Parallel Numerics '02, 204-214 R. Trobec, P. Zinterhof, M. Vaiteršic, A. Uhl (Eds.)
Chapter 1: Numerical Integration ISBN 961-6303-39-2

Embedding Parallel Quadrature Software into a HPC Environment

Pasqua D'Ambra[1], Daniela di Serafino[2], Marco Lapegna [3,*]

[1] National Research Council, Institute for High-Performance Computing and Networking (ICAR), via Cintia - Monte S. Angelo, I-80126 Naples, Italy.

[2] Department of Mathematics, Second University of Naples, via Vivaldi 43, I-81100 Caserta, Italy

[3] Department of Mathematics and its Applications, University of Naples "Federico II", via Cintia - Monte S. Angelo, I-80126 Naples, Italy.

We present our experiences in embedding parallel software, based on a multidimensional quadrature algorithm, into an advanced programming environment for building parallel and distributed high-performance applications. The programming environment, named ASSIST, is based on a combination of structured parallel programming and component-based programming. It is under development in the context of Italian national research projects and some activities are devoted to the definition and implementation of a methodology for its extension with a set of efficient, accurate and reliable numerical modules. To this aim, we have proposed a general approach for integrating MPI-based numerical routines into the ASSIST environment. We focus here on the integration of a quadrature routine based on an adaptive algorithm aimed at achieving dynamic load balancing and scalability. Results of experiments, devoted to analysing the performance behaviour of the embedded quadrature software and the cost of embedding, are reported.

1 Introduction

The development of modern scientific applications requires high-performance computing systems and sophisticated software tools. In such a context, numerical libraries have a relevant role, since they provide accurate, efficient and reliable solutions to computational kernels common to different applications.

*Corresponding author. E-mail: marco.lapegna@dma.unina.it

On the other hand, while high-quality libraries are available for the solution of many classes of problems, integrating and assembling them into large software systems is very difficult, because of data management and interoperability problems.

The component programming model is a way to obtain software integration and interoperability, reuse of existing codes and seamless access to distributed and loosely-coupled resources. Component standards and implementations have been initially proposed by the business world (see [1, 2, 3]) and a large effort is currently addressed to characterize the component model for high-performance computing [4]. The main impact of this scenario on numerical software production is the definition and implementation of component-based numerical toolkits, encapsulating new and existing computational kernels (see, for example, [5, 6]).

A programming environment for building and running parallel and distributed applications is under development in the context of some Italian research projects, supported by the National Research Council (CNR) and by the Italian Ministry for Education, University and Research (MIUR). This environment, named *ASSIST* (A Software development System based upon Integrated Skeleton Technology), implements a hybrid approach, based on a combination of the component model with the structured parallel programming model, in order to realize an extensible general-purpose tool for parallel and distributed computing [7]. Our work in the projects is devoted to extending the ASSIST environment with a numerical toolkit, including modules for Linear Algebra, Quadrature and FFT computations.

The focus here is on the integration of multidimensional quadrature software. 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 little or no changes, preserving as much as possible its performance, and exploiting static and dynamic optimization mechanisms provided by the ASSIST environment.

This paper is organized as follows. In Section 2 we briefly describe the multidimensional quadrature routine that has been integrated into the ASSIST environment. In Section 3 we outline basic features of the programming environment and describe our approach for embedding MPI-based numerical kernels into the ASSIST basic software unit for building parallel and distributed applications. Results of experiments devoted to analysing the performance behaviour of the embedded quadrature routine, carried out on a Beowulf-class Linux cluster, are discussed in Section 4. Concluding remarks and future work are reported in Section 5.

2 A parallel routine for multidimensional quadrature

As a starting point for the integration into the ASSIST environment, we considered a multidimensional quadrature routine, targeted at MIMD distributed-memory architectures. This routine implements an adaptive algorithm for the computation of the following multidimensional integral:

$$I(f) = \int_U f(\underline{t})\, d\underline{t} = \int_U f(t_1, ..., t_n)\, dt_1 \cdots dt_n, \tag{1}$$

where $U = [a_1, b_1] \times \cdots \times [a_n, b_n]$ is a n-dimensional hyper-rectangular region. For problem (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 [8].

An adaptive algorithm for computing (1) is based on an iterative procedure that, at the generic j-th iteration, evaluates a composite quadrature rule $Q^{(j)}$, approaching $I(f)$, and an absolute error estimate $E^{(j)}$, approaching 0. The quadrature rule $Q^{(j)}$ is computed on a partition $\mathcal{S}^{(j)} = \{s_1^{(j)}, \ldots, s_{q_j}^{(j)}\}$ of the domain U and the error $E^{(j)}$ is the sum of the absolute error estimates, $e_k^{(j)}$, in the subdomains $s_k^{(j)}$ of the partition. 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 $\hat{s}^{(j-1)}$ of $\mathcal{S}^{(j-1)}$ 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. This kind of procedure is called *global adaptive algorithm* and it is at the basis of several routines for multidimensional quadrature [9, 10, 11].

At a first look, it may seem that the multidimensional quadrature is a naturally parallel problem, in the sense that the integration domain U can be splitted among the processors and the previous algorithm can be applied by each processor in its subdomain. There are, however, two problems in the parallelization of an adaptive algorithm. The first one is that the sequence $\{\mathcal{S}^{(j)}\}$ of domain partitions is unpredictable, so the workload cannot be uniformly distributed among the processors before the computation [8, 12]. Therefore, to ensure proper load balancing, the processors have to periodically compare the data in their private memories and to reorganize the work subdivision. A further problem is the *scalability* of the algorithm. Roughly speaking, given the execution time $T(1; \mathcal{J})$ of a job $\mathcal{J}$ on one processor, the scalability can be

defined as the ability of the algorithm to perform an H-times larger job $H\mathcal{J}$, on H processors, in the time $T(H; H\mathcal{J}) = T(1; \mathcal{J})$ [13]. In this case, by H-times larger job we mean a job requiring H times the number of floating-point operations of the job $\mathcal{J}$. A necessary condition for scalability is that the communication time $T_{comm}(H; H\mathcal{J})$ is independent of the number of processors H, when the computation time $T_{calc}(H; H\mathcal{J})$ is kept constant in each processor. To achieve scalability, in our parallel algorithm all the global communications have been removed, because their cost is at least $T_{comm}(H; H\mathcal{J}) = \mathcal{O}(\log_2 H)$, hence making the algorithm poorly scalable [8].

Let $\mathcal{S}_i^{(j)}$ be the subpartition of $\mathcal{S}^{(j)}$ made by the subdomains $s_{i,k}^{(j)}$ that reside in the local memory of the processor P_i, at the j-th iteration. The basic idea of our parallel adaptive algorithm is to split, at each iteration, one subdomain $s_{i,k}^{(j)}$ in each subpartition $\mathcal{S}_i^{(j)}$.

Let us consider H processors, logically arranged into a periodical bidimensional mesh, and denote by 0 and 1 the horizontal and the vertical direction of this mesh, respectively. At the beginning of the algorithm ($j = 0$), the integration domain U is partitioned into H subdomains $s_{i,1}^{(0)}$ ($i = 0, \ldots, H-1$) of the same size, and each subdomain $s_{i,1}^{(0)}$ is assigned to the processor P_i, which therefore holds the subpartition $\mathcal{S}_i^{(0)} = \{s_{i,1}^{(0)}\}$. P_i computes the quadrature rule $Q_i^{(0)}$, which approximates the restriction of $I(f)$ to $\mathcal{S}_i^{(0)}$, and the corresponding absolute error $E_i^{(0)}$. At the generic iteration j, with $j > 0$, each processor P_i considers the direction $dir = mod(j, 2)$ of the mesh and attempts to send its subdomain $\hat{s}_i^{(j)}$ with the largest error estimate, $\hat{e}_i^{(j)}$, to the processor P_i^+ which follows P_i in the direction dir. The subdomain $\hat{s}_i^{(j)}$ is sent if $\hat{e}_i^{(j)}$ is larger than the maximum error estimate held by P_i^+. Then P_i selects the domain with the maximum error estimate in the current subpartition, divides it into two parts and updates $Q_i^{(j)}$ and $E_i^{(j)}$. A local error criterion is applied in each subpartition $\mathcal{S}_i^{(j)}$, to stop the iterative procedure.

With the above strategy, the "hard" subdomains are shared among the processors, allowing a successful workload distribution. Furthermore, at the generic iteration, each processor sends to the next one (and hence receives from the previous one), in the horizontal or vertical direction, at most $2n + 3$ scalar data, i.e. the extrema of the intervals defining the n-dimensional subdomain with the maximum error estimate, the corresponding quadrature rule and error values, and some additional information. This communication cost is independent of the number of processors, H, because the communication occours only between processors directly connected in the bidimensional mesh, and the resulting algorithm can be expected to be scalable. Details on a more general version of this algorithm, along with an analysis of its performance on

an MPP machine, are given in [14].

We implemented the previous algorithm in the mathematical software item, `dsmint`, written in the C language, in double precision, with the MPI message passing library, using a degree 9 integration rule developed by Genz and Malik in [15], for the computation of the integral approximations $r_i^{(j)}$, and the error estimate developed in [16], for the computation of $e_i^{(j)}$. The application of the above integration rule requires, at each iteration, $2[2^n + (4/3)n(n-1)(n-2) + 6n(n-1) + 8n + 1]$ function evaluations per processor.

3 Embedding the quadrature routine into an ASSIST parallel module

In order to describe our methodology for embedding the quadrature routine into an ASSIST module, we shortly describe the main features of the ASSIST environment and its programming model. For more details we refer to [7].

3.1 Main features of ASSIST

The ASSIST programming model is based on a combination of the concepts of structured parallel programming and component-based programming. An ASSIST program is structured as a graph, where the nodes are the components and the edges are the component abstract interfaces, defined in terms of typed I/O streams. The basic unit of an ASSIST program is a component named *parmod (parallel module)*, which allows to represent different forms of parallel computation. 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 logically parallel activities of 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 and objects in the code of the virtual processors; as an example, CORBA calls can be issued to access CORBA objects.

The user interface of the ASSIST environment is a coordination language, named *ASSIST-cl*. In the sintax 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*

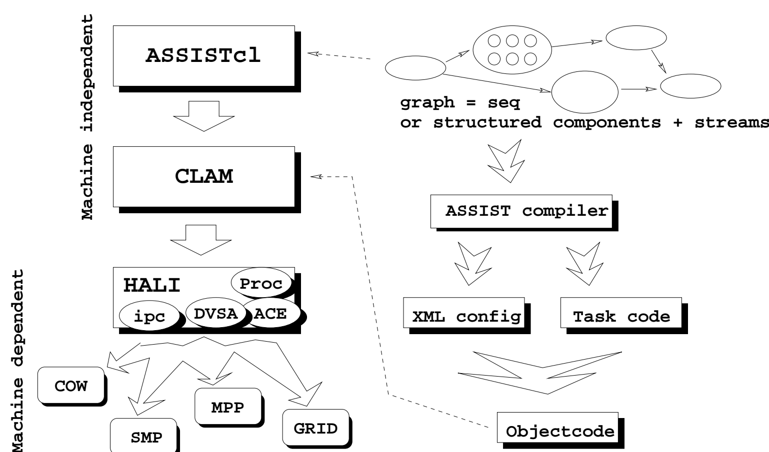

Figure 1: ASSIST architecture.

section, which describes the computation to be executed by the virtual processors and can include existing Fortran 77 or C code.

A layered software architecture has been implemented (see Fig. 1) to support the above programming model on the target hardware architectures, including SMPs, MPPs, and NOWs. Extensions to Computational Grids are under development. 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, named *Hardware Abstraction Layer Interface (HALI)*, which currently exports functionalities from the ACE multithreading and interprocess communication library [17], from the DVSA distributed virtual shared-memory library [18], and from a standard CORBA implementation [19], for using external CORBA objects within ASSIST applications. The ASSIST compiler translates ASSIST-cl source code into C++/HALI processess and threads, using pre-defined implementation templates. In running this code, the 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.

3.2 Process configuration of the numerical parmod

In the following, we describe our methodology for embedding an MPI-based numerical library routine into an ASSIST *parmod*, which from now on, is referred to as *numerical parmod*. The parallel routine code is encapsulated into

the virtual processor section of the parmod, while the parmod 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.

Following the template of a general parmod, for the numerical parmod we implemented a process template consisting of $k + 2$ processes: the *Input Stream Manager (ISM)*, the *Output Stream Manager (OSM)* and k *Virtual Processor Managers (VPMs)*. ISM and OSM esentially support input and output sections, respectively, while VPMs support the activities of the virtual processors. A single VPM is run onto each processing element participating to the execution of the numerical parmod. In particular, these VPMs are in charge of executing MPI computations. We note that this choice allows MPI processes to run under the CLAM control and hence to be taken into account in the process allocation, scheduling and mapping performed by the ASSIST environment.

Since the VPMs are also the MPI processes that execute the computations of the encapsulated numerical routine, 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 performed by the `MPI_Init` function (we refer to the C language API of MPI); the complementary one is `MPI_Finalize`, that cleans up all the MPI state. However, these functions cannot be directly used by the VPMs, because their execution "interfere" with the CLAM world.

To deal with this problem, we considered MPICH [20, 21], a well known and widely used free implementation of MPI. More precisely, we considered the MPICH 1.2.3 implementation for a cluster of Unix workstations. 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 a software layer, which generates MPI slave processes executing their parallel code, and pass them 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 the above information. The MPI master makes this 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 the numerical parmod, the MPI processes are the VPMs and hence they have already been started by the CLAM. Therefore, they cannot use `MPI_Init` to initialize the MPI environment. Therefore, we applied some modifications to MPICH, to allow each VPM to setup the MPI environment parameters, without any further process generation. 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 the information which the MPI setup phase can be started from. To this aim, `MPI_Init` and some other functions have been modified, to avoid process spawning and to send/receive the above information, using the ASSIST communication system. In this way, the ASSIST and the MPI world live in the same processes, without any conflict.

3.3 Implementation of the numerical parmod

The numerical parmod encapsulating the quadrature routine has been implemented in the C++ language, by using a preliminary version of HALI. In our implementation, the ISM generates the procgroup file, once it has received the TCP/IP addresses of the nodes hosting the VPMs. The ISM also manages two input streams to the parmod, one for the integer parameters of the quadrature routine (number of dimensions and maximum number of function evaluations), and the other for the double precision input parameters (relative and absolute tolerances and extrema of intervals that define the integration domain, a_i and b_i). These data are sent to the VPMs, that support the activities of the MPI processes. As already observed, one of the VPMs plays the role of the MPI master, and hence reads the procgroup file and sends to the other VPMs the minumum information (TCP/IP port and MPI master machine) needed to start the MPI setup. Then, all the VPMs call the the MPI-based quadrature routine. The computed results are sent by the VPMs to the OSM, that, in turn, sends them out of the parmod, onto two streams, one for the integer output data (actual number of function evaluations and error flag) and the other for the double precision output data (integral approximation and estimated error).

4 Timing results

Numerical experiments have been carried out using a Beowulf-class Linux cluster, available at ICAR-CNR (Naples Section). 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 GNU 2.96 C++ compiler, and with MPICH 1.2.3. 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.

Numerical tests have been performed by using a family of functions taken from the Genz's package [22]. This package is based on six different families of functions, each of them characterized by some peculiarities (peaks, oscillations, etc.). Here we report results obtained computing the integrals of 10 functions belonging to the following family, made of oscillating functions:

$$f(\underline{x}) = cos(2\pi\beta_1 + \sum_{i=1}^{n} \alpha_i x_i),$$

where the parameters α_i and β_i determine the location of the "difficulty" in the integrand function and its sharpness. Following [22], the 10 functions have been obtained by a random selection of these parameters. The number of dimensions, n, of the integration domain has been set equal to 8.

We first analyze the scalability of the routine `dsmint`, i.e. the MPI-based quadrature routine, when the number of nodes hosting the VPMs, H, increases and the work is kept constant in each node. Let $\mathcal{J}_l$ denote the computation of the integral of the l-th function of the family, $l = 1, .., 10$. The quadrature routine has been used to compute, on one node, the 10 integrals, with a given user tolerance $\varepsilon = 10^{-8}$, obtaining the execution times $T(1; \mathcal{J}_l)$ and the number B_l of evaluations of the integrand function required to compute $\mathcal{J}_l$. These times have been computed by using the return value of the Unix system function `times`. Then, for a number of nodes $H > 1$, the "H-times larger" integrals $H\mathcal{J}$ have been computed, with tolerance $\varepsilon = 0$ and HB_l as maximum number of integrand evaluations, to obtain the execution times $T(H; H\mathcal{J}_l)$. The scalability of the quadrature routine has been measured by using the mean value of the scaled efficiency for the 10 integrand functions:

$$R_H = \frac{1}{10} \sum_{l=1}^{10} \frac{T(1; \mathcal{J}_l)}{T(H; H\mathcal{J}_l)},$$

The mean value R_H and the corresponding standard deviation σ_{R_H} are reported in Table 1, for 2, 4 and 8 nodes, and confirm the expection of a good scalability of the quadrature routine.

	$H = 2$	$H = 4$	$H = 8$
R_H	1.0	0.83	0.76
σ_{R_H}	0.10	0.05	0.06

Table 1: Mean and standard deviation of scaled efficiency values, for $H = 2, 4, 8$ nodes hosting the VPMs.

We now report the results of experiments devoted to measuring the execution time of the quadrature parmod and of its single tasks, i.e. 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 routine `dsmint`. As in the scalability analysis, the "size" of the problem has been scaled with the number of nodes hosting the VPMs, H, to keep constant the workload per processor. The user tolerance $\varepsilon = 10^{-8}$ has been considered for $H = 1$, as in the previous case.

Mean, standard deviation, minimum and maximum of the execution times with the 10 test functions, on H nodes, $H = 1, 2, 4, 8$, are reported in Table 2, for each single task and for the whole parmod.

We note that the total minimum and maximum values are not the sum of the minimum and maximum task values; in particular, a maximum value for the computational kernel does not correspond, in general, to maximum values for the other stages, according to the fact that the workload of these stages does not depend on the choice of the test function. For a given number of nodes, a relatively large standard deviation can be observed for all tasks. Actually, for each task but the computational kernel, only few time values are sensibly different from the other time values, but they affect the computation of mean and standard deviation. This time difference is probably due to system processes running on the Beowulf machine and also the low resolution of the time unit. We also see that most of the time is spent in the setup of the ASSIST communication channels. This time increases as the number of nodes increases, while it does not depend on the workload of the routine `dsmint`. Therefore, we expect this time becomes less relevant in embedding more computing-intensive kernels. However, further investigation is needed to achieve a more efficient ASSIST setup in the numerical parmod. Finally, the time required to inizialize the MPI environment can be considered acceptable if compared with the execution time of the quadrature kernel. Furthermore, it is expected to become negligible, if the workload of the computational kernel increases.

$H = 1$	ASSIST setup	MPI setup	data distr./coll.	`dsmint`	parmod
mean	1.20	0.01	0.02	0.13	1.36
stand. dev.	0.42	0.01	0.01	0.07	0.41
min	1	0	0.01	0.09	1.12
max	1.99	0.03	0.05	0.32	2.13

$H = 2$	ASSIST setup	MPI setup	data distr./coll.	`dsmint`	parmod
mean	1.80	0.02	0.02	0.13	1.97
stand. dev.	0.42	0.01	0.02	0.08	0.37
min	1	0.01	0.01	0.08	1.16
max	2	0.04	0.06	0.35	2.16

$H = 4$	ASSIST setup	MPI setup	data distr./coll.	`dsmint`	parmod
mean	4.31	0.05	0.03	0.15	4.54
stand. dev.	1.25	0.02	0.02	0.08	1.30
min	2	0.02	0.01	0.11	2.18
max	6.03	0.09	0.08	0.39	6.54

$H = 8$	ASSIST setup	MPI setup	data distr./coll.	`dsmint`	parmod
mean	5.99	0.06	0.03	0.17	6.25
stand. dev.	0.03	0.03	0.02	0.09	0.09
min	5.93	0.01	0.01	0.12	6.14
max	6.03	0.08	0.07	0.43	6.46

Table 2: Mean, standard deviation, minimum and maximum of the execution times (in seconds) of the single tasks and of the whole parmod, on H nodes, $H = 1, 2, 4, 8$ hosting the VPMs.

5 Conclusions and future work

We described some research activities devoted to integrating a high-performance software module for multidimensional quadrature into a recently proposed programming environment for parallel and distributed computing, named ASSIST. An MPI-based adaptive quadrature routine has been embedded into the basic programming unit of ASSIST, allowing the composition of the quadrature module with any other software unit in the ASSIST environment. Furthermore, our approach for embedding numerical routines appears to be a general methodology enabling the reuse of MPI-based legacy codes into ASSIST applications. On the other hand, the additional execution time due to

the wrapping procedure is high if the embedded software has a low computational cost. Therefore, future work will be devoted to reduce the cost of wrapping and to experience with other MPI-based numerical software.

References

[1] M. Horsmann, M. Kirtland, DCOM Architecture, Microsoft White Paper (1997), available at http://www.microsoft.com/com/wpaper/.

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

[3] 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.

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

[5] Common Component Architecture at ORNL, home page, http://www.csm.ornl.gov/cca/.

[6] Common Component Architecture at Argonne, home page, http://www-unix.mcs.anl.gov/cca/.

[7] M. Vanneschi, ASSIST: an Environment for Parallel and Distributed Portable Applications, Tech. Rep. TR-02-07, University of Pisa, Italy (2002), available at http://www.di.unipi.it/research/TR.

[8] A. Krommer and C. Ueberhuber, *Numerical integration on advanced computer systems*, Lecture Notes in Computer Science, 848 (Springer-Verlag, 1994).

[9] J. Berntsen, T. Espelid and A. Genz , Algorithm 698: DCUHRE - An adaptive multidimensional integration routine for a vector of integrals, *ACM Transaction on mathematical software*, 17 (1991), pp. 452–456.

[10] D. Khaner and O. Rechard, TWODQD: an adaptive routine for two-dimensional integration, *Journal of Computational and Applied Mathematics*, 17 (1987), pp. 215–234.

[11] 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.

[12] E. de Doncker and J. Kapenga, A parallelization of adaptive integration methods, in P. Keast and G Fairweather eds., *Numerical integration*, (Reidel Publ. Comp., 1987).

[13] J. Gustafson, G. Montry and R. Benner , Development of parallel methods for a 1024 processor hypercube, *SIAM Journal on Scientific and Statistic Computing*, 9 (1988), pp. 580–588.

[14] 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.

[15] A. Genz and A. Malik, An embedded family of fully symmetric numerical integration rules, *SIAM Journal on Numerical Analysis*, 20 (1983), pp. 580–588.

[16] J. Berntsen, Practical error estimation in adaptive multidimensional quadrature routines, *Journal of Computational and Applied Mathematics*, 25 (1989), pp. 327–340.

[17] 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.

[18] 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 9th Euromicro Workshop on Parallel and Distributed Processing (2001), pp. 165–172.

[19] 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

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

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

[22] A. Genz, *Testing multiple integration software*, in B. Ford, J.C. Rault and F. Thommaset eds., *Tools, methods and languages for scientific and engineering computation* (North Holland, New York, 1984).