# Parallel Software Interoperability by Means of CORBA in the ASSIST Programming Environment*

S. Magini, P. Pesciullesi, and C. Zoccolo

University of Pisa, Dip. di Informatica - Via Buonarroti 2, 56127 Pisa, Italy
zoccolo@di.unipi.it

**Abstract.** Parallel software reuse and easy integration between parallel programs and other sequential/parallel applications and software layers can be obtained exploiting the software component paradigm. In this paper we describe the ASSIST approach to interoperability with CORBA objects and components, presenting two different strategies to export a parallel program in the CORBA world. We will discuss their implementations and provide some experimental results.

## 1 Introduction

The development of complex applications for the emerging high-performance large-scale computing platforms (ranging from huge Clusters up to computational GRIDS) poses productivity problems. High-level parallel programming tools give a partial solution, easing the development of complex parallel algorithms, while the exploitation of the software component paradigm can improve parallel software reuse and integration in larger applications.

ASSIST (A Software System based on Integrated Skeleton Technology) is a general-purpose high-level parallel programming environment, based on the skeleton and coordination language technology [1–3]. It combines the benefits of software reuse and integration with those of high-level programming, providing full interoperability with CORBA objects and components. It can easily import (or use) other pieces of software encapsulated in external CORBA objects and components, as in a traditional sequential language (i.e. C++); moreover, it can export parallel algorithms and applications as well: an ASSIST program can be encapsulated in a CORBA object and integrated into a larger application using standard CORBA invocation mechanisms.

CORBA, a distributed object- and component- based middleware, is a well established commercial standard, with thousands of users all over the world. It offers interoperability with several sequential languages and is supported by several

* This work has been supported by: the Italian MIUR FIRB Grid.it project, No. RBNE01KNFP, on High-performance Grid platforms and tools; the Italian MIUR Strategic Project L. 449/97-2000, on High-performance distributed enabling platforms; the Italian MIUR Strategic Project L. 449/97-1999 on Grid-computing for e-science.

M. Danelutto, D. Laforenza, M. Vanneschi (Eds.): Euro-Par 2004, LNCS 3149, pp. 679–688, 2004.
© Springer-Verlag Berlin Heidelberg 2004

big software companies. The interoperability with ASSIST provides the ability to develop computation- or data-intensive components of complex applications using an high-level parallel programming language, enabling the construction of scalable applications.

Recent studies recognized that CORBA technology could be leveraged to support the development of advanced Grid applications: the CORBA CoG kit [4], for example, provides access to the Grid services provided by the Globus Toolkit to CORBA applications. Several studies investigated how CORBA could be adapted to enable the construction of high-performance applications, enabling data-parallel programs to interact efficiently (see Sect. 5).

Our approach extends CORBA interoperability to stream-parallel programs, so we will focus on an efficient implementation of stream communication using standard CORBA mechanisms. This study is a preliminary step towards a complete CCM (CORBA Component Model, introduced in the CORBA 3 standard) integration and a possible generalization towards other component-based standards (e.g. Web/GRID services[5]). In previous works we showed that ASSIST programs could invoke external CORBA objects [6] and claimed that ASSIST programs could be automatically exported as CORBA objects [2]. In the following sections, we describe two different strategies to export an ASSIST program in the CORBA world (Sect. 2), we discuss their implementations (Sect. 3), provide experimental results (Sect. 4) and we survey related work (Sect. 5).

## 2 Exporting an ASSIST Program in the CORBA World

### 2.1 Structure of an ASSIST Program

ASSIST applications are developed by means of a coordination language [1] (ASSIST-CL), in which sequential as well as parallel computation units can be expressed and coordinated. Sequential code units, that are the bricks to build sequential and parallel modules, are described by a C-like procedure interface, in which input and output parameters are identified and associated with their type, and can be written in popular sequential languages (C, C++, Fortran 77), allowing maximal sequential code reuse. Parallel activities, designed following regular parallelism exploitation patterns, are marked, in the ASSIST-CL syntax, by means of the *parmod* construct: it can be considered a sort of "generic" skeleton, that can emulate the "classical" ones without performance degradation, but with more expressive power; for example, it allows both task and data parallelism to be exploited inside the same parallel module, and can be used to program lower (w.r.t. classical skeleton) level forms of parallelism. Parallel activities are decomposed in sequential indivisible units, which are assigned to abstract executors, called *virtual processors*. ASSIST allows them to communicate by reading and modifying in a controlled, consistence preserving way the shared computation state, which is partitioned among the virtual processors.

ASSIST modules interact by means of input and output interfaces. In module interfaces, ASSIST recognizes the standard CORBA types, which are mapped to native (C++) types following the standard C++ to CORBA type mapping[7].

The coordination model is based on the concept of *stream*. A stream is an ordered sequence, possibly infinite, of typed values. Streams connect modules in generic graphs, and drive the activation of their elaborations: either data-flow or nondeterministic behaviour can be selected, and the activation can be controlled by the internal module state. A composition of modules may be, in turn, reused as a component of a more complex application, provided that the interfaces match. The programming model includes also the concept of *external objects*, as a mean of inter-component interaction, as well as interaction with external world. Existing services (such as parallel file systems, databases, interactive user interfaces,etc.) can be encapsulated in an object (e.g. CORBA) and invoked inside an ASSIST application.

### 2.2 ASSIST Programs as Components of Larger CORBA Applications

A computation intensive component of a CORBA application can be conveniently expressed in ASSIST to exploit parallelism and achieve good performance, provided that it can interoperate with the application in a simple manner. We devised two possible ways of interconnecting an ASSIST subprogram to a distributed application, that address two different classes of problems:

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

**stream-like asynchronous** data passing (realized using the standard CORBA event channel mechanism), which is useful when the production of data to be processed takes longer and can be overlapped with the elaboration, or when the data structure to be processed is so big that it cannot be handled as a whole by the application, or as well when we want to receive partial results as soon as possible (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 (i.e. no module produces/consumes them) 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 the program receives from the input source (by means of the input stream), it must produce one and only one datum on the output stream.

## 3 Implementation

The process of exporting an ASSIST program in the CORBA world has been automatized. We had two options: modify the original ASSIST compiler in order to insert specialized support code for interoperation with CORBA in the compiled modules or develop a separate transformation tool that produces an

interoperable ASSIST program that can be compiled with the standard ASSIST compiler. The second options is more flexible: it allows experimenting with different solutions and different ORB implementations, and is easily extensible to other distributed objects middleware (CCM, GridServices). It can be slightly less efficient because the code interacting with CORBA runs as user code in the ASSIST program, and this imposes some restrictions on its interaction with the ASSIST runtime. We opted for the second solution: the ASSIST program undergoes an initial analysis phase (detailed in 3.1); then, if the conditions stated earlier are met, it is transformed, according to the option (RMI-synchronous vs. stream-asynchronous interaction, detailed in 3.2 and 3.3 respectively) chosen by the programmer, adding support code to interact with the CORBA runtime and services. Appropriate CORBA IDL interfaces are automatically generated for the transformed ASSIST program.

### 3.1 Analysis of the ASSIST Program

The analysis and transformation of the ASSIST program is carried out by a separate tool (the integration within the ASSIST compiler is ongoing), designed to parse and automatically manipulate an ASSIST program in source form.

The program is parsed and its representation is built as a set of C++ objects. Those objects provide methods to inspect and modify the structure of the program, that can finally be rewritten in the ASSIST syntax and compiled using the standard tools. The informations needed about the streams chosen to interface the program to CORBA are extracted from the parsed program. If no errors are encountered in this phase, the transformation proceeds according to the selected interaction method.

### 3.2 Exporting as a Synchronous Object Method

In the case of synchronous interaction, an ASSIST program (for example, see Fig. 1(a)) is exported as an object with a single `execute` method, with argument and return types equal respectively to the types of the input and output stream in the program interface. The transformation tool generates an IDL file (see Fig. 1(b)) describing the object interface, including any type definition (translated to IDL syntax) or preprocessor directive specified in the source program.

The original ASSIST program is enriched with a module that acts as a bridge towards CORBA (see Fig. 2). It instantiates an ORB in a separate execution thread, creates an instance of the CORBA object representing the exported algorithm and publishes it to a CORBA Naming Service, so that CORBA clients can easily search for and connect to it; finally it enters the ORB main loop, waiting for incoming requests.

Whenever a request is accepted by the ORB, the implementation of the execute method (running in the ORB thread) delivers the argument to the ASSIST program and then stops waiting for a response. The effective message delivery and reception is executed in the main thread of the server module, and controlled by the ASSIST runtime support. When the response is received, the execute method implementation returns it to its caller.

```
#define N 20

generic main() {
  stream long[N][N] Aaa;
  stream long[N][N] Bbb;

  spt(input_stream Aaa output_stream Bbb);
}

parmod spt (input_stream long A[N][N]
            output_stream long B[N][N])
{
// parallel code omitted
}
```

(a) ASSIST code.

```
// Autogenerated file - DO NOT EDIT
// CORBA interface for the spt ASSIST program

#define N 20

#define bool boolean

typedef long idl_ret_t[N][N];
typedef long idl_call_t[N][N];

interface spt {
 idl_ret_t execute(in idl_call_t _var);
};
```

(b) Generated IDL interface.

**Fig. 1.** Synchronous interaction example.

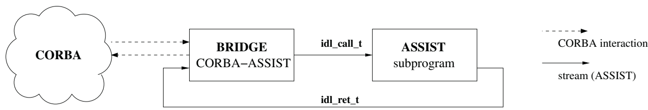



**Fig. 2.** Deployed ASSIST program (synchronous invocation).

### 3.3 Exporting as a Component Interconnected Through Event Channels

Using the former interaction method it is not possible to execute multiple requests simultaneously, therefore it is not effective for stream-parallel programs, like pipelines. To fully exploit stream parallelism, we developed a second mechanism based on CORBA event channels to implement asynchronous communications. The transformation tool generates an IDL file containing the definitions of event types for the input and output streams of the ASSIST subprogram, and a special termination event used to stop gracefully a running ASSIST server (see Fig. 3). In this case, no RMI interface is needed, because the ASSIST program communicates with CORBA only by means of events.

CORBA events can have any type and event reception is governed by the type, so we opted for the convention of encapsulating the datum in a named structure with a single field, to ease the discrimination of ingoing and outgoing messages.

The ASSIST program is transformed adding two modules (see Fig. 4), one (S1) that intercepts ingoing events and sends them to the input stream of the subprogram, and the other (S2) that receives message from the output stream and forwards them as outgoing events.

Each module, at initialization, connects to the CORBA naming service and obtains a reference to the COS Event Service, that is the CORBA service that manages event publishing, subscription and notification. Then the first one subscripts for the ingoing event and the special termination event and keeps forwarding messages until it receives a termination notification. At the same time,

```
// Autogenerated file - DO NOT EDIT
// CORBA event definitions for
// the spt ASSIST program

#define N 20

#define bool boolean

typedef long idl_ret_t[N][N];
typedef long idl_call_t[N][N];

struct Termination {
  bool stop;
};

struct source {
  idl_call_t data;
};

struct sink {
  idl_ret_t data;
};
```

**Fig. 3.** Generated IDL event definitions.

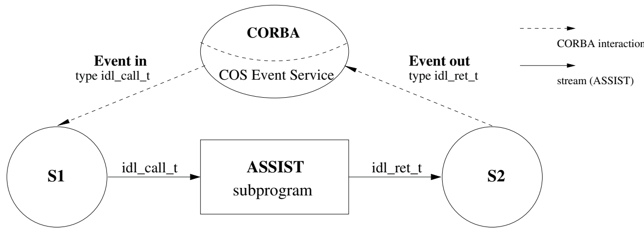



**Fig. 4.** Deployed ASSIST program (asynchronous message passing).

the other publishes the outgoing event type and produces an event whenever receives a datum from the ASSIST subprogram.

## 4 Performance Evaluation

The presented methodology is a viable solution to parallel software integration into larger applications.

In order to demonstrate this, we present some experiments, targeted to measure the performance of invoking parallel algorithms from sequential code.

*Environment.* The following experiments were performed on a Blade cluster consisting of 23 Intel Pentium III Mobile CPU 800MHz computing elements, equipped with 1GB of RAM and interconnected by a switched Fast Ethernet dedicated network. The CORBA implementation adopted is TAO, a free, portable and efficient ORB written in C++, based on the ACE library [8, 9]. All the programs were compiled with gcc/g++ v.3 with full optimizations.

*Basic performance metrics for the mechanisms employed.* The first set of experiments are constructed to measure the maximum performance achievable by the current implementation of the two interaction methods, independently from the chosen algorithm. These performance metrics are compared with the ones obtained on the same platform using plain MPI and ASSIST equivalent programs. For the asynchronous interaction, we compared the bandwidth (varying the size of the message, see Fig. 5) of a pipeline of two processes implemented in

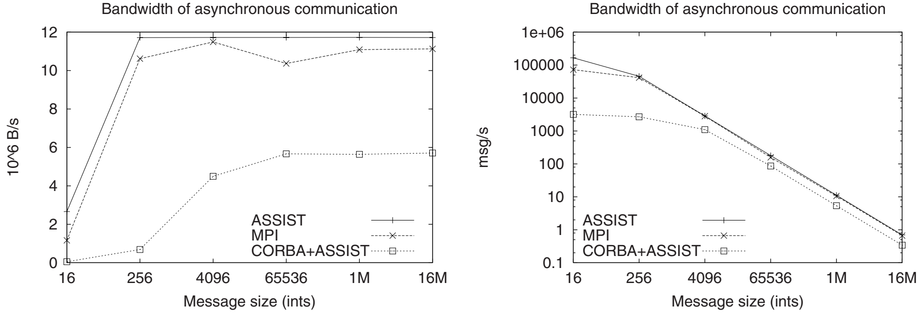



**Fig. 5.** Asynchronous communication bandwidth (left: $10^6$B/s, right: messages/s).

- MPI (plain MPI_Send/MPI_Receive),
- ASSIST (plain stream),
- C++ and ASSIST (CORBA event channels).

The first process produces the stream as fast as possible, while the second consumes it and computes the bandwidth. The communicating processes are mapped on different machines, in order to measure the bandwidth of a remote MPI/ASSIST/CORBA communication.

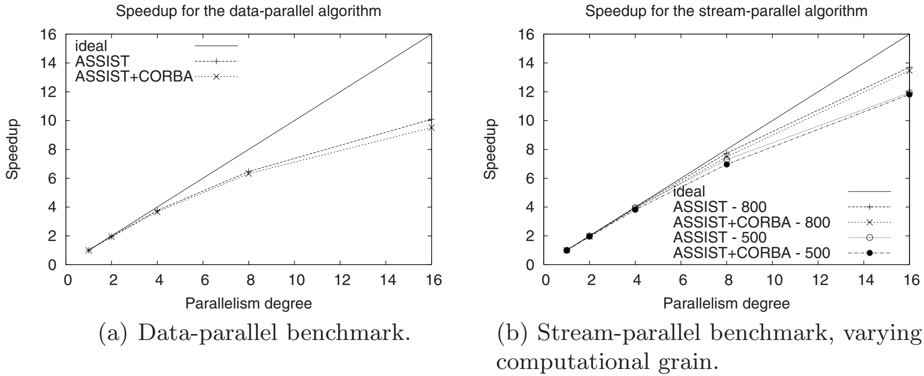
For the synchronous interaction, we measured the average round-trip time (see Table 1) of a remote service request, in which the server simply echoes the received message as a reply, varying the size of the request (and response). The server and the client are mapped onto different machines. We simulated this behaviour in MPI (by means of pairs of MPI_Send/MPI_Receives) and ASSIST (using two streams to send the request and the response). We compared the results to a CORBA+ASSIST version, in which the server is an ASSIST sequential module and the service, exported through RMI, is invoked by a C++ client.

*Service time of an exported CORBA object.* This experiment wants to show the impact of the invocation overhead introduced by CORBA on the performance of a data-parallel object. The service implements a synthetic data-parallel algorithm, operating on large matrices (the argument and result are $700 \times 700$ floats) and is written in ASSIST. The algorithm, given a matrix as input, transposes it in parallel (all to all communication) and then for every pair of rows $(a_i, b_j)$ computes $c_{ij} = \sum_k a_{ik} \times b_{jk} \times \sin(k)$.

The performance of the CORBA object is compared to an equivalent solution expressed in the ASSIST-CL language. In this solution the sequential program

**Table 1.** Round trip times for MPI, ASSIST, and CORBA RMI encapsulating ASSIST.

| Request/response size (ints) | MPI (s) | ASSIST (s) | CORBA+ASSIST (s) |
|---:|---:|---:|---:|
| 4096 | 0.00533 | 0.0045 | 0.0048 |
| 65536 | 0.053 | 0.0452 | 0.0587 |
| 1048576 | 0.762 | 0.7178 | 0.923 |
| 16777216 | 12.09 | 11.47 | 14.79 |




(a) Data-parallel benchmark.

(b) Stream-parallel benchmark, varying computational grain.

**Fig. 6.** Performance comparison: ASSIST vs. ASSIST+CORBA.

is encapsulated in an ASSIST sequential module and interacts with the parallel module implementing the algorithm by means of ASSIST streams[1].

The sequential version of the benchmark completes in 82.8s; a plain CORBA version in which the server is sequential (C++) completes in 83.1s, with an overhead of 0.3s. The overhead introduced by the program transformation is comparable (see table 2) to the one in the sequential case, and is in line with the raw overhead measured for the adopted mechanisms; this means that the implementation doesn't introduce inefficiencies, and that parallelism exploitation doesn't interfere with the transformation. So, if the overhead is negligible compared to the service time, as in this case, we obtain a good scalability, comparable with the one obtained with a pure ASSIST implementation (see Fig. 6(a)).

**Table 2.** Parallel service times for the data-parallel algorithm.

| Par. degree | ASSIST-CL program | ASSIST + CORBA | overhead |
|---|---|---|---|
| 2 | 42.2s | 42.7s | 0.5s (1.2%) |
| 4 | 22.0s | 22.6s | 0.6s (2.7%) |
| 8 | 12.8s | 13.1s | 0.3s (2.3%) |
| 16 | 8.2s | 8.7s | 0.5s (5.7%) |

*Speedup of stream-parallel algorithms.* The last experiment shows that stream-parallel programs with sufficient computation to communication ratio can gain high speedups. The stream-parallel algorithm chosen is the classical Mandelbrot set computation, parallelized as a task farm, in which each task is a set of contiguous points that must be evaluated; it produces a stream of computed lines (arrays of 500-800 values), with a rate that varies during the execution.

[1] This introduces overhead w.r.t. an ASSIST program in which the whole computation is carried in a parallel module, in fact this implies that the argument is scattered or gathered whenever a communication occurs between the sequential and the parallel module, but makes the performance comparable with the CORBA based solution.

Even in this case, the overhead introduced by the CORBA interaction vs. a pure ASSIST implementation is negligible (see Fig. 6(b)), allowing a good scalability up to 16 processors (the maximal configuration we tested).

## 5 Related Work

Several studies investigated how CORBA could be adapted to enable the construction of high-performance applications, focusing mainly on communication optimization with/between data-parallel programs.

PARDIS [10] project extended an existing CORBA ORB implementation, to allow both sequential and parallel clients to interact with sequential or parallel servers, where the processes composing a parallel unit can communicate via MPI message passing. They extended also the IDL language, in order to represent, by means of the new *distributed sequence type*, the possible data distribution policies often found in SPMD programs.

In PaCO [11] parallel (MPI) SPMD servers can be exported as a single parallel CORBA object and invoked by a sequential client; simple IDL extensions are introduced to support the classical BLOCK/CYCLIC data distributions.

GridCCM [12] extends the PaCO approach to CORBA components: SPMD MPI programs can be exported as CCM components, described by usual IDL3 interfaces and additional XML files describing the parallelism-oriented aspects.

Our approach can handle not only data-parallelism or SPMD programs, but also task-parallelism (e.g. pipeline / task farm) and mixed task- and data- parallelism, exploiting standard CORBA mechanisms such as object method invocation and event channels.

## 6 Conclusions

In this work we described the implementation of a compiling tool that automatizes the exportation of an ASSIST subprogram as a CORBA object. The ASSIST programming language eases this task, because it is based on a modular coordination model, that explicits all interactions between parallel components by means of interfaces. The translation of those interfaces to CORBA IDL notation and the construction of bridging ASSIST modules enables ASSIST subprograms to be integrated in larger applications using the CORBA standard as an interoperability layer. This simplifies the use of parallel algorithms within a sequential application and allows, as well, the composition of different parallel algorithms inside the same parallel application, configured at load- or run- time. The experiments showed that this is a viable solution to parallel software integration, in fact the two interaction methods allow the integration of low latency (data-parallel) or high-throughput (stream-parallel) parallel components with good performances, comparable to the ones that can be achieved in a parallel application written entirely in a high-level parallel programming language like ASSIST. These results makes ourselves confident that the integration between object- or component-based frameworks and high-performance computing is feasible, therefore we intend to extend the approach to generate bridges for CCM

components (introduced in the CORBA 3 standard), and to generalize it to handle other component-based standards (e.g. Web/GRID services [5]) as well.

## Acknowledgments

We wish to thank M. Vanneschi, M. Danelutto, M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, G. Giaccherini, A. Paternesi, A. Petrocelli, E. Pistoletti, L. Potiti, R. Ravazzolo, M. Torquati, P. Vitale.

## References

1. Vanneschi, M.: The programming model of ASSIST, an environment for parallel and distributed portable applications. Parallel Computing **28** (2002) 1709–1732
2. Aldinucci, M., Campa, S., Ciullo, P., Coppola, M., Magini, S., Pesciullesi, P., Potiti, L., Ravazzolo, R., Torquati, M., Vanneschi, M., Zoccolo, C.: The Implementation of ASSIST, an Environment for Parallel and Distributed Programming. In Kosch, H., László Böszörményi, Hellwagner, H., eds.: Euro-Par 2003: Parallel Processing. Number 2790 in LNCS (2003) 712–721
3. Aldinucci, M., Campa, S., Ciullo, P., Coppola, M., Danelutto, M., Pesciullesi, P., Ravazzolo, R., Torquati, M., Vanneschi, M., Zoccolo, C.: A Framework for Experimenting with Structured Parallel Programming Environment Design. In: ParCo 2003 Conference Proceedings, to appear, Dresden, Germany (2003)
4. Parashar, M., Laszewski, G., Verma, S., Gawor, J., Keahey, K., Rehn, N.: A CORBA Commodity Grid Kit. Concurrency Practice and Experience, special issue on GRID Computing Environments **14** (2002) 1057–1074
5. Foster, I., Kesselman, C., Nick, J.M., Tuecke, S.: Grid services for distributed system integration. Computer **35** (2002)
6. 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 Kosch, H., László Böszörményi, Hellwagner, H., eds.: Euro-Par 2003: Parallel Processing. Number 2790 in LNCS (2003) 1295–1300
7. Object Management Group: The Common Object Request Broker: Architecture and Specification. (2000) Minor revision 2.4.1, `http://www.omg.org`
8. 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)
9. Schmidt, D.C., Levine, D.L., Mungee, S.: The design of the TAO real-time object request broker. Computer Communications **21** (1998)
10. Keahey, K., Gannon, D.: PARDIS: A parallel approach to CORBA. In: Proceedings of 6th High Performance Distributed Computing, IEEE (1997) 31–39
11. Pérez, C., Priol, T., Ribes, A.: PaCO++: A parallel object model for high performance distributed systems. In: Distributed Object and Component-based Software Systems Minitrack in the Software Technology Track of HICSS-37, Big Island, Hawaii, USA, IEEE Computer Society Press (2004) To appear
12. Denis, A., Pérez, C., Priol, T., Ribes, A.: Bringing high performance to the CORBA component model. In: SIAM Conference on Parallel Processing for Scientific Computing. (2004) To appear