Integrand and Performance Analysis with ParInt/ParVis

Elise de Doncker, Rodger Zanny, Karlis Kaugars and Laurentiu Cucos

Department of Computer Science Western Michigan University Kalamazoo, MI 49008, USA E-mail: {elise,rrzanny,kkaugars,lcucos}@cs.wmich.edu

Abstract

We present a survey of the contents and capabilities of the parallel multivariate integration package ParInt and its companion visualization tool ParVis, while focusing on the analysis of both the integrand behavior and the parallel performance of the integration methods. Current developments include asynchronous adaptive methods for hyperrectangular and simplex regions. A new parallel quasi-Monte Carlo technique suitable for heterogeneous platforms has been incorporated for general-purpose integration in higher dimensions, as well as for the computation of multivariate normal and t-distributions.

Key words: parallel adaptive integration, scalability, Quasi-Monte Carlo, visualization

1 Introduction

Following the release of ParInt1.0, the ParInt group (including E. de Doncker, A. Genz, A. Gupta, K. Kaugars, R. Zanny and L. Cucos) released the second stable version of its integration software, ParInt1.1, in the Summer of 2001. Currently, a beta version of the next release, ParInt1.2, is available. The software is available for download at the ParInt website `http://www.cs.wmich.edu/parint`. This paper presents integrand analysis results obtained with ParInt1.1 and ParVis. It gives a survey of the features offered through ParInt1.2.

⋆ Supported in part by the National Science Foundation under grants ACR-0000442, EIA-0130857, ACI-0203776

The computational objective is to obtain an approximation Q to the multivariate integral

$$I = \int_{\mathcal{D}} \mathrm{f}(\mathbf{x}) d\mathbf{x} \tag{1}$$

and an absolute error bound E_a such that

$$\hat{E} = |I - Q| \leq E_a \leq \hat{\varepsilon} = \max\{\varepsilon_a, \varepsilon_r |I|\}, \tag{2}$$

for given absolute and relative error tolerances ε_a and ε_r, respectively. The integration domain $\mathcal{D}$ in (1) is a $d-$dimensional hyper-rectangular region or simplex. More generally the integrand is defined as a vector function $\mathrm{f} : \Re^d \to \Re^n$ and the errors determined according to the maximum norm.

Our guiding principle in the design of ParInt is a layered approach. Figure 1 shows the architecture of the system. At the lowest level is the integration engine, relying upon a communication library. ParInt uses mpi [1] to provide for a degree of platform independence.

The integration engine encompasses several different techniques for integration, including the adaptive application of cubature rules, quasi-Monte Carlo techniques, variations in data structures, etc. These different methods can be selected by the user.

Above the integration engine is the set of functions that make up the ParInt Application Programming Interface (api). These functions allow for the initialization and solving of an integration problem. Users can write their own parallel applications which call functions in the ParInt api, allowing for the results of the integration to be used in later steps in a larger scientific application. The ParInt functions are written in C (and can be called from C and other languages).

The ParInt executable front end interfaces with the api, and can be executed on the Unix/Linux command line; command line parameters allow for the specification of the problem to solve and various other options. We currently provide a Java-based gui (see Section 3) as a platform independent interface for specifying the problem and related parameters, while offering real-time visualization of some parallel execution performance characteristics. In addition, users can run the gui client at a site remote from where the problem is actually solved.

The separate, post-mortem visualization tool ParVis (see Section 5) allows for a data-driven examination of the integration algorithm's behavior, even for large and complex problems. For various *events* during the execution of the algorithm, ParInt can be instructed to log information about these events to a log file. The log file is then read by ParVis, post mortem, to allow analyzing the execution.

While ParInt is written in C, users can specify the integrand function in most common programming languages; we have so far implemented functions in c and Fortran. Results can be reported in both `double` and `long double` accuracy. A full user manual [2] is available.

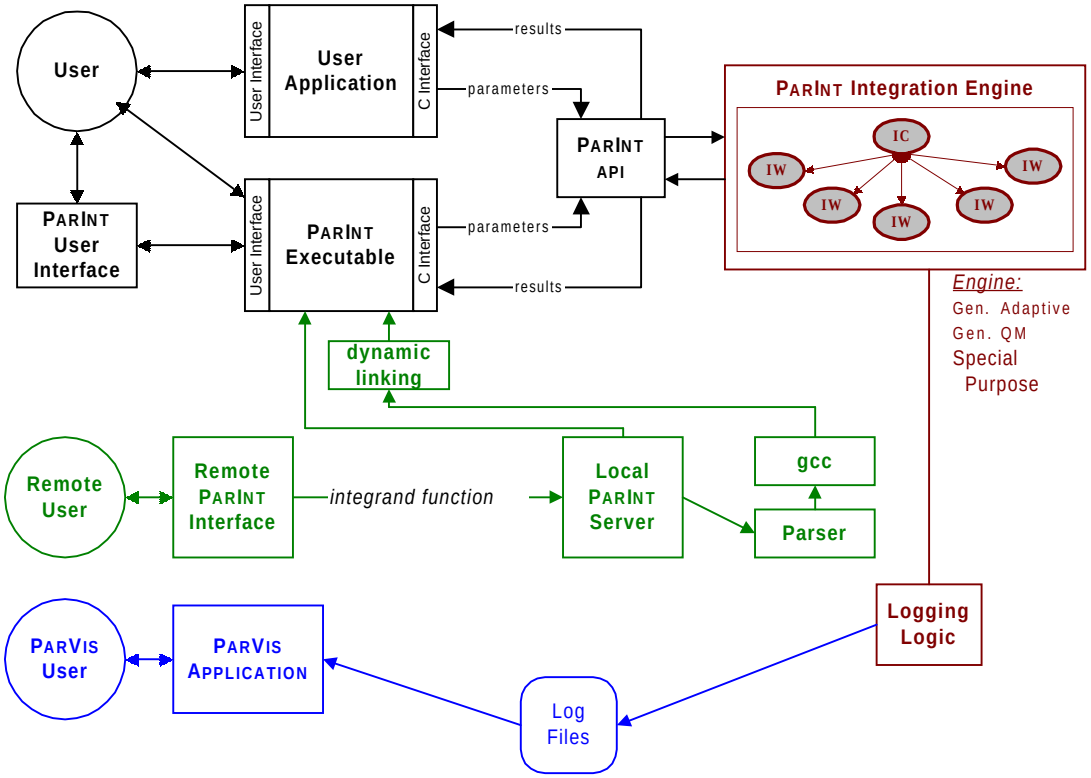

Fig. 1. PARINT Architecture

The remainder of this paper is organized as follows: Section 2 outlines the base adaptive algorithm, Section 3 covers the GUI for PARINT, and Section 4 analyzes the effect of singularity loss on scalability. Section 5 discusses the PARVIS tool, developed for visualizing the execution of the adaptive algorithm. Section 6 explains the quasi-Monte Carlo integration technique, an alternative to adaptive integration which is commonly used for higher dimensional problems. It also describes a new scalable technique for its parallel implementation. Section 7 gives some applications, and Section 8 discusses current research directions, and concludes the paper.

2 Adaptive Algorithm

This section describes the base adaptive algorithm used in PARINT, including data structures and load balancing techniques.

2.1 Local Heap Algorithm

We use a *global adaptive partitioning algorithm* (adhering to the *meta-* algorithm of Figure 2), in order to concentrate the integration points in areas of the domain $\mathcal{D}$ where the integrand

```
Evaluate initial region and update results
Initialize priority queue with initial region
while (evaluation limit not reached and estimated error too large)
    Retrieve region from priority queue
    Split region
    Evaluate new subregions and update results
    Insert new subregions into priority queue
```

Fig. 2. Adaptive Integration Meta-Algorithm

is the least well-behaved. The bulk of the computational work is in the calculation of the function f at the integration points. The granularity of the problem primarily depends on the time needed to evaluate f and on the dimension; an expensive f generally results in a problem of large granularity, such as the Bayesian statistics problems in [3,4] and the high energy physics problem in [5,6].

Multivariate integration rules for hyper-rectangular regions from Genz and Malik [7,8] (with error estimation techniques from [9]), simplex rules from [10], and univariate (Gauss-Kronrod) rules (from Quadpack [11]) are used to provide a result $Q(\mathcal{R})$ and an error estimate $E(\mathcal{R})$ for any subregion $\mathcal{R}$ obtained from the initial domain $\mathcal{D}$. Thus $Q(\mathcal{R})$ is of the form $Q(\mathcal{R}) = \sum_{k=1}^{r} w_k \mathrm{f}(\mathbf{x_k})$. We will refer to the calculation of the pair $Q(\mathcal{R})$ and $E(\mathcal{R})$ for a region $\mathcal{R}$ as a *region evaluation.*

The meta-algorithm of Figure 2 subsumes many different designs and implementations [12,13]. In the ParInt distributed implementation of the adaptive partitioning algorithm, all processes act as *integration worker* processes; one process additionally assumes the role of an *integration controller.* The initial region is divided up among the workers. Each executes the adaptive integration algorithm on their own portion of the initial region, largely independent of the other workers, while maintaining a local priority queue of regions, stored as a heap and keyed with the estimated error of the regions.

In the initial steps the workers evaluate their part of the domain and initialize their local heap with it. The regular adaptive loop iteration consists of: removing the region with the highest estimated error from the priority queue; splitting this region in half and evaluating the two new subregions; updating the worker's overall result and error estimate; and inserting the new subregions into the queue.

All workers periodically send updates of their results to the controller. In turn, the controller provides the workers with updated values of the estimated required accuracy ε, calculated as

$$\varepsilon = \max\{\varepsilon_a, \varepsilon_r |Q|\}, \tag{3}$$

which approximates $\hat{\varepsilon}$ of (2). The workers use this value to determine if they have achieved a satisfactory approximation of the integral over their initial subregion. If they have, they become *idle.* To maintain efficiency, a dynamic load balancing technique is employed to move work to the idle workers (see Section 2.2).

The computation terminates when the total estimated error calculated by the controller drops below the threshold ε, or, a user-specified limit on the number of function evaluations is reached.

2.2 Load Balancing

Through ParInt1.2 we have used a *receiver initiated, scheduler based* load balancing technique. A worker i informs the controller via a regular update message that the worker is not idle; the controller selects (in a round-robin fashion) an idle worker j and sends i a message containing the id of j. Worker i will then send j a region or a message indicating that no work is available. Worker j receives this message and either resumes working or informs the controller that it is still idle.

We are testing alternatives to this load balancing approach for potential adoption in future releases [6]. The flow of the load can be increased by sending more regions per load balancing step. However, one has to account for trade-offs with respect to communication overhead and possible thrashing. The latter occurs when regions are re-sent in subsequent load transfers.

One promising load balancing startegy is *random polling*: in random polling load balancing, busy workers select other workers at random for work transfers. It was found in [6] that random polling load balancing allows for a more effective distribution of the work and leads to a scalable solution for irregular problems.

A useful model for analyzing the behavior of an adaptive partitioning algorithm is the *region subdivision tree* [12,14]. Each node in the tree corresponds to a region that was evaluated during the execution of the algorithm. The root is the initial region $\mathcal{D}$; each other node has a parent node corresponding to the region from which it was formed and either zero or two children. The leaf nodes correspond to the regions on the priority queue(s) at the end of execution, and the number of nodes in the tree is equivalent to the number of region evaluations during execution. These trees generally consist of anywhere from one region or a few dozen to a few tens of thousands of regions, depending upon the difficulty of the problem and the desired accuracy.

Note that as any adaptive partitioning algorithm will have methods for prioritizing, selecting, and partitioning work, the notion of a region subdivision tree exists in any problem domain for which some sort of dynamic, adaptive partitioning can be utilized, including adaptive mesh refinement, hierarchical progressive radiosity and branch and bound algorithms.

Prolonged execution of the local heap algorithm without limiting memory use may lead to the exhaustion of local memory, as each subdivision (in the case of bisection) increases the heap by one node. Therefore, ParInt1.1 offers the option to use a local *deap* (double-ended heap) data structure [15] instead of a heap, which makes it possible to limit the number of nodes. A deap consists of a min-heap and a max-heap (without overlap, i.e., without duplicating the stored data). The insert, delete-min, and delete-max operations are of logarithmic time

(as a function of the size). When the deap reaches the limiting size, each addition of a node requires deletion of the smallest element, so the size remains constant from then on (other than when load balancing occurs).

3 The ParInt Graphical User Interface

We currently supply a graphical user interface written in Java, which is based on [16] and provides the user with a simple interface to ParInt.

The GUI is implemented as a Java applet which runs on the client's computer. The corresponding server runs as a daemon on a local or remote workstation; it accepts connections from the client and starts up ParInt processes as needed. The server is able to solve problems sequentially or in parallel on its network.

The client applet allows the user to specify an integrand function and all of the corresponding parameters: the ε_a and ε_r values, the integration region, the quadrature rule, the function count limit, etc. It also allows for the specification of a number of processors to use, as well as various algorithm parameters which modify the behavior of the algorithm, e.g., turning load balancing on or off, etc. After the parameters have been specified, the user starts the integration; after completion, the results are displayed to the user.

The current implementation limits the user to integrating functions that are either available a priori, i.e., at compile-time, on the server, or that can easily be supplied by the user via the integrand function window of the GUI (which excludes more complicated integrand definitions, e.g., those that span multiple files). For a function specified at run-time, the source will be sent to the server, compiled, and the object dynamically linked with ParInt (see Figure 1).

4 Work Anomaly

Adaptive task partitioning is an important paradigm in numerical integration. However, it is also used in other problem domains, such as ray tracing, function approximation, optimization, and mesh refinement [17]. Our investigations indicate a possible adverse anomaly in the amount and pattern of work performed with distributed versions of an adaptive partitioning algorithm. Test results and measures of this phenomenon were given for distributed adaptive integration and branch and bound methods in [18]. For some problems, the occurrence of this anomaly inhibits the construction of an efficient parallel solution. For other problems it is important to recognize its influence in order to tailor the solution technique.

When a parallel execution results in a region subdivision tree which is larger than the corresponding sequential tree, we will characterize the work increase by the term *work redundancy*, defined as the ratio $R_p = \frac{W_p}{W_1}$ of the parallel work W_p to the sequential work W_1, in terms of the number of regions evaluated to solve the problem using p processes and using one process, respectively. The occurrence of considerable work redundancy will be referred to as *work anomaly*. It is possible for the user to keep the total work size constant (thus alleviating work redundancy) by limiting the allowed number of function evaluations (or region evaluations). The accuracy may be affected if redundant (useless) subdivisions are performed locally before useful tasks could be acquired through load balancing.

It may seem natural to try to force a subdivision pattern which resembles the sequential pattern in order to avoid work anomaly (equivalent to forcing the parallel region subdivision tree to be similar to the sequential one). Note, however, that for a single problem there may be many region subdivision trees of a particular size that get an adequate answer, and they may have very different appearances (in height, width, etc.). Thus a region may not simply be considered "redundant" if it does not appear on the sequential tree, for it may appear on a variation of the latter that is nearly as small or smaller. Furthermore, it may not be necessary for the most important task to be selected at each step, as long as a useful task is selected which would be selected eventually.

We have shown in [14,19] that the problem of work anomaly transcends many different algorithms and techniques, including algorithms with local or globally distributed priority queues (and accessed synchronously or asynchronously by the worker processes). The effects of the anomaly are furthermore similar across multivariate and univariate problems.

For the purposes of analyzing work anomaly, we implemented a synchronous global heap simulator which sequentially executes a simulation of a synchronous distributed global heap algorithm on an arbitrary number of processors. In a synchronous global heap algorithm on p processors, p (or, a multiple of p) tasks are deleted from the heap and assigned to the processors in each iteration. Unless the error estimates are wrong, these are currently the most important tasks; thus one could expect the load balancing in a synchronous global heap to be nearly ideal (in a homogeneous environment). Yet the problem of work anomaly appears in the simulation as well.

Another available distributed integration algorithm which we have tested is that used by the NAG routine `D01FAFP`, developed by Lapegna [20]. This is a synchronous algorithm operating upon a virtual mesh of processors characterized by a neighbor based synchronous load balancing scheme. Our preliminary experiments with the NAG parallel integration routine demonstrate a similar level of work anomaly.

Subsequently we examine two factors contributing to work anomaly: unsatisfactory error estimates and singularity loss.

The estimated absolute error $E(\mathcal{R})$ of the integral approximation over a region $\mathcal{R}$ is used as its priority ranking in the priority queue. All region error estimates add up to the global error estimate for the integral. If the error is overestimated over a number of regions, the effect will cascade and result in redundant, unnecessary regions being evaluated, so that the subdivision tree will be enlarged considerably. Furthermore, underestimated errors may lead to incorrectly selected regions.

Let us introduce the *error ratio* defined as $E_\rho = \frac{E_a}{\varepsilon}$, and the actual error ratio $\hat{E}_\rho = \frac{\hat{E}}{\hat{\varepsilon}}$, where ε and $\hat{\varepsilon}$ represent the estimated and actual requested accuracy defined in (2) and (3), respectively. Consider the function

$$\mathrm{f}_1(x, y, z) = \frac{1}{(x + y + z)^2} \tag{4}$$

over the 3−dimensional unit cube. Note that f_1 has a singularity at the origin. Figure 3 plots the initial portions of the curves of $\log_{10} E_\rho$ and $\log_{10} \hat{E}_\rho$ for function f_1, versus the current number of regions generated, for a sequential (single processor) run.

We found that the top of each peak in the curve corresponds to the subdivision of a cube $\mathcal{C}$ containing the vertex singularity. This is followed by two more subdivisions in different coordinate directions, resulting in a smaller subcube at the origin, the sides of which are half the length of the sides of $\mathcal{C}$. Figure 4 verifies the expected rate of convergence of the adaptive process in the neighborhood of the singularity. It shows the actual error ratio corresponding to the completion of each of these subcubes as a function of the size h of the subcube side. The horizontal axis of the graph represents $-\log_2(h)$. The early part of the subdivision sequence aims directly at the singularity (apparently through $h \approx 2^{-3}$, corresponding to approximately co-linear points satisfying

$$\log_{10} \hat{E} \sim -a(-\log_2 h) + \text{const} \approx 3.32a \log_{10} h + \text{const}.$$

The data in Figure 4 for h through 2^{-3} adhere to $a \approx \frac{1}{3}$; i.e., the observed convergence behavior corresponding to successively halving the grid size around the singularity is $\log \hat{E} \sim O(\log h)$.

As long as the error over the region with the singularity is much larger than over the surrounding regions, the subdivisions conform to the error expansion for this problem [21]. Indeed, for a function which is *homogeneous of degree* α around the origin (and a uniform subdivision),

$$Q \sim I + \sum_{k \geq 0} (a_k + c_k \log(h)) h^{N+\alpha+k} + \sum_{\ell \geq 1} b_\ell h^\ell. \tag{5}$$

In this example, $\alpha = -2$; the terms for k odd in the first sum of (5) are not present because

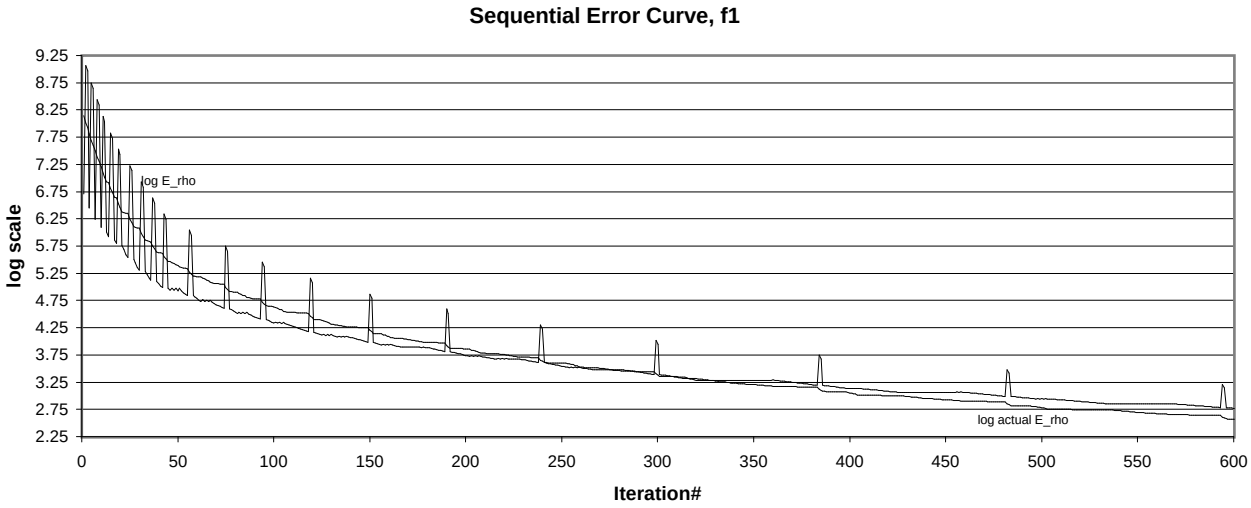

Fig. 3. Sequential error curve for $\mathrm{f}_1(x,y,z) = \frac{1}{(x+y+z)^2}$

of integrand symmetry; in the second summation, the terms of odd index drop out in view of the symmetry of integration rule; furthermore, the second summation only includes terms where ℓ exceeds the polynomial degree of the quadrature formula (here, $\ell > 7$). Thus the behavior of the error is dominated by the contribution for $k = 0$ in the first summation of (5) (which agrees with the observed behavior of $\log \hat{E}$ above). This formula holds if the integration domain is partitioned uniformly using a grid size h. Since this is not done here (as many more larger regions would need to be subdivided), the total error drops off at a slower rate.

Figure 3 also shows that the error estimates are unsatisfactory during a large portion of the computation. It appears that distributed implementations of adaptive integration algorithms are sensitive to mis-prioritizing of tasks, far more so than the sequential algorithm.

Furthermore, the error estimates are often conservative (over-estimated). So even though it may be that the estimated error exceeds ε for p processes and a certain total number of regions (where, sequentially, the estimated error $\leq \varepsilon$), the actual error may be $\leq \varepsilon$ in the parallel run.

The phenomenon of *singularity loss* occurs when a task is partitioned before another, globally more important task can be selected and partitioned [17]. This may happen as a result of singular integrand behavior where a less important task is selected by a process before a more important task is generated by a process holding the singularity. Indeed, for a sequence of successive levels of the region evaluation tree, there may be only a single region that needs to be evaluated (the region containing the singularity), so that only a single process can be doing useful work at any given time. More generally this may happen whenever a process' most important task is less useful than a task yet to be generated at another process. We refer to this characteristic as *inherent* singularity loss, and it inhibits scalability, particularly if each task is partitioned into a fixed number of subtasks.

Furthermore, if an integrand function has varying degrees of difficulty over the integration

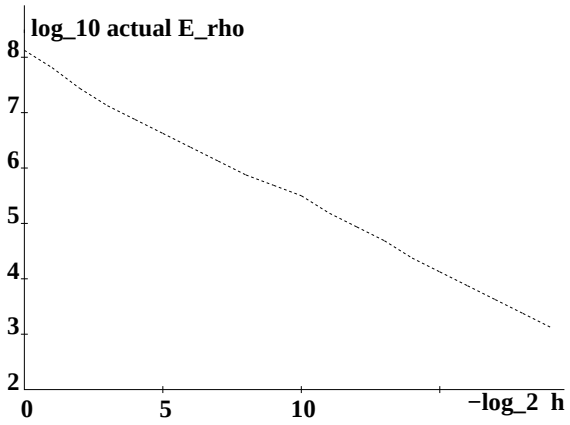

Fig. 4. Actual error ratio vs. grid size for f_1

region, then one process' most important task may be at an unimportant level with respect to another process. Ideally one would expect this can be handled through load balancing, though at the cost of increased communication.

Any parallel adaptive partitioning algorithm may suffer from singularity loss. In the realm of parallel branch and bound it is termed *detrimental anomaly*, see, e.g., [22–24].

There are other contributing forms of loss of efficiency. *Starvation loss* occurs when a process does not have any work to perform due to delays in balancing the load. *Breaking loss* corresponds to the extra work performed at the end of the computation: at some point in time, some worker performs the last piece of work that is actually necessary. The workers, though, will continue until the controller receives the corresponding update and terminates the workers. This extra work done by the workers results in *breaking loss*. Furthermore, if a very simple problem is solved by a large number of workers, then even an initial region division into p pieces may be more work than is required to solve the problem, causing *iteration loss*.

4.3 Influence of Work Anomaly

In a global heap implementation, the p most important tasks are selected simultaneously, while only a portion of these (those with important subtasks) may have actually needed further attention. For example, Figure 5 *(right)* shows the top portion of a typical sequential subdivision tree for a singular integrand function of depth $\delta = 5$ and *(left)* a corresponding parallel tree for $p = 2$ (corresponding to what might be obtained using a synchronous global heap, where two processes at a time obtain and evaluate a region). The node labels indicate the iteration when that node was generated. The extra nodes in the parallel tree are redundant; the redundancy in this case is $R_p = \frac{19}{11}$.

If $p = 2^k$ and it is assumed that levels $0, 1, \ldots, \log_2 p$ are generated sequentially, the remaining $\delta - \log_2 p$ levels are of width $2p$, since each iteration provides p processes with a region to

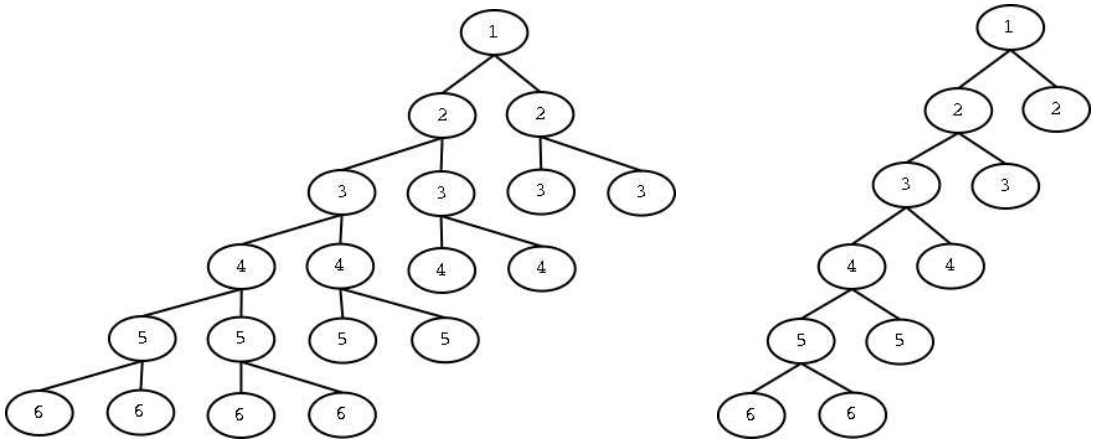

Fig. 5. Subdivision trees leading to singularity loss

be bisected. The work redundancy is then $R_p = \frac{W_p}{W_1} = \frac{\mathcal{O}(\delta p)}{\Theta(\delta)} = \mathcal{O}(p)$. The resulting speedup is as poor as $\mathcal{O}(1)$.

Even though it is impossible to deal efficiently with inherently sequential behavior, the above observations indicate that improvements may generally be obtained by getting more regions in place where they are needed.

5 ParVis: A Tool for Visualizing Adaptive Techniques

Following the approach of interactive visual exploration [25], we developed a visualization system which allows the users of parallel adaptive partitioning programs to gain an intuitive understanding of program behavior and result validity without examining the code (see also [26,27]).

The ParVis tool is a post-mortem event viewing prototype. The ParInt system can be instructed to track significant events during the computation of integrals, logging these events to local disk on a per-process basis. After a complete computation, a second parallel program collects the local log files and assembles them into a coherent log of the execution. Logging events locally and performing all visualization after execution minimizes the perturbation of the computation to be visualized.

5.1 Region Trees

As described in Section 2, ParInt seeds each processing element with an initial region which is then repeatedly subdivided to produce a tree of regions. This process produces a forest of binary trees and ParVis allows the user to interactively explore this forest. ParVis supports four different types of tree layouts formed by combining two layout shapes with two display

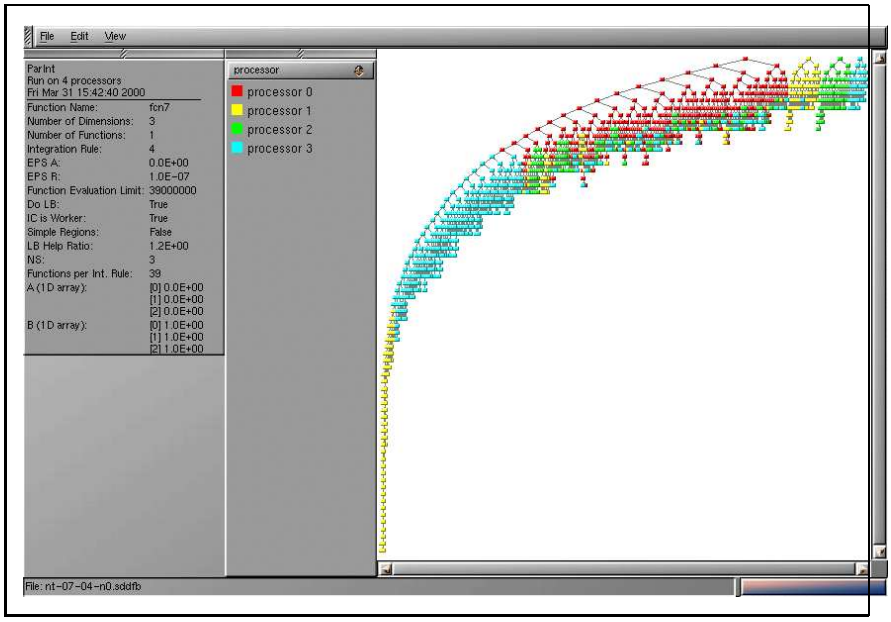

Fig. 6. The forest display, for function f_1. The far left of the display surface is devoted to the presentation of run details. The next panel shows the legend and the main display area shows the forest of region evaluations. In this image the regions are colored by processor. The bottom edge of the display lists the current file and also contains a progress indicator showing the current load status of the file.

aesthetics. The forest may be drawn using either a rooted tree or circular tree shapes, and nodes may be positioned to conserve display space or as a partial binary tree (where each portion of the tree is only given as much width as it would get in a complete binary tree - see the example of Figure 10 in Section 5.2 below). Figure 6 shows the main interface of ParVis after loading a region tree for the integration of function f_1 of (4). This computation was run on four processors. The point singularity at the origin causes the first of the four subtrees to contain most of the region evaluations, with the four processors working on it as they attempt to refine the estimates around this point.

The forest display commonly contains between 200 and 10,000 regions. In order to display the entire forest, the region displays are almost inevitably scaled to the point where each region is represented by at most a few pixels (subject to a minimum size constraint, usually 3x5 pixels). While size constraints at these scales preclude the display of individual data values, ParVis still attempts to display useful information by encoding one of several region attributes in the color of the node. The user may color the nodes using any of the variables logged by the ParInt computation, such as the generating processor, the time of creation or the estimated error of the region.

The forest display gives an overview of the computation, which has proved to be a valuable tool for understanding the activities of the algorithm. In order to be truly useful, the interface must also provide the user with the details of computation, allowing access to the contents of a region. The interface therefore allows the user to interactively pan and zoom the forest. The sequence of images in Figure 7 show two steps in the zooming process, from the display of Figure 6.

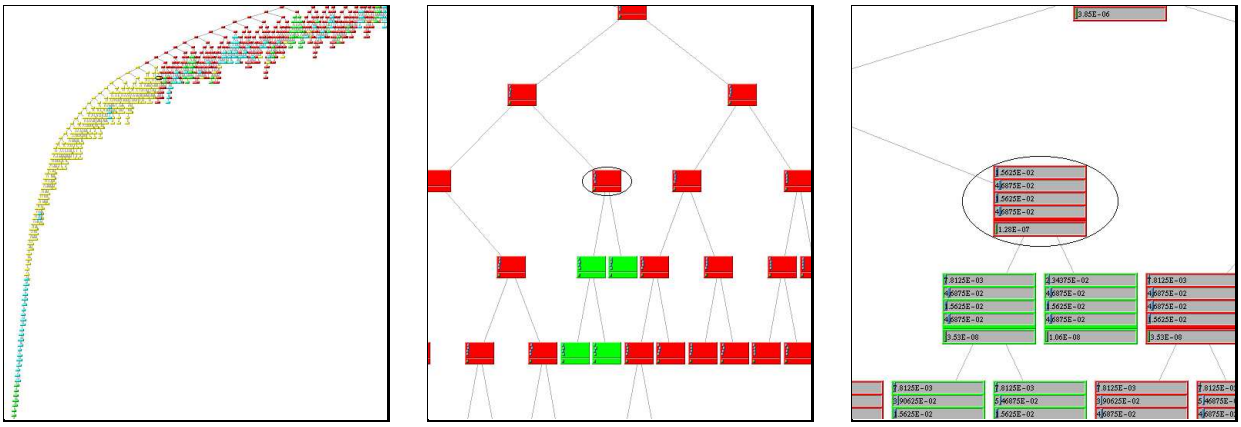

Fig. 7. Zooming in on the region forest display from Figure 6

In addition to the zooming forest views, ParVis also supports the display of region subdivisions over the initial region (see, e.g., Figure 9). Only hyper-rectangular regions are handled. Projections onto the coordinate planes are given for dimensions greater than 2.

The ParVis software is written in C++ for the Linux operating system. ParVis is written to read and manipulate Self Defining Data Format (SDDF) [28] files and can be applied to any algorithm featuring adaptive, dynamic subdividing of the problem space, including branch and bound algorithms, as long as the log files store sufficient information to reconstruct the forest.

5.2 Using ParVis

In this section we illustrate prototype usage of ParVis for evaluating problem and algorithm behavior, on the basis of a few case studies with singular integrand functions. One of the tree node color encodings is with respect to the process that owns the node. For the point singularity of Figure 6 this display reveals the occurrence of thrashing in the parallel integral computation for a load balancing strategy which transfers the top priority region. The majority of work in this case occurs in the leftmost tree of the forest. This tree is initially "owned" by the controller process. The other processes quickly complete their assigned tasks and offer to help. In the top part of the tree, this procedure appears to work fairly well, but eventually Worker #1 is given the region containing the singularity. The region containing the singularity is then passed to Worker #2 and eventually to Worker #3. At each transfer, work on the important part of the initial region is delayed by the inherent messaging latency. It thus emerges that thrashing contributes to singularity loss and work anomaly. In the case of a point singularity, the problem can be avoided by not transferring the top priority region in a load balancing step, but rather a region or regions near the top.

Although the presence of a point singularity at the origin causes thrashing, situations in which the load is well balanced are illustrated by the images in Figure 8, which show region subdivision trees generated by *(left)* four or *(right)* five worker processes while integrating

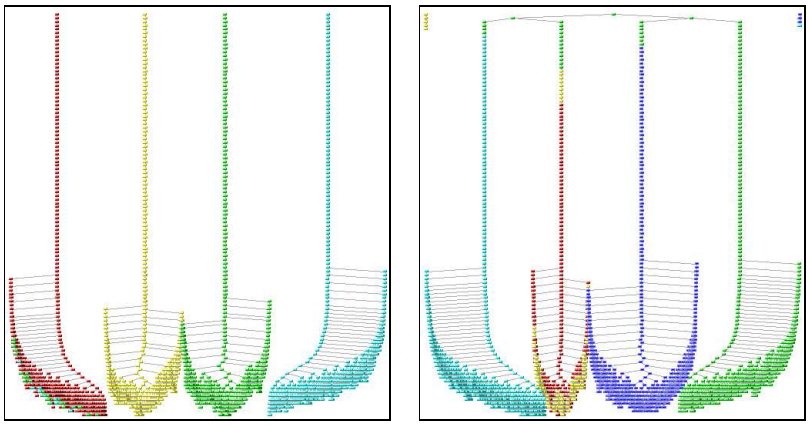

Fig. 8. Region trees showing load balancing for the integration of function f_2. The left image shows a run with four processors, the right image shows the same function run with five processors.

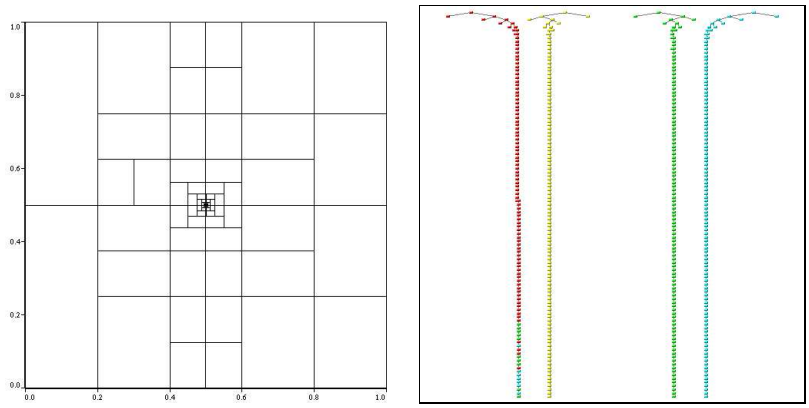

Fig. 9. Region subdivision and binary tree plot views for function f_2.

the function

$$f_2(x, y) = \frac{1}{((x - 0.5)^2 + (y - 0.5)^2)^{0.95}},$$

with a radial point singularity at $(x, y) = (0.5, 0.5)$, over the unit square. Using four workers, the initial region is divided into four sub-squares sharing a vertex at the singularity. This allows each worker to immediately begin working on a region that includes the singularity. Each worker is kept busy doing important work, and little load balancing needs to be done.

In Figure 8, the very small trees (there are actually two in the upper left corner and two in the upper right corner of the figure) represent the initial work done by the workers without the singularity. They quickly go idle on these unimportant regions, but, as shown by the figure, the load balancing mechanism is able to reassign important regions to all of the workers. The coloring of regions by worker has proven to be beneficial in qualitatively evaluating new, experimental load balancing techniques.

The use of the space-conserving rooted tree display in Figure 8 shows the individual trees of the forest occupying larger amounts of space deeper in the tree, due to the increased "bushiness" at these parts of the tree. These trees correspond in fact to subdivisions towards a single point singularity and evaluations of only a small part of the possible region subdivision

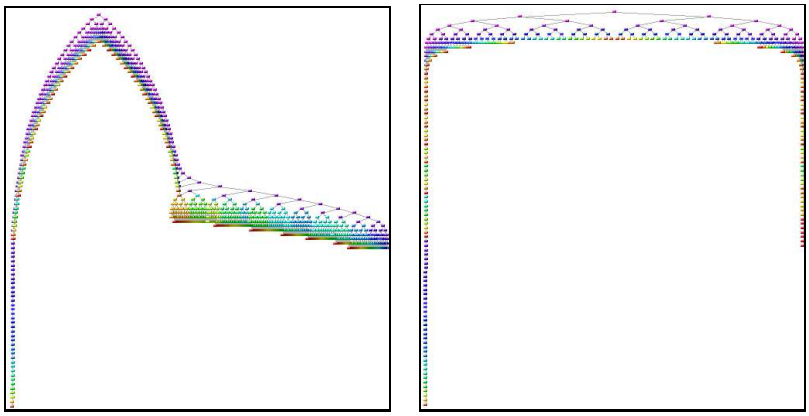

Fig. 10. Region trees for function f_3. The left tree is a normal region view; the right tree is a partial binary tree layout.

space. Using the region view and partial binary tree layouts as in Figure 9 (which shows the same integration run as in Figure 8) allows the user to see both aspects of the computation.

Figure 10 highlights the use of comparing the region plot and the partial binary tree layout to determine the behavior of the algorithm. The region tree in this figure corresponds to the integration of the function

$$f_3(x) = \begin{cases} \frac{1}{\sqrt{x}} & \text{if } x \leq 1 \\ \frac{1}{\sqrt{2-x}} & \text{otherwise} \end{cases}$$

over $0 \leq x \leq 2$. Note that there are singularities at $x = 0$ and at $x = 2$. The derivative discontinuity at $x = 1$ is not troublesome here as it occurs in the middle. In the "sickle" shaped tree on the left in Figure 10, the right subtree does not look like the typical partial region tree corresponding to a singularity. The right side of this figure is a partial binary tree layout. This clearly shows the expected two narrow areas of the tree corresponding to the singularities. The odd shape in the left tree is due to how the floating point library deals with very narrow regions, particularly the difference in the extent of subdivision possible near $x = 0$ and $x = 2$.

6 Asynchronous Quasi-Monte Carlo Method

6.1 Motivation and Notations

When the dimension d of the integral is small (say, $d \leq 10$), adaptive partitioning methods generally work well for finding highly accurate solutions. However, when d is large and the integrand behavior calls for considerable partitioning, the dimensional effect in the required number of evaluation points grows to an unacceptable level [29]. In higher dimensions, lattice

rules have been found useful for functions which are reasonably smooth. Equidistributed point sequences have been used effectively in high dimensions, including Richtmyer rules [29], and low discrepancy point sequences such as Faure and Sobol sequences [30,31].

A (deterministic) point sequence $\mathbf{x}_1, \mathbf{x}_2, \ldots$ in the unit hypercube $\mathcal{H}$ is equidistributed or uniformly distributed in $\mathcal{H}$ if

$$\lim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{N} \mathrm{f}(\mathbf{x}_i) = I = \int_{\mathcal{H}} \mathrm{f}(\mathbf{x}) d\mathbf{x}$$

for all Riemann integrable functions f.

In view of their equidistributed nature, these methods are designated as *quasi-Monte Carlo* methods - as opposed to Monte Carlo integration which uses (pseudo-) random sample points. We will also use the term *quasi-Monte Carlo* for lattice rules.

While synchronous parallel implementations of these techniques appear to be straightforward on tightly coupled parallel architectures, various factors push for an asynchronous solution in heterogeneous, coarse-grained architectures, such as a network of workstations (NOW), where synchronization points at the completion of a rule would generally lead to unacceptable delays.

We use lattice (Korobov) rules from [32] up to dimension 100. Their convergence properties are generally better than the $\mathcal{O}(1/\sqrt{N})$ rate of the classic Monte-Carlo method, where N is the number of points used. Two drawbacks are that good lattice rules are hard to obtain in high dimensions and that the error is hard to estimate.

Let us represent the lattice (Korobov) rule approximation of the integral I of f over $\mathcal{H}$ by

$$K_N = \frac{1}{N} \sum_{i=1}^{N} \mathrm{f}(\{\frac{i}{N}\mathbf{v}\}), \tag{6}$$

where $\{x\}$ = the fractional part of x and $\mathbf{v}$ is a predetermined generator vector with integer coefficients, and denote $\hat{E}_N = |I - K_N|$. Furthermore, using the notations of [29], $\mathcal{E}^k$, $k > 1$, is defined as the class of all functions f periodic with period 1 in each variable and for which the Fourier coefficients $c_{\mathbf{m}}$ satisfy $|c_{\mathbf{m}}| \leq C r_{\mathbf{m}}^{-k}$, for all $\mathbf{m} \neq \mathbf{0}$, a constant $C > 0$ and $r_{\mathbf{m}} = \prod_{j=1}^{d} \max\{1, |m_j|\}$.

For increasing $N = N_0, N_1, \ldots$, the error of a *good lattice point sequence* $\mathbf{v}(N)$ satisfies $\hat{E}_N = \mathcal{O}(\frac{(\log N)^{k\gamma^*}}{N^k})$, for a common γ^* known as the *index* of the rules, for $\mathrm{f} \in \mathcal{E}^k$ $(k > 1)$.

The rules of (6) are randomized in order to obtain a stochastic family [33], by translating the points over a uniformly distributed random vector $\boldsymbol{\beta}$. Then, using a random sample set of size q and averaging gives

$$\bar{K}_N = \frac{1}{q} \sum_{j=1}^{q} K_N(\boldsymbol{\beta}_j), \tag{7}$$

where

$$K_N(\boldsymbol{\beta}) = \frac{1}{N}\sum_{i=1}^{N} \mathrm{f}(\{\frac{i}{N}\mathbf{v} + \boldsymbol{\beta}\}).$$

A standard error estimation for (7) is obtained via $\hat{\sigma}_q^2 = E_N = \frac{1}{q(q-1)}\sum_{j=1}^{q}(K_N(\boldsymbol{\beta}_j) - \bar{K}_N)^2$.

For dimensions larger than 100 we use a Richtmyer rule approximation,

$$R_N = \frac{1}{N}\sum_{i=1}^{N} f(\{i\theta_1\}, \ldots, \{i\theta_d\}),$$

where $\theta_1, \theta_2, \ldots, \theta_d$ are d irrational numbers such that $\lambda_0 + \lambda_1\theta_1 + \cdots + \lambda_d\theta_d \neq 0$ for rational λ coefficients not all zero. Then $\hat{E}_N = |I - R_N|$ satisfies $\hat{E}_N = \mathcal{O}(1/N)$, for $\mathrm{f} \in \mathcal{E}^k, \quad k > 1$ [29]. The points $P_\ell : (\{\ell\theta_1\}, \{\ell\theta_2\}, \ldots, \{\ell\theta_d\})$ are equidistributed over the unit hypercube. We use $\theta_i = \sqrt{\pi_i}$ where π_i is the i^{th} prime [32]. Note that, in view of the above periodicity requirements, a periodizing transformation is applied to the original integrand function.

6.2 Algorithm

The algorithm calculates the $\bar{K}_N$ values of (7) for successively larger values of N until either an answer is found to the user-specified accuracy or the function count limit is reached. As these $\bar{K}_{N_1}, \bar{K}_{N_2}, \ldots$, values are calculated, the overall result is calculated as the weighted sum $\bar{K} = (\sum_i \frac{\bar{K}_{N_i}}{E_{N_i}})/(\sum_i \frac{1}{E_{N_i}})$ (where $\bar{K}_{N_i}$ is weighted with the inverse of its corresponding squared standard error estimate) and, correspondingly, $\bar{E} = 1.0/(\sum_i \frac{1}{E_{N_i}})$. We denote the randomized lattice rules in (7) as $\kappa_{ij} = K_{N_i}(\boldsymbol{\beta}_j)$, $j = 1, \ldots, q$, and represent row i by $R_i = \{k_{ij} \mid 1 \leq j \leq q\}$.

In order to achieve good efficiency in a distributed system, we developed a strategy where the (p) workers are assigned by the controller slices of the rows [34]. First, consider every κ_{ij} split into p pieces $k_{ij}^m, m = 1, \ldots, p$ of equal size. Every worker is assigned a work unit $u_i^m = \{k_{ij}^m \mid 1 \leq j \leq q\}$, which requires N_i/p points to compute. After the controller has received the results of all the work units u_i^m, it reassembles the κ_{ij} values $\kappa_{ij} = \sum_m k_{ij}^m$ along row i, furthermore computing $\bar{K}$ and $\bar{E}$. The algorithm is detailed in Figure 11.

Note that the work unit size increases from one row to the next and the error generally decreases in a stepwise manner. We found that, for this application, the main source of a possible degradation of efficiency is breaking loss, which occurs when some workers compute part of a row that is not going to be included in the results. Furthermore, the avoidance of breaking/starvation loss appears to play a key role in achieving good scalability.

The idea behind the above work splitting strategy is to allow all the workers to work in the same row at the same time. If the precision is not reached after completing a particular row, they would then move on to the next row. Particularly in a heterogeneous environment,

Worker:

```
Compute initial u_i^m
do
    Send u_i^m to Controller
    Receive taskmsg (m, i, Termination)
    Compute u_i^m
until Termination
```

Controller:

```
Termination ← 0
do
    Receive u_i^m from worker w
    if (received all u_i^m, m = 1 ... p)
        for(j=1 ... q) Compute κ_{i,j}
        Compute K̄_{N_i}, Ē_{N_i}, K̄, Ē
        if ( Ē ≤ max(ε_a, ε_r|K̄|) )
            Termination ← 1
    if (not Termination)
        Select a new (m, i)
        Send taskmsg (m, i, Termination) to worker w
until Termination
```

Fig. 11. Asynchronous quasi-Monte Carlo algorithm (controller not acting as a worker).

where the processors may be operating at different speeds, setting the workers up to do useful work in the same row at the same time requires a modification of the strategy.

One possible way to handle this situation is by "proportional" row splitting. For each row, a worker gets a slice of work proportional with its processor power. Although this method seems natural, it may be difficult to obtain a good estimate for the processor power.

Another method, which we implemented, is "constant" row splitting. This strategy splits each row in a constant number of work units: $C = Kp$ where K is a constant. If some processors are slower than others, they will process fewer units from each row. This technique has the advantage that the total number of messages is constant across the rows. It is possible for this strategy to introduce a small amount of breaking loss if the last required work unit is assigned to one of the slower processors.

This algorithm provides us with an asynchronous method for general-purpose QMC integration. PARINT furthermore relies on it for the computation of multivariate normal and

multivariate t-distribution integrals of the form

$$P(\mathbf{a}, \mathbf{b}, \Sigma) = \frac{1}{\sqrt{|\Sigma|(2\pi)^d}} \int_{\mathbf{a}}^{\mathbf{b}} e^{-\frac{1}{2}\mathbf{x}^\tau \Sigma^{-1}\mathbf{x}} d\mathbf{x} \tag{8}$$

and

$$T(\mathbf{a}, \mathbf{b}, \Sigma, \nu) = \frac{\Gamma(\frac{\nu+d}{2})}{\Gamma(\frac{\nu}{2})\sqrt{|\Sigma|(\nu\pi)^d}} \int_{\prod_{i=1}^{s}[a_i,b_i]} (1 + \frac{1}{\nu}\mathbf{x}^\tau \Sigma^{-1}\mathbf{x})^{-\frac{\nu+d}{2}}\, d\mathbf{x}, \tag{9}$$

respectively, where Σ is a symmetric positive definite covariance matrix and the integration domain can be finite or infinite in the different coordinate directions. T adheres to the definition of the standardized d-variate t-distribution function with ν degrees of freedom.

These problems are treated via a series of transformations after a Cholesky decomposition of the Σ matrix [32,35]. We approximate the resulting integral using the asynchronous QMC algorithm.

7 Applications

The integral $\int_0^1 \int_0^1 \frac{2\alpha y}{(x+y-1)^2+\alpha^2}\, dx\, dy$, where the parameter α ranges between 10^{-1} and 10^{-6} mimics the behavior of a class of cross section calculations in particle physics [5]. The integrand has a peaked (ridged) behavior at $y = 1 - x$. With $\alpha = 10^{-6}$, the height of the ridge is $2 \times 10^6 y$. In benchmarks, of the order of 10^8 function evaluations were required to achieve 5 digit accuracy. Yet the actual integrals are more complicated and lengthy to compute, and of higher dimensions.

The process studied in [5] is concerned with the $e+\ e- \rightarrow \mu^+\ \mu^-\ \gamma$ interaction (one electron and one positron collide to produce two muons and a photon). After some analytic manipulation the computation requires solving a four-dimensional integration problem. Using an adaptive Monte Carlo method, the computation time reported in [5] is around 45 hours (on one processing element of a Fujitsu VPP500 machine). The adaptive method in PARINT used around 36 minutes on a subcluster of 12 Athlon processor-based PC's (each an Athlon 1.2Ghz with 512MB memory).

Figure 12 reports the speedups obtained with the parallel QMC algorithm on a homogeneous network, for an integral arising in computational finance [36,37,34]. The integrand f($\mathbf{x}$) has a Gaussian weight and a factor of the form

$$g(\mathbf{x}) = C \sum_{k=1}^{d} \frac{((1 - w_k(\mathbf{x})) + w_k(\mathbf{x})c_k) \prod_{j=1}^{k-1}(1 - w_j(\mathbf{x}))}{\prod_{j=0}^{k-1}(1 + i_j(\mathbf{x}))},$$

where $i_j(\mathbf{x}) = i_0 K_0^j e^{\tilde{\sigma}(x_1+\ldots+x_j)}$, $K_0 = e^{-\tilde{\sigma}^2/2}$, $w_k(\mathbf{x}) = K_1 + K_2 \tan^{-1}(K_3 i_k(\mathbf{x}) + K_4)$, $c_k = \sum_{j=0}^{n-k}(1 + i_0)^{-j}$, and the domain of integration is $\Re^d$. The integral represents the current

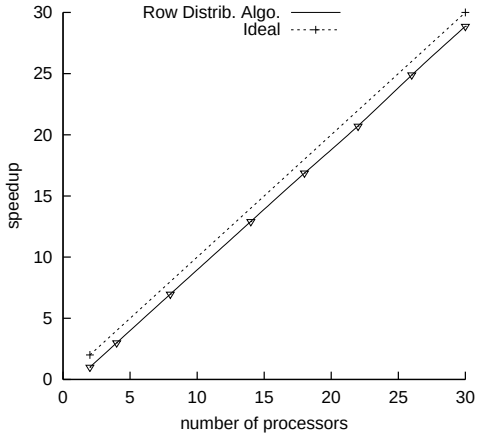

Fig. 12. Speedup and efficiency for a high dimensional integral on a heterogeneous network

value of a security backed by mortgages of length d months (so, a common value of d would be 360, corresponding to a typical 30 year mortgage) with fixed monthly interest rate i_0.

In order to map the infinite domain to the $d-$dimensional unit hypercube, the transformation $z_k = \Phi(x_k), \quad k = 1, \ldots, d$ is performed, where Φ_k is the univariate normal distribution function. The resulting integral is $\int_{\mathcal{H}_d} \mathrm{f}(\mathbf{z}) d\mathbf{z}$ with $\mathrm{f}(\mathbf{z}) = g(\Phi^{-1}(z_1), \ldots, \Phi^{-1}(z_d))$. Using the constants $C = 1$, $i_0 = .007$, $\tilde{\sigma} = .02$ and $K_1 = .01$, $K_2 = -.005$, $K_3 = 10$, $K_4 = .5$ yields Caflish and Morokoff's "nearly linear" problem [36]. Figure 12 shows speedup results for the finance problem solved for 360 dimensions. The error tolerance was set low so that termination occurred by reaching the function count limit.

Figure 13 presents preliminary results of constant row splitting strategy for an MVT problem of the form of (9) on a heterogeneous cluster. The first 32 nodes have a 1.2Ghz CPU, the other ones have a 0.8Ghz CPU (correspondingly, the ideal speedup line has a bend at 32).

For $K = 1$, each row is split in as many work units as there are workers, with all units of the same size. Note that as long as the system is homogeneous, the algorithm exhibits near ideal speedup; however the efficiency decreases sharply when the system has at least one or more workers of lower power.

For $K = 2$ and $K = 3$, the efficiency improves (as compared to $K = 1$) for the heterogeneous system; however the performance is slightly worse as long as the system is homogeneous. We attribute the latter to small breaking loss, as well as an increase in the number of messages required to finish the same problem.

8 Conclusions and Future Work

This paper gives a survey of the current status of the PARINT project and our available software. We are continuing to implement new techniques and refinements, with respect to the distributed algorithms as well as the numerical integration techniques.

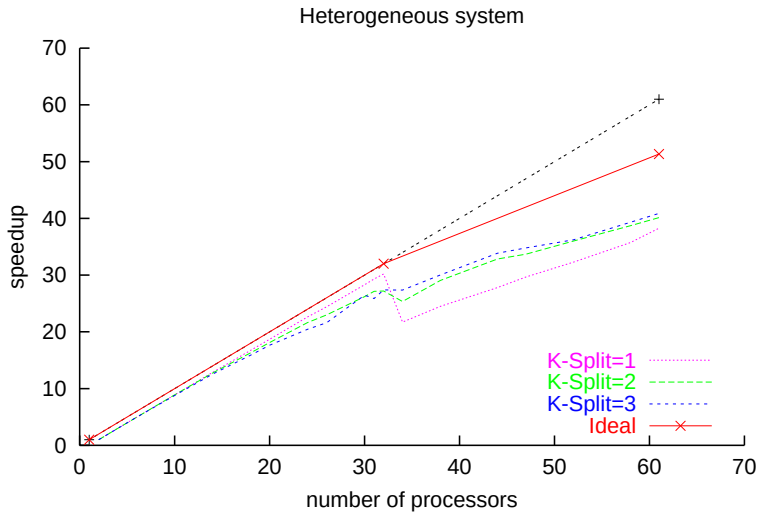

Fig. 13. Speedup for MVT integral on a homogeneous network

As one area of development, for the adaptive algorithms we want to focus on scalability by tackling the work anomaly problem. Further analysis and experimentation is needed to explore this problem, including following up on some interesting similarities with the scalability of branch and bound algorithms. To ensure the availability of work where and when it is needed (and thus avoid performing useless work), we are investigating modifications of the load balancing methods. These include, for example, determining the amount of work transferred in load balancing steps dynamically, as well as performing "pre-emptive" load balancing. Different notions on how to measure an "amount" of integration work have been investigated. Alternative solutions involve decentralizing the current strategies.

We have added techniques to the package to expand the range of problems that can be solved. These include asynchronous quasi-Monte Carlo methods for solving problems of moderately high to high dimensions, with a general-purpose algorithm and with special-purpose code for MVN/MVT applications. Methods for integration over a simplex have been added and will be extended to integration over a set of simplexes. We are working on a transformation interface to allow for various types of domains, and a parallel Monte Carlo code for erratic functions and regions. We have furthermore explored extrapolation techniques for singular problems, and a hierarchical strategy for the concurrent computation of large collections of integrals (as for finite element problems).

Corresponding developments in the package's graphical user interface allow for easy use across problem areas and research disciplines. As a goal of a current project to build PARINT into a Problem Solving Environment (PSE), a server will be provided allowing users to submit integration problems remotely, using a web browser based GUI as the interface and allowing platform independence for the end-user.

References

[1] The MPI Forum, The MPI message-passing interface standard, `http://www.mcs.anl.gov/mpi/standard.html` (May 1995).

[2] R. Zanny, E. de Doncker, PARINT1.1 *User's Manual*, available at `http://www.cs.wmich.edu/parint` (2001).

[3] E. de Doncker, A. Gupta, J. Ball, P. Ealy, A. Genz, ParInt: A software package for parallel integration, in: 10^{th} ACM International Conference on Supercomputing, Kluwer Academic Publishers, 1996, pp. 149–156.

[4] E. de Doncker, A. Gupta, A. Genz, P. Ealy, J. Liu, A. Sureka, Use of ParInt for the parallel computation of statistics integrals, Computing Science and Statistics 27.

[5] K. Tobimatsu, S. Kawabata, Multi-dimensional integration routine DICE, Tech. Rep. 85, Kogakuin University (1998).

[6] E. de Doncker, R. Zanny, K. Kaugars, L. Cucos, Performance and irregular behavior of adaptive task partitioning, in: Lecture Notes in Computer Science, Vol. 2074, Springer-Verlag, 2001, pp. 118–127.

[7] A. Genz, A. Malik, An adaptive algorithm for numerical integration over an n-dimensional rectangular region, Journal of Computational and Applied Mathematics 6 (1980) 295–302.

[8] A. Genz, A. Malik, An imbedded family of multidimensional integration rules, SIAM J. Numer. Anal. 20 (1983) 580–588.

[9] J. Berntsen, T. O. Espelid, A. Genz, An adaptive algorithm for the approximate calculation of multiple integrals, ACM Trans. Math. Softw. 17 (1991) 437–451.

[10] A. Genz, An adaptive numerical integration algorithm for simplices, in: N. A. Sherwani, E. de Doncker, J. A. Kapenga (Eds.), Lecture Notes in Computer Science, Vol. 507, 1990, pp. 279–285.

[11] R. Piessens, E. de Doncker, C. W. Überhuber, D. K. Kahaner, QUADPACK, A Subroutine Package for Automatic Integration, Springer Series in Computational Mathematics, Springer-Verlag, 1983.

[12] J. R. Rice, A metalgorithm for adaptive quadrature, Journal of the Association for Computing Machinery 22 (1975) 61–82.

[13] H. D. Shapiro, Increasing robustness in global adaptive quadrature through interval selection heuristics, ACM Transactions on Mathematical Software 10 (2) (1984) 117–139.

[14] R. Zanny, E. de Doncker, Work anomaly in distributed adaptive partitioning algorithms, in: Proceedings of the High Performance Computing Symposium (HPC'00), 2000, pp. 178–183.

[15] E. Horowitz, S. Sahni, D. Mehta, Fundementals of Data Structures in C++, Computer Science Press, 1995.

[16] J. Li, E. de Doncker, Dynamic visualization of computations on the internet, in: Lecture Notes in Computer Science, Vol. 1593, Springer-Verlag, 1999, pp. 360–369.

[17] J. Kapenga, E. de Doncker, A parallelization of adaptive task partitioning algorithms, Parallel Computing 7 (1988) 211–225.

[18] R. Zanny, K. Kaugars, E. de Doncker, Scalability of branch-and-bound and adaptive integration, in: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'01), 2001, pp. 674–680.

[19] R. R. Zanny, Efficiency of distributed priority queues in parallel adaptive integration, Master's thesis, Western Michigan University (1999).

[20] A. M. M. D'Apuzzo, M. Lapegna, Scalability and load balancing in adaptive algorithms for multidimensional integration, Parallel Computing 23 (1997) 1199–1210.

[21] J. Lyness, An error functional expansion for n-dimensional quadrature with an integrand function singular at a point, Mathematics of Computation 30 (1976) 1–23.

[22] J. Clausen, J. L. Träff, Do inherently sequential branch-and-bound algorithms exist?, Parallel Processing Letters 4 (1 & 2) (1994) 3–13.

[23] A. de Bruin, G. Kindervater, H. Trienekens, Asynchronous parallel branch and bound and anomalies, Tech. rep., Erasmus University, Department of Computer Science (1995).

[24] G.-J. Li, B. Wah, Coping with anomalies in parallel branch-and-bound algorithms, IEEE Transactions on Computers C-35 (6) (1986) 568–573.

[25] D. Hart, E. Kraemer, G.-C. Roman, Interactive visual exploration of distributed computations, in: Proceedings of 11th International Parallel Processing Symposium, 1997, pp. 121–127.

[26] E. de Doncker, K. Kaugars, A new paradigm for scientific visualization, in: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), 1999, pp. 1384–1389.

[27] K. Kaugars, E. de Doncker, R. Zanny, ParVis: Visualizing distributed dynamic partitioning algorithms, in: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'00), 2000, pp. 1215–1221.

[28] R. A. Aydt, The Pablo self-defining data format, latest revision 2000. Available at `http://www-pablo.cs.uiuc.edu` (1992).

[29] P. J. Davis, P. Rabinowitz, Methods of Numerical Integration, Academic Press, New York, 1984.

[30] A. Papageorgiou, J. F. Traub, New results on deterministic pricing of financial derivatives, Tech. rep., Columbia University, Department of Computer Science, cUCS-028-96 (1996).

[31] A. Papageorgiou, J. F. Traub, Faster evaluation of multidimensional integrals, Computers and Physics 11 (6) (1997) 574–578.

[32] A. Genz, MVNDST Mult. Normal Dist. software, available from web page at http://www.sci.wsu.edu/math/faculty/genz/homepage (1998).

[33] R. Cranley, T. N. L. Patterson, Randomization of number theoretic methods for multiple integration, SIAM J. Numer. Anal. 13 (1976) 904–914.

[34] L. Cucos, E. de Doncker, Distributed QMC algorithms: New strategies and performance evaluation, in: Proceedings of the High Performance Computing Symposium (HPC'02), 2002, pp. 155–159.

[35] A. Genz, F. Bretz, Numerical computation of multivariate t-probabilities with application to power calculation of multiple contrasts, J. Stat. Comp. Simul. 63 (1999) 361–378, http://www.sci.wsu.edu/math/faculty/genz/homepage.

[36] R. Caflisch, W. Morokoff, Quasi-Monte Carlo computation of a finance problem, in: Proceedings of the Workshop on Quasi-Monte Carlo Methods and their Applications, Statistics Research and Consultancy Unit, Hong Kong Baptist Univeristy, 1996, pp. 15–30.

[37] A. Genz, J. Monahan, A stochastic algorithm for high dimensional integrals over unbounded regions with Gaussian weight, J. Comp. Appl. Math. To appear, preprint available from website at http://www.sci.wsu.edu/math/faculty/genz/homepage.