CONCURRENCY: PRACTICE AND EXPERIENCE, VOL. 4(6), 413–426 (SEPTEMBER 1992)

Global adaptive quadrature for the approximate computation of multidimensional integrals on a distributed-memory multiprocessor

MARCO LAPEGNA

Dipartimento di Matematica ed Applicazioni
Università degli studi di Napoli 'Federico II'
Via Mezzocannone 16, 80134—Napoli, Italy

SUMMARY

In this paper we discuss the problem of computing a multidimensional integral on a MIMD distributed-memory multiprocessor.

Adaptive quadrature is known as a good approach to the problem of achieving accuracy and reliability while attempting to minimize the number of function evaluations. The implementation makes use of dynamical data structures able to manage subinterval partition. On a distributed-memory multiprocessor, each processor is able to execute code and to manipulate data structures in its own local memory only, and data are sent from one processor to another one by explicit message-passing. Efficient implementation of an adaptive algorithm for the multidimensional quadrature on a parallel computer is quite difficult, because of the need for continuous information exchange between processors.

Our algorithm is based on a global adaptive strategy which dynamically balances the workload and reduces the data communication between processors in order to use the message-passing environment efficiently.

The results and timings for several tests are given.

1. INTRODUCTION

Let $C^d = [0,1]^d$ be the d-dimensional unitary cube and $f(\mathrm{t}) = f(t_1, \ldots, t_d)$ be a continous function on C^d; we discuss the problem of computing

$$If = \int_{C^d} f(\mathrm{t})\,d\mathrm{t} \tag{1}$$

on a distributed-memory multiprocessor. The solution of (1) is fundamental for a large range of scientific applications like chemistry, physics and statistics[1], and interest in it is growing in the scientific community.

In the past few years several algorithms and software have been developed in order to solve the above problem in one dimension, but the high computational cost inhibits solution of the multidimensional case using a serial computer. However, as in the one-dimensional case, adaptive quadrature is a good approach to the problem of achieving accuracy and reliability while attempting to minimize the number of function evaluations. Thus several routines have been developed for the multidimensional quadrature in the last years by using adaptive algorithms[2,3]. Further, in the past few years, some authors[4,5] have pointed out that parallel computing seems to be able to overcome the computational cost drawback because multidimensional quadrature has an intrinsic

1040–3108/92/060413–14$12.00
©1992 by John Wiley & Sons, Ltd.
Received 10 August 1990
Revised 14 January 1992

parallelism: for example, we can split the integration domain into several subdomains and then concurrently integrate the function in each of them.

In this paper we present an algorithm based on a global adaptive strategy using a Cavalieri–Simpson product rule for an MIMD distributed-memory multiprocessor. It is well known that, on this kind of parallel computer, the communications have to be strongly reduced. Our algorithm has been developed with the aim of balancing the workload dynamically and reducing the data communications among the processors, in order to use a MIMD distributed memory multiprocessor efficiently with $p = 2^s$ processors (for example a hypercube)[6].

In Section 2 we explore the relations between adaptive quadrature and parallel computing. Specifically we explain the costs and the benefits of the two basic strategies. In Section 3 we describe our algorithm, including the influence of various steps on the performance. In Section 4 we briefly describe the computational environment used and some implementation issues. Finally, in Section 5 we discuss the numerical results and in Section 6 some concluding remarks.

2. ADAPTIVE QUADRATURE AND PARALLEL COMPUTING

An adaptive algorithm for numerical quadrature is an algorithm which processes a family of subdomains $\{D_k\}$ of the integration domain C^d with the aim of computing an approximation $\tilde{If}$ of If such that

$$Ef = |\tilde{If} - If| < \varepsilon$$

where ε is a user-specified tolerance. To do this, one computes a sequence of couples:

$$\{I_i, E_i\}_{i \in \mathbf{N}} \qquad (2)$$

approximating If and Ef, respectively, such that

$$\lim_{i \to \infty} I_i = If, \qquad \lim_{i \to \infty} E_i = 0$$

The approximations I_i and E_i are computed by evaluating If and Ef in each subdomain of a partition of C^d and then by summing the partial results. Usually the estimates of If and Ef in every subdomain are computed by using two rules R_n and R_m. If they are based respectively on n and m abscissas with $n < m$, an efficient implementation requires that $T_n \subset T_m$, where T_n and T_m are the abscissa sets of R_n and R_m. Thus we can use the first one as an estimate of If and $|R_n - R_m|$ as an estimate of Ef with only m integrand function evaluations. Such rules are called *nested rules*.

The main problems in an adaptive algorithm for numerical quadrature are the criteria for processing the subdomains $\{D_k\}$, and the stopping criterion in (2). For the one-dimensional case two basic strategies are known[7]:

local adaptive strategy: at each stage of the algorithm the approximate value of If in a subdomain D_k is accepted if a local acceptance criterion is satisfied (generally it is required that the error estimate in this subdomain is smaller than $\varepsilon * volume(D_k)$); if

this does not happen, then it is subdivided into several subdomains that are added to the set of the subdomains not yet examined (*pending set*).

global adaptive strategy: all subdomains remain in the pending set until an acceptance criterion for the entire pending set is satisfied (generally it is required that the sum of the error estimates in each subdomain is smaller than ε). At each stage of the algorithm the subdomain with the largest error estimate is subdivided.

Therefore a first-level description of an adaptive algorithm is the following:

*Initia*lize RESULT and ERROR;
while (*the acceptance criterion is not satisfied*) **do**
attempt to reduce ERROR and update RESULT;
endwhile

where '*attempt to reduce ERROR and update RESULT*' is the procedure implementation of one of the previous strategies. Since all subdomains in the pending set can be processed successively, in both strategies some suitable data structures to store them are needed. Thus at every iteration the algorithm is able to find easily the subdomain that has to be subdivided.

Briefly, we can say that the global strategy attempts to reduce the local error in every subinterval in a uniform way, whereas the local strategy attempts to satisfy the acceptance criterion sequentially in every subinterval. Then it is possible to see that in the one-dimensional case the global strategy requires a reduced function evaluation number with respect to local strategy, but it needs a bigger memory space in order to store all the subdomains of the pending set[7]. Thus in the one-dimensional case it is possible to develop efficient routines by using either the local adaptive strategy[8] or the global adaptive strategy[9,10].

In the multidimensional case it is known that the major target in algorithm development is to reduce the computational cost; thus efficient routines for this problem can be developed by using mainly the global adaptive strategy. When we implement such an algorithm on a distributed-memory multiprocessor it is important to preserve the global strategy advantages because a general principle of parallel computing is to use local methods in place of global methods.

In this regard we note that, in a parallel adaptive algorithm for the multidimensional quadrature which makes use of a global strategy, the computation is dominated by the quadrature rule evaluation. Further the global control for algorithm stopping criterion generally requires the sum of the error estimates that are located in the local memory of the processors. That can be done by using cascade schemes, with a computational cost of $\log_2 p$, where p is the number of processors. Therefore the global control needed by the stopping criterion can be implemented quite easily with a communication overhead smaller than the extra computational cost required by the local strategy[5].

3. ALGORITHM DESCRIPTION

Our algorithm is based on a global adaptive strategy. In order to estimate If and Ef in every subdomain D_k, we use a 2-panel and a 4-panel Cavalieri–Simpson product rule. Calling these rules S_2 and S_4, respectively, we use the first one as an estimate for If, and $|S_2 - S_4|$ as an estimate for Ef.

In the global adaptive strategy, the data structure used to store the pending set has to be an ordered one. Further, it can be very large, because, if we bisect the subdomain sides in all dimensions, 2^d new subdomains are added at each iteration. Thus a very efficient list management is required. About this we can note that a completely ordered list is not necessary for our aims because we are interested only in the maximum value in the list and it is sufficient to have a partially ordered binary tree, sometimes called *heap*[7]. In a heap, the value of each node is greater than or equal to that of its subnodes. In this way it is easy to see that the maximum value in the tree is in the root. Sorting such a data structure requires $O(N \log_2 N)$ comparisons.

To implement this algorithm on an MIMD distributed-memory multiprocessor with the aim of balancing the workload among the processors, we split up the subdomains with the maximum error estimates. In this way the processors build a heap in their own private memory, and they have to exchange only the root of the heap in order to find the subdomain that they have to split up.

Once all the processors have located such a subdomain, they share it by bisecting the sides in every dimension, and thus it always preserves its hypercube form.

Therefore, at each iteration cycle, 2^d new subdomains are generated, and, by using a multiprocessor with $p = 2^s$ processors, we have that

$$2^d/p = 2^{d-s} \tag{3}$$

of them are assigned to each processor. Therefore every processor can compute, independently from the other ones, the new approximate values of If and Ef. To assign at least one subdomain to every processor, we note from (3) that this algorithm cannot be used when $d < s$.

Therefore the algorithm executed by each processor is the following:

initialize ERROR;
while (*ERROR*$>\varepsilon$) **do**
1: *compare your heap root with the other ones*
in order to find the maximum error estimate;
2: **if** (*maximum error estimate is in the heap root*) **then**
send the subdomain in the heap root to all processors;
else
receive the subdomain with maximum error estimate;
endif
3: *share such a subdomain with the other processors,*
compute the approximate values of If and Ef
in the new 2^{d-s} *subdomains and reorder the heap;*
4: *update ERROR;*
endwhile
compute RESULTS;

Steps 1, 2 and 4 are communication steps. As previously stated, to optimize them we used a *cascade scheme* which has a computational cost of $\log_2 p$. Further, we used four heaps for each processor, to store the error estimate, the integral estimate, a vertex of the subdomains (the closest one to the origin) and its side length, respectively.

The first step of the algorithm exchanges the maximum error estimate and the processor identifier among the processors. On an MIMD distributed-memory multiprocessor with hypercube topology and p processors this step has a computational cost of

$$2\, t_{comm} \log_2 p$$

where t_{comm} is the time spent in communication when a single-precision real number is exchanged between two processors. In the second step of the algorithm the processor which has the subdomain with maximum error estimate sends such a subdomain to the other ones. In this communication step the vertex of the subdomain (a d long vector) and the side length are sent. Thus, such step has a computational cost of

$$(d+1)\, t_{comm} \log_2 p$$

In the third step the subdomain with maximum error estimate is shared among the processors. This step is dominated by the computation of the estimate of If and Ef in $2^d/p = 2^{d-s}$ new subdomains. Since the composite Simpson's product rule in d dimensions needs 5^d function evaluations, the computational cost of this step is

$$2^{d-s} 5^d$$

integrand function evaluations. This step does not require any kind of communication. Finally, for the fourth step, only a *cascade sum* is necessary in order to compute the new error estimate in the whole integration domain. Then this step has a computational cost of

$$(t_{comm} + t_{calc}) \log_2 p$$

where t_{calc} is the time spent in a floating point calculation.

Now we note that steps 1, 2 and 4 of the algorithm depend on the processor number and implement overall communication among the processors. Thus the efficiency will decay when the processor number grows, while it will increase with the space dimension d, because steps 1, 2 and 4 become negligible compared to step 3, depending on the space dimension. In Figure 1 there is a graphical description of the previous algorithm for the 2-dimensional case using four processors.

4. COMPUTATIONAL ENVIRONMENT AND IMPLEMENTATION ISSUES

To develop the previous algorithm we used the Fortran version of EXPRESS. This is a flexible communication environment based on the CrOS III routines, which allow communication between arbitrary pairs of processors without specification about the message path. To do this EXPRESS assigns to all processors a *processor identifier* which is used, in the routines arguments list, to specify the source and the destination of the messages. Thus the user code can be virtually identical for every computer which supports the same communication environment. EXPRESS supplies routines for sending and receiving messages, synchronizing the processors and collecting data from processors in a user-specified way. In addition, EXPRESS supplies an I/O routine set which allows

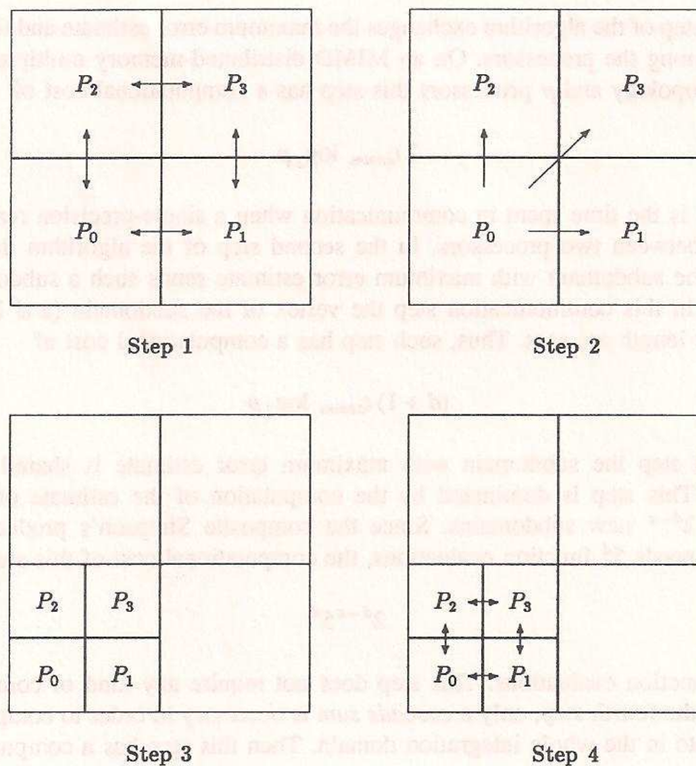

Figure 1. Algorithm graphical description

every processor to access the host computer operating system utilities (CUBIX) and a parallel graphics system (PLOTIX)[11].

In particular we used the EXPRESS routines `KXCOMB` to implement algorithm steps 1 and 4, and the routine `KXBROD` for step 2.

To test the efficiency of our algorithm, several experiments are carried out on two different and widely known multiprocessors: an Ncube/ten and a Meiko Computing Surface. These multiprocessors have quite different architectures. The Ncube/ten is a hypercube of 1024 relatively slow processors (about 0.1 Mflops at peak performance). Each of them is a proprietary Ncube chip with an architecture comparable to a VAX 11/780 with a floating-point unit, and 512 Kbytes of local memory. Every node is connected to ten other nodes by a communication channel of 4 Mbytes per second. The Meiko Computing Surface is a network of faster Inmos T800 Transputer (about 1.5 Mflops at peak performance) with 4 Mbytes of local memory per node. Each transputer has four channels providing a communication bandwidth of 10 Mbytes per second. In the Meiko Computer Surface there is an electronic switching mechanism which allows these message channels to be connected in a software-defined topology.

5. NUMERICAL RESULTS

To give an idea of the application range of this algorithm, we integrate the following kinds of integrand function over the unitary cube in two, three and four dimensions, which are classical examples of test problems for the multidimensional quadrature[12]:

f_1: oscillating function
f_2: function with a singularity near the boundary of the integration domain
f_3: function with singular first derivative
f_4: Gaussian-like function.

In two dimensions we carried out mainly graphical tests using PLOTIX in order to show how the algorithm is able to locate the areas in which it is hard to integrate the function in the unit square $[0,1]^2$ and to share them among the processors. Figures 2–5 show the results using four processors and requiring tolerances between 2.5×10^{-4} and 5×10^{-5}. The Figures show that the work of the processors depends on the integrand function type. For example, in Figure 3 we note that, in order to integrate the function f_2 in the square $[0,1]^2$, more subdomain divisions (represented by decreasing-side squares) are required near the singularity at (0.5, −0.1). Furthermore it is observable that the workload is well distributed among the processors as it is shown by the shading in the picture. Analogous considerations can be applied for the other functions.

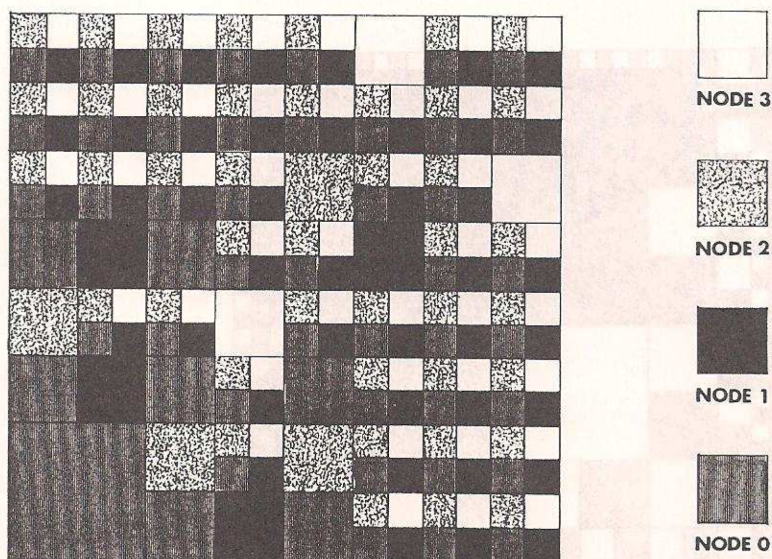

Figure 2. 2-dimensional case: domain decomposition for the function f_1 = sin cos 15xy; required tolerance: 2.5×10^{-4}

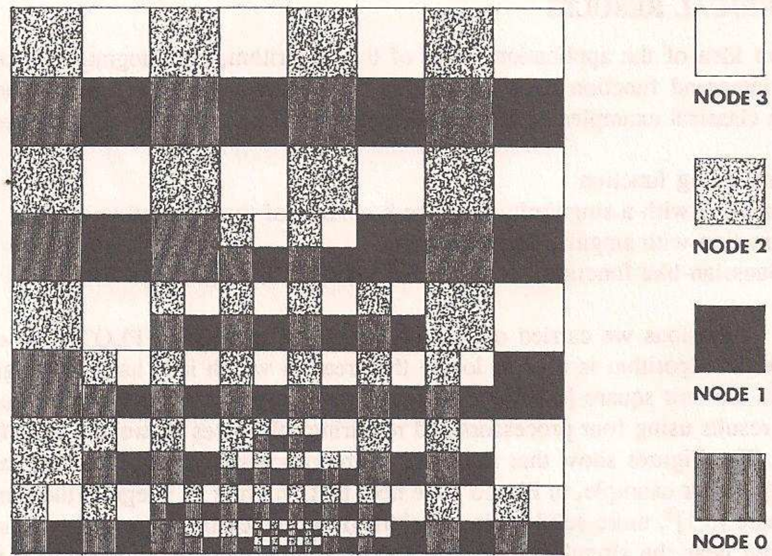

Figure 3. *2-dimensional case: domain decomposition for the function* $f_2 = 1/((x - 0.5)^2 + (y + 0.1)^2)$*; required tolerance:* 5×10^{-5}

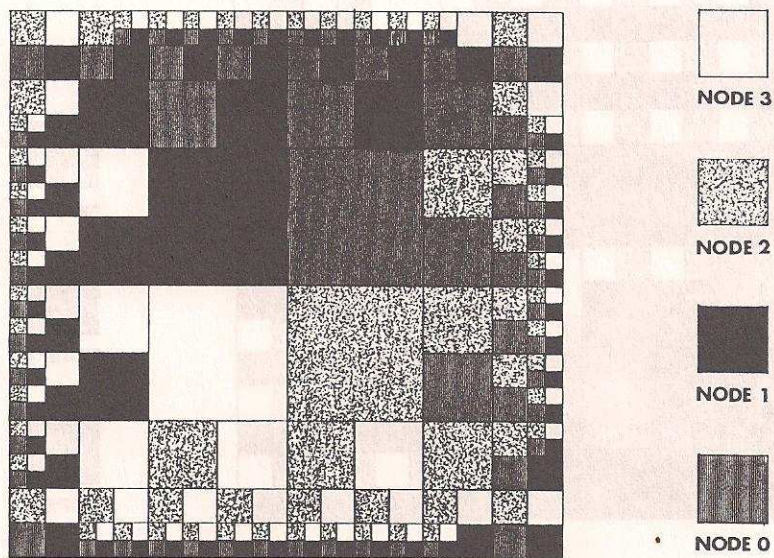

Figure 4. *2-dimensional case: domain decomposition for the function* $f_3 = \sqrt{[xy(1 - x)(1 - y)]}$*; required tolerance:* 2.5×10^{-5}

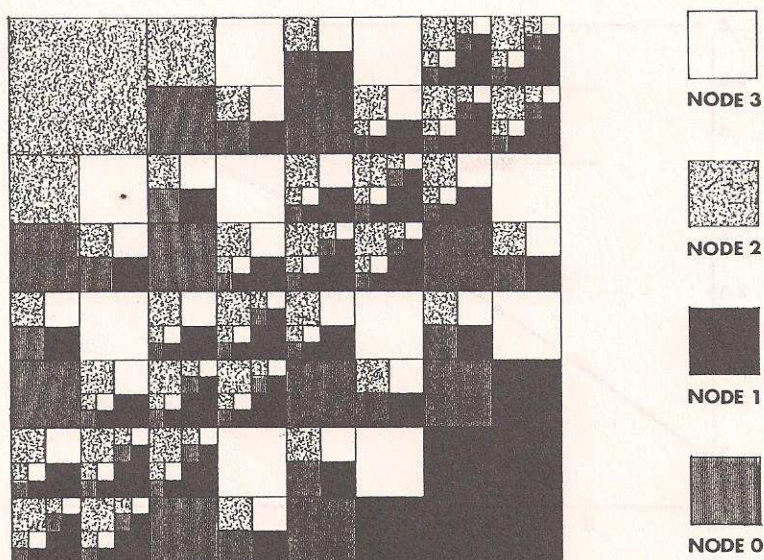

Figure 5. 2-dimensional case: domain decomposition for the function $f_4 = e^{-50(x-y)^2}$*; required tolerance:* 2.5×10^{-5}

Figures 6–8 refer to the 3-dimensional case for the function

$$f_3 = \sqrt{[xyz(1-x)(1-y)(1-z)]}$$

They report respectively speed-up, efficiency and execution time, respectively, on the Ncube/ten and on the Meiko Computing Surface, requiring a tolerance $\varepsilon = 7.5 \times 10^{-4}$. Similar results are obtained for the other functions. Figure 9 shows the integrand function evaluation number as a function of various required tolerances for all the test functions.

Likewise, for the 4-dimensional case, Figures 10–12 show speed-up, efficiency and execution time to integrate the function

$$f_3 = \sqrt{[xyzw(1-x)(1-y)(1-z)(1-w)]}$$

on the two computers considered, requiring a tolerance $\varepsilon = 10^{-3}$, while Figure 13 shows the integrand evaluation number as a function of various required tolerances.

From these results we note that our algorithm fully exploits the intrinsic parallelism of the multidimensional quadrature problem, because it is able to keep the processors busy and to balance the workload. This can be noted also from the quite good speed-up and efficiency values as shown in Figures 7, 8, 11 and 12. Further, our expectation about the efficiency behaviour, based on the algorithm analysis, is validated: the efficiency in Figures 8 and 12 increases with the space dimension, whereas it decreases when the number of processors grows.

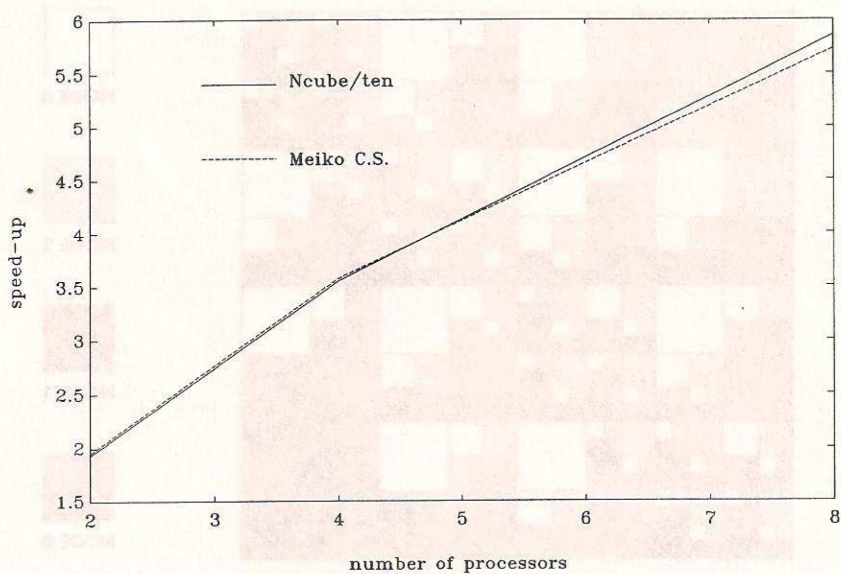

Figure 6. *3-dimensional case: speed-up as function of the number of processors for the function* $f_3 = \sqrt{[xyz(1-x)(1-y)(1-z)]}$; *required tolerance* $\varepsilon = 7.5 \times 10^{-4}$

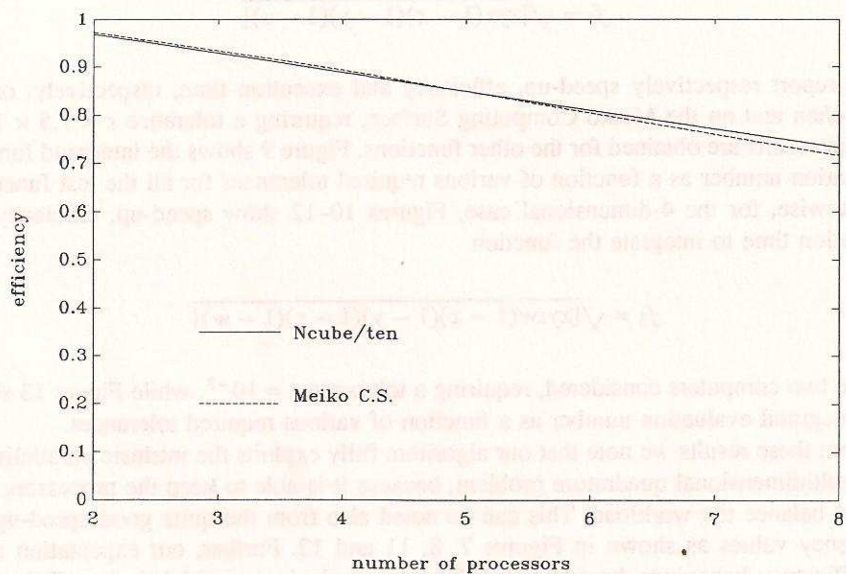

Figure 7. *3-dimensional case: efficiency as a function of the number of processors for the function* $f_3 = \sqrt{[xyz(1-x)(1-y)(1-z)]}$; *required tolerance* $\varepsilon = 7.5 \times 10^{-4}$

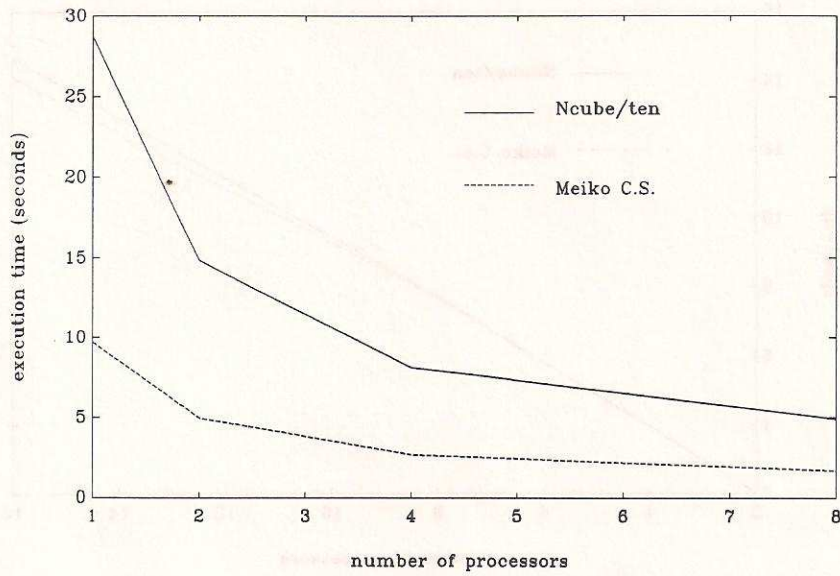

Figure 8. *3-dimensional case: execution time as a function of the number of processors for the function* $f_3 = \sqrt{[xyz(1-x)(1-y)(1-z)]}$*; required tolerance* $\varepsilon = 7.5 \times 10^{-4}$

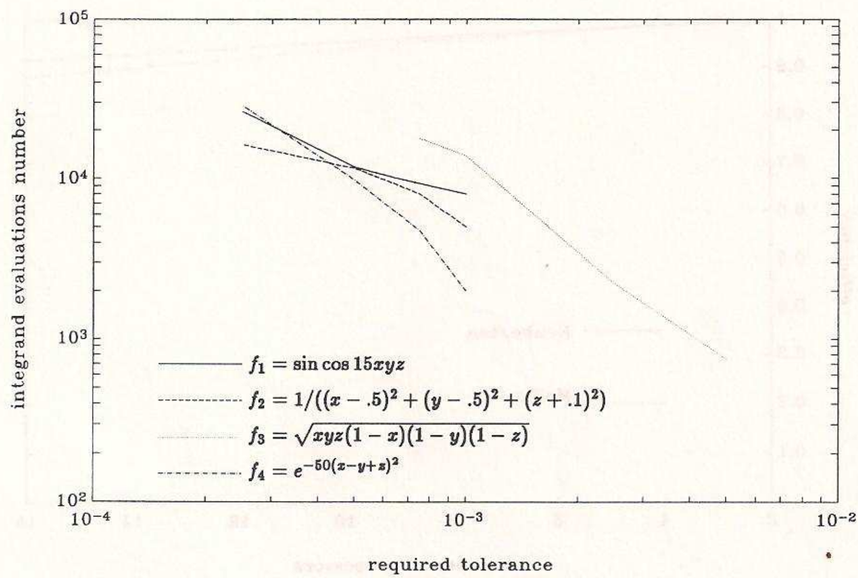

Figure 9. *3-dimensional case: integrand evaluations number as function of various required tolerances*

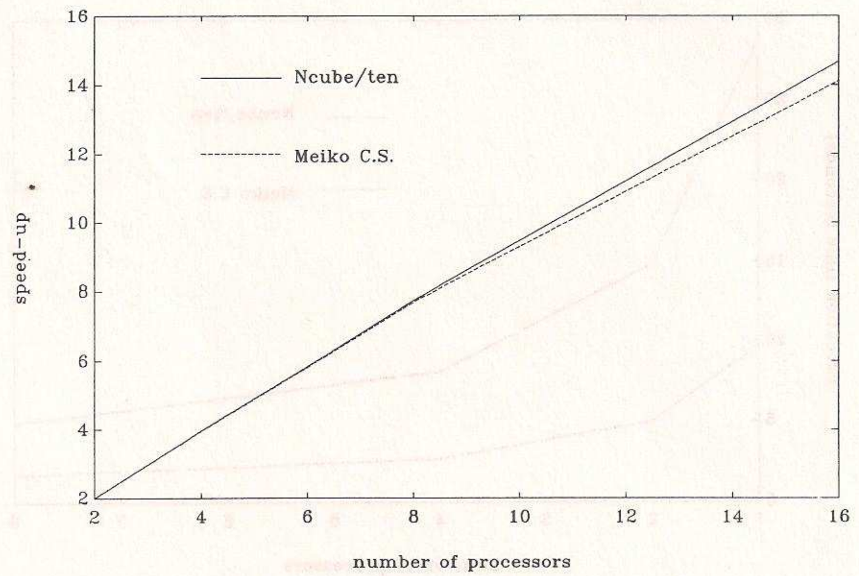

Figure 10. *4-dimensional case: speed-up as a function of the number of processors for the function* $f_3 = \sqrt{[xyzw(1-x)(1-y)(1-z)(1-w)]}$; *required tolerance* $\varepsilon = 10^{-3}$

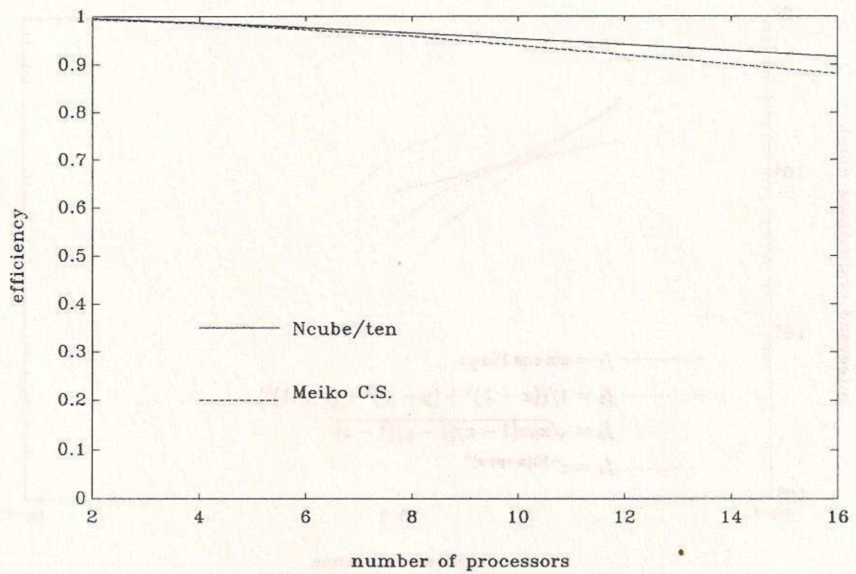

Figure 11. *4-dimensional case: efficiency as a function of the number of processors for the function* $f_3 = \sqrt{[xyzw(1-x)(1-y)(1-z)(1-w)]}$; *required tolerance* $\varepsilon = 10^{-3}$

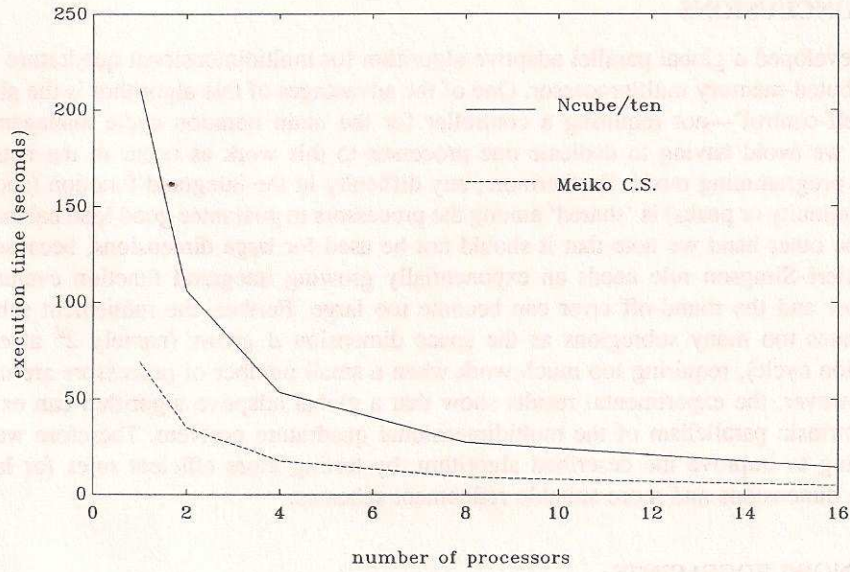

Figure 12. 4-dimensional case: execution time as a function of the number of processors for the function $f_3 = \sqrt{[xyzw(1-x)(1-y)(1-z)(1-w)]}$; *required tolerance* $\varepsilon = 10^{-3}$

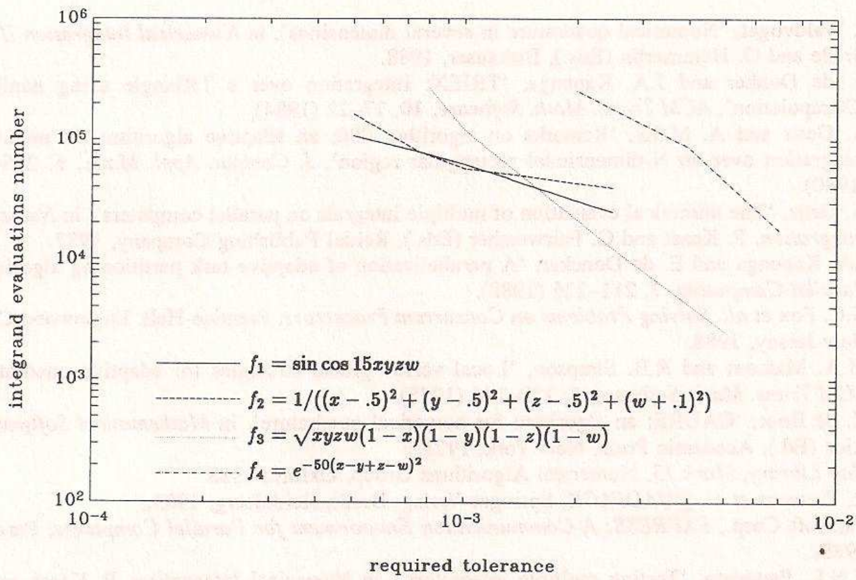

Figure 13. 4-dimensional case: integrand evaluations number as function of various required tolerances

6. CONCLUSIONS

We developed a global parallel adaptive algorithm for multidimensional quadrature on a distributed-memory multiprocessor. One of the advantages of this algorithm is the ability to 'self-control'—not requiring a controller for the main iteration cycle management. Thus we avoid having to dedicate one processor to this work as occur in the master–slave programming mode. Furthermore, any difficulty in the integrand function (such as discontinuity or peaks) is 'shared' among the processors to guarantee good load balancing. On the other hand we note that it should not be used for large dimensions, because the Cavalieri–Simpson rule needs an exponentially growing integrand function evaluation number and the round-off error can become too large. Further, the refinement scheme generates too many subregions as the space dimension d grows (namely 2^d at every iteration cycle), requiring too much work when a small number of processors are used.

However, the experimental results show that a global adaptive algorithm can exploit the intrinsic parallelism of the multidimensional quadrature problem. Therefore we are working to improve the described algorithm, by testing more efficient rules for larger space dimensions and more suitable refinement schemes.

ACKNOWLEDGEMENTS

This work was partially supported by a grant of the Consiglio Nazionale delle Ricerche (Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo). We also thank Dr. Paul Messina for making available the computational resources of Caltech Concurrent Supercomputing Facilities.

REFERENCES

1. J. Waldvogel, 'Numerical quadrature in several dimensions', in *Numerical Integration III*, H. Braße and G. Hämmerlin (Eds.), Birkäuser, 1988.
2. E. de Donker and J.A. Kapenga, 'TRIEX: Integration over a TRIangle using nonlinear EXtrapolation', *ACM Trans. Math. Software*, **10**, 17–22 (1984).
3. A. Genz and A. Malik, 'Remarks on algorithm 006: an adaptive algorithm for numerical integration over an N-dimensional rectangular region', *J. Comput. Appl. Math.*, **6**, 295–299 (1980).
4. A. Genz, 'The numerical evaluation of multiple integrals on parallel computers', in *Numerical Integration*, P. Keast and G. Fairweather (Eds.), Reidel Publishing Company, 1987.
5. J.A. Kapenga and E. de Doncker, 'A parallelization of adaptive task partitioning algorithm', *Parallel Computing*, **7**, 211–226 (1988).
6. G.C. Fox *et al.*, *Solving Problems on Concurrent Processors*, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.
7. M.A. Malcom and R.B. Simpson, 'Local versus global strategies for adaptive quadrature', *ACM Trans. Math. Software*, **1**, 129–146 (1975).
8. C. de Boor, 'CADRE: an algorithm for numerical quadrature', in *Mathematical Software*, J. Rice (Ed.), Academic Press, New York, 1971.
9. *Nag Library, Mark 13*, Numerical Algorithms Group, Oxford, 1988.
10. R. Piessens *et al. QUADPACK*, Springer-Verlag, Berlin Heidelberg, 1983.
11. Parasoft Corp., *EXPRESS: A Communication Environment for Parallel Computers*, Parasoft, 1988.
12. T.N.L. Patterson, 'Testing multiple integrators', in *Numerical Integration*, P. Keast and G. Fairweather (Eds.), Reidel Publishing Company, 1987.