Int J Parallel Prog (2012) 40:397–409
DOI 10.1007/s10766-011-0191-4

A Double Adaptive Algorithm for Multidimensional Integration on Multicore Based HPC Systems

Giuliano Laccetti · Marco Lapegna · Valeria Mele · Diego Romano · Almerico Murli

Received: 20 January 2011 / Accepted: 18 November 2011 / Published online: 6 December 2011

Abstract In this work, a parallel double adaptive algorithm for the computation of a multidimensional integral on multicore based multicomputer systems is described. This new algorithm is the revision of a procedure developed by one of the present authors for multicomputer systems, with the aim to introduce features for an efficient implementation in multicore based hierarchical environments. Two different adaptive strategies have been combined together in the algorithm: a first procedure is responsible for load balancing among the system nodes and a second one is responsible for coordinating the cores within a single node. The performance is analyzed and experimental results on a Blade Server with 8 nodes and 2 quad-core CPUs per node have been achieved.

Keywords Multicomputer system · Multicore node · Hierarchical environment · Multidimensional integration · Parallel adaptive algorithm

G. Laccetti · M. Lapegna (✉) · V. Mele
Department of Mathematics and Applications, University of Naples Federico II,
Via Cintia Monte S.Angelo, 80126 Naples, Italy
e-mail: marco.lapegna@unina.it

G. Laccetti
e-mail: giuliano.laccetti@unina.it

V. Mele
e-mail: valeria.mele@unina.it

D. Romano
ICAR-CNR, Via Pietro Castellino 111, 80131 Naples, Italy
e-mail: diego.romano@na.icar.cnr.it

A. Murli
SPACI c/o Department of Mathematics and Applications, University of Naples Federico II,
Via Cintia Monte S.Angelo, 80126 Naples, Italy
e-mail: almerico.murli@unina.it

1 Introduction

From an architectural point of view, a multicore based High Performance Computing system can be described by means of a hierarchical two-level structure: at the highest level there are several nodes based on multicore CPUs connected among them by dedicated fast networks or high performance switches (Node level) and at the lowest level there are several computing elements, the cores, sharing resources in a single CPU (Core level). In the following we consider an homogeneous system, composed by NP nodes P_h $(h = 1, \ldots, NP)$ each of them with NC cores C_{hk} $(k = 1, \ldots, NC)$. These two architecture levels have very different features and they require different algorithmic development methodologies. The nodes in a multicomputer system have a private memory and they communicate only by means of explicit message passing. On the other hand in a single node the cores cannot be considered as separate computing units because they share both on-chip resources like caches, as well as the external main memory that is used to exchange data among the cores [5]. For such a reason, the development of algorithms and scientific software for multicomputer systems based on multicore CPUs implies a suitable combination of several methodologies to deal with the different kinds of parallelism corresponding to each architectural level (distribution of the computation among the nodes of the system, as well as among the cores of a single node), so that such kind of algorithms are often called *Hybrid* or *Hierarchical Algorithm*. The development of such hybrid and hierarchical algorithms, able to be aware of the underlying topology, is one of the recommended action in the research agenda for the next generation HPC systems [6].

The aim of this work is the development of a new algorithm for the computation of multidimensional integrals on these innovative computing systems. The resulting algorithm is a deep revision of an already existing algorithm, originally developed by one of the present authors for multicomputer systems with traditional CPUs [4,10]. Computational kernels for this problem are on the basis of several applications in fields like quantum chemistry, high energy physics and computational finance, and several techniques and methodologies are available to this aim (e.g. [9]). In this paper we pay special attention to the class of the parallel adaptive algorithms, because they are able to achieve high accuracies with a reasonable computational cost, so that they are on the basis of several mathematical software libraries and routines. Furthermore at the moment of writing this document we have not found significant works for implementing such a class of algorithms on HPC systems with multicore CPUs.

The development of parallel adaptive algorithms for multidimensional integration is a challenging task, because in such algorithms the execution is strongly dependent from the input data (requested tolerance, integrand function, integration domain, …), so that it is possible to balance the workload among the computing units only by using a run time approach, with a significant cost in terms of communications and synchronizations. Considering all these facts, we propose an algorithm where two workload distribution strategies are combined together with the aim to reduce synchronization points and communications among the computing units: a first one is responsible for task scheduling in the cores of a single node according to the availability of the local resources; a second one periodically rearranges the tasks among the computing nodes of the system to balance the workload.

The document is then structured as follow: after a brief introduction of a general framework used to describe the adaptive algorithms for multidimensional integration in Sect. 2, in Sect. 3 we describe a hierarchical adaptive algorithm for multicore based HPC systems. Therefore, in Sect. 4 we analyze the performance of the algorithm by means of a model based on a estimate of the communication and synchronization overhead and, in Sect. 5, we describe the tests conducted on a Blade Server with nodes equipped with 2 quad-core CPUs. Finally we report conclusions and future works.

2 Adaptive Algorithms for Multidimensional Integration

Given $U = [a_1, b_1] \times \cdots \times [a_d, b_d] \subset R^d$ a d-dimensional hyper-rectangular region, an adaptive algorithm for the numerical computation of a multidimensional integral:

$$I(f) = \int_U f(t_1, \ldots, t_d) dt_1 \cdots dt_d \tag{1}$$

is an iterative procedure that, at each iteration j, process the subdomains $s_i^{(j)}$ $(i = 1, \ldots, K)$ of a partition $\wp^{(j)}$ of U, where K is the number of subdomains in the partition. More precisely, at each iteration j the algorithm computes an approximation $Q^{(j)}$of $I(f)$ and an error estimate $E^{(j)}$ of the absolute error $|I(f) - Q(f)|$ by means of composite quadrature rules defined on the subdomains of $\wp^{(j)}$. Since the convergence properties of the composite quadrature rules, the sequence $Q^{(j)}$ approaches $I(f)$ and the sequence $E^{(j)}$ approaches zero. Therefore the procedure is repeated until:

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

where ε is a user-requested tolerance. Alternative stopping criterion are based on allowed bounds on the number of function evaluations or on the number of iterations. The last computed approximation $Q^{(j)}$ is the result of the algorithm. For dimension up to $d = 15$ are available efficient rules for standard regions [3] as well reliable procedures for the error estimate $e_i^{(j)}$ in the subdomains $s_i^{(j)}$ [1]. Since the convergence rate of this procedure depends on the analytical properties of the integrand function (presence of peaks, singularities, oscillations, etc), in order to reduce as fast as possible the error $E^{(j)}$, at the generic iteration j, the subdomain $\hat{s}^{(j-1)} \in \wp^{(j-1)}$ with maximum absolute error estimate $\hat{e}^{(j-1)} = \max_{s_i^{(j-1)} \in \wp^{(j-1)}} e_i^{(j-1)}$ is split in two parts s_λ and s_μ that take the place of $\hat{s}^{(j-1)}$ in the new partition $\wp^{(j)}$. Consequently the partition $\wp^{(j)}$ is updated through $\wp^{(j)} = \wp^{(j-1)} - \left\{\hat{s}^{(j-1)}\right\} \cup \left\{s_\lambda, s_\mu\right\}$. In a similar way the approximations $Q^{(j)}$ and $E^{(j)}$ over U are updated by using the quadrature rule and the error estimates evaluated in the new subdomains s_λ and s_μ. The described refinement procedure is generally called global adaptive algorithm for multidimensional integration, and more general details can be found in [9].

Let us now focus on some implementation issues of a global adaptive algorithm. In this algorithm, at each iteration j, it is necessary to arrange all the subdomains $s_i^{(j)} \in \wp^{(j)}$ in a suitable data structure able to supply the subdomain with maximum error estimate with the lowest computational cost. In this sense, a widely used data structure is a partially ordered binary tree called *heap*. For a heap with K elements, the management steps have a computational cost equal to $\log_2(K)$ access to the memory. This cost is much smaller than the cost of evaluating a multidimensional integration rule, mainly for large values of the space dimension d. Actually a multidimensional integration rule has a cost typically equal to $\gamma_1 d^4$ integrand function evaluations where $0.1 < \gamma_1 < 1$ is a real constant (see e.g. [3]), each of them having a cost of at least d floating point operations. Consequently we can assume that the computational cost of a multidimensional quadrature rule is $\gamma_1 d^5$ floating-point operations.

3 A Parallel Double Adaptive Algorithm

We start this section with a short description of the adaptive algorithm for multidimensional integration on MIMD distributed memory multicomputers developed by one of the present authors in [4]. The algorithm is based on a Parallelism at Subdomain Level approach [9], where the subdomains of $\wp^{(j)}$ are distributed in h subpartitions $\wp_h^{(j)}$, one for each node P_h $(h = 1, \ldots, NP)$, so that it is possible to process them concurrently. The main problem of this approach is that the sequence of the subdomains is unpredictable because it depends on the analytical features of the integrand function. Therefore, in order to avoid the splitting on unimportant subdomains with small error and to ensure a proper load balancing, the algorithm uses a *dynamic load balancing* strategy based on a periodic redistribution of the subdomains $\hat{s}_h^{(j-1)}$ with largest error $\hat{e}_h^{(j-1)}$ among the nodes P_h, connected in a 2-dimensional periodical mesh. The described procedure is repeated until a node stopping criterion is satisfied in each node. For example it is possible to use a local stopping criterion, without global communications, based on the allowed number of iterations in each node P_h or based on the stopping criterion (2) scaled on the total volume of the subdomains $s_h^{(j)} \in \wp_h^{(j)}$:

$$E_h^{(j)} = \sum_{s_i^{(j)} \in \wp_h^{(j)}} e_i^{(j)} < \varepsilon \frac{[vol(\wp_h^{(j)})]}{[vol(U)]}$$

More details can be found in [4]. Algorithm 1 represent the described algorithm by using the Single Program Multiple Data (SPMD) model. In each nodeP_h, the steps a), b) and c) balance the workload by exchanging the subdomains with maximum error estimate among other nodes, and the step d) updates the partial results. This algorithm has been chosen for the implementation of the mathematical software subroutine D01FAFP in the NAG Parallel Library [11].

Algorithm 1 : procedure executed by each process in the node P_h at the Node Level

```
Initialize $\wp_h^{(0)}$, $Q_h^{(0)}$ and $E_h^{(0)}$
while (node stopping criterion not satisfied) do iteration $j_1$
    a) define 2 connected nodes $P_{h+}$ and $P_{h-}$ in one of the two directions of the
       2-dimensional periodical grid
    b) if ($\hat{e}_h > \hat{e}_{h+}$) send $\hat{s}_h$ to $P_{h+}$
    c) if ($\hat{e}_{h-} > \hat{e}_h$) receive $\hat{s}_h$ from $P_{h-}$
    d) split $\hat{s}_h$ in two subdomains $s_\lambda$ and $s_\mu$ and update $Q_h^{(j_1)}$ and $E_h^{(j_1)}$ on $\wp_h^{(j_1)}$
endwhile
compute $Q(f) = \sum_h Q_h^{(j_1)}$ and $E(f) = \sum_h E_h^{(j_1)}$
```

In the following we describe how to exploit the parallelism among the cores of a single CPUs, in the step d) of Algorithm 1. At the Core Level level, each node P_h has NC cores C_{hk} $(k = 1, \ldots, NC)$, that cannot be considered as completely independent processing units, because they share resources such as caches, main memory or I/O devices. These hardware features outline a scenario where several computing units need to access shared resources, so that a natural programming model for an efficient use of these devices is based on the Thread Level Parallelism, where each thread is assigned to a single core. The issues related to the use of these devices have been extensively studied for other problems [2], so that it is possible to identify some common properties of the algorithms in order to achieve the highest performances from these devices:

- Fine granularity: by splitting the computation in small tasks and by increasing the ratio of floating point computation on data movement, the cores can reuse data in their caches while reducing the number of memory accesses through the shared bus;
- Asynchronicity: by reducing the synchronization points, even a large number of threads can achieve good performance, because idle times are eliminated.

The easiest way to parallelize step d) of Algorithm 1 is based on the fair distribution of the function evaluations of the quadrature rules applied to s_λ and s_μ among the threads assigned to the cores C_{hk}, because a quadrature rule is a weighted summations of independent function evaluations. After the computation of the partial sum in each thread, only one global synchronization point occurs to evaluate the final sum. This approach is referred as Parallelism at Integration Formula Level [9] and it exploits the parallelism only at a very low level, because it produces a very synchronous execution: parallel tasks (evaluations of the integration function) are interleaved by sequential ones (error estimations, integral approximations and management of the data structures), with large intervals of idle time. More generally the Parallelism at Integration Formula Level is a special case of the *fork-join approach*, that it is known as a very inefficient approach for parallel algorithms in multicore environments also in other fields [2]. For these reasons, our strategy to introduce parallelism a the Core Level is based on the concurrent execution of a new adaptive procedure by several threads, each of them executed by a core. Therefore, the threads assigned to the cores C_{hk} of the node P_h can refine the subdomains of $\wp_h^{(j)}$ independently of the threads of the other nodes. However it should be noted that the data structure used to store the subdomains $s_i^{(j)} \in \wp_h^{(j)}$ and the values $Q_h^{(j)}$ and $E_h^{(j)}$ resides in the main memory of P_h, and they have to be accessed by all the threads in the steps in charge of the data structure management, so that they have to be executed in critical sections because of the risk of race condition. These are the only critical sections in the procedure, and the computational cost of such steps is smaller than the cost of the multidimensional integration rule that are concurrently evaluated by the threads, as stated at the end of the Sect. 2. In any case these critical sections are aimed only to avoid the risk of race condition when two or more threads are accessing the shared data and it cannot considered a global synchronization point as in fork-join approach. Furthermore we remark that this strategy has a very favorable ratio of floating point computation on data movement, because there are several symmetrical rules defined over regular regions, like the cube or the sphere, that are identified by very few data [3]. For example, a d-dimensional hyperrectangular region is described by its center and the dimensions of the edges, that is only two array with d elements that must be taken from the memory, compared with the much larger computational cost equal to $\gamma_1 d^5$ floating point operations required by a multidimensional quadrature rule. Such a favorable ratio produces a better use of the cache memories with a reduced number of accesses to the main memory.

The described redistribution procedure is repeated until a core stopping criterion is satisfied in each thread. More precisely it is possible to use a local stopping criterion, without global synchronizations, similar to the ones described for the parallelization at the Node Level.

Algorithm 2 : procedure executed by each thread in the core C_{hk} at the Core Level

```
Initialize $\wp_h^{(0)} = \wp_h^{(j_1)}$, $Q_h^{(0)} = Q_h^{(j_1)}$ and $E_h^{(0)} = E_h^{(j_1)}$
while (core stopping criterion not satisfied) do iteration $j_2$
    {enter critical section}
    1) pick up $\hat{s}^{(j_2-1)} \in \wp_h^{(j_2-1)}$ with maximum error estimate
    {exit critical section}
    2) divide $\hat{s}^{(j_2-1)}$ in two parts $s_\lambda$ and $s_\mu$
    3) evaluate the quadrature rules and compute error estimates in $s_\lambda$ and $s_\mu$
    {enter critical section}
    4) insert $s_\lambda$ and $s_\mu$ in $\wp_h^{(j_2)}$
    5) update $Q^{(j_2)}$ and $E^{(j_2)}$
    {exit critical section}
endwhile
update $\wp_h^{(j_1)} = \wp_h^{(j_2)}$, $Q_h^{(j_1)} = Q_h^{(j_2)}$ and $E_h^{(j_1)} = E_h^{(j_2)}$
```

Algorithm 2 is the adaptive procedure described for the Core Level in a hierarchical computational environment and it is the refinement of Step d) in Algorithm 1. It is replicated in each core C_{hk} of P_h as a thread function. where each thread start its execution from data in the subpartition $\wp_h^{(j_1)}$, defined after the workload redistribution in the Algorithm 1. The main loop is in charge for tasks distribution among the cores in a single node. More precisely each thread access the data structure containing the subdomains to be process independently of the other ones, according to the availability of the local resources. In any case it should be noted that the critical sections in steps 1), 4) and 5), necessary to avoid race conditions on the shared data, does not include the much more expensive evaluation of the multidimensional quadrature rule in steps 2) and 3), with a significant increase of the concurrency in the algorithm and a reduction of the idle time in the threads executions. We conclude this section by remarking that the obtained algorithm is not a simple revision of the original algorithm in a hybrid programming environment (MIMD distributed/shared memory systems), but a new hierarchical algorithm obtained by the combination of two adaptive strategies for different architectures: a first strategy described for the Node Level (Algorithm 1) and a second one for the Core Level (Algorithm 2). Therefore we can define the new algorithm as a *Parallel Double Adaptive Algorithm for Multidimensional Integration.*

4 Performance Analysis

To evaluate the performance of the algorithm, we use the well known Scaled Speed-up introduced by Gustafson in [8], where it is proposed that, mainly for a large number of processing units, the Speed-up should not be measured taking a fixed-size problem and run it on various numbers of processors, but scaling the problem size to the number of processing units. In other words it is remarked that, when measure Speed-up, it is most realistic to assume that the computational cost in each processing unit, not the problem size, is constant when the number of processing units increases. Now we remember that for a multidimensional quadrature algorithm the computational cost is usually measured by using the number of function evaluations, so that it is possible to define $T(N; F)$ as the elapsed time to compute (1) with F integrand function evaluations on N processing units. From [8], the classical definition of Scaled Speed-up is:

$$SS_N = \frac{T(1; N \cdot F)}{T(N; N \cdot F)} \tag{3}$$

The ideal value is $SS_N = N$ but in practice a slight degradation is acceptable. In order to assess a real model for SS_N, in [8] the computing time $T(N; N \cdot F)$ is decomposed into:

$$T(N; N \cdot F) = T_S + T_C$$

where T_C is the part of the algorithm that can be decomposed in N fully independent tasks and T_S is the part of the algorithm that cannot be parallelized and that it contains the communication and synchronization steps of the algorithm. Therefore it is possible to define the impact of T_S on $T(N; N \cdot F)$ by means of the serial fraction $\sigma = T_S/T(N; N \cdot F)$, that represents a measure of the overhead introduced parallelizing the algorithm. From the above follows:

$$\begin{aligned} T(1; N \cdot F) = T_S + N \cdot T_C = \sigma\,(T(N; N \cdot F) - N \cdot T(N; N \cdot F)) \\ + N \cdot T(N; N \cdot F) \end{aligned}$$

Substituting into the definition of SS_N in (3), a model for the Scaled Speed-up in term of the serial fraction σ is achieved:

$$\overline{SS_N} = \frac{T(1; N \cdot F)}{T(N; N \cdot F)} = N - \sigma(N-1) \tag{4}$$

We conclude this section with an estimate of the serial fraction σ in the case of our double adaptive parallel algorithm introduced in Sect. 3. To this aim we remark that Algorithm 1 is based on a iterative procedure, where we say Nit_1 total the number of iterations. Then, at each iteration, steps a), b) and c) redistribute the subdomains with maximum error estimate among the nodes connected in a 2-dimensional periodical grid without global communications. Since the subdomains are described by means 2 arrays with dimension d, the communication cost of such steps is $\tau_1 = 2d\ tcom$ where $tcom$ is the time spent for the communication of a word between two nodes. Then, each node P_h generates NC concurrent threads in step d), where Algorithm 2 is executed as thread function. Also Algorithm 2 is based on a iterative procedure where we say Nit_2 the number of iterations. In such iterative procedure, steps 1), 4) and 5) manage the data structures required for the subdomains to be process, so that, as stated in Sect. 3, they must to be executed in two critical sections in order to avoid race conditions on the shared data structure. These steps are dominated by step 1), where it is necessary to sort the heap with a cost equal to $\log_2(K)$ where K is the number of items in the heap. Since at each iteration, in steps 1), 4) and 5) the number of subdomains in the heap increases by one, each thread traverses the critical section in $\log_2(Nit_2)tmem$, where $tmem$ is the memory access time. Then we remark that steps 1), 4) e 5) are executed in critical sections, so that the 8 threads in each single node have, as worst case, a total cost of $\tau_2 = 8\log_2(Nit_2)tmem$. Finally steps 2) and 3) evaluate the quadrature rule in two new subdomains s_λ and s_μ, and they dominate the computational cost of the algorithm. These steps have computational cost of $\tau_3 = \gamma_1 d^5 tcal$ where $tcal$ is the time for a floating point operation, and they are the only steps that we consider run in parallel by the threads. Then we assume $tcal$ as a measure unit, and we introduce the two reasonable assumptions:

$$\frac{tcom}{tcal} = \gamma_2 10^3 \qquad \frac{tmem}{tcal} = \gamma_3 10^2$$

where $0.1 < \gamma_i < 1 \quad i = 2, 3$ are real constant. Therefore we achieve:

$$\tau_1 = 2d\gamma_2 10^3 \qquad \tau_2 = \gamma_3 \log_2(Nit_2)10^2 \qquad \tau_3 = \gamma_1 d^5$$

and, from the definition of serial fraction, we have:

$$\begin{aligned}\sigma &= \frac{T_S}{T(N; N \cdot F)} = \frac{\tau_1 + Nit_2\tau_2}{\tau_1 + Nit_2(\tau_2 + \tau_3)} \\ &= \frac{2d\gamma_2 10^3 + Nit_2(\gamma_3 8\log_2(Nit_2)10^2)}{2d\gamma_2 10^3 + Nit_2(\gamma_3 \log_2(Nit_2)10^2 + \gamma_1 d^5)}\end{aligned} \tag{5}$$

5 Test Results

In this section we present the experimental results achieved on a DELL blade server with 8 nodes, each of them equipped with 2 Intel Xeon quadcore CPU (Harpertown) and 16 GBytes of shared main memory. The nodes are connected by means of a Infiniband network DDR 4X at 16 Gbit/s (Mellanox Technology 25418). In this system we implemented our algorithm in double precision using C and Fortran language, with the libraries Mellanox OFED 1.3.1 (a MPI implementation for the communication among the nodes) and POSIX thread and semaphores (for the synchronization among the cores in a single node).

For the experiments we used a standard procedure based on the well known Genz's package [7]. This package is composed by six different families of functions, each of them characterized by some issues making the problem (1) hard to integrate numerically (peaks, oscillations, singularities..). Each family is composed by 10 different functions where the parameters α_i and $\beta_i$change and related test results are computed (execution time, error,...). Here we report the results for the following three families:

$$\begin{aligned}
\Phi_1 &= \cos\left(2\pi\beta_1 + \sum_{i=1}^{d} \alpha_i x_i\right) && \text{oscillating function}\\
\Phi_2 &= \left(1 + \sum_{i=1}^{d} \alpha_i x_i\right)^{-d-1} && \text{corner peak}\\
\Phi_3 &= \exp\left(-\sum_{i=1}^{d} \alpha_i \left|x_i - \beta_i\right|\right) && C^{(0)} \text{ function}
\end{aligned}$$

where $U = [0, 1]^d$ with dimension $d = 10$. We selected these functions because their different analytical features. However, for other functions in the Genz's package we achieved similar results. In our experiments we measured for each test family:

– The average of the experimental Scaled Speed-up (SS_N) as defined in (3) on the 10 functions of each family
– The estimated Scaled Speed-up ($\overline{SS_N}$) as defined in (4)
– The minimum (MinErr) and the maximum (MaxErr) relative error $|I(f) - Q(f)| /|I(f)$ on the 10 functions of each family

In order to estimate the serial fraction σ in (5), we use $\gamma_i = 1, d = 10$ and $Nit_2 \leq 10,000$, so that it is reasonable to assume $\sigma \approx 0.1$. Finally we remark that in our algorithm we use the Genz and Malik quadrature rule with $\varphi = 1,245$ function evaluations when $d = 10$ so that at each iteration $2\varphi = 2,490$ function evaluations are computed in the two new subdomains s_λ and s_μ.

A first set of experiments is aimed to study the Double Adaptive Algorithm on several cores by using only one node. Therefore such experiments are aimed to test the adaptive strategy at Core Level described in Algorithm 2. In these tests the number of processing units in (3) is the number of cores $N = NC = 1, 2, 4, 8$. To the aim of computing the Scaled Speed-up, the computational cost in each core is $F = 10 \times 10^6$ function evaluations, so that the total number of function evaluations is $Fval = NC \times 10 \times 10^6$ when the number of cores increases. The core stopping criterion in Algorithm 2 is based on the maximum allowed number of iterations in each core $Nit_2 = F/2\varphi = 4{,}016$, while the node stopping criterion in Algorithm 1 is based on only one iteration $Nit_1 = 1$. Figure 1 reports the experimental Scaled Speed-up in (3) for the three families of functions Φ_1, Φ_2 and Φ_3 compared with the model in (4). From the Figure we observe a good scalability when the number of cores increases. The experimental Scaled Speed-up are generally well estimated by the model in (4), mainly with 2 and 4 cores. Actually it can be observed a more evident reduction for SS_N when 8 cores are used. As already remarked in Sect. 3, the evaluation of the multidimensional integration rules are tasks with a favorable ratio of floating point computation on data movement. Then their data can be easily stored in the CPUs caches and reused in the next iterations. When our Parallel Double Adaptive

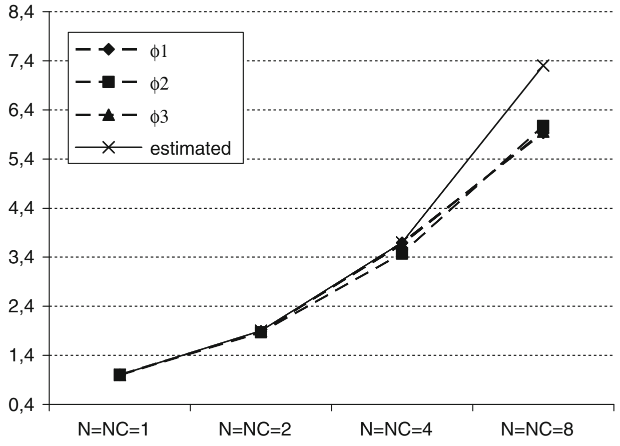

Fig. 1 Scaled Speed-up for the three families of functions ϕ_1, ϕ_2 and ϕ_3 with only 1 node and 1, 2, 4 and 8 cores. The experimental values are compared with the estimated values of (4). The workload in each processing unit is $F = 10 \times 10^6$ when the number of core increases. The average execution times with 1 core for the three families of functions are: Time(ϕ_1) = 1, 67, Time (ϕ_2) = 1.45, Time (ϕ_3) = 1.76

Table 1 The minimum (MinErr) and the maximum (MaxErr) achieved relative error, on the ten functions, when the number of core increases

	$N = NC = 1$	$N = NC = 2$	$N = NC = 4$	$N = NC = 8$
Family Φ_1				
MinErr	0.92 (−9)	0.44 (−9)	0.15 (−9)	0.48 (−10)
MaxErr	0.98 (−7)	0.47 (−7)	0.23 (−7)	0.11 (−7)
Family Φ_2				
MinErr	0.97 (−7)	0.32 (−7)	0.78 (−8)	0.41 (−7)
MaxErr	0.63 (−6)	0.63 (−6)	0.58 (−6)	0.47 (−6)
Family Φ_3				
MinErr	0.24 (−4)	0.40 (−4)	0.29 (−4)	0.40 (−4)
MaxErr	0.11 (−2)	0.73 (−3)	0.55 (−3)	0.46 (−3)

The workload in each processing unit is $F = 10 \times 10^6$

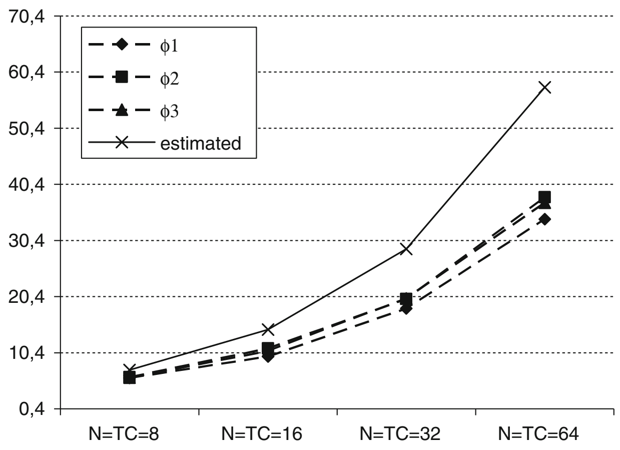

Fig. 2 Scaled Speed-up for the three families of functions ϕ_1, ϕ_2 and ϕ_3 with 1, 2, 4 and 8 nodes and 8 cores for node. The experimental values are compared with the estimated values of (4). The workload in each processing unit is $F = 10 \times 10^6$ when the number of core increases. The average execution times with 1 node for the three families of functions are: Time $(\phi_1) = 2.25$, Time $(\phi_2) = 1.91$, Time $(\phi_3) = 2.36$

Table 2 The minimum (MinErr) and the maximum (MaxErr) achieved relative error, on the ten functions, when the number of core increases

	$N = TC = 8$ $(NP = 1)$	$N = TC = 16$ $(NP = 2)$	$N = TC = 32$ $(NP = 4)$	$N = TC = 64$ $(NP = 8)$
Family Φ_1				
MinErr	0.48 (−10)	0.15 (−10)	0.65 (−11)	0.22 (−11)
MaxErr	0.11 (−7)	0.75 (−8)	0.45 (−8)	0.33 (−8)
Family Φ_2				
MinErr	0.41 (−7)	0.4 (−7)	0.4 (−7)	0.4 (−7)
MaxErr	0.47 (−6)	0.35 (-6)	0.22 (-6)	0.13 (−6)
Family Φ_3				
MinErr	0.40 (−4)	0.45 (−4)	0.42 (−4)	0.42 (−4)
MaxErr	0.46 (−3)	0.31 (−3)	0.25 (−3)	0.19 (−3)

The workload in each processing unit is $F = 10 \times 10^6$

Algorithm is executed with only 4 cores there is an extensive use of cached data. On the other hand, the use of 8 cores, that is two CPUs, needs a more frequent access to the main memory because the larger number of caches miss. Table 1 reports the minimum (MinErr) and the maximum (MaxErr) achieved relative errors for the three families of function. Families Φ_1 and Φ_2 show generally smaller numerical errors than family Φ_3 because of their better analytical properties.

A second set of experiments is aimed to study the algorithm by using 8 cores per node and $NP = 1, 2, 4, 8$ nodes of the system, so that the number of processing units is the total number of cores $N = TC = 8 \times NP$. In this way both strategies (parallelism among cores and parallelism among nodes), combined as described in Sect. 3, are tested together. To this aim, we allow $Nit_1 = 10$ iterations as the node stopping criterion of Algorithm 1 and $Nit_2 = F/20\varphi = 402$ iterations as core stopping criterion in Algorithm 2. The computational cost in each core is $F = 10 \times 10^6$ integrand function so that the total number of function evaluations is $Fval = NP \times 8 \times 10 \times 10^6$ when the number of nodes increases. Figure 2 reports the experimental Scaled Speed-up in (3) for the three families of functions Φ_1, Φ_2 and Φ_3 compared with the model in (4). Also for these experiments, acceptable values for experimental Scaled Speed-up SS_N are achieved, that are quite well estimated by the model (4) also for large number of computing units. Finally Table 2 reports the numerical errors for the three families of function, where the same considerations of Table 1 holds.

6 Conclusions

In this work we address the problem of the development of an adaptive algorithm for the multidimensional quadrature on a hierarchical HPC systems with nodes based on multicore CPUs. For this environment we need to combine two different algorithm development methodologies: a message passing based paradigm for the dynamic load balancing of the algorithm among the nodes of the system, and a shared memory based paradigm for the synchronization of the cores in each node. The resulting algorithm can be defined as a Parallel Double Adaptive Algorithm, in order to recall the two different adaptive strategies combined together in a single hierarchical algorithm. The obtained results confirm our expectation of good scalability for the proposed algorithm on this High Performance Computing environment.

In any case further investigations are needed. As the number of processing units will grow in the next generations of multicore CPUs, a key factor for the performance will be the memory levels management (caches and main memory), since the memory bandwidth usually grows much slower than the CPUs speed and limitations on space will not permit a growth of caches dimension proportional with the number of cores. Furthermore, modern HPC systems exhibit hybrid architectures mixing together traditional CPUs with other components, like hardware accelerators or GPUs, making much more difficult to define general strategies good enough for a large set of HPC systems. A notable example in this sense is the Tianhe-1A HPC system [12], that is based on hybrid nodes equipped with multicore Intel Xeon CPUs and NVIDIA Tesla general purpose GPUs. It requires different and specialized strategies in order to extract the maximum performance from the different components.

References

1. Berntsen, J.: Practical error estimation in adaptive multidimensional quadrature routines. J. Comput. Appl. Math. **25**, 327–340 (1989)
2. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Comput. **35**, 38–53 (2009)
3. Cools, R., Rabinowitz, P.: Monomial cubature rules since "Stroud": a compilation. J. Comput. Appl. Math. **48**, 309–326 (1993)
4. D'Apuzzo, M., Lapegna, M.: Scalability and load balancing in adaptive algorithms for multidimensional integration. Parallel Comput. **23**, 1199–1210 (1997)

5. Dongarra, J., Gannon, D., Fox, G., Kennedy, K.: The impact of multicore on computational science software. CTWatch Q. **3**(1), 3–10 (2007)
6. Dongarra J., Beckman P., et al.: The international exascale software roadmap. Int. J. High Perform. Comput. Appl. **25**, 3–60 (2011)
7. Genz, A.C.: A package for testing multiple integration subroutines. In: Keast, P., Fairweather, G. (eds.) Numerical Integration, pp. 337–340. D. Reidel Publishing Co., Dordrecht (1987)
8. Gustafson, J.: Reevaluating Amdahl's law. Commun. ACM **31**, 532–533 (1988)
9. Krommer, A., Ueberhuber, C.: Computational Integration. SIAM, Philadelphia (1998)
10. Laccetti, G., Lapegna, M.: PAMIHR. A parallel FORTRAN program for multidimensional quadrature on distributed memory architectures. In: Proceedings of EUROPAR99 Conference, LNCS 1685, pp. 1144–1148. Springer-Verlag, Berlin, Heidelberg (1999)
11. Numerical Algorithms Group Ltd. NAG Parallel Library release 3. NAG Oxford
12. Top500 list www.top500.org