ELSEVIER

Parallel Computing 23 (1997) 1199–1210

PARALLEL COMPUTING

Practical aspects and experiences

# Scalability and load balancing in adaptive algorithms for multidimensional integration

Marco D'Apuzzo, Marco Lapegna * Almerico Murli

*Research Center on Parallel Computing and Supercomputers (CPS), via Cintia-Monte S. Angelo, 80126 Napoli, Italy*

Received 15 September 1995; revised 20 May 1996; accepted 15 December 1996

**Abstract**

A parallel adaptive algorithm for the approximate computation of a multi-dimensional integral over an hyperrectangular region is described. This is a more general version of the well known algorithms presented in Genz (1987) and Lapegna and D'Alessio (1993) and it has been developed for an efficient implementation on a MIMD distributed memory multiprocessor. In order to achieve a good scalability, all global communications have been removed from the algorithm. The processor's network has been configured as a multidimensional periodical mesh, to distribute information about the integrand behavior fast enough in order to achieve a good load balancing. Test results on the Intel Touchstone Delta System are given.

*Keywords:* Multidimensional integration; Adaptive algorithms; Parallel computing; Scalability and load balancing

## 1. Introduction

The numerical computation of multidimensional integrals

$$I(f) = \int_U f(\underline{t})\mathrm{d}\underline{t} = \int_U f(t_1, \ldots, t_n)\mathrm{d}t_1 \ldots \mathrm{d}t_n, \tag{1}$$

* Corresponding author.


*PII* S0167-8191(97)00017-3

is required in several application areas like quantum chemistry, high energy physics and statistics. In Eq. (1) $U = [a_1, b_1] \times \ldots \times [a_n, b_n]$ is a $n$-dimensional hyperrectangular region. Because of such a need, in the 1980's several efficient routines have been developed for the solution of this problem. Most of them (see for example [4,13,18]) are based on adaptive algorithms, that allow high accuracy with a reasonable computational cost.

In spite of the importance of this problem, few papers have been published about the parallelization of adaptive algorithms for multidimensional integration. De Doncker et al. [6] and Berntsen et al. [3,4] implemented some algorithms on MIMD shared memory computers, Genz [9] shows several ways to introduce parallelism in the adaptive algorithms for shared memory computers, and for distributed memory computers. In the last years one of the authors of this paper [15] and de Doncker and Kapenga [7] developed adaptive algorithms for MIMD distributed memory computers. A good book containing as well these references as the state of the art in numerical integration on advanced architectures is [14].

Our activity is aimed to develop adaptive algorithms for the computation of Eq. (1) on a MIMD distributed memory computers. One of the main target of the research is the scalability of the algorithms in the sense that they must be able to keep constant the efficiency when the problem size and the number of the processors increase. Preliminary experiences about this work are shown in [16]. In the present paper an improved scalable algorithm is described. This algorithm differs from that described in [16] in two aspects: (1) a more general and efficient communication strategy, described in Section 3, and (2) a stopping criterion based on the absolute error in stead of the relative one, showing inefficiency. Further, more extensive tests have been produced. On the basis of this algorithm the development of a portable mathematical software routine based on standard software tools like the Fortran language and the BLACS communication system is in progress in a collaboration with the NAG.

In Section 2 a brief outline of a common adaptive algorithm for the multidimensional integration and the problems related to their elective parallelization are given. Then in Section 3 a scalable adaptive algorithm is presented describing the data mapping, the topology of the network of processors and the communication algorithm. Finally, in Section 4 test results for our algorithm on the Intel Touchstone Delta System are shown.

## 2. Parallelization of adaptive algorithms

An adaptive algorithm for the computation of Eq. (1) is an iterative procedure that, at each iteration $j$, evaluates a composite integration rule on a family of subdomains $s_k^{(j)}$ $(k = 1, \ldots, K)$ of a partition $\mathcal{S}^{(j)}$.

The aim is to compute an approximation $Q(f)$ of $I(f)$ and an estimate $|E(f)|$ of the error $|Q(f) - I(f)|$ such that

$$|Q(f) - I(f)| < |E(f)| < \varepsilon, \tag{2}$$

where $\varepsilon$ is an user required tolerance. To achieve this, the algorithm computes a sequence $Q^{(j)}$ of composite quadrature rule approaching $I(f)$ and a sequence $|E^{(j)}|$ of

approximations of the error $|Q(f) - I(f)|$ approaching 0, until the user required tolerance is satisfied or it is found that the error criterion of Eq. (2) cannot be achieved within a allowed bound on the number of function evaluations. For dimensions up to 15 we have good rules available for standard regions [5].

Since the convergence rate of this procedure depends on the behavior of the integrand function (presence of peaks, oscillations, etc.), in order to reduce as soon as possible the error, at the iteration $j$, the subdomain $\hat{s}^{(j-1)}$ of $\mathcal{S}^{(j-1)}$ with maximum error estimate $\hat{e}^{(j-1)}$ is split in two parts $s_\lambda^{(j)}$ and $s_\mu^{(j)}$ that take the place of $\hat{s}^{(j-1)}$ in the partition $\mathcal{S}^{(j)}$. In the same way the approximations $Q^{(j)}$ and $E^{(j)}$ are updated.

Usually the subdomain $\hat{s}^{(j-1)}$ is split in two parts by halving the side where the fourth divided difference of the integrand function is larger than in the other directions. Such a value is taken as a measure of the difficulty of the integration in that direction, so that this strategy has been used to develop efficient routines for serial computers [4,18]. This kind of procedure is called *global* adaptive algorithm.

A framework for a sequential global adaptive algorithm for the computation of multidimensional integrals is therefore the following [17]:

**Algorithm 1**
**begin** Algorithm 1
Initialize $\mathcal{S}^{(0)}$ $Q^{(0)}$ and $E^{(0)}$
**while** $E^{(j-1)} > \varepsilon$ **do** stage $j$
(1) select $\hat{s}^{(j-1)} \in \mathcal{S}^{(j-1)}$ such that $\hat{e}^{(j-1)} = \max_{k=1,\ldots,K} e_k^{(j-1)}$
(2) divide $\hat{s}^{(j-1)}$ in two parts $s_\lambda^{(j)}$ and $s_\mu^{(j)}$
(3) compute $r_\lambda^{(j)}$ and $e_\lambda^{(j)} r_\mu^{(j)}$ and $e_\mu^{(j)}$
(4) sort the subdomains according to their errors
(5) update $Q^{(j)}$ and $E^{(j)}$
**endwhile**

The main problem in the parallelization of Algorithm 1 is that the sequence of $\mathcal{S}^{(j)}$ is unpredictable, so that it is impossible to distribute uniformly the workload among the processors before the computation. Therefore, to ensure proper balancing, periodically, the processors of a MIMD distributed memory computers have to compare the data in their private memory and have to reorganize a more suitable work subdivision. This type of load balancing (sometimes called *dynamic load balancing*) reacts to the current state of the algorithm during the execution of parallel tasks and it can improve the performance of parallel systems when there is unpredictable workload or system behavior, like in the case of the adaptive algorithms.

It is well know (see for example [9,14]) that, for a MIMD distributed memory computer with $H$ processors, an efficient way to reorganize the workload in Algorithm 1 is to process several subregions at the same time and to share the subdomains with the largest error estimate among the processors, because of the sufficiently large granularity of the processes.

A further aim in the development of efficient parallel algorithms is the *scalability*, that is the ability to perform a job $\mathcal{J}$ on a MIMD distributed memory multiprocessor with 1 processor, in about the same time of a $H$-times larger job $H\mathcal{J}$ (in the sense that

the job $H\mathcal{J}$ requires $H$ time more floating point operations than the job $\mathcal{J}$) on $H$ processors [12]. Usually the scalability of an algorithm is measured by

$$R_H = \frac{T(1;\mathcal{J})}{T(H;H\mathcal{J})}, \tag{3}$$

where, in general, $T(H;\mathcal{J})$ is the total elapsed time to perform a job $\mathcal{J}$ on a MIMD distributed memory multiprocessor with $H$ processors. Therefore the ideal value for the scalability is $R_H = 1$ but in practice a slight degradation is acceptable.

Further we remember that, since a necessary condition for the scalability is the data locality, the communication time $T_{\text{comm}}(H;H\mathcal{J})$ must be independent on the size of the system (that is the number of processors $H$) when the computation time $T_{\text{calc}}(H;H\mathcal{J})$ is kept constant in each processor.

## 3. A scalable algorithm

To improve the scalability of our algorithm, all global communications are removed, because such communication tasks have a cost at least

$$T_{\text{comm}}(H;H\mathcal{J}) = \mathcal{O}(\log_2 H)$$

and they make the algorithm poorly scalable. At the same time, the workload is balanced by performing communications only between pairs of directly connected processors in a $G$-dimensional periodical mesh $\mathcal{M}_G$. This is a set of $\Lambda_0 \times \ldots \times \Lambda_{G-1} = H$ processors, arranged along the points of a $G$-dimensional space with integer non negative coordinates in which a connection link between nearest nodes is established. In addition, the corresponding processors on the opposite faces of the mesh are connected too, so that the mesh is periodical. In Fig. 1 a 2-dimensional periodical mesh with $H = 12$ processors is shown for $\Lambda_0 = 4$ and $\Lambda_1 = 3$.

Note that if some $\Lambda_i = 1$ the $G$-dimensional periodical mesh is equivalent to a mesh of lower dimension. For example this is the case when $H = 2^k$ for some integer $k$ (that is the system can be configured like an hypercube) and $G > k$. Note also that this topology contains the *ring* when $G = 1$ and the *torus* when $G = 2$.

### 3.1. Data mapping

The basic idea of our adaptive algorithm is to split, at each iteration, $H$ subdomains (one for each processor), each of them chosen in a subpartition $\mathcal{S}_i^{(j)}$ of $\mathcal{S}^{(j)}$. Each

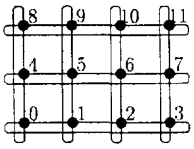


Fig. 1. A 2-dimensional periodical mesh with 12 processors.

subpartition $\mathscr{S}_i^{(j)}$ is composed by the subdomains $s_{i,k}^{(j)}$ that reside in the local memory of the processor $P_i$.

With the previous definitions, at the beginning of the algorithm (iteration $j=0$), the integration domain $U$ is divided into $H$ parts $s_{i,1}^{(0)}$, $(i=0,\ldots,H-1)$ of the same size, that are assigned to the $H$ processors. Then, each processor $P_i$ stores its subdomain $s_{i,1}^{(0)}$ in its own private memory. In this way each processor initializes, independently from the other ones, the subpartition $\mathscr{S}_i(0)$.

Therefore, the main iteration cycle of the algorithm is composed by the following steps:

**Step 1.** At the generic iteration $j$, each processor $P_i$ selects the subdomain to be split, $\hat{s}_i^{(j-1)} \in \mathscr{S}_i^{(j-1)}$, with error estimate $\hat{e}_i^{(j-1)}$ such that

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

only in the subpartition $\mathscr{S}_i^{(j-1)}$.

**Step 2.** Once the subdomain $\hat{s}_i^{(j-1)}$ has been selected, it is split by the processor $P_i$ in two 2 parts by halving the side where the fourth divided difference of the integrand function is larger than in the other directions. The two new subregions $s_{i,\lambda}^{(j)}$ and $s_{i,\mu}^{(j)}$ replace $\hat{s}_i^{(j-1)}$ in $\mathscr{S}_i^{(j-1)}$.

**Step 3.** Once the subdomain $\hat{s}_i^{(j-1)}$ with maximum error estimate has been split, each processor $P_i$ computes the integral and the error approximations in the new subregions.

**Step 4.** Each processor $P_i$ sorts the subdomains in its own memory.

**Step 5.** In this step are updated the absolute error estimate and the integral estimate in the subpartition $\mathscr{S}_i^{(j)}$ by each processor $P_i$. Let $E_i^{(j)}$ and $Q_i^{(j)}$ be such estimates and

$$\varepsilon_i = \varepsilon \frac{\text{Volume}\left(\mathscr{S}_i^{(j)}\right)}{\text{Volume}(U)}$$

the user required tolerance $\varepsilon$ scaled to the volume of the subpartition $\mathscr{S}_i^{(j)}$, if

$$|E_i^{(j)}| \le \varepsilon_i \tag{4}$$

then, the absolute error estimate in the whole integration region is

$$|E^{(j)}| \le \sum_{i=1}^{H} |E_i^{(j)}| \le \sum_{i=1}^{H} \varepsilon_i = \varepsilon$$

that is Eq. (2). Eq. (4) is used in our algorithm as a stopping criterion of the main iteration cycle. Note that this stopping criterion is local to the processor $P_i$, as it uses only information that resides in its private memory.

### 3.2. Redistribution procedure

In a $G$-dimensional periodical mesh, each processor $P_i$ has $2G$ neighbors: 2 for every direction. In the $G$ directions *dir*, we define

$$P_{i+}^{(\mathrm{dir})} \text{ and } P_{i-}^{(\mathrm{dir})} \mathrm{dir} = 0, \ldots, G-1,$$

respectively the next and the previous processor of $P_i$. More precisely let be the coordinates $(C_0^{(i)}, \ldots, C_{G-1}^{(i)})$ in the mesh $\mathcal{M}_G$ of the current processor $P_i$. These are

$$C_k^{(i)} = \mathrm{mod}\left( \frac{i}{\prod_{l=0}^{k-1} \Lambda_l}, \Lambda_k \right) \quad k = 0, \ldots, G-1.$$

Therefore the coordinates of $P_{i+}^{(\mathrm{dir})}$ are given by

$$\left(C_0^{(i)}, \ldots, C_{\mathrm{dir}}^{(i)} + 1, \ldots, C_{G-1}^{(i)}\right).$$

Obviously, if $C_{\mathrm{dir}}^{(i)} + 1 > \Lambda_{\mathrm{dir}} - 1$, to make $\mathcal{M}_G$ periodical, is defined $C_{\mathrm{dir}}^{(i)} = 0$. In an analogous way the coordinates of $P_{i-}^{(\mathrm{dir})}$ are defined.

In our algorithm, at the generic iteration $j$, each processor attempts to send to $P_{i+}^{(\mathrm{dir})}$, the next processor in the mesh direction $\mathrm{dir} = \mathrm{mod}(j, G)$, its subdomain with largest error estimate $\hat{s}_i^{(j)}$, in order to share, with the other processors, its 'hard' subdomains.

More precisely, in a fixed direction dir let $\hat{e}_i^{(j)}$, $\hat{e}_{i+}^{(j)}$ and $\hat{e}_{i-}^{(j)}$ be respectively the largest error estimates in the current processor $P_i$, in the next processor $P_{i+}^{(\mathrm{dir})}$ and in the previous processor $P_{i-}^{(\mathrm{dir})}$. If $\hat{e}_i^{(j)} > \hat{e}_{i+}^{(j)}$ then the current processor $P_i$ sends the subdomain $\hat{s}_i^{(j)}$ with error estimate $\hat{e}_i^{(j)}$ to the processor $P_{i+}^{(\mathrm{dir})}$. In the same way if $\hat{e}_{i-}^{(j)} > \hat{e}_i$ the current processor receives the subdomain $\hat{s}_{i-}^{(j)}$ with error estimate $\hat{e}_{i-}^{(j)}$ from the processor $P_{i-}^{(\mathrm{dir})}$.

About the scalability, it should be noted that in the proposed workload redistribution, the communication cost is independent on the number of processors $H$, because, at each iteration $j$, each processor $P_i$ exchanges messages only with the two processors $P_{i+}^{(\mathrm{dir})}$ and $P_{i-}^{(\mathrm{dir})}$. Therefore the resulting algorithm is scalable.

With the previous communication algorithm, an important topic is the diameter of the mesh of processors $G$ that is the maximum distance between any pair of processors. At this regard it is important to note that a good load balancing is achieved only if the subdomains with largest error estimate are quickly distributed among the processors, so that no processor wastes time on unimportant subdomains.

With this communication strategy, at each iteration, information about the subdomains with large errors are transmitted from a processor to another one along the $G$ directions of the mesh. Therefore such information reach all the processors after $D(\mathcal{M}_G)$ iterations, where $D(\mathcal{M}_G)$ is the diameter of the $G$-dimensional periodical mesh $\mathcal{M}_G$.

The diameter of a $G$-dimensional periodical mesh is $D(\mathcal{M}_G) = G\sqrt[G]{H}$ when the mesh is *symmetric*, that is when the sides have the same number of processors. In this case the 'hard' subdomains reach all processors after at most $G\sqrt[G]{H}$ iteration. Because $D(\mathcal{M}_G)$ is a decreasing function of $G$, the load balancing improves when $G$ increases.

The described algorithm includes, as special case when $G = 1$, the *ring-algorithm*, briefly reported in [9]. For this algorithm the author emphasizes the slow distribution of

the subdomains with largest error estimate. In fact when $G = 1$ the diameter of the system is $\mathcal{O}(H)$, that is much higher than the diameter of the system when $G > 1$. Therefore our algorithm will distribute the subregion with large error estimate faster than the ring algorithm, with an improvement of the error reduction speed.

At the end of this section a description of the algorithm is given. The following Algorithm 2 is the node algorithm in the single program multiple data (SPMD) model for the node $P_i$. In order to describe the communications tasks it is convenient to introduce the commands (see for example [11], ch. 6)

**send** (*data*, *dest*)
**recv** (*data*, *orig*)

The first of these commands realizes the transmission of *data* to the processor identified by *dest* and the second one realizes the reception of *data* from the processor identified by *orig*. Further it is necessary to use the command ([11], ch. 6)

**init** ($P_i$, $H$)

that initializes $H$ as the requested number of processors and assign to $P_i$ a single processor identifier (an integer) in $0, \ldots, H-1$.

The input data of the algorithm are the integration domain $U$, the user required tolerance $\varepsilon$, the integrand function $f$ and the mesh dimension $G$. The output of the algorithm is an approximation $Q(f)$ of the integral $I(f)$ and an approximation $|E(f)|$ of the error $|Q(f) - I(f)|$. It is assumed that the algorithm is executed on processor $P_i$ so that the 'scalable parallelization' of Algorithm 1 (Algorithm 2) is:

**Algorithm 2**

**begin** Algorithm 2
**init** ($P_i$, $H$)
Determine the processors $P_{i-}^{(\mathrm{dir})}$ and $P_{i+}^{(\mathrm{dir})}$, (dir $= 0, \ldots, G-1$)
Initialize $Q_i^{(j)}$, $E_i^{(j)}$, $\varepsilon_i$, $j$
Compute the global value of $Q^{(0)}$
**while** $|E_i^{(j)}| > \varepsilon_i$ **do** stage $j$
Compute dir = mod($j$, $G$)
**if** $|\hat{e}_i^{(j-1)}| > |\hat{e}_{i+}^{(j-1)}|$
**send** ($\hat{s}_i^{(j-1)}$, $P_{i+}^{(\mathrm{dir})}$)
update $Q_i^{(j)}$ and $E_i^{(j)}$
**endif**
**if** $|\hat{e}_{i-}^{(j-1)}| > |\hat{e}_i^{(j-1)}|$
**recv**($\hat{s}_{i-}^{(j-1)}$, $P_{i-}^{(\mathrm{dir})}$)
update $Q_i^{(j)}$ and $E_i^{(j)}$
**endif**
(1) select $\hat{s}_i^{(j-1)} \in \mathcal{S}_i^{(j-1)}$ such that $\hat{e}_i^{(j-1)} = \max_{k=1,\ldots,K} e_{i,k}^{(j-1)}$
(2) divide it in two parts $s_{i,\lambda}^{(j)}$ and $s_{i,\mu}^{(j)}$
(3) compute in this parts $r_{i,\lambda}^{(j)}$, $r_{i,\mu}^{(j)}$, $e_{i,\lambda}^{(j)}$ and $e_{i,\mu}^{(j)}$
(4) sort the subdomains according to their errors
(5) update $Q_i^{(j)}$, $E_i^{(j)}$, $\varepsilon_i$, $j$
**endwhile**
Compute the global approximation $Q(f)$ and $|E(f)|$

## 4. Test results

Algorithm 2 has been implemented in a FORTRAN double precision program and has been tested on the Intel Touchstone Delta System operated by the California Institute of Technology in the NX/2 message passing environment.

For our experiment we used a degree 9 integration rule developed by Genz and Malik in [8] for the unitary cube and the error estimate developed in [1], in order to compare our algorithm with the routine DCUHRE [4] that uses the same integration rule and error estimate procedure.

The numerical tests have been performed by using three families of functions taken from the Genz's package [10]. This package is based on six different families of functions, each of them characterized by some peculiarity (peaks, oscillations,...). The families have been tested using several dimension up to 9 and different tolerances. The families we used are

$$f^{(1)}(\underline{x}) = \prod_{i=1}^{n}\left(\alpha_i^{-2} + (x_i - \beta_i)^2\right)^{-1} \quad \text{product peak functions}$$

$$f^{(2)}(\underline{x}) = \exp\left(-\sum_{i=1}^{n}\alpha_i|x_i - \beta_i|\right) \quad C^{(0)} \text{ functions}$$

$$f^{(3)}(\underline{x}) = \cos\left(2\pi\beta_1 + \sum_{i=1}^{n}\alpha_i x_i\right) \quad \text{oscillating functions} \tag{5}$$

Each family is composed by 20 different functions where the parameters $\alpha_i$ and $\beta_i$ change. The parameters determine the location of the difficulty and its sharpness. The parameters $\beta_i \in [0, 1]$ are random values. The parameters $\alpha_i \in [0, 1]$ are also random values, but they are scaled according to

$$n^{e_j}\sum_{i=1}^{n}\alpha_i = d_j$$

in order to control the difficulty of the integrand when $n$ increases. For our experiments we used $\mathbf{d} = (300, 300, 15)$ and $\mathbf{e} = (1.5, 2, 0)$ as suggested in [10].

### 4.1. Sequential test

The first set of experiments is aimed to measure the reliability and the accuracy of Algorithm 2 with one processor, by comparing its results with that of the two routines DCUHRE [4] and ADAPT [18]. We remember that DCUHRE uses the same integration rule and the same error estimate procedure of Algorithm 2.

This kind of experiment is crucial because, in general, all parallel software introduces some additional operations respect to the sequential algorithm and it is important to verify that such a computational overhead is not expensive. In Tables 1–3 (one for every family functions) are reported the obtained results in 3 dimensions. In the first column of each table there is the required tolerance, then there are, for each routine, the average number of function evaluations needed to reach the required tolerance and the number of

Table 1
Product peak functions

| Req. toler. | Algorithm 2 | | DCUHRE | | ADAPT | |
|---|---|---|---|---|---|---|
| | func. eval. | digits | func. eval. | digits | func. eval. | digits |
| $10^{-1}$ | 8470 | 2.23 | 7909 | 2.86 | 756 | 1.25 |
| $10^{-2}$ | 16816 | 3.35 | 23503 | 4.38 | 4772 | 2.87 |
| $10^{-3}$ | 30830 | 4.55 | 44570 | 5.27 | 13953 | 4.26 |
| $10^{-4}$ | 54192 | 5.81 | 68831 | 5.96 | | |

exact digits. The number of exact digits is computed through the average of the actual errors on the 20 functions in the generic family, that is

$$\text{exact digits} = -\log_{10}\left(\frac{1}{20}\sum_{l=1}^{20}|Q(f_l) - I(f_l)|\right).$$

The tables show that our algorithm is reliable as the routine DCUHRE and it is much more reliable than ADAPT. More precisely Algorithm 2 evaluates the integrand function about the same number of time of DCUHRE with about the same exact digits. The results of DCUHRE (the old name of this routine was ADMINT) and ADAPT are taken from [2].

### 4.2. Scalability test

The second set of experiments has been performed to measure the scalability of Algorithm 2 when the number of the processors $H$ increases and the work is constant in each processor.

More precisely, for each family of functions let $\mathcal{F}_l$, $(l = 1, \ldots, 20)$ be the computation of the 20 integrals, Algorithm 2 has been used to compute, with one processor, the 20 integrals in 3 dimensions with a given user required tolerance $\varepsilon$. Then $T(1; \mathcal{F}_l)$ $(l = 1, \ldots, 20)$ and $B_l$ $(l = 1, \ldots, 20)$ have been aimed, where the $B_l$ are the number of function evaluations required to compute $\mathcal{F}_l$ with the given user required tolerance. Successively, for a number of processors $H > 1$, given the integrals $\mathcal{F}_l$, the $H$-time larger integrals $T(H; H\mathcal{F}_l)$ are defined as the integrals computed by using the user required tolerance $\varepsilon = 0$ and by using $B_l$ as maximum number of function

Table 2
$C^{(0)}$ functions

| Req. toler. | Algorithm 2 | | DCUHRE | | ADAPT | |
|---|---|---|---|---|---|---|
| | func. eval. | digits | func. eval. | digits | func. eval. | digits |
| $10^{-1}$ | 6945 | 2.08 | 6580 | 2.60 | 539 | 1.35 |
| $10^{-2}$ | 28875 | 3.37 | 27037 | 3.66 | 3923 | 2.51 |
| $10^{-3}$ | 78878 | 4.30 | 81303 | 4.61 | 15887 | 3.71 |
| $10^{-4}$ | 182028 | 4.94 | 187471 | 5.34 | | |

Table 3
Oscillating functions

| Req. toler. | Algorithm 2 | | DCUHRE | | ADAPT | |
|---|---|---|---|---|---|---|
| | func. eval. | digits | func. eval. | digits | func. eval. | digits |
| $10^{-1}$ | 970 | 3.72 | 1427 | 4.35 | 1326 | 3.21 |
| $10^{-2}$ | 1909 | 4.81 | 2603 | 4.83 | 3920 | 4.39 |
| $10^{-3}$ | 3311 | 5.78 | 5708 | 5.88 | 13073 | 5.64 |
| $10^{-4}$ | 6914 | 6.71 | 12966 | 7.13 | | |

Table 4
Average scalability for the product peak functions varying the number of processors $H$ and the mesh dimension $G$

| $P$ | $G=1$ | $G=2$ | $G=3$ | $G=4$ | $G=5$ | $G=6$ | $G=7$ |
|---|---|---|---|---|---|---|---|
| 2 | 0.82 | 0.86 | 0.87 | 0.89 | 0.89 | 0.89 | 0.90 |
| 8 | 0.80 | 0.81 | 0.82 | 0.85 | 0.86 | 0.87 | 0.87 |
| 32 | 0.79 | 0.80 | 0.81 | 0.82 | 0.83 | 0.83 | 0.85 |
| 128 | 0.78 | 0.71 | 0.74 | 0.74 | 0.75 | 0.75 | 0.80 |

Table 5
Average scalability for the $C^{(0)}$ functions varying the number of processors $H$ and the mesh dimension $G$

| $P$ | $G=1$ | $G=2$ | $G=3$ | $G=4$ | $G=5$ | $G=6$ | $G=7$ |
|---|---|---|---|---|---|---|---|
| 2 | 0.92 | 0.96 | 0.97 | 0.97 | 0.97 | 0.97 | 0.97 |
| 8 | 0.96 | 0.96 | 0.96 | 0.97 | 0.98 | 0.97 | 0.97 |
| 32 | 0.91 | 0.91 | 0.91 | 0.92 | 0.93 | 0.93 | 0.94 |
| 128 | 0.91 | 0.91 | 0.90 | 0.91 | 0.90 | 0.90 | 0.90 |

Table 6
Average scalability for the oscillating functions varying the number of processors $H$ and the mesh dimension $G$

| $P$ | $G=1$ | $G=2$ | $G=3$ | $G=4$ | $G=5$ | $G=6$ | $G=7$ |
|---|---|---|---|---|---|---|---|
| 2 | 0.89 | 0.89 | 0.90 | 0.91 | 0.90 | 0.89 | 0.90 |
| 8 | 0.87 | 0.89 | 0.88 | 0.90 | 0.90 | 0.89 | 0.89 |
| 32 | 0.86 | 0.88 | 0.88 | 0.88 | 0.89 | 0.88 | 0.88 |
| 128 | 0.85 | 0.87 | 0.87 | 0.87 | 0.88 | 0.88 | 0.87 |

Table 7
Average values for the actual absolute error

| $H$ | $f^{(1)}$ func. | | | $f^{(2)}$ func. | | | $f^{(3)}$ func. | | |
|---|---|---|---|---|---|---|---|---|---|
| | $G=1$ | $G=2$ | $G=5$ | $G=1$ | $G=2$ | $G=5$ | $G=1$ | $G=2$ | $G=5$ |
| 2 | 0.14(-4) | 0.12(-4) | 0.15(-4) | 0.51(-4) | 0.51(-4) | 0.56(-4) | 0.25(-4) | 0.21(-4) | 0.20(-3) |
| 8 | 0.38(-6) | 0.32(-6) | 0.32(-6) | 0.25(5) | 0.11(-5) | 0.10(-5) | 0.23(-6) | 0.33(-6) | 0.30(-6) |
| 32 | 0.46(-7) | 0.66(-8) | 0.86(-8) | 0.21(-6) | 0.38(-7) | 0.21(-7) | 0.15(7) | 0.23(-8) | 0.14(-8) |
| 128 | 0.11(-7) | 0.11(-8) | 0.26(-9) | 0.61(-7) | 0.64(-8) | 0.91(-9) | 0.75(8) | 0.58(-9) | 0.99(-10) |

evaluations for each processor. For this integrals the elapsed times using $H$ processors $T(H;\ H\mathscr{F}_l)$, $(l=1,\ \ldots,\ 20)$ are measured. In Tables 4–6 the average values of the scaled efficiency

$$R_H = \tfrac{1}{20}\sum_{l=1}^{20}\frac{T(1;\ \mathscr{F}_l)}{T(H;\ H\mathscr{F}_l)}$$

varying the number of processors $H$ and the mesh dimension $G$, are reported for the three functions families of Eq. (5). Such a tables confirm the expectation of good scalability of the pro posed algorithm in the sense that it is possible to evaluate the integrand functions on one processor in about the same time of $H$ times this number on $H$ processors.

Further in Table 7 the average values of the actual absolute errors

$$\tfrac{1}{20}\sum_{l=1}^{20}|Q(f_l) - I(f_l)|,$$

for the three families of functions of Eq. (5) are reported when the number of processors $H$ increases and the maximum number of function evaluations $B_l$ are fixed in each processor. This kind of experiment shows that, when $G > 1$, the 'hard' subdomains are distributed faster than when $G = 1$ and the processors do not waste time on unimportant subdomains. For this reason the actual error, reported in the table, decreases more quickly. When $G = 2$ and $G = 5$ we found about the same error reduction speed, because the value of the diameter is about the same (for example with $H = 32$ processors we have a diameter of $D(\mathscr{M}_2) = 12$ and $D(\mathscr{M}_5) = 10$).

Table 7 also shows which error is possible to achieve in a fixed time when the number of processors $H$ increases.

## Acknowledgements

This work was supported by the grant No. 93.01615.PF69 of the CNR (Special Project Informatic Systems and Parallel Computing) and it was performed in part by using the Intel Touchstone Delta System operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by the California Institute of Technology. Finally we thank the anonymous referees for their suggestions.

## References

[1] J. Berntsen, Practical error estimation in adaptive multidimensional quadrature routines, J. Comput. Appl. Math. 25 (1989) 327–340.
[2] J. Berntsen, T. Espelid, A. Genz, A test of ADMINT, Tech. Rep. No. 31/1988, Dept. of Informatics, Univ. of Bergen, Norway.
[3] J. Berntsen, T. Espelid, A. Genz, An adaptive algorithm for the approximate calculation of multiple integrals, ACM Trans. Math. Software 17 (1991) 437–451.
[4] J. Berntsen, T. Espelid, A. Genz, Algorithm 698: DCUHRE: An adaptive multi-dimensional integration routine for a vector of integrals, ACM Trans. Math. Software 17 (1991) 452–456.
[5] R. Cools, P. Rabinowitz, Monomial cubature rules since 'Stroud': A compilation, J. Comput. Appl. Math. 48 (1993) 309–326.
[6] E. de Doncker, J. Kapenga, A parallelization of adaptive integration methods, in: P. Keast, G. Fairweather (Eds.), Numerical integration, Reidel Publ. Comp., 1987.
[7] E. de Doncker, J. Kapenga, Parallel cubature on loosely coupled systems, in: T. Espelid, A. Genz (Eds.), Numerical Integration: Recent developments, software and applications, Kluwer, 1992.
[8] A. Genz, A. Malik, An embedded family of fully symmetric numerical integration rules, SIAM J. Numer. Anal. 20 (1983) 580–588.
[9] A. Genz, The numerical evaluation of multiple integrals on parallel computers, in: P. Keast, G. Fairweather (Eds.), Numerical Integration, Reidel Publ. Comp., 1987.
[10] A. Genz, Testing multiple integration software, in: B. Ford, J.C. Rault, F. Thommaset (Eds.), Tools, methods and language for scientific and engineering computation, North Holland, New York, 1984.
[11] J. Golub, Matrix computation, 2nd ed., The Johns Hopkins University Press, 1989.
[12] J. Gustafson, G. Montry, R. Benner, Development of parallel methods for a 1024 processor hypercube, SIAM J. Sci. Stat. Comput. 9 (1988) 580–588.
[13] D. Khaner, O. Rechard, TWODQD: an adaptive routine for two-dimensional integration, J. Comput. Appl. Math. 17 (1987) 215–234.
[14] A. Krommer, C. Ueberhuber, Numerical integration on advanced computer systems, Lecture Notes in Computer Science 848, Springer Verlag, 1994.
[15] M. Lapegna, Global adaptive quadrature for the approximate computation of multidimensional integral, Concurr. Pract. Exp. 4 (1992) 413–426.
[16] M. Lapegna, A. D'Alessio, A scalable parallel algorithm for the adaptive multidimensional quadrature, in: Proc. 6th SIAM Conf. on Parallel Processing for Scientific Computing, SIAM, 1993.
[17] M. Malcom, R. Simpson, Local versus global strategies for adaptive quadrature, ACM Trans. Math. Software 1 (1975) 129–146.
[18] P. Van Dooren, L. De Ridder, An adaptive algorithm for numerical integration over an $n$-dimensional cube, J. Comput. Appl. Math. 2 (1976) 207–217.