The Implementation of ASSIST, an Environment for Parallel and Distributed Programming

Marco Aldinucci[2], Sonia Campa[1], Pierpaolo Ciullo[1], Massimo Coppola[2], Silvia Magini[1], Paolo Pesciullesi[1], Laura Potiti[1], Roberto Ravazzolo[1], Massimo Torquati[1], Marco Vanneschi[1], and Corrado Zoccolo[1]

[1] University of Pisa, Dip. di Informatica – Via Buonarroti 2, 56127 Pisa, Italy
[2] Ist. di Scienza e Tecnologie dell'Informazione, CNR – Via Moruzzi 1, 56124 Pisa, Italy, coppola@di.unipi.it

Abstract. We describe the implementation of ASSIST, a programming environment for parallel and distributed programs. Its coordination language is based of the parallel skeleton model, extended with new features to enhance expressiveness, parallel software reuse, software component integration and interfacing to external resources. The compilation process and the structure of the run-time support of ASSIST are discussed with respect to the issues introduced by the new characteristics, presenting an analysis of the first test results.

1 Introduction

The development of parallel programs to be used in the industrial and commercial fields is still a difficult and costly task, which is worsened by the current trend to exploit more heterogeneous and distributed computing platforms, e.g. large NOW and Computational Grids.

In our previous research [1,2] we focused on *structured parallel programming* approaches [3], and the ones based on parallel algorithmic skeletons in particular. We exploited skeletons as a parallel coordination layer of functional modules, eventually made up of conventional sequential code. Our experience is that this kind of skeleton model brings several advantages, but does not fulfill all the practical issues of efficient program development and software engineering.

In this paper we discuss the implementation of ASSIST [4], a general-purpose parallel programming environment. Through a coordination language approach we aim at supporting parallel, distributed and GRID applications. ASSIST eases software integration also by exploiting the software component paradigm.

In ASSIST, the skeleton approach is extended with new ideas to allow for (1) sufficient expressiveness to code more complex parallel solutions in a modular, portable way, (2) the option to write skeleton-parallel applications interacting with, or made up of components, and (3) the option to export parallel programs as high performance software components.

⋆ This work has been supported by the *Italian Space Agency*: ASI-PQE2000 Programme on "Development of Earth Observation App.s by Means of Systems and Tools for HPC", and by the *National Research Council*: Agenzia 2000 Programme.

H. Kosch, L. Böszörményi, H. Hellwagner (Eds.): Euro-Par 2003, LNCS 2790, pp. 712–721, 2003.
© Springer-Verlag Berlin Heidelberg 2003

Sect. 2 summarizes the motivation and main features of the ASSISTcl coordination language. We describe in Sect. 3 the structure and implementation of the first prototype compiler, and in Sect. 4 the compilation process and the language run-time, with respect to the constraints and the issues raised by the language definition. First results in verifying the implementation are shown in Sect. 5, Sect. 6 discusses related approaches in the literature, and Sect. 7 draws conclusions and outlines future development.

2 The ASSIST Approach, Motivation, and Features

Our previous research about structured parallelism and skeleton-based coordination has verified several advantages of the approach. However, our experience is that data-flow functional skeletons are not enough to express in a natural, efficient way all parallel applications, especially those that *(i)* involve irregular or data-driven computation patterns, *(ii)* are data-intensive in nature (I/O bound or memory constrained), *(iii)* are complex and multidisciplinary, hence require access to specific external resources and software modules by means of mandated protocols and interfaces. Stateless, isolated modules linked by simple and deterministic graphs can lead to inefficient skeleton compositions in order to emulate state-dependent or concurrent activities in a parallel program.

Moreover, the approach did not allow us the reuse of *parallel* programs as components of larger applications, nor to easily exploit dynamically varying computational resources. These are major issues in view of the affirmation of high performance, large-scale computing platforms (from huge clusters to Computational Grids), which require us both to promote cooperation of software written within different frameworks, and to exploit a more informed management of computational resources.

The ASSIST environment has been designed with the aim of providing

- high-level programmability and productivity of software development
- performance and performance portability of the resulting applications
- enhanced software reuse, and easier integration between ASSIST programs and other sequential/parallel applications and software layers.

These requirements have a visible impact on the structure of the coordination language. While pipeline, farm and map parallel skeletons are still supported by ASSISTcl, new, more expressive constructs are provided by the language. The language support has been designed to more easily interact with heterogeneous program modules and computational resources (e.g. different communication supports and execution architectures). Due to lack of space, we can only summarize here the distinguishing features of ASSIST, which are detailed in [5,4].

Coordination Structure. A module is either a unit of sequential code, or a skeleton construct coordinating more sequential or parallel modules. In addition to the older skeletons, a generic *graph* container is available, that specifies arbitrary connections among modules, allowing multiple input and

output channels per module. This implies a radical departure from the simpler and more restrictive notion of module coordination we adopted in past research.

Parallel Expressiveness. The *ParMod* skeleton [4] is introduced, which exploits increased programmability. A ParMod coordinates a mixed task/data parallel computation over a set of *virtual processors* (VP), to be mapped to real processing units. The set of VPs has a topology, data distribution policy, communications and synchronization support. The ParMod allows managing multiple, independent streams of data per VP, different computations within a VP being activated in a data-driven way. It is possible to manage *nondeterminism* over input streams, using a CSP-like semantics of communication, as well as to explicitly enforce synchronous operation of all the VPs. When needed, data structures from the streams can be broadcast, distributed on-demand, or spread across the VPs, according to one of the available *topologies*. With respect to the topology, fixed and variable stencil communications are allowed, as well as arbitrary, computation dependent communication patterns.

Module State. Program modules are no longer restricted to be purely functional, they can have a local state. A *shared* declaration allows ASSIST variables to be accessed from all the modules of a skeleton. Consistency of shared variables is not rigidly enforced in the language, it can be ensured (*i*) by algorithmic properties, (*ii*) by exploiting the synchronization capabilities of the ParMod skeleton, or (*iii*) by a lower-level implementation of the data types as *external objects*.

External Resources. The abstraction of *external objects* supports those cases when module state includes huge data, or unmovable resources (software layers or physical devices, databases, file systems, parallel libraries). External objects are black-box abstract data types that the ASSIST program can communicate with, sequentially and in parallel, by means of object method calls. As an essential requirement, external objects are implemented by a separate run-time tool, which must avoid hidden interactions with the execution of parallel programs. The approach is being used to develop a run-time support of module state which handles out-of-core data structures as objects in virtual shared memory, and is independent from ASSIST shared variables.

Software Components. ASSIST can both use and export component interfaces (e.g. CORBA). The new features of the language and the improvements in the run-time design make it possible to use external, possibly parallel software components as ASSIST program modules, and to export an ASSIST application as a component in a given standard.

The expressive power of the graph and ParMod constructs encompasses that of conventional skeletons[1] and allows the expression of combinations of task and data parallelism not found in other skeleton-based languages or in data-parallel models (e.g. systolic computations and large/irregular divide and conquer algorithms). The new features, and the management of dynamic, heterogeneous

[1] In principle we can exploit VPs to write low-level, unstructured parallel programs.

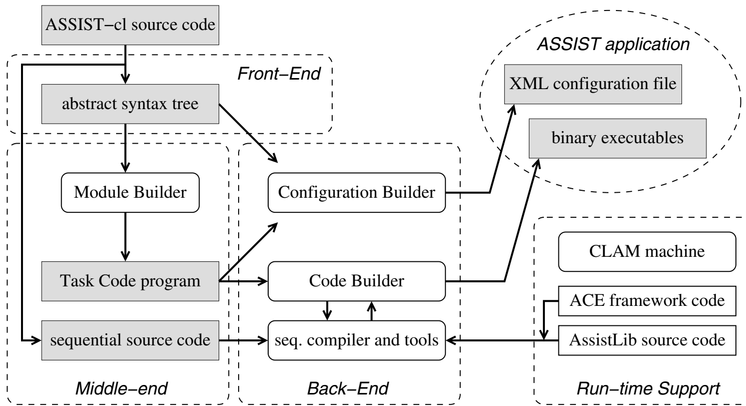

Fig. 1. Compilation steps in ASSIST

resources, lead to new open issues in the implementation, in particular w.r.t. compile-time optimizations and run-time performance models.

3 Implementation of the ASSIST Compiler

The ASSISTcl compiler (Fig. 1) has been designed to be modular and extendable. In particular, *(i)* the compilation process is divided into completely separate phases, *(ii)* portable file formats are used for the intermediate results, *(iii)* the compiler exploits object-oriented and design pattern techniques (e.g. visitor and builder patterns are consistently used from the Front-End to the Back-End of the compiler). The intermediate formalisms of all compilation phases have been designed with great care for the issues of portability and compatibility of the compilation process with heterogeneous hardware and software architectures.

Front end. The compiler front-end is based on a fairly standard design, in which the high-level source code is parsed, turned into an abstract syntax tree and type-checked. Host language source code (C, C++, Fortran) is extracted from ASSIST program modules and stored separately, gathering information about all sequential libraries, source files and compilation options used.

Middle End. The middle-end compilation (the *Module Builder* unit) employs an intermediate representation called *Task Code*, described in Sect. 3.1. Contrary to the high-level syntax, Task Code represents a materialized view of the process graph used to implement the program, with each skeleton compiled to a subgraph of processes that can be thought of as its implementation template (Fig. 2b). It is actually a mixed-type process network, as some of the graph nodes represent internally parallel, SPMD activities. Global optimizations can then be performed on the whole Task Code graph.

Back End. The compiler Back-End phase contains two main modules.

The *Code Builder* module compiles Task Code processes to binary executables, making use of other (sequential) compilers and tools as appropriate for each module. Most of the run-time support code is generated from the Task-Code specification and the sources of the C++ AssistLib library.
In the same compilation phase, the *Configuration Builder* module generates a program configuration file, encoded in XML, from the abstract syntax tree and Task Code information. This `assist_out.xml` file describes the whole program graph of executable modules and streams, including the information needed by the run-time for program set-up and execution (machine architecture, resources and a default process mapping specification).

3.1 Program Representation with Task Code and Its Run-Time Support

Task Code (Fig. 2b) is the intermediate parallel code between the compiler Front-End and its Middle-End. It is a memory resident data structure, with full support for serialization to, and recovery from a disk-resident form. This design choice enhances compiler portability, and allows intermediate code inspection and editing for the sake of compiler engineering and debugging. The Task Code definition supports sequential and SPMD parallel "processes", each one having

- a (possibly empty) set of input and output stream definitions, along with their data distribution and collection policy
- sequential code information previously gathered
- its parallel or concurrent behaviour as defined by the high-level semantics (e.g. nondeterminism, shared state management information).

Task Code supports all of the features of ParMod semantics. Different parts of the abstract machine that supports these functions are implemented at compile time, at load-time and at run-time. The parallel primitives we have mentioned so far are mainly provided in the AssistLib library. The Task Code graph is also a target for performance modelling and optimization. While results about the classical skeletons can be applied to simple enough subgraphs, in the general case approximate or run-time adaptive solutions will be required to deal with the increased complexity of the ASSIST coordination model, and the more dynamic setting of heterogeneous resources.

AssistLib is a compile-time repository of object class definitions and method implementations. Heavy use is made of C++ templates and inline code in order to reduce the run-time overhead in the user code. AssistLib comprises:

- routines for communication and synchronization
- implementation of stream data distribution policies
- implementation of the SMU (state management unit)
- interface to shared data structures handled by the SMU
- interface to "external object" resources.

Most of the concurrency and communication handling is currently performed by Reactor objects of the ACE library (see Sect. 4). Template and inline code implements communications exploiting the information statically available about communication channels (data types, communication policy).

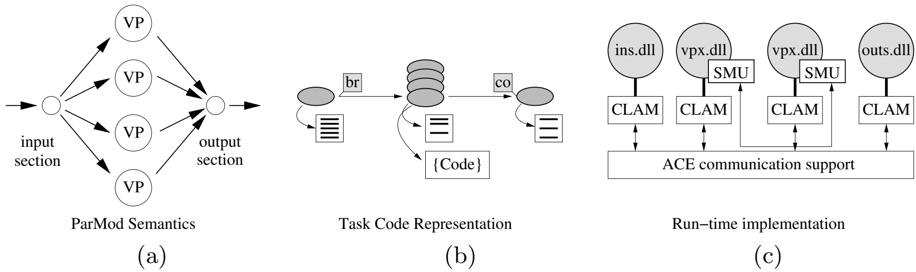

Fig. 2. ParMod compilation steps.

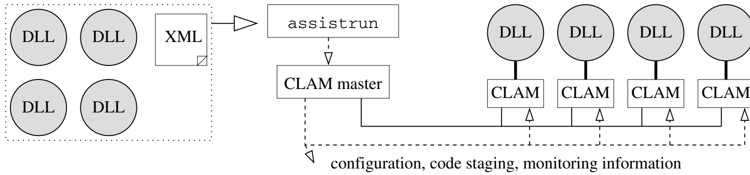

Fig. 3. ASSIST program execution scheme.

The SMU code encapsulates shared state management within the user code, mapping shared variables to data that is replicated or distributed across VPs according to the declarations in the ASSIST program. The SMU core essentially becomes part of the user code (Fig. 2c), providing the run-time methods to access shared data, and to ensure state coherence among the VPs when required by ParMod semantics. Optimizations are performed if multiple VPs run as threads in the same process, and owner-initiated communications are employed if the shared data access pattern is easily predictable (e.g. a fixed stencil).

The AssistLib library uses communication and resource management primitives provided by the run-time support. In the current implementation, some functionalities are provided by the ACE library, and their implementation is chosen at compile-time, while other ones can still be changed at run-time.

4 Run-Time Architecture

The current prototype of the ASSIST environment has been developed on a cluster of LINUX workstations, exploiting the communication and O.S. abstraction layers of the ACE library [6] (Fig. 2c). The use of the ACE object-oriented framework to interface to primary services (e.g. socket-based communications, threads), besides symplifying the run-time design, ensures a first degree of portability which includes all POSIX compliant systems.

The communication support provided by ACE relies on standard UNIX sockets and the TCP/IP stack, which are ubiquitous, but cannot fully exploit the performance of modern communication hardware. The language run-time is expandable either by extending the ACE classes to use new communication libraries, or by rewriting part of the communication primitives of AssistLib.

The standard ASSIST loader `assistrun` uses the XML configuration file to set-up the process net when running the program (Fig. 3). Manual, automatic and visual tool assisted editing of program configuration is also possible, enhancing the flexibility of the compiled application. The collected executable modules are designed to run with the help of CLAM, the coordination language abstract machine. The CLAM is indeed no complete virtual machine, it provides run-time services (resource location and management, computation monitoring) within heterogeneous and dynamically configurable architectures. To accomplish this goal, the CLAM either implements its own configuration services (current version) or interfaces to existing ones, e.g. [7].

A CLAM master process monitors the computation and manages the global configuration, interfacing to a set of CLAM slave processes that run on the available computing nodes. CLAM slaves can load code in the form of dynamically linked libraries (DLL), and they can act as intermediaries between the executable modules and the communication software layer. The CLAM master "executes" the `assist_out.xml` file by assigning the slaves a proper mapping of executable modules in the program. CLAM support processes react to changes in the program and in machine configuration. They communicate using a simple and portable internal protocol, while application communications are performed according to CLAM internal process mapping information. When several VPs are mapped to the same phisycal node, CLAM slaves execute their DLL either as separate processes, or as threads. The first method is more robust, as separation of the user and support code is complete, while the second introduces less run-time overheads. A third executable form, useful for testing and for statically configured programs and architectures, simply bypasses the CLAM and employs ACE processes and primitives.

Interoperability with standards like CORBA is currently supported in two ways.

– ASSIST programs can access CORBA functions from within the user code sections by gaining access to an external ORB. The ORB reference can be provided in the application configuration file.
– CORBA services can be implemented by ASSIST modules, for instance by activating a CORBA servant within a program module. Exporting CORBA services at the user code level is a blocking operation, so the module becomes unavailable as long as the ORB is alive, and the servant must actually be a sequential module.

A different, compiler-supported solution, which we are currently developing, allows using as a servant a generic parallel ASSIST subprogram f with unique input and output. The compiler inserts into the program a support process p (actually a ParMod with a single virtual processors) that runs ORB code as a thread. Process p forwards external CORBA requests to f and gets back answers on the ASSIST internal interfaces.

A variant of this scheme has been used to run Java bytecode inside VPs, by starting Java virtual machines as separate processes and forwarding them the tasks to compute. Building on these integration experiences, we plan to

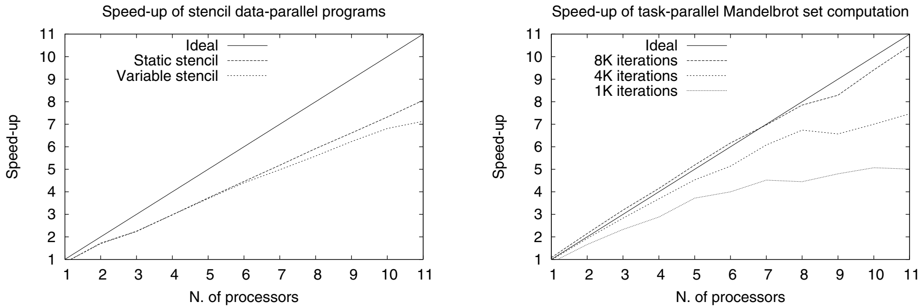

Fig. 4. Speed-up results. (left) Computation of a function with stencil access over an array of 1600x1600 points, computation is 470 ns per point. — (right) Farm load-balancing computation over the same array, Mandelbrot function at 1K, 4K and 8K iterations per point, computation grain is 1280 points per packet.

fully integrate high performance components in more effective ways in the next versions of the ASSIST run-time.

5 First Results

We are currently running simple tests on a small Beowulf cluster in order to test compiler and support functionalities. We have verified that the ParMod implementation can efficiently handle both geometric algorithms (data parallelism with stencil access to non-local data, Fig. 4-left) and dynamic load balancing farm computations, already at a small computation grain (470 ns is 30-300 machine instructions on the Pentium II platform used), Fig. 4-right. Although some support optimization are still missing, test results in Fig. 4-left show that the SMU support for dynamically computed stencil patterns is almost as efficient as that of static (unchanging) stencils.

These results are confirmed by preliminary tests on a computational kernel of molecular dynamic simulation, developed using the ASSIST technology (AssistLib) as part of the ASI-PQE2000 Research Program. More results are found in [8]. To verify the feasibility of mixing data parallelism and task parallelism within the same application kernel, we are currently implementing with ASSIST the C4.5 decision-tree classifier described in [4]. Decision tree classifiers are divide and conquer algorithms with an unbalanced, irregular and data-driven computation tree, and require using data and control-parallel techniques in different phases of the execution, while at the same time managing out-of-core data structures. This kind of applications can be written only with low-level, unstructured programming models (e.g. MPI), and clearly show the limits of pure data-parallel programming models. In [4] we propose a solution based on two ParMod instances and external object support to shared memory data.

6 Related Work

In [9] an extensive comparison of the characteristics of several parallel programming environments is reported, including ASSIST, and developing standards for High Performance and Grid components (CCAFFEINE, XCAT) are referenced. Another notable example reported is CO_2P_3S, a language based on parallel design patterns. CO_2P_3S is aimed at SMP multithreaded architectures, and its expressive power comes in part from exposing the implementation of patterns to the application programmer. The Ligature project [10] also aims at designing a component architecture for HPC, and an infrastructure to assist multicomponent program development. Kuchen [11] describes a skeleton-based C++ library which supports both data and task parallelism using a two tier approach, and the classical farm and map skeletons. As a closer approach, we finally mention the GrADS project [12]. It aims at smoothing Grid application design, deployment and performance tuning. In GrADS the emphasis is much stronger on the dynamic aspects of program configuration, including program recompilation and reoptimization at run-time, and thus performance models and contracts issues. Indeed, in the GrADS project the programming interface is a problem solving environment which combines components.

7 Conclusions

We have presented a parallel/distributed programming environment designed to be flexible, expandable and to produce efficient and portable applications. Primary features of language are the ease of integration with existing applications and resources, the ability to mix data and control parallelism in the same application, and the ability to produce application for heterogeneous architectures and Computing Grids. By design, it is possible to exploit different communication support methods by extending the CLAM or the ACE library. Key resources in this direction are libraries for faster communication on clusters (e.g. active messages) and Computational Grid support libraries like Nexus [7].

Future improvements will be the result of a multidisciplinary work, in collaboration with several other research groups. We already made efforts to integrate the ASSIST environment with existing parallel numerical libraries [13], and a tool has been developed to help run ASSIST applications over Grid Computing Environments [14].

The support of the language has been verified on Beowulf clusters. At the time of this writing, we are developing more complex applications to stress the environment and tune performance optimization in the compiler. Among the “heavyweight” applications in development there are data mining algorithms for association rule mining and supervised classification; earth observation applications using SAR interferometry algorithms; computational geometry kernels.

The parallel cooperation model of ASSIST allows consideration of a module, especially a ParMod, as a software component. It is already possible to use CORBA sequential objects, and to export a CORBA interface from an ASSIST program. In a more general setting, ASSIST modules can be seen as parallel

software components. Thus, future development of the ASSIST environment will interact with the development of parallel and high-performance component interfaces, exploiting them to connect program modules.

Acknowledgments. We wish to thank D. Laforenza, S. Orlando, R. Perego, R. Baraglia, P. Palmerini, M. Danelutto, D. Guerri, M. Lettere, L. Vaglini, E. Pistoletti, A. Petrocelli, A. Paternesi, P. Vitale.

References

1. Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S., Vanneschi, M.: A structured high level programming language and its structured support. Concurrency Practice and Experience **7** (1995) 225–255
2. Bacci, B., Danelutto, M., Pelagatti, S., Vanneschi, M.: SkIE : A heterogeneous environment for HPC applications. Parallel Computing **25** (1999) 1827–1852
3. Skillicorn, D.B., Talia, D.: Models and languages for parallel computation. ACM Computing Surveys **30** (1998) 123–169
4. Vanneschi, M.: The programming model of ASSIST, an environment for parallel and distributed portable applications. Parallel Computing **28** (2002) 1709–1732
5. Vanneschi, M.: ASSIST: an Environment for Parallel and Distributed Portable Applications. Technical Report TR-02-07, Dip. di Informatica, Università di Pisa (2002)
6. Schmidt, D.C., Harrison, T., Al-Shaer, E.: Object-oriented components for high-speed network programming. In: Proceedings of the 1st Conference on Object-Oriented Technologies and Systems (COOTS), Monterey, CA, USENIX (1995) extended version online at `http://www.cs.wustl.edu/~schmidt/ACE-papers.html`.
7. Foster, I., Kesselman, C.: The Globus toolkit. In Foster, I., Kesselman, C., eds.: The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann (1998)
8. Aldinucci, M., Campa, S., Ciullo, P., Coppola, M., Danelutto, M., Pesciullesi, P., Ravazzolo, R., Torquati, M., Vanneschi, M., Zoccolo, C.: Assist demo: A high level, high performance, portable, structured parallel programming environment at work. In: *Proceedings of EuroPar'03*. (2003) to appear.
9. D'Ambra, P., Danelutto, M., di Serafino, D., Lapegna, M.: Advanced environments for parallel and distributed applications: a view of current status and trends. Parallel Computing **28** (2002) 1637–1662
10. Keahey, K., Beckman, P., Ahrens, J.: Ligature: Component architecture for high performance applications. The International Journal of High Performance Computing Applications **14** (2000) 347–356
11. Kuchen, H.: A skeleton library. In Monien, B., Feldman, R., eds.: Euro-Par 2002 Parallel Processing. Number 2400 in LNCS (2002) 620–629
12. : The GrADS Project. `http://hipersoft.cs.rice.edu/grads/` (2003)
13. D'Ambra, P., Danelutto, M., di Serafino, D., Lapegna, M.: Integrating MPI-based numerical software into an advanced parallel computing environment. In: Proc. of 11th Euromicro Conf. PDP2003, IEEE (2003) 283–291
14. Baraglia, R., Danelutto, M., Laforenza, D., Orlando, S., Palmerini, P., Perego, R., Pesciullesi, P., Vanneschi, M.: Assistconf: A grid configuration tool for the assist parallel programming environment. In: Proc. of 11th Euromicro Conf. PDP2003, IEEE (2003) 193–200