A framework for experimenting with structured parallel programming environment design*

M. Aldinucci[a], S. Campa[b], P. Ciullo[b], M. Coppola[a], M. Danelutto[b], P. Pesciullesi[b], R. Ravazzolo[b], M. Torquati[b], M. Vanneschi[b] and C. Zoccolo[b]

[a]Institute of Information Science and Technologies (ISTI) – National Research Council (CNR), Via Moruzzi 1, I-56124 Pisa, Italy

[b]Department of Computer Science, University of Pisa, Via Buonarroti 2, I-56127 Pisa, Italy

ASSIST is a parallel programming environment aimed at providing programmers of complex parallel application with a suitable and effective programming tool. Being based on algoritmical skeletons and coordination languages technologies, the programming environment relieves the programmer from a number of cumbersome, error prone activities that are required when using traditional parallel programming environments. ASSIST has been specifically designed to be easily customizable in order to experiment different implementation techniques, solutions, algorithms or back-ends any time new features are required or new technologies become available. In this work we discuss how this goal has been achieved and how the current ASSIST programming environment has been already used to experiment solutions not implemented in the first version of the tool.

1. Introduction

Our research group recently developed a structured parallel programming environment based on the skeleton and coordination language technology. The programming environment (ASSIST, A Software development System based on Integrated Skeleton Technology [13,14]) is intended to solve some of the problems we had in the past, while designing other structured parallel programming environments such as P3L [4] and SkIE [5]. Those problems were mainly related to language expressiveness and interoperability. The whole ASSIST environment has been designed and implemented exploiting well known software engineering techniques. These enable the easy extension of programming environment features. The result is an high performance, structured, parallel programming environment that produces code for plain POSIX/TCP workstation networks/clusters. The object code produced by the compiling tools demonstrated good efficiency and scalability on medium to coarse grain parallel applications. Furthermore, some of the features inserted in the structured coordination language of ASSIST (ASSISTcl) allow fair interoperability levels to be achieved (e.g. with respect to the CORBA framework) as well as to use different kind of existing, optimized library codes within an ASSISTclparallel application. The result is a programming environment that can be suitably used to program complex, interdisciplinary applications.

The ASSIST features are discussed elsewhere [13,14,6,8,3,2]. In this work we want to point out how the ASSIST programming environment can be used to experiment new implementation techniques, mechanisms and solutions within the framework of structured parallel programming models. Therefore we briefly outline the ASSIST implementation structure and then we proceed discussing some experiments we performed aimed at extending the environment. Those experiments were aimed at modifying ASSIST environment in such a way it can be used to program GRID architectures, include existing libraries in the application code, target heterogeneous cluster architectures, etc.

The paper is organized as follows: Section 2 outlines the ASSIST environment features. Section

*This work has been partially supported by the Italian MIUR Strategic Project "legge 449/97" year 1999 No. 02.00470.ST97 and year 2000 No. 02.00640.ST97, and by the Italian MIUR FIRB Project *GRID.it* No. RBNE01KNFP.

3 describes some experiments performed that took advantage of the ASSIST features. Eventually, Section 4 outlines the current experiments performed with the ASSIST environment.

2. ASSIST

ASSIST is a programming environment oriented to the development of parallel and distributed high performance applications. It is based on a coordination language providing programmers with a structured coordination language (ASSISTcl) based on customizable parallel skeletons [10,12,7] that can be used to model the most common parallelism exploitation patterns, and a developing toolkit to compile ASSISTcl programs to homogeneous clusters of POSIX workstations (`astCC`).

ASSISTcl allows arbitrary graphs of concurrent, possibly parallel activities to be defined in a program. Items in the graph are interconnected by means of data flow streams. In turn, parallel activities appearing in the graph can be expressed using the ASSISTcl skeletons, as well as using plain sequential C, C++ or F77 code. ASSISTcl skeletons include the `parmod` (*parallel module*, a configurable skeleton that can be used to model most of the classical skeletons (pipelines, task farms, data parallel skeletons). A complete description of the coordination language can be found in [13,14].

The ASSIST environment has been designed since the very beginning with the target of being efficient (i.e. able to produce fast object code) as well as modifiable on the need. Therefore, the whole ASSISTcl compiling tools have been given a three tier design:

front-end (the top tier), parses ASSISTcl 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, parallel assembly language. The step transforming internal representation into task code is completely implemented exploiting design pattern technology [9]. A façade pattern decouples compiler internals from the compiler engine;

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 basically built on POSIX processes/threads and communication (SysV and TCP/IP sockets) primitives. All those primitives are used via the ACE (Adaptive Communication Environment) library [1].

The result of the whole compilation process consists in two distinct items: the object code files (either as compiled code or as shared libraries) and an XML configuration file, holding all the information needed to run the program: which objects need to be loaded/executed on which processing element(s), how streams are mapped to INET addresses ($\langle host, port\rangle$ pairs), which existing libraries or (external) object codes must be loaded on the processing elements, etc.)

The three tiers 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 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, ASSISTcl compiled code is run by means of a dedicated loader (the `assistrun` command), that in turn activate CLAM run-time support. 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 run on the target architecture processing nodes load and execute the suitable code[2] after the set up of the communication infrastructure (i.e. after the proper TCP/IP sockets have been published on the nodes). A more detailed description of the ASSIST implementation can be found in [3]. As CLAM (and the object code itself) access POSIX features via ACE wrappers, and as ACE is available on different operating systems (Linux and Windows, as an example), CLAM actually behaves as the fourth tier of the compile/run process and guarantees a degree of portability of the whole programming environment.

[2] either coming from ASSISTcl source code or belonging to external libraries properly named by the programmer in the ASSISTcl source code

3. Experimenting with ASSIST

The first version of ASSIST has been designed in 2001 in the framework of an Italian National project involving Italian National Research Council and Italian Spacial Agency: the ASI-PQE project. This version correctly compiled a significant subset of the original ASSISTcl language. This subset did not include, as an example, pipeline and task farm skeletons. They rather can be implemented customizing a `parmod` skeleton. Furthermore, the compiler was able to produce code only for homogeneous[3] Linux/ACE clusters. Exploiting the ASSIST implementation features, we performed different experiments on this first prototype environment, that are described in the following Sections.

3.1. Optimized library inclusion

In many cases the application programmer would cope with specified problems by relying on optimized libraries. A notable example of such libraries include mathematical libraries build on top of the MPI programming environment. ASSISTcl provides different facilities to integrate existing sequential or parallel libraries. Mainly, programmers can denote in the program which external sources/object codes have to be compiled/linked to the ASSISTcl modules. The programmer may invoke completely external services from within the ASSISTcl modules as well. As an example, CORBA object services can be claimed from within the `parmod` sequential modules denoting parallel activities (or virtual processors, in our terminology). However, there was no facility in the original version of the environment that allowed programs to benefit from the execution of already existing, optimized, MPI-based library code. Exploiting the compiler and CLAM structure, we performed an experiment that allowed to use such libraries directly within the ASSISTcl code [8]. Basically, the `mpirun` command of the `mpich` version of MPI has been slightly modified in such a way that its services can be invoked from within CLAM processes. The whole library code has then been wrapped in such a way that it looked like a normal `parmod` code to the programmer. Overall, this allowed MPI libraries to be run in an ASSISTcl program without requiring an explicit programmer intervention.

Although the whole library integration process has not been included in the compiler yet, this experiment demonstrated that the compiler/CLAM pair is flexible enough to allow to completely independent, parallel library code to access the services provided by CLAM (e.g. to know the INET address of the input and output stream), as well as to provide services to the rest of the ASSISTcl program (e.g. to provide "compute" calls to other modules appearing in the ASSISTcl program graph).

3.2. GRID

As discussed, the output produced by `astCC` is made up by a set of object code/DLLs and an XML file, representing the "structure" of the parallel code. Exploiting this feature, ASSISTcl programs can be run on a GRID configuration as follows performing the following three steps. First, the XML configuration file can be analyzed and resources needed to execute the program can be individuated. Such resources are defined in terms of generic "processing elements" in the original compiler XML file. Second, the resources needed to execute the program can be gathered (and reserved) from GRID using the normal GRID middle-ware tools (e.g. those provided by the Globus toolkit). Last, the XML file can be modified in such a way that the resources gathered are used to run the ASSIST code.

In order to to demonstrate the feasibility of the approach we developed a tool that support such kind of manipulation of the original XML configuration file [6]. Actually, the tool only supports programmer decisions: 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.

In Section 4, we should outline how this process is being fully integrated in the ASSIST programming environment in such a way the environment could efficiently target GRID architectures also.

3.3. Heterogeneous clusters

The first version of ASSIST produces code for homogeneous cluster of Linux PC/WS only. The missing items needed to produce code for heterogeneous architectures are basically two: the inclusion of some kind of XDR (external data representation) wrapping messages flowing among heterogeneous

[3] in terms of processor type

processing elements, and the generation of proper makefiles to compile final object code[4]. Both this problems can be solved exploiting the ASSISTcl compiling tools structure. As 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, we are currently intervening on these two builders in order to modify them in such a way that:

- 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 config 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.

Again, the algorithms that solve these problems are currently being incorporated in the ASSISTcl compiling tools. We plan to have a working version of the compiler targeting Intel Linux and Mac OsX networks by the end of this year.

3.4. ASSISTcl improvement

The first version of the ASSISTcl coordination language demonstrated some problems and pitfalls mainly due to the strict timings involved in language design and implementation in the ASI-PQE project. Currently, we are enhancing the coordination language features, mainly those related to the language expressivity and to the possibility to use external libraries from within the sequential portions of code included in the ASSISTcl source code.

Different enhancements have been designed and implemented exploiting the three tier structure of the ASSIST environment compiling tools. As an example, a smarter syntax has been designed to allow items of the output data streams of a `parmod` to be gathered from `parmod` virtual processors (i.e. internal, concurrent/parallel `parmod` activities). These enhancements did not imply any changes in the compiler core and back-end layers. Just the front end has been modified.

Other enhancements regard the possibility to have variable length arrays in the ASSISTcl type systems. This is dramatically important in order to efficiently implement the activities (and the communications) involved in Divide&Conquer like computations. Again, the introduction of variable length data structures[5] only interested the front-end and part of the core compiler layer. The existing task code implementation perfectly supports these changes.

4. Ongoing activities

The experiments described in Section 3 have already been performed and currently the related experience is being moved to the production compiler. However, as the ASSIST environment was exactly meant to be a sort of test-bed to experiment new solutions to efficiently support structured parallel programming on a wide range of target architectures, we are currently studying different new enhancements, in the context of several National Research projects. These experiences are described in the following sections.

4.1. Component-based ASSISTcl

In the context of the FIRB project, we are re-designing the ASSIST environment as a full-fledged component based programming model. This means that the upper part of the coordination language will move to a component framework and that existing ASSISTcl skeletons will be provided as parametric components to be used in the construction of parallel applications. This is possible as

[4]The `astCC` compiler actually produces a set of C++ files that include calls to ACE and CLAM services, but these need to be compiled using a standard C++ compiler. This is because we do not aim at entering the sequential code compiling area.

[5]in the sense of C++ Vector objects

the modules that are used to construct the generic graphs appearing in ASSISTcl programs are already conceived as close modules interacting with the external world (other modules) via streams and events. As a matter of fact, this means that ASSISTcl modules already behave as (non-standardized) components. On the other side, the task code is already organized as a class hierarchy providing objects that model common parallel program components. Therefore, while restructuring the coordination language level, we are also trying to expose part of the task code at the component level, in such a way that experienced users can program directly the component task code level to provide new higher level components to applicative programmers (i.e. to the end users of the ASSIST programming environment)[6].

4.2. Full GRID ASSISTcl

In the meanwhile, in the context of both strategic project "legge 449/97" and FIRB project we are currently trying to make automatic the targeting of GRID architectures. In order to produce efficient code, many factors have to be taken into account, which are traditionally handled by expert parallel programmers: resource co-allocation, code and data staging, task scheduling and the alike. The structure of existing ASSIST programming environment can be exploited in this case as follows:

- resource co-allocation can be decided on the basis of the contents of the XML configuration file produced by the ASSISTcl 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 GRIDs may easily process the XML config file in such a way that resources are looked for that match the needs stated in that file.
- Code and data staging can also be managed by the CLAM setup process. Also on clusters, the first phase in the execution of an ASSISTcl program consists in deploying the proper object/library code to the interested processing nodes. Data items instead are delivered to the processing nodes needing them either as data items on the streams connecting the ASSISTcl program modules (and this happens under direct programmer control), or as data items belonging to the underlying distributed shared virtual memory subsystem (this is automatically managed by the ASSISTcl runtime).
- Task scheduling is completely under the control of CLAM and follows the directives taken from the XML configuration file.

Therefore, by moving the GRID configuration and management phase to the XML config file processing phase and to the CLAM we expect that the whole ASSIST environment can be made "GRID aware".

5. Conclusions

In this work we outlined the features of the ASSIST programming environment and we discussed several experiences aimed at improving the programming environment features. We are currently consolidating the experimental results achieved with the ASSIST programming environment. That means that current "production" compiling tools do not include some of the results already assessed. Rather, development versions of the ASSISTcl compiling tools include these results and are currently being debugged and used by our team. In all cases, once the compiler debugging has been completed, by using the ASSIST programming environment programmers took hours to develop complete parallel applications out of existing sequential code. And these applications demonstrated good (close to

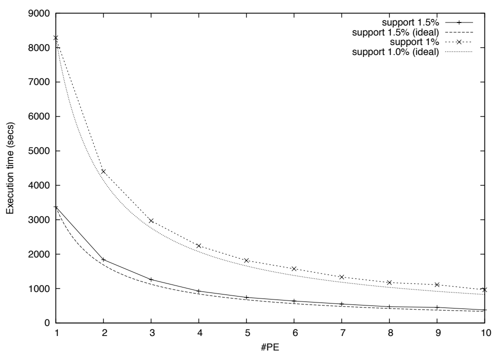

Figure 1: Typical ASSISTcl program performance (data mining code on Linux PC cluster)

[6]Much in the style of what already happens in the design pattern environment COPS [11].

ideal) scalability on workstation clusters with either Fast or Gbit Ethernet. As a typical example of performances achieved using ASSIST, Figure 1 plots the execution times (ideal and measured) of an ASSIST data mining application run on a network of Linux PCs (the different curves are relative to different dimensions of the support set).

Never, once debugged versions of the compilers has been used, programmers needed to enter the classical debug/compile/run cycle. In particular, they didn't care about correctness of process/parallel activities setup and scheduling, communications, termination etc. Instead, programmers could spend time in experimenting different parallelization strategies for the application at hand. This activity requires to rewrite from scratch a few lines of the coordination/skeleton code rather than entire parts of the program. Overall, this represent a consistent advantage with respect to what happens when using other parallel programming environments, at least in our experience.

REFERENCES

[1] The Adaptive Communication Environment home page. http://www.cs.wustl.edu/∼schmidt/ACE-papers.html, 2003.

[2] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, M. Danelutto, P. Pesciullesi, R. Ravazzolo, M. Torquati, M. Vanneschi, and C. Zoccolo. ASSIST demo: a high level, high performance, portable, structured parallel programming environment at work. In *Proceedings of EuroPar'03*, LNCS. Springer Verlag, august 2003. to appear.

[3] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, S. Magini, P. Pesciullesi, L. Potiti, R. Ravazzolo, M. Torquati, M. Vanneschi, and C. Zoccolo. The Implementation of ASSIST, an Environment for Parallel and Distributed Programming. In *Proceedings of EuroPar'03*, LNCS. Springer Verlag, august 2003. to appear.

[4] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A Structured High level programming language and its structured support. *Concurrency Practice and Experience*, 7(3):225–255, May 1995.

[5] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SkIE: a heterogeneous environment for HPC applications. *Parallel Computing*, 25:1827–1852, December 1999.

[6] R. Baraglia, M. Danelutto, D. Laforenza, S. Orlando, P. Palmerini, R. Perego, P. Pesciullesi, and M. Vanneschi. AssistConf: A Grid Configuration Tool for the ASSIST Parallel Programming Environment. In *Proceedings of the Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 193–200. Euromicro, IEEE, February 2003. ISBN 0-7695-1875-3.

[7] M. Cole. Bringing skeletons out of the closet. available at author's home page, december 2002.

[8] P. D'Ambra, M. Danelutto, D. di Serafino, and M. Lapegna. Integrating MPI-Based Numerical Software into an Advanced Parallel Computing Environment. In *Proceedings of the Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 283–291. Euromicro, IEEE, February 2003. ISBN 0-7695-1875-3.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[10] H. Kuchen. A skeleton library. In *Proceedings of the Euro-Par 2002 Conference*, LNCS. Springer Verlag, August 2002.

[11] S. MAcDonald, J. Anvik, S. Bromling, J. Schaeffer, D. Szafron, and K. Taa. From patterns to frameworks to parallel programs. *Parallel Computing*, 28(12):1663–1684, december 2002.

[12] J. Serot and D. Ginhac. Skeletons for parallel image processing: an overviwe of the SKIPPER project. *Parallel Computing*, 28:1685–1708, December 2002.

[13] M. Vanneschi. ASSIST: an environment for parallel and distributed portable applications. Technical Report TR 02/07, Dept. Computer Science, Univ. of Pisa, May 2002.

[14] M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28:1709–1732, December 2002.