Adaptive Numerical Integration on Message-Passing Systems

Rudolf Schürer

Department of Scientific Computing, University of Salzburg, AUSTRIA
Abstract

Parallel adaptive algorithms for the approximation of a multi-dimensional integral over an hyper-rectangular region are described. Algorithms with centralized global region collection are compared to algorithms using local region collections, which should result in better scalability since global communication is avoided. Tests are performed using Genz’s test functions and speed-up results are given.

1 Introduction

We consider the problem of estimating the multidimensional integral

$$I_f := \int_{C^s} f(x)dx,$$

for a given function $f : C^s \to \mathbb{R}$, where $C^s$ denotes a $s$-dimensional hyper-rectangular region $[a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_s, b_s] \subseteq \mathbb{R}^s$.

Adaptive algorithms based on an embedded cubature rule are used to approximate the integral. These algorithms divide the initial region into subregions and repeat this process on the subregion with the largest estimated error until a certain termination criterion is met. For examples of this algorithm applied to hyper-rectangular regions, see e.g. [1, 6].

Maintaining the collection of subregions and keeping it sorted is easy in a sequential program. However, if multiple processing nodes are working on the approximation of the integral, maintaining this region collection becomes a problem:

Using one global region collection (e.g. [9, 10]) results in bottlenecks, because all other nodes have to access this data structure by communicating with the manager node.

Using local region collections (e.g. [2, 4, 7]) avoids global communication. However, “bad” regions may not be distributed to other nodes fast enough to ensure good approximations with a small number of integrand evaluations.

2 Algorithms

2.1 Sequential Algorithm

The key concept of this algorithm is to apply a basic cubature rule successively to smaller subregions of the original integration domain. The selection of these subregions adapts to difficult areas in the integration domain by refining subregions with large estimated errors:

1. The basic rule is applied to the whole integration domain to estimate the value of the integral and the error of this approximation.
   An embedded degree 7 rule due to Genz and Malik [6] is used in our programs. It includes a degree 5 rule for error estimation.

2. The region and the estimations for result and error are stored in the region collection.
   A heap, ordered by the estimated error, is used to implement this collection efficiently.

3. The region with the largest estimated error is taken from the region collection.
4. This region is split into two subregions by halving it along the coordinate axis where the integrand has the largest local absolute fourth difference (For details, see e.g. [6]). Calculating this fourth divided difference is embedded in the basic rule, so no extra integrand evaluations have to be performed.

5. Estimations for result and error for these subregions are calculated

6. Both regions are stored in the region collection

7. Goto step 3

The loop can be terminated either when a certain absolute or relative error is reached, when the number of integrand evaluations exceeds an upper bound, or if some other criterion is met.

2.2 Parallel Algorithm using a global region collection

This algorithm is the straight-forward parallelization of the sequential algorithm.

There is a dedicated manager node maintaining the (global) region collection. When one of the other nodes becomes idle, the manager sends it a new region to be processed. The worker node calculates estimations for its value and error and returns these results to the manager, who stores them in the region collection. If further processing is required, the manager sends a new region to the worker. Otherwise, a message signalizing termination is sent.

Starting this algorithm with only one subregion is inefficient, because nodes would be idle until the number of subregions reaches the number of worker nodes. One common approach to avoid this is to have each node execute the sequential algorithm until the number of subregions reaches the number of working nodes.

The approach taken here is slightly different: Each worker node performs the following steps to find its initial region:

1. Set level to 0. Set region to the initial region

2. Split region in two parts by halving it along the coordinate axis with the largest fourth divided difference of the integrand. It takes $4 \cdot \dim + 1$ integrand evaluations to accomplish this.

3. If bit number level in the binary representation of the node number is 0, choose the left subregion. If it is 1, choose the right subregion.

4. If there are other nodes working on the same subregion (this can be determined easily using bit operations involving the node number and the number of nodes), increment level and continue with step 2.

5. Start integration algorithm with region.

The number of iterations of this loop is approximately $\log_2 n$, with $n$ denoting the number of worker nodes.

The manager node has to calculate the initial regions for all worker nodes. This task requires $n - 1$ calculations of a fourth difference. For a large number of nodes, this can be
substantially more expensive than the work required on the worker nodes. However, the worker nodes have to apply the basic rule on their region before they contact the manager for the first time. This gives the manager some extra time to catch up.

2.3 Parallel Algorithm using local region collections

The main disadvantage of the the parallel algorithm with a global region collection is that it does not scale to a large number of processing nodes. Due to the fact that there is only a single manager node which has to serve all worker nodes, this node will become the bottleneck if the number of nodes increases.

To improve scalability, all global communication has to be removed. This implies that there can not be a dedicated manager node. To balance workload, some communication between processing nodes is required. However, the communication of a certain node can be restricted to a small number of nodes in its neighborhood.

To accomplish this, all nodes are arranged in a $G$-dimensional periodical mesh $M_G$. This is a set of $n_0 \times n_1 \times \cdots \times n_{G-1} = n$ nodes, identified by the tuples $(x_0, x_1, \ldots, x_{G-1})$ with $0 \leq x_i < n_i$. Nodes are connected if they differ in exactly one coordinate $i$, and if this difference is $1 \mod n_i$. Note that this topology contains the ring when $G = 1$, the torus when $G = 2$, and the hypercube when $n_i = 2$ for $0 \leq i < G$.

In our implementation only meshes with dimension 2 are used (Experiments reported in [4] suggest that for up to 32 nodes, nothing is gained by using a mesh of higher dimension). $n_0$ and $n_1$ are chosen to be the integers closest to $\sqrt{n}$, with $n_0 \cdot n_1 = n$.

The basic idea of this algorithm is that every node executes the sequential algorithm on a subset of subregions of the whole integration domain. At the beginning, the algorithm outlined in the previous section is used by each node to find its initial subregion. When it is found, each node uses the sequential algorithm and its own (local) region collection to split regions into subregions and to adapt to difficulties found there.

2.3.1 Redistribution procedure

If this algorithm is used without further load balancing, there is a good chance that eventually most of the processing nodes will work on irrelevant refinements on regions with low (global) estimated errors, while only a few nodes tackle “bad” regions. To avoid this, regions with large estimated errors have to be redistributed evenly among processing nodes. This is accomplished by the following procedure:

For a certain node $P_i$ we define $P^{(dir)}_{i+}$ and $P^{(dir)}_{i-}$ as the next and previous node of $P_i$ in direction $dir$ in the mesh $M_G$. More formally, if $P_i = (x_0^i, x_1^i, \ldots, x_{G-1}^i)$, then

$$P^{(dir)}_{i+} = (x_0^i, \ldots, x_{dir-1}^i, (x_{dir}^i + 1) \mod n_{dir}, x_{dir+1}^i, \ldots, x_{G-1}^i).$$

$P^{(dir)}_{i-}$ is defined accordingly.

Each time the basic rule is evaluated, the following steps are performed:

1. Decide if redistribution shall take place at this iteration. If not, skip it.

Redistribution requires communication between nodes which is usually expensive. So

---

1There are also experimental results suggesting that load balancing is not worth the cost of communication. See [3] for an example.
it is preferable to perform redistribution only in certain intervals. The optimal size of these intervals depends on the number of performed iterations, the number of processing nodes, the number of function evaluations in the basic rule, and the amount of time needed for a single integrand evaluation and for communication.

In our implementation we perform redistribution after 5 basic rule evaluations.

2. Increment \( \text{dir} \). If \( \text{dir} = G \) set \( \text{dir} = 0 \).

3. Send the total estimated error of the local region collection to \( P_i^{(dir)} \).
   Receive the total estimated error of \( P_{i+}^{(dir)} \)'s region collection.

4. Send the \( k \) regions with the largest estimated error to \( P_{i+}^{(dir)} \). \( k \) shall be the largest number so that, after the transfer, the total estimated error of \( P_{i+}^{(dir)} \)'s region collection is still smaller than the total estimated error of the own region collection. Note that \( k \) may be 0.
   Receive regions from \( P_i^{(dir)} \) and store them in the local region collection.

Send and receive operations in step 3 and 4 have to be executed concurrently, or some other method has to be used to avoid deadlocks.

If \( n_{dir} = 2 \), both communicating nodes know the total estimated error of each others region collection after step 3. By comparing these values it is possible to determine if a send or a receive will be required in step 4, so only 3 instead of 4 messages have to be sent/received in this case.

2.3.2 Termination detection

Implementing an upper bound on the number of integrand evaluation is simple, because all nodes are synchronized by the redistribution procedure and perform the same number of basic rule evaluations.

However, termination depending on a requested total error is impossible without additional communication. Even if a node detects that the estimated error in its region collection is less than the requested error for its volume, it has no way of knowing this about other nodes. If this information is sent in the redistribution procedure, it takes several turns to reach all nodes. During this time the original node might have received a new “bad” region.

3 Testing

Numerical tests have been performed using three families of functions from a package proposed by Genz [5]. This package contains six families, each of them characterized by some peculiarity. The following three families were used:

\[
\begin{align*}
 f_1(x) &= \prod_{i=1}^{s} \frac{1}{a_i^2 + (x_i - u_i)^2} \\
 f_2(x) &= e^{-\sum_{i=1}^{s} a_i^2(x_i^2 - u_i^2)} \\
 f_3(x) &= e^{-\sum_{i=1}^{s} a|x_i - u_i|} 
\end{align*}
\]

Product Peak

Gaussian

\( C_0 \) Function

Parallel Numerics 2000/Bratislava, Slovakia/September 18-20
Results

For every family, 10 instances were created by choosing parameters $u_i$ and $a_i$ with a pseudo-random number generator. The unaffective parameters $u_i$, which do not affect the difficulty of the integrand, were chosen from the interval $[0.05, 0.95]$. The affective parameters $a_i$ were calculated by choosing temporary values $a'_i$ (pseudo)randomly from $[0.05, 0.95]$, and then by defining $a_i$ using $a_i = ca'_i$, with $c$ chosen so that

$$\sum_{i=1}^{n} a_i = d.$$ 

The single parameter $d$ controls the difficulty of the test function. $d = 15, 14, 10$ were used for the function families $f_1$, $f_2$, and $f_3$, respectively.

After each test, the number of correct digits was calculated using the formula

$$d_{act} = -\log_{10} | I f - value | .$$

The accuracy for a certain integration algorithm and a certain function family was calculated by taking the average of $d_{act}$ from all 10 tests.

4 Results

All algorithms were implemented in a C++ double precision program using MPI [8] for inter-process communication.

Tests were performed on a SGI Power Challenge GR located at the RIST++ (University of Salzburg). This is a shared memory machine based on 20 R10000 MIPS processors with 2.5 GB memory.

10 test functions were chosen from each family in dimension 5 and 10. This set of functions was used to test both parallel algorithms, running on 2, 4, 8, 12, 16, and 20 processors. We have also performed all calculations with the sequential algorithm to measure speed-up and evaluate the accuracy of the parallel algorithms.

Figure 1 shows the results for test function $f_1$ (Product peak) in 10 dimensions. The algorithm with local region collections (right side) shows the expected good scalability. The time to achieve a certain accuracy decreases with every additional processor.
The algorithm with global region collection performs poorly with two processors, because only one node is actually evaluating the integral, while the other node is maintaining the region collection. The performance with two processors is almost identical to the performance of the sequential algorithm.

With 4 processors, the global algorithm does better, but it takes 8 processors to beat the local algorithm. It seems, that at this point the local algorithm is not capable of distributing “bad” regions fast enough to avoid unnecessary work. With 16 processors, the global algorithm performs better than the local algorithm.

With 20 processors, however, the performance of the global algorithm breaks down completely. It seems that at this point the capacity of the manager node is reached. Adding additional processors will not result in any performance improvements.

Figure 2 shows the results for test function $f_2$ (Gaussian) in dimension 10. The charts are very similar to Figure 1.

Figure 3 shows the results for the family of $C^0$-Functions $f_3$. For this kind of functions we see acceptable speed-up for both algorithms. However, the results seem to be too unpredictable to allow any further conclusions.

Figure 4 shows the result for $f_1$ in 5 dimensions. In this case, the global algorithm behaves catastrophic on 16 and 20 processors. Due to the short time it takes the worker nodes to
evaluate the basic rule in 5 dimensions, and the large number of subregions to be maintained in the region collection, the manager node can not serve all worker nodes. It seems that some worker nodes have no chance to return their region at all, leaving a “bad” region with a large error in the final region collection. The error from this region dominates the final result. The local algorithm again scales well up to 16 processors.

5 Conclusion

Both parallel algorithms can be implemented with relatively good speed-ups compared to the sequential algorithm.

However, the global algorithm does not scale well and breaks down even for a medium number of nodes, especially in low dimensions where the basic rule can be evaluated quickly and the region collection becomes large. Due to the fact that it uses one node exclusively for management, it is inferior to the local algorithm if the number of nodes is small (up to 4). In high dimensions, however, it can beat the local algorithm.

Both algorithms might allow substantial improvements:

The global algorithm needs to take precautions not to starve out one of its worker nodes. If fairness is not provided by the communication library, it has to be implemented as part of the algorithms. Another way of optimizing this algorithm might be to send multiple regions for processing to the clients, or allow them to split them several times before returning the results. This can cut down communication overhead and might be especially useful for lower dimensions.

The local algorithm could be approved by (dynamically) adjusting the number of performed redistribution procedures. It might also be an advantage to test meshes with 3 or more dimensions, especially if the number of nodes is not a square number.

6 Acknowledgments

This work was partially supported by Österreichische Nationalbank, Jubiläumsfonds project no. 6788.
References


