Parallel and Distributed Computing Practices
Volume 5, Number 3, pp. 279-288

ISSN 1097-2803

PARALLEL COMPONENTS FOR MULTIDIMENSIONAL QUADRATURE: SOME EXPERIENCES*

PASQUA D'AMBRA†, DANIELA DI SERAFINO‡, AND MARCO LAPEGNA§

Abstract. In this paper we present our experiences in wrapping a parallel multidimensional quadrature routine, based on the BLACS message-passing library, in order to obtain a software component computing multiple multidimensional integrals. The reference framework, where the component lives, is a programming environment called ASSIST, under development in the context of Italian research projects, which combines the component model with the structured parallel programming model. The approach described here not only allows to reuse the quadrature routine, preserving all its features, but also defines a general methodology for encapsulating into components existing parallel software that uses MPI or MPI-based implementations of BLACS, without changes to the software itself.

Key words. Parallel and Distributed Computing, Component Programming, BLACS/MPI-based Numerical Software, Multidimensional Quadrature.

AMS subject classifications. 65Y05, 65D30, 68N19, 68U99.

1. Introduction and motivations. Current high-performance simulations in Computational Science are typically multi-physics and multi-model and require the combined use of sophisticated tools from Numerical Mathematics and Computer Science. As a consequence, the software architecture for high-performance scientific computing is evolving toward environments where numerical software libraries, packages for data analysis and visualization, software development tools, etc., often distributed over a network, can interact easily and reliably to build large-scale applications [5]. In this context, paradigms that drive the design and development of numerical algorithms and software must be rethought in light of the new architectures and programming models. In particular, they must take into account the modern requirements of immediate interfacing and seamless interoperability with complementary and distributed software tools, and they must allow, at the same time, an easy reuse of existing software, in order to preserve years of research and development.

Answers to the above requirements can be found in the component-based programming model, that was developed in the business world [13, 20, 27], but has been only recently considered for high-performance, parallel and distributed, scientific computing [4, 17, 22]. In this context, a new trend in the production of high-performance numerical software is the definition and implementation of components encapsulating novel and existing computational kernels, in order to build numerical toolkits for parallel and distributed scientific computing, that can be used inside component frameworks obeying well-defined rules (see, for example, [1, 21]).

In this paper we discuss our experiences in the development of a *numerical component* encapsulating parallel multidimensional quadrature software. The computation of multidimensional integrals is, indeed, a basic kernel in several application fields, such as quantum chemistry, high-energy physics, finance, etc. As an example, the procedure described in [24], for molecular electronic structure calculations, involves the computation of multiple six-dimensional integrals of the form

$$I_{pqrs} = \int \chi_p^*(\mathbf{r})\chi_q(\mathbf{r})\frac{1}{\mathbf{r}-\mathbf{r}'}\chi_r^*(\mathbf{r}')\chi_s(\mathbf{r}')\mathbf{dr}\,\mathbf{dr}',$$

where the function $\chi_j(\mathbf{r})$ represents an atomic orbital and $\chi_j^*(\mathbf{r})$ denotes its complex conjugate. The number of integrals to be computed is $O(m^4)$, where m is the number of $\chi_j(\mathbf{r})$ functions, and a significant electronic structure calculation requires at least $m = 200$.

*This work has been supported 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.*

†National Research Council (CNR), Institute for High-Performance Computing and Networking (ICAR), via Cintia - Monte S. Angelo, I-80126 Naples, Italy. Email: pasqua.dambra@na.icar.cnr.it.

‡Department of Mathematics, Second University of Naples, via Vivaldi 43, I-81100 Caserta, Italy. Email: daniela.diserafino@unina2.it.

§Department of Mathematics and its Applications, University of Naples "Federico II", via Cintia - Monte S. Angelo, I-80126 Naples, Italy. Email: marco.lapegna@dma.unina.it.

Our reference framework is a programming environment for building parallel and distributed applications, called *ASSIST* (A Software development System based upon Integrated Skeleton Technology), which combines the component model with the structured parallel programming model [29]. A prototype version of ASSIST has been developed in the context of Italian research projects supported by the Italian Space Agency (ASI) and by the National Research Council (CNR). Our activities in the projects have been devoted to developing some numerical components, wrapping existing and widely used numerical library routines, to be used in this programming environment. First experiences in wrapping, into ASSIST components, MPI-based numerical routines written in the C language are reported in [6, 7]. The component described in this paper exploits a parallel Fortran routine using the BLACS communication library, in order to compute multiple multidimensional integrals.

We note that our goal was not only providing the programming environment with accurate, efficient and reliable numerical tools, but also defining, implementing and testing a methodology that allows to compose, and hence reusing, heterogeneous numerical library software to build scientific computing applications.

The remainder of this paper is organized as follows. A short description of the ASSIST programming environment is provided in Section 2, in order to outline the features that have been used in the development of the quadrature component. The quadrature routine encapsulated into the component is presented in Section 3, giving details about the algorithm used and its implementation. The methodology used to develop ASSIST-compliant numerical components is described in Section 4, with details on the specific implementation of the quadrature component. Results of experiments carried out running this component in the ASSIST framework, on a Beowulf-class Linux cluster, are discussed in Section 5. Projects related to our work are presented in Section 6 and concluding remarks are reported in Section 7.

2. A short description of the programming environment. ASSIST is an environment for the development of parallel and distributed applications, that merges the concepts of structured parallel programming and component-based programming [29]. Target hardware includes a wide range of architectures, from SMPs to MPPs to homogeneous and heterogeneous workstation clusters. Plans have been made to extend ASSIST in order to use it on Computational Grids. The version considered in this paper has been developed for workstation clusters.

An ASSIST program is structured as a graph, where the nodes are the components and the edges are the component interfaces, defined in terms of I/O streams of typed values. Streams are indeed the main mechanism for composing and interfacing ASSIST components. The basic parallel component is the so-called *parmod (parallel module)*, which allows to represent different forms of parallel computation. A parmod is defined by an *input/output section*, handling the parmod stream interface, a set of *virtual processors*, possibly with a given topology, that are the logical cooperating entities performing the parallel computation, and a *state*, holding the state variables that can be accessed by any virtual processor in the parmod. Furthermore, "external" objects, such as CORBA objects or virtual shared memory libraries, can be called in the code of the virtual processors.

The user interface of the ASSIST environment is a coordination language, named *ASSIST-CL*. In the syntax of this language, 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.

An ASSIST-CL code is compiled and then it is loaded and run onto the target architecture by a *Coordination Language Abstract Machine (CLAM)*. The CLAM is decoupled from the target hardware by a run-time support (RTS), named *Hardware Abstraction Layer Interface (HALI)*, which currently exports functionalities from the ACE multithreading and interprocess communication library [25], from the DVSA distributed virtual shared-memory library [2], and from a standard CORBA implementation [26], for using external CORBA objects within ASSIST applications. The ASSIST compiler translates the ASSIST-CL source code of a parmod into C++/HALI processes,

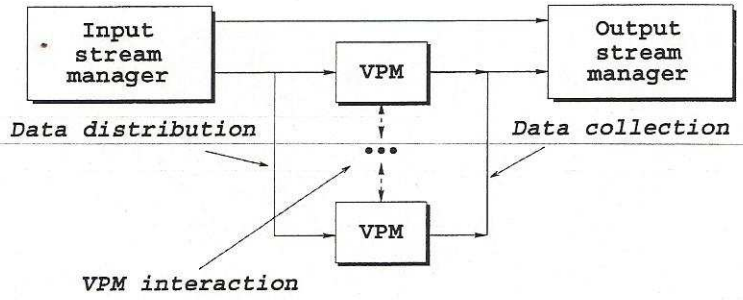

FIG. 2.1. *The process template of a parmod.*

using a pre-defined implementation template, shown in Figure 2.1. In this template, an *Input Stream Manager (ISM)* and an *Output Stream Manager (OSM)* perform the input and the output activity of the parmod, respectively, while k *Virtual Process Managers (VPMs)* execute the activities of the virtual processors. A single VPM is run onto each processing element participating to the parmod execution. An XML configuration file, produced by the ASSIST compiler, contains all the information needed to load and run the C++/HALI code. More details on ASSIST can be found in [6, 29].

3. The multidimensional quadrature routine. The component described in this paper is based on a multidimensional quadrature routine, that computes an approximation of the integral

$$I(f) = \int_U f(t_1, ..., t_n)\, dt_1 \cdots dt_n, \tag{3.1}$$

where $U = [a_1, b_1] \times \cdots \times [a_n, b_n]$ is a n-dimensional hyper-rectangular region. This routine implements a parallel version of a *global adaptive algorithm*, targeted at MIMD distributed-memory architectures. We note that, for problem (3.1), adaptive algorithms are known as good procedures able to achieve high accuracy with a reasonable computational cost, both in sequential and in parallel environments [15].

The global adaptive algorithm is based on an iterative procedure that, at the generic j-th iteration, evaluates a composite quadrature rule $Q^{(j)}$ and an absolute error estimate $E^{(j)}$, such that

$$\lim_{j\to\infty} Q^{(j)} = I(f), \quad \lim_{j\to\infty} E^{(j)} = 0.$$

The quadrature rule $Q^{(j)}$ is computed on a partition

$$\mathcal{S}^{(j)} = \left\{ s_k^{(j)},\ k = 1, \ldots, K^{(j)} \right\}$$

of the domain U, and the error $E^{(j)}$ is computed as

$$E^{(j)} = \sum_{k=1}^{K^{(j)}} e_k^{(j)},$$

where $e_k^{(j)}$ is the absolute error estimate in the subdomain $s_k^{(j)}$. Since the convergence rate of the sequence of rules depends on the behaviour of the integrand function (presence of peaks, oscillations, etc.), in order to reduce the error the integral estimate $Q^{(j)}$ is computed from $Q^{(j-1)}$ by splitting into two parts the subdomain $\hat{s}^{(j-1)} \in \mathcal{S}^{(j-1)}$ with the maximum error estimate, $\hat{e}^{(j-1)}$. The algorithm terminates when the error satisfies a user required tolerance, e.g. $E^{(j)} < \varepsilon$, or it is found that this error criterion cannot be satisfied within an allowed bound on the number of integrand function evaluations.

A straightforward parallelization of the previous algorithms can be obtained by partitioning among the processors, at the beginning, the integration domain U, and then, at each iteration, the subdomain with the maximum error estimate. However, this strategy has two main drawbacks. First, the sequence $\{\mathcal{S}^{(j)}\}$ of domain partitions is unpredictable, so the workload cannot be uniformly distributed *a priori* among the processors. Second, the cost of global communications, needed at least to identify the subdomain holding the maximum error estimate and to compute the global error, makes the algorithm poorly scalable [15].

A scalable version of the previous algorithm, with good load-balancing properties, can be obtained with a hybrid approach, i.e. performing the global quadrature algorithm in each one of the domain partitions assigned to the processors, and using a termination criteria based on a local error estimate along with a mechanism that "quickly" distributes among all processors, via nearest-neighbour communications, the subpartitions with largest error estimates.

Let us consider p processors, logically arranged into a periodical bidimensional mesh, and denote by 0 and 1 the horizontal and the vertical direction of this mesh, respectively. Furthermore, let

$$\mathcal{S}_i^{(j)} = \left\{ s_{i,k}^{(j)},\ k = 1, \ldots, K_i^{(j)} \right\} \text{ for i=0,..., p-1}$$

be the subpartition of $\mathcal{S}^{(j)}$ made by the subdomains that reside in the local memory of the processor P_i, at the j-th iteration.

At the beginning of the algorithm ($j = 0$), the integration domain U is partitioned into p subdomains $s_{i,1}^{(0)}$ $(i = 0, \ldots, p-1)$ of the same size, and each subdomain is assigned to the processor P_i, which therefore holds the subpartition

$$\mathcal{S}_i^{(0)} = \left\{ s_{i,1}^{(0)} \right\}.$$

P_i computes the quadrature rule $Q_i^{(0)}$, approximating the restriction of $I(f)$ to $s_{i,1}^{(0)}$, and the corresponding absolute error $E_i^{(0)}$. At the generic iteration j, with $j > 0$, each processor P_i selects its subdomain with the largest error estimate, divides it into two parts and updates $Q_i^{(j)}$ and $E_i^{(j)}$. Then P_i identifies, in its new sub partition, the subdomain $\hat{s}_i^{(j)}$ with the largest error estimate,

$$\hat{e}_i^{(j)} = \max_{k=1,\ldots,K_i^{(j)}} e_{i,k}^{(j)},$$

and attempts to send this subdomain to the processor P_{i+}^{dir} which follows P_i in the direction $dir = mod(j, 2)$. The subdomain $\hat{s}_i^{(j)}$ is sent if

$$\hat{e}_i^{(j)} > \hat{e}_{i+}^{(j)},$$

where $\hat{e}_{i+}^{(j)}$ is the largest error held by P_{i+}^{dir}. A local error criterion is used in each subpartition $\mathcal{S}_i^{(j)}$ to stop the iterative procedure, with a tolerance $\varepsilon_i^{(j)}$ obtained by scaling a global input tolerance with respect to the size of the subdomain defined by the local subpartition. A description of the parallel algorithm, using the *Single Program Multiple Data (SPMD)* paradigm, is given in Figure 3.1.

The redistribution strategy used in the algorithm is such that the domains containing computational "difficulties" are dynamically partitioned among the processors in a time proportional to the diameter of the processor mesh, with a communication cost which is independent of the number of processors, since communications involve only nearest neighbours. Therefore, the algorithm is able to achieve good load balancing and scalability [8]. Furthermore, a small amount of data has to be communicated between two processors at each iteration, i.e. at most $2n + 3$ scalar values, including the extrema of the intervals defining the n-dimensional subdomain with the maximum error estimate, the corresponding approximations of the integral and the error, and some additional information.

begin *(on processor P_i)*

- *determine the processors $P_{i-}^{(dir)}$ and $P_{i+}^{(dir)}$, $dir = 0, 1$*
- *set $j = 0$ and initialize $Q_i^{(j)}$, $E_i^{(j)}$, $\varepsilon_i^{(j)}$*
- *compute the global values $Q^{(j)}$ and $E^{(j)}$*

while $E_i^{(j)} > \varepsilon_i^{(j)}$ **do**

- *divide the subdomain with the largest error into two subdomains $s_{i,\lambda}^{(j)}$ and $s_{i,\mu}^{(j)}$*
- *compute, in these subdomains, the integrals $q_{i,\lambda}^{(j)}$ and $q_{i,\mu}^{(j)}$ and the errors $e_{i,\lambda}^{(j)}$ and $e_{i,\mu}^{(j)}$*
- *update $Q_i^{(j)}$, $E_i^{(j)}$, $\varepsilon_i^{(j)}$*
- *identify the current subdomain, $\hat{s}_i^{(j)}$, with the largest error estimate, $\hat{e}_i^{(j)}$*
- *compute $dir = mod(j, 2)$*

- **send** $\hat{e}_i^{(j)}$ **to** $P_{i+}^{(dir)}$ *and* $P_{i-}^{(dir)}$

- **receive** $\hat{e}_{i-}^{(j)}$ **from** $P_{i-}^{(dir)}$ *and* $\hat{e}_{i+}^{(j)}$ **from** $P_{i+}^{(dir)}$

if $\hat{e}_i^{(j)} > \hat{e}_{i+}^{(j)}$

- **send** $\hat{s}_i^{(j)}$ **to** $P_{i+}^{(dir)}$
- *update $Q_i^{(j)}$, $E_i^{(j)}$, $\varepsilon_i^{(j)}$*

endif

if $\hat{e}_{i-}^{(j)} > \hat{e}_i^{(j)}$

- **receive** $\hat{s}_{i-}^{(j)}$ **from** $P_{i-}^{(dir)}$
- *update $Q_i^{(j)}$, $E_i^{(j)}$, $\varepsilon_i^{(j)}$*

endif

endwhile

- *compute the global values $Q(f)$ and $E(f)$*

end

FIG. 3.1. *The parallel adaptive quadrature algorithm.*

The previous algorithm has been implemented in a Fortran 77 double precision routine called `PAMIHR` [16] using the BLACS library [9] to perform message passing. It has also been included in the release 2 of the NAG Parallel Library with the name `D01FAFP` [19]. We note that BLACS has been originally developed to manage Linear Algebra oriented communications, but it is also widely used in parallel numerical software libraries providing different computational kernels. An implementation of BLACS on the top of the MPI communication environment is available at the URL `http://www.netlib.org/ blacs/index.html`. A Genz-and-Malik quadrature rule of degree 9 [10] and the error estimate in [3] have been applied for the computation of integrals and related errors. A more detailed description of the algorithm and its implementation is given in [8].

4. A software component for multidimensional quadrature. We now describe our work to wrap into an ASSIST component the quadrature routine described in the previous section. This work actually identifies a general approach for encapsulating numerical software, using MPI [12] or

an MPI-based version of BLACS [9], into an ASSIST parallel module. This module, that we call *numerical parmod*, provides ASSIST applications with the functionalities of the embedded numerical kernel, by means of a suitable interface. Main guidelines in our work have been reusing numerical software with no or minimum changes, preserving as much as possible the original performance characteristics of the software, and, possibly, exploiting the static and dynamic optimization mechanisms provided by the ASSIST RTS, such as process allocation, scheduling and mapping, load balancing and so on.

In the ASSIST environment, a parallel component is implemented by instantiating the predefined template outlined in Section 2. Following that template, we use ISM process for receiving streams of input data and handling their distribution to the numerical routine, the VPMs for running the numerical routine code, and OSM process for collecting the output data and sending them, as streams, to other ASSIST components. However, we did not instantiate any pre-defined template and all the (glue) code needed to wrap the numerical routine was written by hand. This wrapper, in C++ with calls to HALI functions, is responsible for communications between the CLAM and the numerical routine and between the numerical routine and any other ASSIST component, through HALI communication channels. The wrapper is also responsible of initializing the "native" message-passing library of the encapsulated numerical routine, in such a way that the routine can use all the functionalities provided by that library.

In our work we used MPICH [18], a well known free implementation of MPI[1], and the MPI-based version of BLACS mentioned in Section 3. Changes have been introduced in the RTS of MPICH to enable the VPMs to set the parameters related to the MPI implementation and execution environment (groups, contexts, communicators, various attributes, etc.) and hence to use all the MPI functionalities within a "communication world" which is well separated from the ASSIST world. In this way, ASSIST and MPI can live in the same process without conflicting each other. Note that the changes to MPICH do not involve the MPI standard APIs. For details on the modifications to MPICH, the reader is referred to [6]. The MPI communicator, identifying the above MPI world, has been "translated" into a BLACS context by a simple call to a BLACS routine in the wrapper code, thus allowing to initialize the BLACS environment and to call any routine using the BLACS library inside the VPM code.

The numerical component (wrapper plus routine code) has been made available to the ASSIST environment as a dynamically linked library. In the development of an ASSIST program which uses this component, proper *pragmas* must be put in the ASSIST-CL source code, in order to force the compiler to generate suitable entries in the XML configuration file, that allow this library to be loaded and executed. When the CLAM is invoked to run a program including the numerical component, a CLAM master process generates slave processes on the processors of the target architecture, and, according to the information stored in the configuration file, some CLAM slave processes load and run the code of the numerical component itself. Interfacing between the CLAM and the numerical component is achieved by means of suitable functions in the numerical parmod code, which provide entry points to CLAM processes.

We note that a result of our work is the integration of the RTS of MPICH into the ASSIST environment, in such a way that MPI-based computations can be encapsulated into ASSIST components and hence scheduled, mapped and executed as CLAM processes. Communications of these components with other ASSIST components are performed via the ASSIST RTS, while inter-component communications are based on the native communication layer of the legacy code.

4.1. Implementation details. The wrapper to the quadrature routine has been written in C++, using library calls from the ASSIST HALI. This wrapper instantiates suitable HALI classes for managing process communications in the ASSIST environment. In particular, ASSIST communication channels are created that allow ISM to receive, from other components, the input data required by the quadrature routine, and to distribute them to VPMs; channels are also created to allow VPMs to send quadrature output data to OSM, and OSM to send output data to other components. As already observed, the wrapper is also in charge of initializing BLACS and the MPI RTS, and of cleaning up the BLACS and MPI states.

[1] More precisely, we considered the MPICH 1.2.3 implementation for networks of Unix workstations.

Taking into account that many applications require the computation of multiple integrals (see, for example, Section 1) and that the cost of creating ASSIST communication channels cannot be neglected (see the performance analyses of the numerical parmods in [6, 7]), a numerical component has been developed which allows users to compute an arbitrary number of multidimensional integrals, with the same instance of the component itself. More precisely, the quadrature component receives, through an input stream, the number of integrals to be computed; then it sets up a cycle where, at each step, the input data concerning a single integral are received via input streams, the integral is computed and the corresponding output data are sent out onto output streams, until all the integrals have been computed. Two input streams have been implemented, one for integer data (the number of dimensions of the integral and the maximum number of function evaluations) and the other for double precision data (the extrema of the intervals that define the integration domain, and the error tolerance). Two more streams have been implemented, for integer output data (the actual number of function evaluations and an error flag), and for double precision ones (the integral and error approximations). Currently, the computation of integrand function values, required by the quadrature routine, is performed by using an external C function, to be linked to the numerical parmod code.

5. Results. Experiments have been carried out to evaluate the performance of the whole quadrature component and the overhead due to wrapping. The following family of functions has been considered for integration in the unit n-dimensional cube:

$$f(t_1, t_2, \ldots, t_n) = \prod_{i=1}^{n} (\alpha_i^{-2} + (t_i - \beta_i)^2)^{-1},$$

where the parameters α_i and β_i have been chosen randomly, according to the Genz testing strategy [11]. The number of dimensions, n, has been set equal to 8.

The tests have been performed using a Beowulf-class Linux cluster, available at ICAR-CNR (Naples branch). 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 (kernel 2.4.7), with the GNU 2.96 C++ compiler, and with MPICH 1.2.3 and BLACS 1.1. A prototype version of the ASSIST environment has been 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 running a parmod which generates and sends input data to the quadrature parmod, and the other running a parmod which receives output data from the quadrature parmod.

Figure 5.1 shows the execution times, in seconds, required by a single instance of the quadrature component to compute $m = 1, 10, 100$ integrals, on $p = 1, 2, 4, 8$ nodes hosting the VPMs. The times of the single tasks performed by the component are also reported, i.e. the times required by the setup of the ASSIST communication channels, the setup of the MPI environment, the data distribution (from ISM to VPMs) and collection (from VPMs to OSM), and the execution of the quadrature routine `PAMIHR`. All these times have been computed by using the return value of the Unix system function `times`. Figure 5.2 shows the percentage of time spent by each single task for the previous values of m and p. We note that the size of the test problem has been scaled with p, to keep constant the workload per node. More precisely, a user tolerance $\varepsilon = 10^{-8}$ has been chosen for $p = 1$, obtaining, for each function f, a number h_f of function evaluations; for $p > 1$, a p-times larger problem has been obtained by setting $\varepsilon = 0$ and a maximum number of function evaluations equal to $p \cdot h_f$ [7].

As already noted in [6, 7], a large time is spent in setting up the ASSIST communication channels. This time increases as the number of nodes increases, but it does not depend on the number of integrals, i.e. on the workload of the computational task. While for $m = 1$ the ASSIST setup accounts for more than 90% of the total execution time, for $m = 100$ it decreases to about 10% with $p = 1$ and to less than 25% with $p = 8$. A further percentage reduction can be observed with a larger number of integrals. On the other hand, the time required to initialize the BLACS/MPI environment is negligible. In the "worst case", i.e. for $m = 1$ and $p = 8$, it is about 1% of the total execution time and its relative cost significantly decreases as the number of integrals increases.

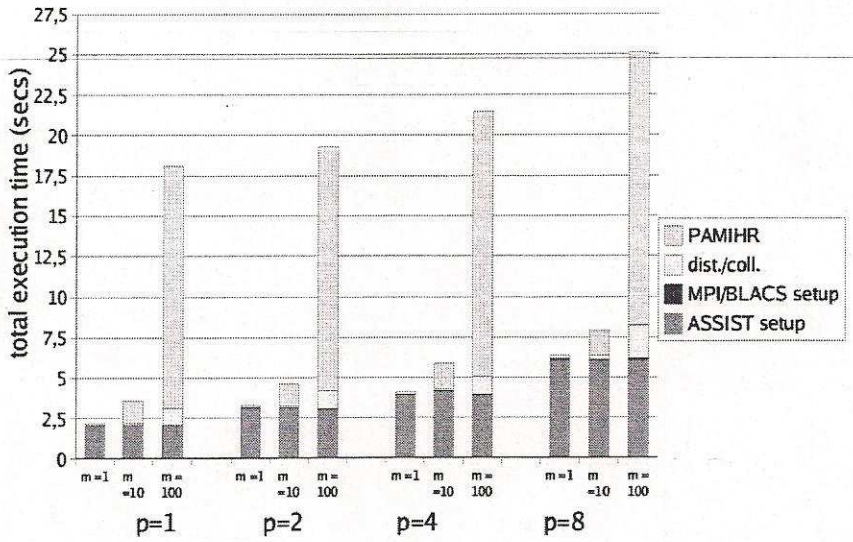

FIG. 5.1. *Execution times (in seconds) of the quadrature component and its single tasks, with* $m = 1, 10, 100$ *integrals and* $p = 1, 2, 4, 8$ *nodes hosting the VPMs.*

Finally, the time for data distribution and collection grows with the number of integrals, since I/O data must be transferred for each integral, but the cost of this task is not really significant with respect to the computational kernel.

6. Related work. Our work can be placed in the line of some recent proposals of using the component-based programming model for scalable, parallel and distributed scientific computing, in order to overcome problems concerning interface definition, composition, interoperability and reuse of heterogeneous software resources [4, 14, 22, 30]. In this context, work has been devoted to supporting the SPMD programming model, which is widely used to develop numerical software on distributed-memory parallel computers, in the context of component architectures.

An example is given by the *MDS-PSE* project, carried out at Cardiff University and targeted at molecular dynamics simulations (MDS) [30]. In this project, SPMD MPI-based legacy code has been wrapped as one or more CORBA objects, with no extensions to the OMGs CORBA specification and IDL compiler. A combination of the MPI RTS with the CORBA environment has been realized, that allows to use MPI to manage intra-communications of parallel components, and the CORBA ORB to manage communications among different components [17]. When invoked from a client, the parallel CORBA object, through its wrapper, initiates two processes: one starting the MPI RTS and starting the MPI-based computation on a cluster of computational nodes, and the other waiting for output of the MPI code and generating a callback to the client for displaying the results. We note similarities with our approach, where functionalities are added to the ASSIST RTS in order to support MPI-based SPMD parallel computations without modifying the ASSIST component model and the ASSIST-CL compiler.

A different approach to encapsulation of MPI-based parallel codes in CORBA objects has been proposed in [23]. In this case, the CORBA object model has been extended by introducing the concept of distributed object and IDL distributed data structures, following the approach proposed

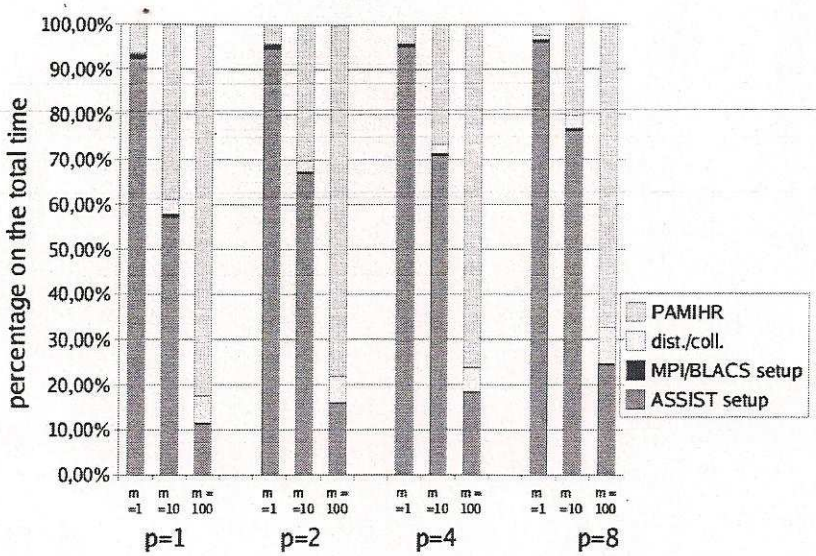

FIG. 5.2. *Percentage of execution time spent in each task, with* $m = 1, 10, 100$ *integrals and* $p = 1, 2, 4, 8$ *nodes hosting the VPMs.*

in [14].

Finally, a large effort in the context of the *Common Component Architecture (CCA) Forum* [4] is devoted to defining a widely-accepted standard of component architecture oriented towards high-performance scientific computing. In this context, work is carried out to developing CCA-compliant frameworks for building SPMD parallel applications [1] and for defining and implementing CCA-compliant interfaces for existing and widely used numerical software [21].

7. Concluding remarks. A general approach for wrapping BLACS/MPI-based software into ASSIST components has been proposed and implemented, obtaining a parallel component for multi-dimensional quadrature, which provides all the functionalities of the encapsulated numerical routine. The overhead introduced by the wrapping strategy is balanced not only by the encapsulation of time-consuming computational kernels, but also by the possibility of reusing parallel legacy software and of easily composing it with heterogeneous software modules.

Further investigation with ASSIST developers is needed, in order to define suitable numerical parmod templates and related ASSIST-CL APIs, in order to automate the wrapping of existing and novel numerical kernels.

REFERENCES

[1] B.A. ALLAN ET AL., *The CCA Core Specification in a Distributed Memory SPMD Framework*, Concurrency and Computation - Practice & Experience, 14 (2002), pp. 323-345.

[2] F. BAIARDI, D. GUERRI, P. MORI, L. MORONI, L. RICCI, *DVSA and SHOB: Support for Shared Data Structure on Distributed Memory Architecture*, in Proc. of the 9^{th} Euromicro Workshop on Parallel and Distributed Processing, pp. 165-172, 2001.

[3] J. BERNTSEN, *Practical Error Estimation in Adaptive Multidimensional Quadrature Routines*, Journal of Computational and Applied Mathematics, 25 (1989), pp. 327-340.

[4] COMMON COMPONENT ARCHITECTURE FORUM home page, http://www.cca-forum.org/.

[5] P. D'Ambra, M. Danelutto, D. di Serafino, M. Lapegna, *Advanced Environments for Parallel and Distributed Applications: a View of Current Status*, Parallel Computing, 28 (2002), pp. 1635-1662.
[6] P. D'Ambra, M. Danelutto, D. di Serafino, M. Lapegna, *Integrating MPI-based Numerical Software into an Advanced Parallel Computing Environment*, in Proc. of the 11^{th} Euromicro Conference on Parallel, Distributed and Network based Processing, IEEE pub., pp. 283-291, 2003.
[7] P. D'Ambra, D. di Serafino, M. Lapegna, *Embedding Parallel Quadrature Software into a HPC Environment*, in Proc. of Parallel Numerics '02, Theory and Applications, R. Trobec, P. Zinterhof, M. Vajtersic, A. Uhl, eds., University of Salzburg, Austria, and Jozef Stefan Institute, Slovenia, pub., pp. 15-27, 2002.
[8] M. D'Apuzzo, M. Lapegna and A. Murli, *Scalability and Load Balancing in Adaptive Algorithms for Multidimensional Integration*, Parallel Computing, 23 (1997), pp. 1199-1210.
[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] A. Genz and A. Malik, *An Embedded Family of Fully Symmetric Numerical Integration Rules*, SIAM Journal on Numerical Analysis, 20 (1983), pp. 580-588.
[11] A. Genz, *Testing Multiple Integration Software*, in Tools, Methods and Languages for Scientific and Engineering Computation, B. Ford, J.C. Rault and F. Thommaset, eds., North Holland, New York, 1984.
[12] W. Gropp, E. Lusk and A. Skiellum, *Using MPI - Portable Parallel Programming with the Message Passing Interface*, second edition, MIT Press, 1999.
[13] M. Horsmann, M. Kirtland, *DCOM Architecture*, Microsoft White Paper (1997). Available from http://www.microsoft.com/com/wpaper/.
[14] K. Keahey, D. Gannon, *PARDIS: a CORBA-based Architecture for Application-Level Parallel Distributed Computation*, in Proceedings of Supercomputing '97, 1997, available at http://www.supercomp.org/sc97/proceedings/TECH/KEAHEY/INDEX.HTM.
[15] A. Krommer, C. Ueberhuber, *Numerical Integration on Advanced Computer Systems*, in Lecture Notes in Computer Science, 848, Springer-Verlag pub., 1994.
[16] G. Laccetti, M. Lapegna, *PAMIHR: A Parallel Fortran Program for Multidimensional Quadrature on Distributed Memory Architectures*, in Proceedings of EUROPAR99, P. Amestoy et al. eds., Lecture Notes in Computer Science, 1685, Springer-Verlag pub., 1999, pp 1144-1148.
[17] M. Li, O.F. Rana, D.W. Walker, *Wrapping MPI-Based Legacy Codes as Java/CORBA Components*, Future Generation Computer Systems, 18 (2001), pp. 213-223.
[18] MPICH home page, http://www-unix.mcs.anl.gov/mpi/mpich/.
[19] Numerical Algorithms Group *The Numerical Parallel Library - release 2*, NAG Ltd., Oxford, 1997.
[20] Object Management Group, Common Object Request Broker Architecture (CORBA/IIOP), revision 3, OMG specification document (2002). See also the CORBA home page, http://www.corba.org.
[21] B. Norris et al., *Parallel Components for PDEs and Optimization: Some Issues and Experiences*, Parallel Computing, 28 (2002), pp. 1811-1831. See also CCA@Argonne home page, http://www-unix.mcs.anl.gov/cca/.
[22] C. René, T. Priol, G. Alléon, *Code Coupling Using Parallel CORBA Objects*, in Proc. of the IFIP Working Conference on Software Architectures for Scientific Computing, Kluwer Pub., pp. 105-118, 2000.
[23] C. René, T. Priol, *MPI Code Encapsulation Using Parallel CORBA Object*, Cluster Computing, 3 (2000), pp. 255-263.
[24] A.L. Sargent, J. Almlöf, M.W. Feyereisen, *Electronic Structure Calculations in Quantum Chemistry: Massively Parallel Algorithms*, SIAM News, 26 (1993).
[25] D.C. Schmidt, *The Adaptive Communication Environment: Object-Oriented Network Programming Components for Developing Client/Server Applications*, 12th Sun Users Group Conference, 1994, available at http://www.cs.wustl.edu/schmidt/ACE.html.
[26] D.C Schmidt, *Evaluating Architectures for Multithreaded Object Request Brokers*, Communication of the ACM, 41 (1998), available at http://www.cs.wustl.edu/schmidt/TAO.html
[27] Sun Java home page, http://java.sun.com.
[28] P. Van Dooren and L. De Ridder, *An Adaptive Algorithm for Numerical Integration Over an N-Dimensional Cube*, Journal of Computational and Applied Mathematics, 2 (1976), pp. 207-217.
[29] M. Vanneschi, *The programming model of ASSIST: an environment for parallel and distributed portable applications*, Parallel Computing, 28 (2002), pp. 1709-1732.
[30] D.W. Walker, M. Li, O.F. Rana, M.S. Shields, Y. Huang, *The Software Architecture of a Distributed Problem-Solving Environment*, Concurrency: Practice & Experience 12 (2000), pp. 1455-1480.

Edited by: Peter Zinterhof
Received: September 23rd, 2002
Accepted: May 21st, 2003