherini, S. Magini, A. Paternesi, A. Petrocelli, E. Pistoletti, L. Potiti, P. Vitale. The research activities on *Grid.it* programming environment are performed in collaboration with the ISTI-CNR group: D. Laforenza, S. Orlando, R. Perego, R. Baraglia, T. Fagni, F. Panciatici, D. Puppin, M. SerranÚ, N. Tonellotto.

References

1. M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, S. Magini, P. Pesciullesi, L. Potiti, R. Ravazzolo, M. Torquati, M. Vanneschi, C. Zoccolo. The Implementation of ASSIST, an Environment for Parallel and Distributed Programming. In *Proc. of Euro-Par2003: Parallel and Distributed Computing*, *LNCS* n. 2790, Springer, August 2003.
2. M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, M. Danelutto, P. Pesciullesi, R. Ravazzolo, M. Torquati, M. Vanneschi, C. Zoccolo. ASSIST demo: a high level, high performance, portable, structured parallel programming environment at work. In *Proc. of Euro-Par2003: Parallel and Distributed Computing*, *LNCS* n. 2790, Springer, August 2003.
3. M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, M. Danelutto, P. Pesciullesi, R. Ravazzolo, M. Torquati, M. Vanneschi, C. Zoccolo. A framework for experimenting with structured parallel programming environment design. In *Proc. of Intl. Conference ParCo2003: Parallel Computing*, Dresden, Germany, September 2003.
4. M. Aldinucci, M. Danelutto, P. Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611-626, Elsevier Science, 2003.
5. 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
6. B. Bacci, M. Danelutto, S. Pelagatti, M. Vanneschi. SkIE: A heterogeneous environment for HPC applications. *Parallel Computing*, 25:1827-1852, Dec. 1999, 1827-1852.
7. H E. Bal. Ibis: A Java-based grid programming environment. In Proc. of *Euro-Par2003* (invited talk), *LNCS* n. 2790, 2003.
8. R. Baraglia, M. Danelutto, D. Laforenza, S. Orlando, P. Palmerini, P. Pesciullesi, R. Perego, and M. Vanneschi. AssistConf: a Grid configuration tool for the ASSIST parallel programming environment. In *Proc. of 11-th Euromicro Conference on Parallel Distributed and Network based Processing (Euro-PDPí03)*, pages 193ñ200, Genova, Italy, February 2003.
9. G. Beneken, U. Hammerschall, M. Broy, M.V. Cengarle, J. J¸rjens, B. Rumpe, and M. Schoenmakers. Componentware - State of the Art 2003. Background paper for the *Understanding Components* Workshop, Venice, 7-9 October 2003.
10. F. Berman, R. Wolski, H. Casanova, et al. Adaptive Computing on the Grid using AppLeS. *IEEE Trans. On Parallel and Distributed Systems*, 14 (5), 2003.
11. E. Bruneton, T. Coupaye, J.B. Stefani. *The Fractal Composition Framework.* The Object Web Consortium, Interface Specification, July 2002.
12. *CCA Forum home page*. http://www.cca-forum.org/
13. M. Cole, *Algorithmic skeletons: structured management of parallel computation.* MIT Press, Cambridge, Mass., 1989.
14. M.Coppola, M. Vanneschi. High Performance Data Mining with Skeleton-based Structured Programming. *Parallel Computing*, 28 (2002), 793-813, special issue on Parallel Data Intensive Algorithms, Elsevier Science.

15. *The CORBA & CCM home page*. http://ditec.um.es/~dsevilla/ccm/
16. F. Burbera et al. Unraveling the Web Services Web: an introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing,* vol. 6, n. 2, 2002, 86-93.
17. H. Dail, F. Berman, H. Casanova. A modular scheduling approach for grid application development environments. *Journal of Parallel and Distributed Computing*, 63 (5), 2003.
18. H. Dail, O. Sievart, F. Berman, H. Casanova, A. Yarkahn, S. Vadhiyar, J. Dongarra, C. Liu, L. Yang, D. Angulo, I. Foster. Scheduling in the Grid Application Development Software Project. In *Grid Resource Management*, Kluwer, 2003.
19. P. DíAmbra, M. Danelutto, D. Di Serafino, M. Lapegna. Integrating MPI-based numerical software into an advanced parallel computing environment. *Euromicro Conference on Parallel, Distributed and Network-based Processing*, Genova, IEEE Comp. Soc. (2003).
20. M. Danelutto. Efficient Support for Skeletons on Workstation Clusters. *Parallel Processing Letters*, vol. 11, n.1 (2001), 41-56.
21. M. Danelutto. Adaptive Task Farm Implementation Strategies. *Euromicro Conference on Parallel, Distributed and Network-based Processing*, La Coruna, IEEE Comp. Soc. (2004), 416-423.
22. B. Ensink, J. Stanley, V. Adve. Program Control Language: a programming language for adaptive distributed applications. *Journal of Parallel and Distributed Computing*, in press.
23. I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, San Francisco, CA, 1999. Chap. 11.
24. I. Foster, C. Kesselman, S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organization. *The International Journal of High Performance Computing Applications*, 15(3):200ñ222, Fall 2001.
25. I. Foster, C. Kesselman, J. Nick, S. Tuecke. *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*, 2002. http://www.globus.org/research/papers/ogsa.pdf
26. I. Foster et al. Grid Services for Distributed System Integration. *Computer*, vol. 35, n. 6, 2002, 37-46.
27. E. Gamma, R. Helm, R. Johnson, and J. Vissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.
28. M. Govindaraju, S. Krishnan, A. Slominski K. Chiu, D. Gannon, and R. Bramley. *XCAT 2.0: A component-based programming model for Grid web services*. Technical Report TR562, Department of Computer Science, Indiana University, 2002.
29. Grid.it Project: *Enabling Platforms for High-Performance Computational Grid Oriented to Scalable Virtual Organizations*. MIUR, FIRB National Research Programme, November 2002.
30. K. Keahey, D. Gannon. PARDIS. a parallel approach to CORBA. *6th IEEE Int. Symp. on High Performance Distributed Computing,* 1997, 31-39.
31. K. Kennedy, M. Mazina, J. Mellor-Crummey, K. Cooper, L. Torczon, F. Berman, A. Chien, H. Dail, O. Sievert, D. Angulo, I. Foster, D. Gannon, L. Johnsson, C. Kesselman, R. Aydt, D. Reed, J. Dongarra, S. Vadhiyar, and R. Wolski. Toward a framework for preparing and executing adaptive Grid

programs. In *Proc. of NSF Next Generation Systems Program Workshop (IPDPS 2002)*, 2002.

32. W.A. Kosters, E. Marchiori, A.A.J. Oerlemans, Mining clusters with association rules. In D. Hand, J. Kok, M. Berthold, eds., *Advances in Intelligent Data Analysis*, *LNCS* 1642 (1999), 39-50.
33. Object Management Group. *The Common Object Request Broker: Architecture and Specification.* (2000) Minor revision 2.4.1, http://www.omg.org.
34. M. Parashar, G. Laszewski, S. Verma, J. Gawor, K. Keahey, N. Rehn. A CORBA Commodity Grid Kit. *Concurrency Practice and Experience*, special issue on GRID Computing Environments 14 (2002) 1057-1074
35. C. Prez, T. Priol, A. Ribes, PaCO++: a parallel object model for high performance distributed systems. In *Distributed Object and Component-based Software Systems*, Hawaii Int. Conf. On System Sciences, IEEE Comp. Soc. (2004).
36. T. Priol. Programming the Grid with Distributed Objects. Proc. of *Workshop on Performance Analysis and Distributed Computing* (PACD 2002), 2002
37. D. C. Schmidt. The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software. In *Proc of Sun User Group conferences*, San Jose, CA, USA, December, 1993 and San Francisco, CA, USA, June 1993.
38. J.M: Schopf. Structural Prediction Models for High-performance Distributed Applications. *Cluster Computing Conferenc*e, 1997.
39. Sun M.sys. *JavaBeans home page*. http://java.sun.com/products/javabeans.
40. D. Thain, T. Tannenbaum and M. Livny. Condor and the Grid. Fran Berman and Geoffrey Fox and Tony Hey editors, *Grid Computing: Making the Global Infrastructure a Reality*, 2002.
41. M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing* 28(12), 2002, 1709-1732.
42. F. Vraalsen, R. Aydt, C. Mendes, D. Reed. *Performance contracts: predicting and monitoring grid application behaviour*. TR, Comp. Sc. Dept, Univ. Illinois at Urbana-Champaign, 2001.
43. W3C.*Web Services home page*. http://www.w3.org/2002/ws/
44. R. Wolski, N. Spring, J. Hayes. The Network Weather Service: a distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 15 (5-6), 1999.