Integrating MPI-based numerical software into an advanced parallel computing environment*

Pasqua D'Ambra
Institute for High-Performance Computing and Networking (ICAR)
National Research Council (CNR)
Via Cintia - Monte S. Angelo, I-80126 Naples, Italy
dambra.p@cps.na.cnr.it

Marco Danelutto
Department of Computer Science
University of Pisa
Via Buonarroti 2, I-56127 Pisa, Italy
marcod@di.unipi.it

Daniela di Serafino
Department of Mathematics
Second University of Naples
Via Vivaldi, 43, I-81100 Caserta, Italy
daniela.diserafino@unina2.it

Marco Lapegna
Department of Mathematics and its Applications
University of Naples "Federico II"
Via Cintia - Monte S. Angelo, I-80126 Naples, Italy
marco.lapegna@dma.unina.it

Abstract

In this paper we present first experiences concerning the integration of MPI-based numerical software into an advanced programming environment for building parallel and distributed high-performance applications, which is under development in the context of Italian national research projects. Such programming environment, named ASSIST, is based on a combination of the concepts of structured parallel programming and component-based programming. Some activities within the projects are devoted to the definition, implementation and testing of a methodology for the integration of a parallel numerical library into ASSIST. The goal is providing a set of efficient, accurate and reliable tools that can be easily used as building blocks for high-performance scientific applications. We focus on the integration of existing and widely used MPI-based numerical library modules. To this aim, we propose a general approach to embed MPI computations into the ASSIST basic programming unit. This approach has been tested using the MPICH implementation of MPI for networks of workstations. Some modifications have been applied to the MPICH process startup procedure, in order to make it compliant with the ASSIST environment. Results of experiments concerning the integration of routines from a well-known FFT package are discussed.

1. Introduction

The development of scientific applications typically requires both a deep knowledge of the application domain and a proper use of sophisticated methods, techniques and tools from Numerical Mathematics and Computer Science. In such a context, numerical libraries play a fundamental role, since they provide computational scientists with many years of experience and know-how of numerical software developers. The emergence of a wide variety of parallel and distributed computer architectures increases the difficulties arising in the development of efficient and reliable software, and emphasizes the role of high-quality software modules in building computational applications. This role is going to be recognized also by the industrial world, where efforts to exploit parallel numerical libraries into existing codes have

*This work has been supported by the ASI-PQE2000 Programme *Development of Applications for Earth Observation with High-Performance Computing Systems and Tools*, and by the CNR Agenzia 2000 Programme *An Environment for the Development of Multi-platform and Multi-language High-Performance Applications based on the Object Model and on Structured Parallel Programming*.

Proceedings of the Eleventh Euromicro Conference on Parallel,Distributed and Network-Based Processing (Euro-PDP'03)
0-7695-1875-3/03 $17.00 © 2003 IEEE

been observed (see, for example, [1, 8]).

The complexity of current large-scale scientific applications often needs the combined use of multiple numerical software packages to address problems concerning Matrix Computations, Differential and Integral Equations, Function Transforms, Optimization, and so on. On the other hand, while accurate, efficient, and reliable libraries are available for the solution of single classes of problems, integrating and assembling them into large software systems is very difficult, because of data management and interoperability problems. New generations of advanced environments for parallel and distributed computing are evolving to take into account the requirements of today's applications and the evolution of the hardware and software technologies. One of the main design objectives of many advanced environments is the exploitation of the component technology to obtain software integration and interoperability, reuse of existing codes and seamless access to distributed and loosely coupled resources. Component standards and implementations, e.g. OMG CORBA [17], Microsoft DCOM [13], and Sun Enterprise JavaBeans [20], have been developed by the business world, that first recognized their importance. However, they do not support basic needs of high-performance computing, such as the abstractions needed by parallel programming and the performance. Therefore, a large effort is currently devoted to characterizing the component technology for high-performance computing [6]. In this scenario, work is carried out to wrap legacy codes and parallel software libraries as high-performance components [14, 16, 18].

In this context, a programming environment for building and running parallel and distributed applications is under development within Italian research projects, supported by the Italian Space Agency (ASI) and the Italian National Research Council (CNR). This environment, called *ASSIST* (A Software development System based upon Integrated Skeleton Technology), is aimed at exploiting the component programming model and the structured parallel programming model [22]. Some activities in the projects are devoted to the integration into ASSIST of parallel numerical library modules, in the areas of Linear Algebra, Fast Fourier Transforms and Quadrature. The final goal is not only providing the programming environment with a set of accurate, efficient and reliable numerical tools, but also defining an integration methodology that allows to reuse existing parallel numerical software with small or no changes, preserving as much as possible its performance, and exploiting static and dynamic optimization mechanisms of the ASSIST environment.

The remainder of this paper is organized as follows. In Section 2 we outline the main features of the ASSIST environment. In Section 3 we describe our approach for encapsulating MPI-based numerical kernels into an ASSIST parallel component, giving also details on its implementation. First experiences concerning the integration of routines from FFTW [11], a well-known package for FFT computations, are presented in Section 4. Concluding remarks and future work are reported in Section 5.

2. Main features of the ASSIST environment

The ASSIST project represents the evolution of research activities on structured parallel programming languages and related supports, carried out in the '90s by a group at the University of Pisa [2, 3]. The ASSIST programming environment merges the skeleton-based approach of early proposals with the emerging component-based approach. The target hardware layer includes different architectures, ranging from SMPs to MPPs, from homogeneous to heterogeneous clusters, till to computational Grids.

A layered software architecture has been designed (see Figure 1), exploiting object-oriented technology. The user interface is a coordination language, named *ASSIST-cl*, that allows "external" code to be used within the sequential portions of code encapsulated in skeletons. The ASSIST-cl code is compiled and then it is loaded and run onto the target architecture by the *Coordination Language Abstract Machine (CLAM)*. The CLAM layer is decoupled from the target hardware by a run-time support, named *Hardware Abstraction Layer Interface (HALI)*, which currently exports functionalities from the ACE multithreading and interprocess communication library [19], from the DVSA distributed virtual shared-memory library [4], and from a standard CORBA implementation [21], to allow external CORBA object usage within ASSIST applications. Furthermore, in the framework of the previously mentioned Italian projects, some activities are aimed at substituting suitable parts of the CLAM/HALI pair by Globus [10] components, in such a way that ASSIST-cl programs, currently running on workstation clusters, can run unmodified on a computational Grid.

The ASSIST-cl compiler parses ASSIST-cl source files and produces an intermediate "task code". In turn, the task code is translated into C++/HALI processes and threads. In addition, an XML configuration file is produced by the ASSIST-cl compiler, that can be used by CLAM to load and run the C++/HALI object code. The C++/HALI code is derived from task code by using a proper *builder* that uses pre-defined implementation templates to produce the actual code. The XML configuration file stores all the information needed to run the object code. In particular, the XML file stores names of the dynamically linked libraries containing the object code, dynamic code load paths, process-to-processing element mapping info, parallelism degree of the parallel components, type and number of processing elements available in the target machine, etc. This infor-

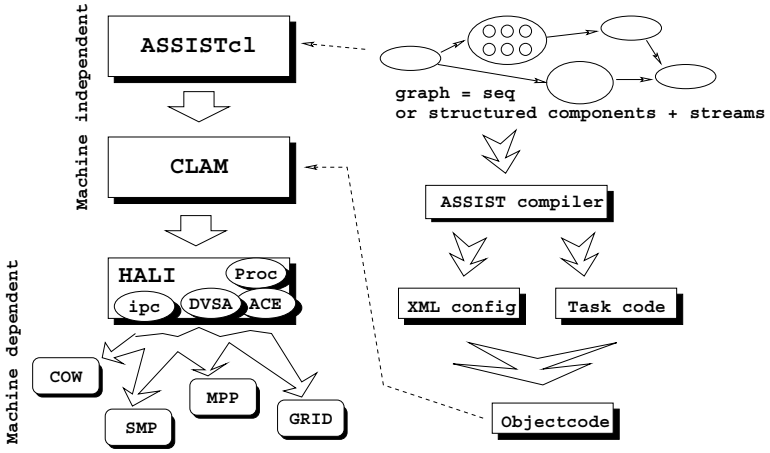

Figure 1. ASSIST architecture.

mation is used by CLAM to load proper object code, as well as to schedule the execution of object code components. Once the ASSIST-cl program has been compiled and the XML configuration file produced, the user may invoke the `assistrun` command, to make CLAM loading all the necessary code and running it, according to the XML configuration file. Furthemore, CLAM sets up all the required communication/synchronization channels (mainly via ACE TCP/IP socket wrappers) as well as distributed shared-memory regions (via proper DVSA calls), and prepares parameters needed by the different processes (process id's, handles for communication channels or shared-memory regions, etc.). Finally, CLAM takes care of handling program termination and cleanup. While executing these tasks, CLAM uses all the facilities provided by HALI and, in particular, by its ACE subsytem, making no assumptions on the existence of other software, running on the same nodes and competing to use the same resources. In other words, CLAM pretends to be *the* abstract machine using the target hardware.

The structure of an ASSIST program is a graph, where the nodes are the components and the edges are the component abstract interfaces, defined in terms of typed I/O streams. Two types of components have been designed: *sequential modules* and *parallel modules (parmods)*. The sequential module has an internal state and is activated by its input stream values, according to a deterministic data-flow behaviour. The parmod represents the basic ASSIST parallel component. A parmod is built out of different items: an *input/output section*, handling the parmod stream interface, a *virtual processor set*, possibly with a given topology, defining the internal parallel behaviour in terms of the acitivities that happen to be *logically* parallel in the parmod, and a *state*, holding the state variables that can be accessed by any virtual processor in the parmod. Furthermore, the parmod allows users to call "external" libraries in the code of the virtual processors; as an example, CORBA calls can be issued to access CORBA objects, or DVSA calls can be issued to handle intra-parmod shared data.

Data parallelism can be expressed by defining a topology of virtual processors, which can operate concurrently on partitioned or replicated datasets. Furthermore, the virtual processors can also be partitioned into disjoint topologies, assigned to distinct tasks/functions of a given task-parallel (or mixed task- and data-parallel) programming model. For each parmod, besides typed I/O interfaces, it is possible to define different policies of activations, i.e. deterministic data-flow or nondeterministic guarded.

In the syntax of ASSIST-cl, a parmod is completely specified by an *interface*, where the parmod name and the input and output streams are defined, a *declaration and setup section*, where the virtual processor topology is defined and the state variables are declared, an *input section* and an *output section*, used to manage the I/O streams and to distribute/collect them among/from virtual processors, and a *virtual processor section*, which describes the computation to be executed by the virtual processors and can include existing Fortran 77 or C code. Note that a composition of modules, expressed by a graph, may be reused as a component of a more complex ASSIST program, provided that the types of I/O streams are compatible. For more details on the ASSIST programming model we refer to [22].

3. Wrapping MPI-based numerical software as ASSIST parallel modules

We describe our methodology for embedding a parallel numerical library routine into an ASSIST *parmod*, that,

from now on, we refer to as *numerical parmod*. Main objectives are reusing library software with no or minimum changes, preserving as much as possible its performance, and, possibly, exploiting the static and dynamic optimization mechanisms provided by ASSIST, such as process allocation, scheduling and mapping, load balancing and so on.

The library contents have been selected taking into account typical computational kernels of scientific and engineering applications and needs of the applications proposed as a test bed for the ASSIST environment. This led to the choice of three areas: Linear Algebra, Fourier Transforms, and Numerical Integration. General criteria for selecting the software to be integrated were accuracy, reliability, performance portability, self-adaptivity, and use of "standard" programming tools. This led to the choice of routines from *ScaLAPACK* [5], a package for dense Linear Algebra computations, which has a modular structure and employs standardized and optimized (possibly self-optimized [23]) basic software, from *FFTW* [11], a self-adaptive package for the execution of FFTs, and from *PAMIHR* [7], a numerical integration software, developed by one of the authors and included in the NAG Parallel Library. All the above software is written in Fortran 77 and/or C, and uses, as communication environment, MPI or BLACS [9] (the latter, in turn, can be implemented on the top of MPI). Here we focus on the integration of routines that use directly MPI.

In our approach, a *numerical parmod* is conceived as a component that provides specific functionalities by means of a clear interface. The parallel routine code is embedded into the virtual processor section, while the I/O sections are used to compose the numerical parmod with other ASSIST modules. Input data coming on streams are distributed among the virtual processors according to the distribution policy required by the embedded routine; ouput data are collected and sent onto output streams in the output section. Sharing data with other ASSIST modules can also be realized through the ASSIST virtual shared-memory mechanism.

3.1. The process template of a numerical parmod and its implementation

Current ASSIST compiling tools process ASSIST source code to derive a process network that implements the source code and runs on the CLAM/HALI abstract machine. Each one of the parallel modules appearing into the source code is implemented by instantiating a parmod template that arranges processes, threads, communication channels, etc., in such a way that the virtual processors (i.e. the logical parallel entities) defined by the parmod are actually scheduled for parallel execution onto the available processing elements. The template used looks like the one depicted in Figure 2. It consists of $k + 2$ processes: the *Input Stream Manager (ISM)*, the *Output Stream Manager (OSM)* and k *Virtual Processor Managers (VPM)*. ISM and OSM essentially take care of the input and output activity of the parmod, respectively. The different VPMs take care of executing all the activities of the virtual processors. A single VPM is run onto each processing element participating to the execution of the parmod.

The numerical parmod exploits the above logical template, introducing all the changes necessary in order to allow virtual processors to execute MPI code. In particular, each VPM is in charge of executing the activities of a single virtual process, which performs MPI computations. However, current implementations of numerical parmods have not been obtained by instantiating any predefined template, but by writing directly the C++/HALI code needed to wrap into a parmod a given MPI-based numerical routine, and making this code available as a dinamically linked library. In developing an ASSIST program which uses numerical parmods, proper *pragmas* have to be used in the ASSIST-cl source code, in order to force the compiler to generate particular entries in the XML configuration file, that allow the numerical parmod code to be loaded and executed.

When CLAM is invoked to run the program including the numerical parmod, a CLAM master process generates slave processes on the processing elements used in the target architecture, according to the information stored in the XML configuration file. Eventually, those slave processes involved in the numerical parmod implementation load the numerical parmod process code. This code is run and parameters are passed that allow to use the communication channels set up by CLAM (i.e. those supporting the I/O streams of the module), to identify the process within the set of processes implementing the whole ASSIST-cl program, etc. Such behaviour is achieved because the parmod process code provides suitable `init` and `run` methods, that are invoked by CLAM immediately after loading, and because CLAM calls are used in the process code that transfer parameter values to the process space. CLAM is also in charge of running the proper ISM and OSM processes on the processing elements involved in the numerical parmod execution. ISM and OSM represent the interface that numerical parmod VPMs access, to get input data as well as to deliver results to the remaining part of the ASSIST-cl program.

In a numerical parmod, the VPMs are also the MPI processes that execute the computations of the encapsulated numerical routine, hence they must setup all the parameters related to the MPI implementation and execution environment (groups, contexts, communicators, various attributes, etc.), that are needed by MPI routines to accomplish their functionalities. In an MPI application, this setup phase is

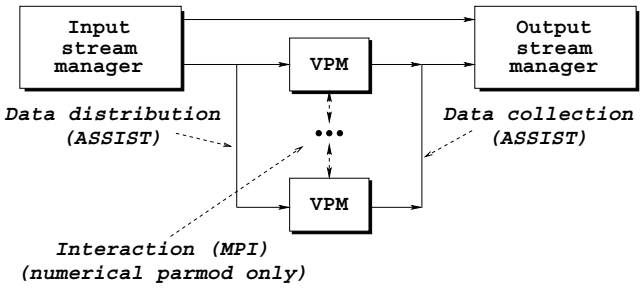

Figure 2. Process template of the parmod.

performed by the `MPI_Init` function[1]; the complementary one is `MPI_Finalize`, that cleans up all the MPI state.

In our experiments devoted to building numerical parmods, we referred to MPICH [15], a well known and widely used free implementation of MPI. More precisely, we considered the implementation for networks of Unix workstations. In this case, the `MPI_Init` function cannot be directly used by the VPMs of the numerical parmod, since its execution conflicts with the CLAM world, as explained below.

The architecture of MPICH consists of multiple layers. The higher layer, which provides the MPI APIs, is written in terms of a lower layer, called *Abstract Device Interface*, which essentially controls the data flow between the APIs and the hardware. A portable implementation of the ADI is built on the top of a further layer, named `Channel Interface`, which manages basic data exchange mechanisms. Multiple implementations of the Channel Interface, depending on the specific communication subsystem, are provided. The one for (possibly heterogeneous) networks of Unix workstations, named `ch_p4`, is based on the `p4` message-passing system, which is in turn implemented using TCP/IP sockets. (For more details on the MPI architecture see, for example, [12].) On a workstation network, usually, the processes running a parallel program cannot be directly started on a requested number of nodes. Therefore, the user, via the `mpirun` command, starts up a master MPI process, which in turn starts the other MPI processes (the slave ones) through the `MPI_Init` function. To do this, `MPI_Init` relies on `ch_p4`, which actually generates, via `rsh`, MPI slaves executing their parallel code, and pass them, as command line arguments, information such as the machine where the master is running on, a flag to distinguish between master and slaves, the executable pathname, the port to be used by TCP/IP sockets, and so on. Once started, the slave processes call `MPI_Init`, take the above information, hence recognize their slave role, and perform the setup phase of the MPI communication environment exploiting that information. The MPI master makes the setup too. The names of the hosts where the MPI processes must be started, their role (master or slave) and the executable names are in the so-called *procgroup* file, usually produced by the `mpirun` command.

In our case, the MPI processes are the ASSIST VPMs, that have already been started by CLAM. Therefore, they cannot use the previously described `MPI_Init` to initialize the MPI environment. Hence we made some modifications in the MPICH process startup procedure for network of Unix workstations, to make it compliant with the ASSIST environment. In our strategy, one of the VPMs is elected as MPI master process, but, instead of spawning MPI slaves, it sends to the other VPMs some information (MPI master machine and TCP/IP port) which the MPI setup phase can be started from. To this aim, some `ch_p4` functions, called inside `MPI_Init`, have been modified, to avoid Unix system calls devoted to slave process creation. Furthermore, `MPI_Init` has been modified in such a way that the MPI slaves use HALI communication channels to receive from the MPI master the above setup information. In this way, the ASSIST and the MPI world live on the same processes, without any conflict.

We note that the modifications to MPICH do not involve the APIs of MPI; therefore MPI-based code encapsulated into an ASSIST parmod does not undergo any change. We also note that the portability of the above ASSIST/MPI environment depends on the portability of ASSIST, currently available for networks of Unix workstations, and on the portability of the `ch_p4`-based MPICH version, also running on networks of Unix workstations. On the other hand, the main modifications to MPICH have been applied to the software layer that directly works on top of the Unix operating system, and concern the inhibition of process creation. Therefore, we think that similar modifications can be carried out on MPICH Channel Interface devices for different platforms. Finally, we observe that the approach used to implement numerical parmods can be exploted to encapsulate general MPI-based codes into ASSIST parmods. Future work could be devoted to develop templates for

[1] From now on, we use the C language APIs of MPI.

```
parmod assist_fftwnd_mpi(input_stream int N1; int N2; int details[2]; int startfft;
                         output_stream int endfft)
{
   topology array [i:NP] VP;
   use shared double precision Z[][];
}
```

Figure 3. ASSIST-cl interface of the 2D FFT parmod.

generic MPI-computations, that can be instantiated by ASSIST users.

4. First experiences

The methodology described in the previous section has been used to build a numerical parmod that encapsulates software from FFTW, a free package, written in C, that performs one-dimensional and multi-dimensional Fast Fourier Transforms (FFTs) on complex and real data, in either sequential or parallel computing environments [11]. FFTW has the capability of automatically adapting itself to the characteristics of the underlying computer architecture (memory hierarchy, processor pipeline, number of registers, etc.), thus having a portable high performance. The computation af an FFT is accomplished by an *executor* routine, which combines, according to the well known Cooley-and-Tukey algorithm, different specialized pieces of code, named *codelets*, that compute FFTs of fixed sizes. A *planner* routine, based on a dynamic programming algorithm, is called before the executor, to determine a composition of codelets that minimizes the execution time of the FFT. More details are given in [11].

FFTW includes routines executing parallel FFTs, for either SMPs or distributed-memory systems. The distributed-memory version is based on MPI. We now focus our attention on the tranform of a bivariate sequence of complex data values $Z = (z_{j_1 j_2})$:

$$\hat{z}_{k_1 k_2} = \sum_{j_1=0}^{m-1} \sum_{j_2=0}^{n-1} z_{j_1 j_2} \times e^{-2\pi i \left(\frac{j_1 k_1}{m} + \frac{j_2 k_2}{n} \right)},$$

where $i = \sqrt{-1}$, $j_1, k_1 = 0, \ldots, m-1$ and $j_2, k_2 = 0, \ldots, n-1$. The parallel algorithm implemented in the corresponding FFTW executor routine, named `fftwnd_mpi`[2], assumes that the matrix Z is distributed among a given set of p MPI processes in a row-block fashion, i.e. blocks of $m_{loc} \simeq m/p$ consecutive rows are distributed among these processes. Each process computes m_{loc} one-dimensional transforms of length n, on the rows assigned to it, contributes to the transposition of the global resulting matrix, computes $n_{loc} \simeq n/p$ one-dimensional transforms of length m, and, finally, contributes to the transposition of the global final matrix. All the communication is carried out in the transposition of the distributed matrices. The computation of FFTs is performed as indicated by the parallel two-dimensional FFT planner routine. To help users in dealing with the above data distributions, FFTW provides also an auxiliary routine, which returns parameters describing the required data layout.

The `fftwnd_mpi`, as well as the corresponding planner and auxiliary routines, have been embedded into the virtual processor section of a numerical parmod, using the methodology described in the previous section. The virtual processors are arranged into a linear array topology. The FFT sizes and data specifying some details (direct or inverse transform and information to planner) are taken in input via streams and hence are managed by the ISM process. The matrix to be transformed is assumed to be stored into a virtual shared-memory area, which must be initialised with proper data by some ASSIST module, before the numerical parmod is activated. This area is accessed by each VPM, that copies a block of rows into a local array, using the DVSA services provided by HALI. To start its computation, the parmod must receive, through an input stream, a flag indicating that the shared-memory area has been initialised. Analogously, a flag indicating that the FFT has been performed is sent by the parmod onto an output stream. The parmod interface, expressed in the ASSIST-cl syntax, is reported in Figure 3.

To evaluate the performance of the numerical parmod wrapping the FFT routine and the overhead due to wrapping, experiments have been carried out at ICAR-CNR (Naples section), using a Beowulf-class Linux cluster. This system has 19 nodes, each with a 1500 Mhz Pentium IV processor, a 256 KB L2 cache and a 512 MB RAM memory. The nodes are connected by a Fast Ethernet switch with a full-duplex bandwidth of 100 Mbit/sec. Each node is equipped with Linux Red Hat 7.2, with kernel 2.4.7, the GNU 2.96 version of the C++ compiler, and MPICH 1.2.3. A prototype version of the ASSIST environment has been

[2] Actually, this routine performs also higher-dimensional transforms.

$p = 1$	ASSIST setup	MPI setup	data distr./coll.	sh.-mem. get	FFT	sh.-mem. put	parmod
mean	3.52	0.04	0.24	4.42	3.06	4.42	15.70
stdev	0.06	0.02	0.07	0.08	0.14	0.09	0.16
min	3.44	0.02	0.37	4.30	2.89	4.28	15.30
max	3.63	0.07	0.34	4.51	3.25	4.50	16.30

$p = 2$	ASSIST setup	MPI setup	data distr./coll.	sh.-mem. get	FFT	sh.-mem. put	parmod
mean	3.73	0.06	0.35	2.70	7.21	2.73	16.77
stdev	0.07	0.01	0.06	0.20	0.16	0.23	0.49
min	3.64	0.03	0.25	2.36	7.00	2.35	15.63
max	3.83	0.09	0.42	2.87	7.47	2.93	17.61

$p = 4$	ASSIST setup	MPI setup	data distr./coll.	sh.-mem. get	FFT	sh.-mem. put	parmod
mean	6.42	0.06	0.39	1.91	6.30	1.89	16.95
stdev	0.12	0.02	0.04	0.07	0.31	0.03	0.19
min	6.23	0.03	0.35	1.84	5.95	1.84	16.24
max	6.58	0.09	0.43	1.99	6.79	1.92	17.80

$p = 6$	ASSIST setup	MPI setup	data distr./coll.	sh.-mem. get	FFT	sh.-mem. put	parmod
mean	8.07	0.21	0.61	1.42	5.76	1.55	17.59
stdev	1.29	0.08	0.19	0.01	0.85	0.09	1.32
min	5.99	0.08	0.27	1.41	4.56	1.47	13.78
max	11.18	0.31	1.09	1.44	6.97	1.65	22.64

$p = 8$	ASSIST setup	MPI setup	data distr./coll.	sh.-mem. get	FFT	sh.-mem. put	parmod
mean	9.26	0.38	0.65	1.18	5.65	1.23	18.34
stdev	0.20	0.02	0.02	0.07	0.07	0.07	0.29
min	8.95	0.34	0.60	1.12	5.50	1.16	17.67
max	9.49	0.41	0.70	1.30	5.75	1.31	18.96

Table 1. Mean, standard deviation, minimum and maximum of the execution times (seconds) of the FFT parmod, on p = 1, 2, 4, 6, 8 nodes hosting the VPMs, for m = n = 2000.

installed on the top of ACE 5.2.

The ISM, OSM and VPMs have been run on different nodes. Two more nodes have been used at each execution. One runs a parmod which generates the data to be sent, via streams, to the FFT parmod, sends them, and creates and inizialises a shared-memory area with the matrix to be transformed. The other runs a parmod which receives the termination flag from the FFT parmod and accesses the shared-memory area to check the correctness of the transformed data.

The experiments have been devoted to measuring the execution times of the FFT parmod and of its single tasks, i.e. the setup of the ASSIST communication system and the creation of communication channels, the setup of the MPI environment, the distribution and collection of data from ISM to VPMs and from VPMs to OSM, respectively, the copy of the input matrix from the virtual shared memory to VPM local arrays, the execution of the FFTW routines, and the copy of the transformed matrix from VPM local arrays to the virtual shared-memory.

In Table 1 we report the execution times, in seconds, obtained with $p = 1, 2, 4, 6, 8$ nodes hosting the VPMs, for the FFT sizes $m = n = 2000$. Five executions have been performed for each p, and mean, standard deviation, minimum and maximum of time values have been computed. The times have been measured by using the return value of the Unix system function `times`.

We see that the MPI setup requires a considerably smaller time than the other tasks. The MPI setup time increases as the number of nodes increases, but it generally accounts for no more than 2.2% of the total execution time. Instead, the time for copying data from/to the virtual shared memory has about the same order of magnitude of the time for performing the FFT, and it decreases as the number of nodes increases. On one node it is almost 300% of the FFT time and it is about 55% of the total execution time, according to the fact that the shared-memory area does not reside in the local memory of the node itself. On 8 nodes it is about 40% of the FFT time and it is less than 15% of the total time. A very large part of the execution time is

$p = 1$	ASSIST setup	MPI setup	data distr./coll.	sh.-mem. get	FFT	sh.-mem. put	parmod
mean	2.61	0.03	0.03	0.74	0.52	0.76	4.70
stdev	0.13	0.02	0.01	0.03	0.04	0.01	0.17
min	2.43	0.01	0.37	0.71	0.48	0.74	4.74
max	2.78	0.05	0.05	0.78	0.59	0.77	5.02

$p = 2$	ASSIST setup	MPI setup	data distr./coll.	sh.-mem. get	FFT	sh.-mem. put	parmod
mean	3.58	0.08	0.02	0.86	1.79	0.83	7.14
stdev	0.01	0.04	0.02	0.03	0.02	0.03	0.05
min	3.54	0.01	0.00	0.83	1.76	0.80	6.94
max	3.63	0.21	0.05	0.89	1.81	0.87	7.95

$p = 4$	ASSIST setup	MPI setup	data distr./coll.	sh.-mem. get	FFT	sh.-mem. put	parmod
mean	6.16	0.22	0.32	1.03	3.78	1.02	12.57
stdev	0.13	0.02	0.04	0.09	0.14	0.09	0.14
min	5.99	0.18	0.19	1.06	3.58	0.94	12.42
max	6.36	0.29	0.40	1.20	3.98	1.18	12.81

$p = 8$	ASSIST setup	MPI setup	data distr./coll.	sh.-mem. get	FFT	sh.-mem. put	parmod
mean	9.26	0.38	0.65	1.18	5.65	1.23	18.34
stdev	0.20	0.02	0.02	0.07	0.07	0.07	0.29
min	8.95	0.34	0.60	1.12	5.50	1.16	17.67
max	9.49	0.41	0.70	1.30	5.75	1.31	18.96

Table 2. Mean, standard deviation, minimum and maximum of the execution times (in seconds) of the parmod, on p = 1, 2, 4, 8 nodes hosting the VPMs, with a local size 1000 x 500.

spent in setting up the ASSIST communication environment and channels. This time grows with the number of nodes, varying from about 20%, on one node, to about 50%, on 8 nodes. This large time takes into account process synchronizations required by the ASSIST setup. However, this point deserves further investigation. Similar results have been obtained from experiments with $m = n = 1000$.

Finally, in order to analyze the scalability of our approach, we show the execution times obtained increasing the number of nodes hosting the VPMs, p, and keeping constant the size of the problem per node, i.e. the amount of matrix entries assigned to each node.[3] In Table 2 we report the results obtained on $p = 1, 2, 4, 8$ nodes with $1000 \times 500, 1000 \times 1000, 2000 \times 1000, 2000 \times 2000$ input matrices, respectively, in order to have 1000×500 matrix entries per node. From a comparison with Table 1, we see that the ASSIST and MPI setup do not depend on the problem size, but only on the number of nodes, as it was expected. Furthermore, the shared-memory access time obviously grows with the global size of the problem; however, for all values of p except $p = 1$, this time is a fraction of the FFT time, and this fraction decreases as p increases.

[3]We note that this choice does not correspond to a fixed number of floating-point operations per processor, because of the O($mn(\log m + \log n)$ FFT operation count.

5. Conclusions and future work

In this paper we described some research activities devoted to extend the ASSIST programming environment, based on a combination of the parallel structured programming and of the component programming models, with a toolkit of parallel numerical components for high-performance scientific computing.

MPI-based parallel numerical routines from the FFTW package have been embedded into a parmod, the basic unit proposed as the building block for developing parallel and distributed applications in the ASSIST environment. Our approach to perform such embedding required some modifications of the MPICH implementation of MPI for network of Unix workstations, in order to have ASSIST and MPI live together without any conflict. Esperiments have shown that the cost of setting up the MPI enviroment inside a parmod is negligible with respect to the whole parmod execution time. On the other hand, the time required by the entire wrapping is large, and further investigation is needed to reduce it. However, we believe that users can accept a reasonable loss in permormance to ease the composition of numerical

library modules with other ASSIST software units. Finally, we note that the proposed approach appears to be a general methodology enabling the reuse of MPI-based legacy code into ASSIST applications.

Further work will be done to develop templates, and related implementations, of ASSIST parmods embedding MPI/BLACS-based numerical routines for dense Linear Algebra.

References

[1] L. Arnone, P. D'Ambra, S. Filippone. A Parallel Version of KIVA-3 based on General-Purpose Numerical Software and its Use in Two-stroke Engine Applications. *International Journal of Computer Research, Special Issue on Industrial Applications of Parallel Computing*, 10:31-46, 2001.

[2] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, M. Vanneschi. P^3L: A Structured High Level Programming Language and its Restricted Support. *Concurrency: Practice & Experiences*, 7:225-255, 1995.

[3] B. Bacci, M. Danelutto, S. Pelagatti, M. Vanneschi. SkIE: A Heterogeneous Environment for HPC Applications. *Parallel Computing*, 25:1827-1852, 1999.

[4] F. Baiardi, D. Guerri, P. Mori, L. Moroni, L. Ricci. DVSA and SHOB: Support for Shared Data Structure on Distributed Memory Architecture. *Proc. of the 9^{th} Euromicro Workshop on Parallel and Distributed Processing*, 165-172, 2001.

[5] L.S. Blackford et al. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, 1997.

[6] Common Component Architecture Forum home page, http://www.cca-forum.org/.

[7] M. D'Apuzzo, M. Lapegna, A. Murli. Scalability and Load Balancing in Adaptive Algorithms for Multidimensional Integration. *Parallel Computing*, 23, 1997.

[8] I. de Bono, D. di Serafino, E. Ducloux. Using a General-Purpose Numerical Library to Parallelize an Industrial Application: Design of High-Performance Lasers. *Euro-Par'98 Parallel Processing, Lecture Notes in Computer Science*, 1470:812-820, 1998.

[9] J.J. Dongarra, R.C. Whaley. A User's Guide to the BLACS v1.1, *LAPACK Working Note 94, Tech. Rep. CS-95-283*, University of Tennessee, 1995 (updated: 1997). Available from http://www.netlib.org/lapack/lawns.

[10] I. Foster, C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Intl J. Supercomputer Applications*, 11(2):115-128, 1997. See also the Globus project home page, http://www.globus.org.

[11] M. Frigo, S.G. Johnson. FFTW: an Adaptive Software Architecture for the FFT, *Proceedings of 1998 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 3, 1998. See also FFTW home page, http://www.fftw.org.

[12] W. Gropp, E. Lusk, A. Skiellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. Available from http://www-unix.mcs.anl.gov/mpi/mpich/papers/mpicharticle/paper.html.

[13] M. Horsmann, M. Kirtland. DCOM Architecture, *Microsoft White Paper*, 1997. Available from http://www.microsoft.com/com/wpaper/.

[14] M. Li, O.F. Rana, D.W. Walker. Wrapping MPI-Based Lagacy Codes as Java/CORBA Components. *Future Generation Computer Systems*, 18(2):213-223, 2001.

[15] MPICH home page, http://www-unix.mcs.anl.gov/mpi/mpich/.

[16] B. Norris et al. Parallel Components for PDEs and Optimization: Some Issues and Experiences. *Parallel Computing, Special Issue on Advanced Environments for Parallel and Distributed Computing*, 2002. See also CCA@Argonne home page, http://www-unix.mcs.anl.gov/cca/.

[17] OMG, the CORBA home page, http://www.corba.org/.

[18] C. René, T. Priol. MPI Code Encapsulation Using Parallel CORBA Object. *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, 3-10, 1999.

[19] D.C Schmidt. The Adaptive Communication Environment: Object-Oriented Network Programming Components for Developing Client/Server Applications. *12th Sun Users Group Conference*, 1994. See also ACE home page, http://www.cs.wustl.edu/ schmidt/ACE.html.

[20] Sun Java home page, http://java.sun.com.

[21] D.C Schmidt. Evaluating Architectures for Multithreaded Object Request Brokers. *Communication of the ACM*, 41, 1998. Available from http://www.cs.wustl.edu/ schmidt/TAO.html.

[22] M. Vanneschi. The Programming Model of ASSIST, an Environment for Parallel and Distributed Portable Applications. *Parallel Computing, Special Issue on Advanced Environments for Parallel and Distributed Computing*, 2002.

[23] R.C Whaley, A. Petitet, J.J Dongarra. Automated Empirical Optimization of Software and the ATLAS Project, *LAPACK Working Note 147*, 2000. Available from http://www.netlib.org/lapack/lawns.