Multivariate Quadrature on MIMD Machines with Shared or Distributed Memory

Ignatios E. Vakalis
Western Michigan University

Follow this and additional works at: https://scholarworks.wmich.edu/dissertations
Part of the Applied Mathematics Commons

Recommended Citation
https://scholarworks.wmich.edu/dissertations/1990
MULTIVARIATE QUADRATURE ON MIMD MACHINES
WITH SHARED OR DISTRIBUTED MEMORY

by

Ignatios E. Vakalis

A Dissertation
Submitted to the
Faculty of The Graduate College
in partial fulfillment of the
requirements for the
Degree of Doctor of Philosophy
Department of Mathematics and Statistics

Western Michigan University
Kalamazoo, Michigan
December 1992
MULTIVARIATE QUADRATURE ON MIMD MACHINES
WITH SHARED OR DISTRIBUTED MEMORY

Ignatios E. Vakalis, Ph.D.
Western Michigan University, 1992

For a procedure as numerical integration, of high computational expense which is used extensively in large-scale computations, it is natural to aim at the design of algorithms which can be used on parallel computers. This work deals with the design of efficient and portable parallel algorithms on MIMD (Multiple Instruction Multiple Data) architectures with shared memory, and on distributed memory systems.

A parallel global adaptive algorithm is presented for multivariate integration over simplex type regions. Process synchronization is achieved through the use of monitors. Macros were developed, for managing the task pool with a heap data structure. Layered over the Argonne monitor macro package, the algorithm is portable to a variety of machines.

Convergence properties and speedup are analyzed for a class of global adaptive multivariate integration algorithms on shared memory MIMD machines. The analysis considers functions with a given number of continuous derivatives in the (cube or simplex) integration region, with possible exception of a type of vertex singularity.
In many problems, integration is required over a triangularized (two-dimensional) region. A parallel global adaptive algorithm is presented for integration over a set of triangles on distributed memory systems. On such systems, the task pool which evolves adaptively in the course of the computations is distributed over the local memories of the processors. The Reactive Kernel/Cosmic Environment is used as the processor communication system, thereby also assuring the algorithm's portability to various systems. Two new dynamic load balancing systems governing the flow of work between processors, were constructed and tested.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Multivariate quadrature on MIMD machines with shared or distributed memory

Vakalis, Ignatios E., Ph.D.

Western Michigan University, 1992
Αφιερώνεται στοις γονείς μου
και στη κορη μου Claire
ACKNOWLEDGEMENTS

I could not have completed this work without the help and support of many people. I must first thank my parents. Right from the start they have always impressed in me the importance of education and without their support I would not be where I am today. I owe them much more than I can ever give.

Many, many thanks must be paid to my advisor, Dr. Elise de Doncker. I would not have been able to complete this work without her expertise, knowledge, and invaluable suggestions and insights. I hope we can continue to work together in the future. Thanks also to the other members of my committee: Dr. Dennis Pence, who was always willing to give advice, Dr. Joseph McKean, Dr. J. Donald Nelson, and Dr. Ian Robinson. I would also like to thank Margo for all her excellent typing, Dr. Jay Treiman for his help with TeX, and my good friend Joe for all his programming advice.

This year, 1992, has been a year of great achievements. One such achievement is finishing this work and obtaining my Ph.D which has been my dream for many years. But, I must confess that the birth of my daughter, Claire, in January 1992 was something that far exceeded any such dream. Because of this, I would like to dedicate this work to Claire, in the hopes that one day she might look at it and yearn for furthering her studies too.
Acknowledgements – Continued

Finally, many thanks to my wife, Marianne, who has helped in many ways too numerous to mention. She and Claire provide me with so much love that I really don’t need anything else. They are my constant companions and a constant source of joy. I hope they know that I love them both dearly.

Ignatios E. Vakalis
# TABLE OF CONTENTS

ACKNOWLEDGEMENTS .............................................................. ii

LIST OF TABLES ........................................................................ vi

LIST OF FIGURES ....................................................................... viii

CHAPTER

I. INTRODUCTION ................................................................. 1

II. REVIEW OF THE LITERATURE ............................................. 9

III. A PARALLEL GLOBAL ADAPTIVE ALGORITHM FOR INTEGRATION OVER THE N-DIMENSIONALS SIMPLEX ON SHARED MEMORY MACHINES ............ 22

3.1. Introduction ................................................................. 22

3.2. Background ................................................................. 23

3.3. Parallel Global Adaptive Algorithm ................................. 26

3.4. Alternative Data Structures ............................................. 31

3.5. Some Test Results ....................................................... 36

IV. SPEEDUP ANALYSIS OF ADAPTIVE INTEGRATION ALGORITHMS ON MIMD-SM MACHINES ............... 46

4.1. Introduction ................................................................. 46

4.2. Classification of Some Adaptive Quadrature Strategies .......... 47

4.3. Error Estimate of a Quadrature Rule ................................ 50
# Table of Contents –Continued

## CHAPTER

4.4. Speedup Analysis ........................................................ 63

V. A PARALLEL GLOBAL ADAPTIVE ALGORITHM FOR INTEGRATION OVER A SET OF TRIANGLES ON LOOSELY COUPLED SYSTEMS ............................ 82

5.1. Introduction ................................................................. 82
5.2. Background ................................................................. 83
5.3. Parallel Algorithm and Implementation .................. 90
5.4. Dynamic Load Balancing ........................................... 97
5.5. Some Test Results ...................................................... 105

VI. SUMMARY AND FUTURE WORK ................................. 116

6.1. Summary ................................................................. 116
6.2. Future Work ................................................................. 117

## APPENDICES

A. List of User-Defined Macros ........................................................... 123
B. Message Polling Loop on a Computational Node ....................... 126
C. The Mediator Program .................................................................... 129

## BIBLIOGRAPHY ................................................................. 137

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
LIST OF TABLES

1. Worst Case Time Complexities of Create, Insert, and Delete-Max Operations on a Structure of \(N\) Items. ................................................................. 34

2. Test Integrand Functions. ................................................................. 37

3. Execution Time and Speedup as a Function of NP for Integrand Function #2 On the Sequent Symmetry and Encore Multimax. .......................... 40

4. Execution Time and Speedup as a Function of NP for Integrand Function #2 On the Alliant FX/8. ................................................................. 41

5. Execution Time and Speedup as a Function of NP for Integrand #3 On the Sequent Symmetry and Encore Multimax. ................................. 42

6. Execution Time and Speedup as a Function of NP for Integrand #3 On the Alliant FX/8. ................................................................. 43

7. Execution Time and Speedup as a Function of NP for Integrand #4 On the Sequent Symmetry and Encore Multimax. ................................. 44

8. Execution Time and Speedup as a Function of NP for Integrand #4 On the Alliant FX/8. ................................................................. 45

9. Test Integrand Functions. ................................................................. 106

10. iPSC/2 Host Times. ........................................................................ 108

11. Symult-2010 Host Times and Speedup for Integrand Function #1. ........................................................................ 109

vi

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
List of Tables—Continued

12. Symult-2010 Host Times and Speedup for
Integrand Function #2. ....................................................... 111

13. Symult-2010 Host Times and Speedup for
Integrand Function #3. ....................................................... 113

14. Symult-2010 Host Times and Speedup for
Integrand Function #4. ....................................................... 114

15. Symult-2010 Host Times and Speedup for
Integrand Function #5. ....................................................... 115
<table>
<thead>
<tr>
<th>Figure Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A Programmer’s View of a Multicomputer.</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>Block Diagram of a Metalgorithm for Adaptive Quadrature.</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>Creation of the Slave Process.</td>
<td>28</td>
</tr>
<tr>
<td>4</td>
<td>FORTRAN Code for the Subroutine SLAVE.</td>
<td>28</td>
</tr>
<tr>
<td>5</td>
<td>Execution Time vs. Number of Processes and Speedup vs. the Number of Processes For Integrand #1 On the Alliant FX/8.</td>
<td>38</td>
</tr>
<tr>
<td>6</td>
<td>Execution Time vs. the Number of Processes and Speed-up vs. the Number of Processes For Integral #1, On the Sequent Symmetry.</td>
<td>38</td>
</tr>
<tr>
<td>7</td>
<td>Execution Time vs. the Number of Processes and Speed-up vs. the Number of Processes For Integral #2, On the Encore Multimax.</td>
<td>39</td>
</tr>
<tr>
<td>8</td>
<td>Flow of Data and Control.</td>
<td>92</td>
</tr>
<tr>
<td>9</td>
<td>Load Balancing Scheme #1.</td>
<td>101</td>
</tr>
<tr>
<td>10</td>
<td>Load Balancing Scheme #2.</td>
<td>103</td>
</tr>
<tr>
<td>11</td>
<td>Sample Triangles and Processor Assignment With 2 Computational Processors.</td>
<td>107</td>
</tr>
<tr>
<td>12</td>
<td>Speedup Curves For Integrand Function #1 on Symult-2010.</td>
<td>110</td>
</tr>
<tr>
<td>13</td>
<td>Symult-2010 Host Times Without Load Balancing</td>
<td>111</td>
</tr>
<tr>
<td>14</td>
<td>Symult-2010 Speedup Curve Using Scheme #2.</td>
<td>112</td>
</tr>
</tbody>
</table>
List of Figures – Continued

15. Symult-2010 Host Times For Oscillating Integral. ............... 113

16. Speedup Curves for Integrand Function #4
    on Symult-2010. ................................................................. 115
CHAPTER I

INTRODUCTION

The numerical evaluation of integrals is one of the oldest problems in mathematics. The beginnings of this subject can be traced at least as far back as 2,300 years ago to the era of the Ancient Greek civilization. A fine example of ancient numerical integration was given by the ingenious Archimedes. His quadrature for the circle was developed by means of inscribed and circumscribed regular polygons. He calculated the area of a series of inscribed polygons as he was successively doubling the number of edges. His ingenuity lies in the fact that by doubling the number of edges he was able to drastically reduce the error for his approximation of the area of the circle. This technique became known as extrapolation to the limit zero of the polygon side. Note that the side $S_{2n}$ of an inscribed $2n$-polygon in terms of the side of an $n$-polygon (radius = 1) is given by $S_{2n} = (2 - (4 - S_n^2)^{1/2})^{1/2}$, while $S_5 = 1$. In this way Archimedes obtained the first strict bounds for the number $\pi$, as $22/7 < \pi < 223/71$ (Engels, 1980). His idea was much later used more generally to raise the order of accuracy for some approximation methods. Richardson (1927) was the first to apply this technique in numerical quadrature and in 1955 the principle was rediscovered yielding the Romberg integration method (Romberg, 1955).

1
After the invention of functions in the modern sense of mappings in this century, the idea of integration was born in order that the areas of regions and volumes of objects might be measured. Much effort has been devoted to the techniques for the analytic evaluation of one- and multi-dimensional integrals. However, most of the routine integrals in practical scientific work are incapable of being evaluated in closed form. Even if an expression can be derived for the value of an integral, often this can be only seen after an extraordinary amount of error prone algebraic manipulations.

In the last century many important contributions have been established towards the creation of successful quadrature formulas, i.e., numerical rules that approximate the value of a definite integral by using information about the integrand function only at a set of discrete points. Many general quadrature formulas have been developed while others target specific classes of integrals where the integrand function has different types of vertex and/or edge singularities or is highly oscillating or the integral is divergent.

Numerical integration is paradoxically both simple and exceedingly difficult. It is simple because it can often be successfully resolved by the simplest methods. It is difficult in several respects. It may require an enormous amount of computing time and possibly lead to some unfavorable situation tending towards impossibility. The construction of quadrature methods sometimes leads to some of the deepest portions of pure and applied analysis as well as other areas of mathematics like linear algebra, group theory, etc. Also the problem
of numerical integration is open ended. No finite collection of techniques is likely to cover all possibilities that arise and so an extra bit of ingenuity and special knowledge may be required in order to handle the numerical evaluation of some integrals.

Recently some computer procedures have been developed which can perform analytic integration whenever it is possible. Some of the most powerful and popular symbolic integration systems are incorporated into algebraic computational systems such as REDUCE, MACSYMA, MAPLE, MATHEMATICA. Successful analytic integration of rather simple functions can result in lengthy formulas. This can also involve the evaluation of complicated functions which can only be computed approximately. Therefore it is natural to attempt to approximate integrals directly for a wide range of integrands rather than performing lengthy and problem specific algebraic manipulations.

As mentioned above, many quadrature rules have been developed which make up the building blocks upon which quadrature algorithms can be built. A numerical integration algorithm can be thought of as an implementation of one or more rules, but it is more than that. A successful algorithm must also include strategies for the selection of the next subregion from a pool of candidates, a method for subdivision of the subregion at hand, and an efficient data structure for storing the pool of intervals. It must also contain procedures for the estimation of the error and an integral value over each subinterval, and demonstrate its ability to gracefully handle a wide range of
error abnormalities which may arise during the course of the computation and which usually dictates termination of the algorithm.

With the development of fast computer systems over the last decades many integration algorithms have been published. The majority of these deal with the numeric approximation of one-dimensional integrals and the most prominent are characterized as being automatic. For automatic routines the most important parameters in the input supplied by the user, are the integrand function, the region of integration, and the level of the desired accuracy. Upon return, the program provides an estimate of the integral as well as an error estimate which hopefully satisfies the accuracy requirements set by the user. The basic problem for automatic integration is the calculation of an approximation \( Q \) to the integral \( I = \int_D f(x)dx \) and of an error estimate or upper bound \( E_a \) satisfying \( |I - Q| < E_a < \varepsilon = \max\{\varepsilon_a, \varepsilon_r|I|\} \) where \( \varepsilon_a \) and \( \varepsilon_r \) are the tolerated absolute and relative error. Some standard areas of integration \( (D) \), are the \( n \)-cube, the \( n \)-simplex, or the \( n \)-sphere. Some regions can be transformed into standard regions or can be split into regions which can then be transformed into standard regions. Automatic routines can be classified as adaptive or non-adaptive. For the non-adaptive algorithms, the evaluation points used in the quadrature rules are fixed and predetermined, except for the case of Monte-Carlo type algorithms. For the adaptive routines, the evaluation points are clustered in the neighborhood of difficult spots for each integrand. Therefore, the adaptive routines are, in general, more accurate.
but also more difficult to program. Adaptive algorithms can also be subdivided into two main categories: the local and the global adaptive, even though many hybrids are possible (Shapiro, 1984). The classification of an adaptive algorithm depends on the strategy used for selecting the next subinterval of integration as well as on the criteria for determining if the appropriate error tolerance is satisfied on an interval. A global adaptive method is characterized by its selection at each step of the algorithm of one or more intervals from the pool of regions, their subdivision into two or more subregions, and the insertion of the latter into the pool until a termination criterion is met relating to global quantities (i.e., to the entire pool). The adaptivity component of the algorithm results from the fact that regions are selected according to their assigned priority. As a consequence the course of computations differs for individual problems since the priorities issued during the integration process are problem dependent. One of the most successful software packages for one dimensional numerical integration is QUADPACK (Piessens, de Doncker, Uberhuber, and Kahaner, 1983) which has been adopted by major numerical libraries such as NAG, IMSL, SLATEC.

In passing from one dimension to several dimensions, the diversity of integrals and the difficulty in handling them is greatly increased. For example, in one dimension we can restrict our attention to three different types of intervals of integration: finite, infinite in one direction, and doubly infinite, whereas in several dimensions there are many different types of integration re-
regions of potential interest. Also the behavior of functions of several variables can be considerably more complicated and our experience and intuition with them is much more limited. Due to the significant increase of the number of integrand evaluations needed for higher dimensions, numerical multivariate integration may be very computationally expensive and in some cases unfeasible, particularly when the dimension is high, the integrand exhibits irregular behavior, or the integration region is highly irregular. A limited number of algorithms and, in particular, adaptive routines have been published for the cases where the dimension of the integration region is \( n \geq 3 \).

To overcome the computational burden which arises in multivariate numerical integration, an exploration for parallelization of existing sequential algorithms as well as the development of new parallel algorithms has appeared in the last few years. Parallel computers with varying architectures offer different degrees of suitability for implementing parallel numerical algorithms. A brief survey of the parallel architectures suitable for our purposes and of the existing parallel multivariate numerical integration routines reported in the literature, is presented in the next chapter. The motivation for this work was to construct portable parallel global adaptive algorithms for multivariate numerical integration on shared and distributed memory machines and analyze their performance. Thus one of the presented algorithms runs successfully on a range of multiple instruction multiple data shared memory machines, while the other is portable among various distributed memory machines and networks.
of workstations.

In the remaining section of this chapter, an outline of the subsequent chapters emphasizing the main points of new contributions is presented. In particular, Chapter II provides a brief review of the literature on three topics. First, an outline is given of the various types of parallel computers. Second, the most important quadrature algorithms used for integration over a triangle or a set of triangles are listed. We also outline algorithms used for integration over the $n$-simplex ($n \geq 3$) and the existing parallel routines for which the integration region is the $n$-simplex where $n \geq 2$. Mention is also made of the types of parallel machines used for their implementation. One of our algorithms targeted at distributed memory machines for integration over a set of triangles uses a new load balancing method derived from a scheme by Smith and Schnabel (1991). The third part of Chapter II presents a review of some suitable load balancing algorithms.

In Chapter III, a new parallel global adaptive algorithm is given for the integration over the $n$-dimensional simplex, which can be used on various multiple instruction multiple data shared memory machines (MIMD-SM). The portability of the algorithm is achieved through the use of the Argonne monitor macros. The macro package was employed as a bottom layer in the task pool management scheme and new primitives were developed to support a heap data structure on the pool of regions. Test results are presented demonstrating the speedup of the algorithm as it runs on an Alliant FX/8, Sequent Symmetry,
and an Encore Multimax. Alternative data structures are also discussed with regard to potential improvement of the performance of the algorithm. Chapter IV presents the speedup analysis of a region-size global adaptive algorithm for multivariate integration on MIMD-SM machines. Theoretical results are obtained for certain classes of integrand functions.

A portable parallel multivariate integration algorithm is given in Chapter V which uses a new load balancing scheme. Test and performance results for a range of integrals as the algorithm runs on a variety of distributed memory machines is presented. The current implementation of the algorithm is in C, and its portability is achieved by the use of the Reactive Kernel/Cosmic Environment (RK/CE) by Seitz et al. (1988) as the node communication system. The "reactive handler" which is part of the reactive kernel of the RK/CE system, dispatches user processes and supports a set of message passing routines called from C programs. Chapter V also presents two different versions of a new load balancing strategy incorporated into the parallel integrator.

Conclusions and future research plans are outlined in Chapter VI. The appendices contain the code for the high level macros used for the implementation of the algorithm on shared memory machines as well as the code for the load balancing algorithm as it runs on a designated node, called the mediator. A segment of C code for handling the message queue on a computational node is also included.
CHAPTER II

REVIEW OF THE LITERATURE

Numerical integration forms the basis of many large scale computations and is often computationally demanding, especially in the cases of high dimensionality, unspecified irregular integral behavior, or irregularity of the integration region. To overcome the computational burden which arises in multivariate numerical integration, the construction of efficient parallel algorithms has begun in the last few years. Since adaptive multivariate numerical integration routines are considered for "general" usage in a low to average number of dimensions and are superior in most cases to the non-adaptive ones, a brief overview of the most commonly used parallel architectures and their suitability for implementing adaptive numerical integration algorithms will be presented (de Doncker, 1991; Genz, 1987).

The vector processor machines use a single stream of instructions. The special vector hardware allows a single instruction to operate on vector objects with much greater speed than a sequence of single instructions accomplishing the same task. Vector processors are best suited for performing operations which may include the scalar multiplication of vectors, the addition of vectors, or the multiplication of matrices. Taking full advantage of the vector capabilities on these machines is hard for adaptive numerical integration, apart from
the rule evaluation which can be seen as a scalar product. Gladwell (1987) presented his experiences with vectorization of one-dimensional quadrature codes.

The multiple instruction multiple data shared memory machines (MIMD-SM) are architectures on which it is relatively easy to parallelize adaptive numerical integration algorithms. These architectures typically allow 2 to 32 processors which access the same memory. To use a MIMD-SM machine as a parallel computer in a problem, the programmer must usually make his parallelization explicit in the program. This is unlike the vector processing case. Also no special load balancing or data movement is needed. In Chapter III, a new portable parallel algorithm for multivariate numerical integration over the $n$-simplex is presented. This routine is portable together with the Argonne monitor macros (Lusk and Overbeek, 1983, 1984) to a variety of machines including an Alliant FX/8, a Sequent Symmetry, and an Encore Multimax.

Another type of parallel architecture is the multiple instruction multiple data distributed memory machines (MIMD-DM). They often consist of 16 to 1042 processors and thus remove the limit on the number of processors which constrains the shared memory machines. The development environment for these machines is much weaker than on MIMD-SM machines, the debugging tools being one of the prime examples. These machines offer the potential for outstanding performance in parallel adaptive integration. The problem is that load balancing and data transfer between processors as well as distribu-
tion and collection of global information updates must be explicitly managed. This makes the program development much harder than on MIMD-SM machines. There are many attempts underway to develop machine independent environments to ease this burden and thus allowing portable parallel algorithms to be developed (EXPRESS, 1987; Seitz et al., 1989). In this work, we have implemented a new load balancing method for adaptive integration on MIMD-DM machines, which handles the flow of tasks between overloaded and underloaded nodes. The integration algorithm is layered over RK/CE and runs on a range of MIMD-DM machines such as iPSC/2, Symult-2010 as well as on a collection of networked workstations.

A multicomputer or a message-passing concurrent computer consists of $N$ computing nodes connected by a message-passing communication network. This terminology will also be used in Chapter V. Figure 1 depicts a simplified programmer's view of a multicomputer (Seizovic, 1988).

![Communication Network Diagram](image)

Figure 1. A Programmer's View of a Multicomputer.
Multicomputers are distributed memory systems including local memory multiprocessors and networked workstations. Each node of a multicomputer has a memory that is physically separate and logically private from those on other nodes. Thus, there is no global memory address. A copy of an operating system exists in each of the computing nodes \((C_0, C_1, \ldots, C_{N-1})\). The node operating system supports multiple processes and provides them with an interface to the communication network. The latter performs the message routing enabling each process to communicate with any other process. The development of first generation multicomputers began with the Cosmic Cube project (Seitz, 1985). All first generation multicomputers used a binary \(n\)-cube interconnection network and software controlled store-and-forward message routing. Recently the second generation of multicomputers has appeared (e.g., iPSC/2, iPSC/860, Symult-2010, and nCUBE-2). The performance of the message routing network has improved dramatically as a combined result of using wormhole routing (Dally and Seitz, 1987) and high performance routing hardware (Flaig, 1987).

Finally, the single instruction multiple data (SIMD) architectures have a single instruction stream which operates on multiple copies of the data at the same time, unlike the previous machines where each processor could have a separate program. SIMD machines like the DAP are best suited to applications like linear algebra and image processing. The Connection Machine from Thinking Machines Inc. is another SIMD machine which can have up to \(2^{16}\)
processors. Special languages like *Lisp, C*, Paris, and Connection Fortran (CMF) are typically used for programming on these architectures. The suitability of parallelization of adaptive integration routines on SIMD machines is similar to that reported for Vector processors. To our knowledge, no parallel adaptive integration routine exists on these machines even though some initial results in this direction were published by Mascagni (1991).

There are several published routines for automatic numerical integration over an n-simplex \((n \geq 2)\), which have been implemented on several computers but a limited number of algorithms exist for integration over a collection of \(n\)-simplices \((n = 2, 3)\). Of the existing routines only a very small subset are for shared or distributed memory machines. Among the most prominent adaptive quadrature algorithms for numerical integration over a triangle are: (a) TRIADA by Haegemans (1977), (b) CUBTRI by Laurie (1982), and c) TRIEX by de Doncker and Robinson (1984) which uses extrapolation by means of the \(\varepsilon\)-algorithm (Wynn, 1956). TRIEX is able to handle not only functions with \(r^a\)-vertex singularities but also various other types of singular behavior including certain line singularities occurring along the side of the integration triangle. Some of the non-adaptive quadrature algorithms which use a triangle as the region of integration include: (a) the EXTRR routine by Hollosi and Keast (1985) which was designed explicitly to handle integrands with singularity of the type \(r^a\), where \(a\) is known by incorporating an algorithm for non-linear extrapolation, and (b) the TRILEV routine by
TRILEV applies extrapolation by the use of the Levin d-transformation (Levin, 1973) and successfully handles functions with different types of singularities. For the case where the original region of integration is composed of a set of triangles, three algorithms are: (a) TWODQD by Kahaner and Rechard (1987), (b) TRISET, an extension of TRIEX, by de Doncker, Kahaner, and Starkenburg (1990) which uses the ε-algorithm and copes with various local singularities on edges of the given triangles, and (c) DCUTRI by Bernsten and Espelid (1990) which can also handle the numerical integration of vector functions $\vec{f}(x, y)$. Note that the routines designed for integration over a collection of triangles can also be used for the case where the region of integration consists of one triangle.

For higher dimensions, $n \geq 3$, the number of published routines for numerical integration decreases substantially. For example, D01PAF in the NAG library is a non-adaptive routine for integration over the $n$-simplex, while SCUSMP is a recently developed global adaptive algorithm (Genz, 1991). DCUTET is the only existing global adaptive routine for integration over a collection of tetrahedrons (Bernsten, Cools, and Espelid, 1990). Parallel computers with shared memory are prime candidates for parallelization of recently developed routines. In particular, DCUTRI and DCUTET have been implemented on MIDM-SM machines using compiler directives to achieve the parallelization. It must be noted that using this approach to handle the synchronization in the parallel code makes the algorithm compiler dependent.
and therefore eliminates its portability. Genz (1987) has also used this approach in constructing a parallel routine for integration over an n-cube where $2 \leq n \leq 10$. On the other hand, a portable parallel algorithm, CUBEX for shared memory machines using monitor macros, was implemented by de Doncker and Kapenga (1988). This algorithm performs numerical integration over a hyperrectangular (n-dimensional) region and also performs extrapolation by means of the ε-algorithm. The same approach with regards to the portability of the algorithm has been used in this work for the construction of a portable parallel algorithm for integration over the n-simplex.

Very limited work has been done in constructing multivariate numerical integration routines for MIMD-DM machines. Special load balancing routines must be incorporated into the integration algorithm in order for the routine to handle a wide variety of integrand functions. Genz (1987) has implemented an algorithm on the Intel hypercube and on the NCUBE machines for integration over a n-cube. Berntsen and Espelid (1987) have implemented a parallel routine on an Intel hypercube for one-dimensional integration. Since no load balancing techniques have been used in the routines reported above on DM machines, significant speedup results were obtained only for the case of smooth or oscillatory integrand functions where almost all the node processors were busy throughout the entire computation. Also no provisions have been made to ensure portability of the code and so significant changes must be performed while porting any of the above routines to a differently loosely coupled system.
A parallel algorithm over a set of triangles for MIMD-DM machines (ADLEV) has been developed by Cariño (1992). It applies extrapolation by means of the Levin d-transformation and utilizes a dynamic load balancing technique. De Doncker and Kapenga (1991) had also implemented a portable parallel algorithm (TRISET) on distributed memory machines for integration over a collection of triangles. The portability of both TRISET and ADLEV is ensured by the use of the RK/CE as the node communication system (Seitz et al. 1988) which exists on a variety of loosely coupled systems. A dynamic load balancing strategy was incorporated into the algorithms. In Chapter V we present alternative load balancing schemes which are imbedded in a quadrature routine for integration over a collection of triangles.

Rice (1974) has used the concept of a metalgorithm to describe a class of adaptive algorithms for one-dimensional integration and prove their convergence for parallel computers with shared memory. Figure 2 shows a block diagram of a metalgorithm for adaptive quadrature routines. The thick lines show the flow of intervals while the single lines indicate the flow of control and other information among the participating processors of the parallel computer. The interval collection is organized into some data structure, and the purpose of the interval collection manager is to create and maintain this structure. Intervals are taken from the collection, and the interval processor does the computations to improve the integral estimates and the associated error bounds. Usually the interval processor produces two intervals for the
one-dimensional case (two or more subregions for higher dimensional regions) while processing an interval. The controller has two distinct components. One is the bound estimator which is involved in the decision to terminate the algorithm. The other is a collection of procedures to detect special types of behavior and to initiate special computations in such cases. The block diagram of our parallel algorithm for integration over $n$-simplex fits the above metalgorithm. In Chapter IV we present a speedup analysis of a quadrature routine extending the method presented by Rice (1974) to handle multivariate integrand functions. The theoretical analysis assumes that the algorithms are region-size global adaptive routines, a variation of the global adaptive strategy introduced by Shapiro (1984).
An important component of a distributed algorithm for numerical integration which is able to handle a variety of different types of integrand functions is the load balancing routine. There are numerous load balancing strategies which are either application independent or tailored to enhance the performance of the solution for a specific type of problem. The different strategies are classified into different categories depending on the properties of their components (Eager, Lazowska, and Zahorjon, 1986a; Casavant and Kuhl, 1988). There are several issues that arise in load balancing that need to be addressed by a scheduling strategy. The first issue is how to decide when to move a task from one processor to another and which task to move. A second issue is how to decide which processor to send tasks to, or get tasks from. The task transfer policy describes how the first decision is made while the task location policy dictates how the second decision is made. A third issue concerns the type of information used to make the decisions. A policy based on global information would use information about the state of each processor to make decisions, whereas a local policy would only use information about the processor’s local state. Besides these two clear-cut policies, there are also load balancing schemes where the decision making on a node depends on the current information that the node has from its nearest neighbors. A final issue is whether a scheduling strategy has a centralized or distributed implementation. The centralized scheduling uses the master-slave approach where each processor, referred to as slave, sends the newly generated tasks to
a designated master processor which makes all the decisions about the redistribution of the load. At the other extreme, for the distributed scheduling, each processor maintains a load queue of tasks and uses either the *sender-initiated* or the *receiver-initiated* strategy to send or receive extra work-load. For the load balancing schemes, which use the *sender-initiated* strategy, the overloaded nodes search for underloaded nodes to which some of their excess load may be transferred to (Hac and Jin, 1990). On the other hand, for systems using the *receiver-initiated* strategy, the situation is reversed and underutilized nodes search for congested nodes from which load may be transferred to enhance the performance by preventing processor inactivity due to lack of task availability (Gulati, Barhen and Iyengar, 1988). Analytical models and simulations have shown that the *sender-initiated* strategies outperform *receiver-initiated* strategies at low to moderate system loads while *receiver-initiated* strategies are preferable at high system loads, assuming that process migration costs under the two strategies are comparable (Eager, Lazowska, and Zahorjon, 1986b).

Policies used by load balancing schemes can also be classified as static or adaptive. Static policies are also referred to as compile-time strategies while adaptive are also known in the literature as dynamic or run-time strategies. Static policies use only information about the average behavior of the system. Transfer decisions are independent of the actual current system state. Static policies could be either deterministic (Stone, 1978; Lo, 1984) or probablistic
(Tantawi and Towsley, 1985). The principal advantage of the static policies is their simplicity, there is no need to maintain and process system state information. These policies are effective only when the average system load can be characterized beforehand. However, they fail to react to transient behavior of the system load. Adaptive policies by contrast are more complex since they employ information on the current system state in making transfer decisions. This information makes significantly greater performance possible, than can be achieved under static policies. Two of the recently published adaptive load balancing schemes are: “Contracting Within a Neighborhood” (CWM) by Kale (1988) and “The Gradient Model” (GM) by Lin and Keller (1986). Both schemes use information from nearest neighbors in the task transfer and task location policies. An extensive list as well as a classification scheme for load balancing methods is given by Casavant et al. (1988).

For the case of multivariate integration on distributed memory machines there is no other published routine, besides the one by de Doncker et al (1992) for integration over a collection of triangles, using any type of load balancing scheme. Recently a routine was developed by Cariño (1992) which incorporates a receiver-initiated strategy for the load balancing routine. For this work we have implemented a load balancing method which uses both aspects of centralized and distributed strategies. We use non-local information for the task transfer and location policies. The strategy is dynamic and the decision making adapts with the current state of the system. The method
combines the \textit{sender-} and \textit{receiver-initiated} aspects of a load balancing scheme. A specific node is designated to store information indicating the overload of any node and the work requests of underloaded nodes. This node tries to minimize the time that the processors will be idle during the computations by controlling the flow of tasks from overloaded to underloaded nodes.
CHAPTER III

A PARALLEL GLOBAL ADAPTIVE ALGORITHM FOR INTEGRATION
OVER THE N-DIMENSIONAL SIMPLEX ON
SHARED MEMORY MACHINES

3.1 Introduction

We consider the problem of the numerical evaluation of integrals of the form

\[ I(f) = \int_{S_n} f(\bar{x}) d\bar{x} \]

where \( \bar{x} = (x_1, x_2, \ldots, x_n) \) and \( S_n \) is the \( n \)-dimensional simplex. Due to the significant increase in the number of integrand evaluations needed for higher dimensions, numerical multivariate integration may be expensive when the dimension is high and/or the function is not smooth. The above situation favors a parallel approach. As long as \( n \) is not too large \( (n \leq 10) \), an adaptive parallel approach is suitable, because of the fairly large granularity of the subdivision step in adaptive quadrature algorithms. This chapter presents such an approach. Background information is given in section 3.2, while section 3.3 presents a portable parallel global adaptive algorithm for integration over \( S_n \), implemented on a range of MIMD shared memory machines. We employ a parallel task pool scheme, based on the use of monitors, for the collection
of subregions which are organized as an error keyed binary heap. Section 3.4 contains a survey of alternative data structures for the maintenance of the task pool. These structures exhibit, in theory, a faster insertion and deletion time than that of a conventional binary heap. Performance results given in Section 3.5 demonstrate the speedup achieved by the implementation of the algorithm on three shared memory machines.

3.2 Background

Two main types of subregion adaptive strategies have been used for algorithm development: the local adaptive and the global adaptive. The first strategy has been very popular for one-dimensional calculations where good results are often obtained in a short time. However, it is, in general, unknown at the beginning of a calculation whether a result accurate to within a given tolerance $\epsilon$ can be found in a reasonable amount of time. In this case, local adaptive algorithms may spend a great portion of the allotted of time working in a small part of the integration region and trying to achieve an unreasonably small error. When the time runs out, all that is available globally is an inaccurate result since a big portion of the integration interval has been left unexplored. Although this can also be a problem with one-dimensional integrals, much smaller times are usually involved. On the other hand, global adaptive algorithms subdivide using information from all the current sub-regions. Global adaptive algorithms usually require more memory space to
maintain the current pool of subregions than do local adaptive algorithms, and possibly more time to select subregions for subdivisions. But at each stage of the calculation process, the global estimate for $I(f)$ is in some sense the best one available using the computation that has been done so far. For difficult problems, early termination of the algorithm usually gives a much more accurate result than that obtained with a locally adaptive algorithm. We have incorporated a global adaptive strategy in the algorithm presented in this chapter.

Since our algorithm runs on MIMD shared memory machines, for the synchronization of the running processes we have to consider their accessing the updating of the global data. These operations give rise to critical sections in the code. The execution of these critical sections is embedded in monitors (Hoare, 1974). Monitors are abstract data types consisting of three parts: (1) the data to be shared among concurrently executable processes, (2) the initialization code for the data, and (3) a set of operations that can be performed on the data. The monitor must guarantee that the initialization code is performed before contention for the shared data begins, and that only one of the operations may be performed at any time. Note that the last guarantee is a strong one and does not permit different processes to be operating on the same data simultaneously. In addition to the states of the control variables, the shared data in our application consists of the task pool, maintained as an error-keyed heap data structure, and the global variables specific to the
algorithm (global integral result, error estimate, and error flags). Monitors are primarily implemented as macros rather than as subroutines in order to avoid the overhead generated from each subroutine call. The portability of the current algorithm is achieved through the use of the Argonne monitor macros introduced by Lusk and Overbeek (1983, 1984). This macro package contains the necessary synchronization primitives and supports a range of MIMD shared memory machines including the Alliant FX/8, the Sequent Symmetry and the Encore Multimax. The macros are invoked from the user FORTRAN program. Thus, before compilation the code is passed through the m4 macro expansion UNIX tool, which replaces the macros with the corresponding FORTRAN text, based on the macro definitions. For some macros this just means an insertion of some statements whereas other macros expand to function calls. Specific syntax rules must be followed in the macro definitions.

It has been recognized in the literature (Genz, 1989) that an adaptive integration algorithm can be parallelized at three different levels: (1) The integral level, (2) The subregion level, and (3) The basic rule level. Parallelization at the integral level may be appropriate in large scale calculations where a number of integrals need to be computed, especially if the integrand functions are sufficiently different so that the subregions resulting from the subdivision of the original region are significantly different for each integral. If similar integrals share a common integration region, then the same subdivision might be good for the accurate computation of all the integrals. Thus,
the time finding this subdivision can be shared among the integrals and the parallelization should therefore be introduced at a lower level. Parallelism at the subregion level is best suited for MIMD shared memory machines. If the function to be integrated is a vector function, then different processors can be given different components of the vector function to integrate over a common subregion. For the case of one integrand function, different subregions should be assigned to different processors so that the integration and the error estimation of those regions will be computed in parallel. Parallelism can also be introduced at the basic quadrature rule level of an algorithm. The approximation to the value of the integral $I(f)$ is given by the basic quadrature rule $Q(f) = (w_1, w_2, \ldots, w_n) \cdot (f(x_1), f(x_2), \ldots, f(x_n))$ where $w_i$ is the $i^{th}$ weight of the rule associated with the function value $f(x_i)$. The computation of $Q(f)$ can be done efficiently by exploiting the vector processing capabilities of the machine (i.e., Alliant FX/8).

3.3 Parallel Global Adaptive Algorithm

To obtain a parallel algorithm portable across various MIMD shared memory machines, a set of macros was layered over the Argonne macro package which provides a portable and efficient set of high level synchronization primitives. The underlying sequential algorithm is SCUMP which was recently developed by Genz (1991). The portability of our integration routine offers an advantage over existing adaptive algorithms for integration over the $n$-simplex.
(n \geq 3). To our knowledge the only parallel adaptive routine reported in the literature for integration over the n-simplex \( (n = 3) \) is DCUTET, which is parallelized at the subregion level with use of compiler directives. Evidently it runs, after some modifications, on MIMD machines where the FORTRAN compiler supports parallelization through the use of compiler directives. This, however, limits the portability of the code. For example, the Encore Multimax uses a parallel FORTRAN language and parallelization of the sequential code through compiler directives is not supported. Furthermore there is no similar approach in the C language. However, the Argonne monitor macros are available in C (Boyle, Butler, Disz, Glickfeld, Lusk, Overbeek, Patterson, and Stevens, 1987).

The algorithm is executed in parallel by \( P \) processes, one of which, designated as the master process, executes Control. In the current implementation, NPROCS is the variable used to store the value of \( P \). Work is executed by the master and slave processes in parallel. The \( P - 1 \) slave processes are created from the master process via the Argonne macro Create using a do-loop structure as depicted in Figure 3. This macro assigns a newly created process to a different processor if one is available. Upon creation each slave process executes subroutine SLAVE which initiates the execution of the Work subroutine, as illustrated in Figure 4. Subroutine Work accepts only one argument with values 0 or 1 in order to distinguish if it is called from the master or a slave process. Note that after the creation of all slave processes,
as given in Figure 3, the master process also calls the Work routine (Lusk and Overbeek, 1984). Using this approach, the master and slave processes are characterized to have identical roles in the parallel portions of the algorithm and therefore the same program can be run with one process only (master process). This allows a fair amount of development on a workstation since the set of Argonne macros for a uniprocessor can be used in order to generate the sequential code of the integrator. The algorithm attempts to obtain an approximation \( Q(f) \) to \( I(f) \), hopefully satisfying the inequality \( |I(f) - Q(f)| \leq \max(\varepsilon_a, \varepsilon_r |I(f)|) \) where \( \varepsilon_a \) and \( \varepsilon_r \) are the specified absolute and relative tolerances respectively. The input to the algorithm consists of the integrand function \( f(x) \), the requested accuracies, the dimension \( \text{NDIM}=n \) of the simplex integration region, and the value of a parameter KEY providing a choice of different polynomial degree Grundmann and Möller quadrature rules (Grundmann and Möller 1978) used for the basic integral and error
estimate over each subregion. The algorithm returns the integral and absolute
error estimates, the number of function evaluations used by the routine and
a flag indicating any abnormalities incurred during the calculations (such as
exceeding the maximum allowed number of function evaluations).

Following the scheme given in de Doncker and Kapenga (1987, 1988),
the algorithm can be represented by its Control and Work parts as follows.

**Control:**
- Initialize and create the slave processes;
- While total error > requested accuracy do
  - **Work**;
- Enddo

**Work:**
- While acceptance criteria is not satisfied do
  - Select simplex for subdivision;
  - Get space for the subsimplices;
  - Subdivide, integrate and calculate local error estimates;
  - Update global variables and flag termination conditions;
  - Add subsimplices to the pool;
- Enddo

The local error estimate is calculated as in Genz (1991). At each iter­
ation in the **Work** routine, the subregion with the largest estimated error is
extracted from the pool and subdivided into two subsimplicies of equal vol­
ume. The direction of subdivision is based on values of the fourth divided
difference of the integrand (Genz, 1991). Measures other than the fourth di­
vided differences could also be used to measure the irregularity of the integrand
(Berntsen, 1984). It may be noted, that a subdivision scheme which produces
a large number of subregions at each step may cause the algorithm to loose its
adaptive nature and result in unnecessary subdivisions. On the other hand,
subdividing each region into more than two parts at a time may be beneficial if the further subdivisions are likely to be needed at a later stage anyway. In a parallel setting, the latter may have the advantage of generating more useful work within an integration, but it may also generate useless work. In this setting, at each loop cycle of the algorithm each processor will perform a number of insert operations and one delete. An improvement could be achieved when the last insert operation is merged with the delete into a slightly modified delete operation as follows. If the error estimate of the region stored in the root of the heap is smaller than the error estimate of the region to be inserted, then the merged operation finishes right away and the processor treats the element to be inserted as the one to be deleted. Otherwise it replaces the root with the item to be inserted and proceeds with the deletion routine.

The task pool is currently implemented as a heap global data structure. Since the task pool specific operations are hidden in the additional layer of macros, the data structure is invisible to the integration problem. In Appendix A, we provide the m4 definitions of the new macros used in the implementation of the algorithm, for supporting the heap data structure. The macro calls in the integration program are expanded to FORTRAN code by the (UNIX) m4 macro processor.

The macro HEAPDEC declares variables and arrays used in subsequent macros and HEAPBOUNDS sets the maximum and minimum number of sub-regions to be stored in the heap. Macro GETSPACE checks if the maximum allowed number of elements of the global data structure is reached and sets
a flag accordingly. Macro ADDGETWORK is passed as an argument to the Argonne ASKFOR macro via the call:

\begin{verbatim}
ASKFOR(MO,RC,NPROCS,ADDGETWORK(MO,SBRGNS),).
\end{verbatim}

The ASKFOR macro supports the creation of tasks from a shared pool of work. The parameter MO is the name of the user's monitor; NPROCS specifies the number of processes used; RC contains a return code with value zero in the case of successful acquisition of the root of the heap; and SBRGNS indicates the number of nodes currently in the heap. After the selection of a simplex from the pool and the integration over its two equally sized subsimplices, the code in macro ADDTOTEMP saves the integral and error contributions over the subsimplices, their vertices and volumes locally, until they are inserted into the heap. Their insertion into the data structure is delayed until the ASKFOR is called; it is achieved in the ADDGETWORK macro by the call to ADDREHEAP. Subroutine DELREHEAP, also called inside the ADDGETWORK macro, extracts the root of the heap (simplex with the largest error estimate) and reheap the generated tree. Note that, since the accessing of the task pool is done through the ASKFOR monitor, only one monitor is used for maintaining the global heap.

### 3.4 Alternative Data Structures

The use of the heap data structure for efficient maintenance of the set of subregions as suggested by Malcolm and Simpson (1985) and Kahaner (1980) allows a subregion priority queue of size $N$ to be updated in $O(\log N)$
time. Experience has shown that the most expensive part of a (sequential) quadrature routine consists of the function evaluations. However, with respect to solely the processing of function evaluations, a "good" parallel algorithm would be able to obtain perfect speedup, if the various losses studied in Kapenga and de Doncker (1988a) are small and all participating processes are kept busy with useful work at each step. Since accessing and updating of the shared pool contributes to the critical time, the role of an efficient data structure for managing the task pool is important. Rice (1973) has stressed the importance of data structures in the construction of efficient integration algorithms by stating:

The interplay between numerical analysis and data structures will become increasingly important as more adaptive algorithms are found to be superior for intensive computations. (Rice, 1973, p. 25.)

In the remainder of this section we will discuss some alternative data structures.

1. The use of the Indirect Heap can speed the computation, due to the fact that no rearranging of the keys (real or double precision items) takes place. The subroutines that maintain the pool of subregions can work with an array heap of indices into the array A, which stores the heap, so that $A(\text{heap}(k))$ is the key of the $k^{th}$ element of the heap. Thus, the Insert and Delete operations will be done with integer data movements but still in $O(\log N)$ time.

2. The conventional heap structure is a binary tree. Using a heap with a branch factor $b > 2$ offers a tradeoff between the efficiency of insertion
and that of extraction. In particular, the larger the value of $b$, the shorter the insertion time and the larger the extraction time. Since, in the current application, the number of insertions is greater than that of deletions, a heap with a larger branch factor may be better suited.

3. The Min-Max (or Max-Min) heap introduced by Atkinson, Sack, Sontoro and Storthotte (1986), demonstrates some improvement over the traditional heap. A Min-Max heap over a set of values is defined as a binary tree $T$ with the following properties: (a) $T$ has the heap-shape, that is all leaves lie on at most two adjacent levels, and the leaves on the last level occupy the leftmost positions while all other levels are complete; (b) $T$ is Min-Max ordered: values stored on even levels are smaller than or equal to the values stored at their descendants (if any) and the root is at level zero. Therefore the smallest value is stored at the root of $T$, whereas the largest value is stored at one of the root’s children. Table 1 shows a comparison of the worst case time complexities for the maintenance of the two structures. The Min-Max heap offers an advantage over the max-heap especially if the number of insertions is large. It may be noted that the implementation of the above operations for the new structure are very similar to that of a heap.

4. An extension of the Max-Min heap is the Generalized-Max-Min heap (Atkinson et al., 1986). Assuming that $K = \{k_1, k_2, \cdots, k_n\}$ where $k_1$ is the rank of the element in the new data structure, the operation $\text{Find}(K)$ can be executed in constant time, $\text{Delete}(K)$ in logarithmic time. For our specific application, if $k_i = i$, $1 <= i <= m$ then the largest $m$ elements in the
structure can be located in $O(1)$ time, and their extraction will be more efficient, requiring one "reheaping" operation instead of $m$ consecutive ones. As a consequence, such usage of a Generalized-Max-Min heap, together with the extraction of $m$ items at once, may lead to a reduction in the overall execution time, since more work (consisting of $m$ subregions) is obtained by the processes within one access of the data structure. The value of $m$ should not be too large, to avoid situations where subregions would be extracted unnecessarily, resulting in useless work.

Table 1
Worst Case Time Complexities of Create, Insert, and Delete-Max Operations on a Structure of $N$ Items

<table>
<thead>
<tr>
<th>Operations</th>
<th>Heap</th>
<th>Min-Max Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create:</td>
<td>$2N$</td>
<td>$\frac{7}{3}N$</td>
</tr>
<tr>
<td>Insert:</td>
<td>$\log_2(N+1)$</td>
<td>$\frac{1}{2}\log_2(N+1)$</td>
</tr>
<tr>
<td>Del.-Max.:</td>
<td>$2\log_2 N$</td>
<td>$\frac{3}{2}\log_2 N$</td>
</tr>
</tbody>
</table>

5. A similar structure introduced by Carlsson (1988) is the Deap. In this data structure, the left subtree of the root is a min-heap and the right subtree is a max-heap. Furthermore, a leaf node in the min-heap is smaller than the corresponding leaf node in the max-heap. Specifically any node $k$ in the min-heap is smaller than the element at node $k+a$, if it exists, else $(k+a)\text{div}2$, where $a = 2^{\lceil \log_2 k \rceil} - 1$. Otherwise, the Deap exhibits the same
properties as the common heap data structure. This structure can be maintained in a linear array and its time complexities reveal a slight improvement over those of the Max-Min heap listed in Table 1.

6. It is known that the time to insert an element plus that to extract the smallest from a priority queue must be at least $O(\log N)$, otherwise we would be able to sort in less than $O(N \log N)$ time. In the context of the integration algorithm we insert twice as many elements as we extract. Therefore, it is better to keep the cost of an insertion down as far as possible. This suggests keeping the time for an insertion constant and the time for extracting an element logarithmic. These time bounds are met with the Implicit Binomial Queue data structure developed by Carlsson, Munro and Pablete (1988).

Kapenga and de Doncker (1988b) introduced the Distributed Array Queue data structure (DAQ) for concurrent management of priority queues in adaptive algorithms. This structure has a constant insertion and removal time. It consists of an array of buckets, each of which contains tasks (subregions) of a particular priority class. For example, a priority class may correspond to a range of estimated error. The number of buckets is fixed and the function which maps a task of a given priority to a bucket is predefined. Since there is no priority queue structure within the buckets, insertion can be achieved in $O(1)$ time. Deletion is from the bucket of the highest priority class. Consequently, the DAQ assures that a region of the highest priority class is selected, not necessarily the one with the largest error estimate.
Experience has shown that the latter is not really needed, since typically the subregions of the highest priority class will be selected eventually. Deo and Prasad (1990) introduced the parallel heap data structure. If \( N \) is the number of items in the parallel heap, the data structure efficiently uses \( P \) processors for \( 1 < P \leq N \). It allows \( \Theta(P) \) inserts and/or deletes to be done in \( O(\log N) \) time and thus exhibits up to \( O(P) \) speedup for priority queue based algorithms. These performance results are only theoretical, since this data structure has not yet been implemented.

3.5 Some Test Results

The test results, which are presented in this section, demonstrate the execution time and the speedup of the implementation of the algorithm on the Alliant FX/8, the Sequent Symmetry and the Encore Multimax at the Argonne National Laboratory. Table 2 lists the integrand functions for which test results were obtained. The region of integration is the unit \( n \)-simplex.

Test integrand function \#1 is \( F(x, y, z) = e^{x+y+z} \cos(x+y+z) \) and the region of integration is the unit tetrahedron (3-simplex). This function was selected from the sample program of the NAG routine D01PAF in the current NAG FORTRAN library manual. We set the absolute and relative error tolerances \( \varepsilon_a = 0 \) and \( \varepsilon_r = 10^{-7} \) and we specify a degree 9 for the basic quadrature rule. The number of function evaluations then reaches the maximum, set to 50,000 for all the runs of the test integral \#1. Figures

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
5 - 7 demonstrate the execution times, in seconds, and the speedup of the implementation of the algorithm for that specific integral.

Table 2
Test Integrand Functions

<table>
<thead>
<tr>
<th>Integral #</th>
<th>n</th>
<th>Integrand</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>$e^{x+y+z} \cos(x + y + z)$</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>$\frac{xyz^2}{(x+y+z)^2}$</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>$xyz^2 \sqrt{\frac{1-x-y-z-w}{1+z+y+x+w}}$</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>$xyz^2 \sqrt[3]{w^{1/2} u^{1/3} v^2 e^{-2(z+y+z+w+u+v)}}$</td>
</tr>
</tbody>
</table>

It must be noted that the Alliant FX/8 provides 8 computational processors and for the case of the Sequent Symmetry, by default a maximum of 10 processors is assigned provided a parallel programming library. For the Encore Multimax the execution time of the algorithm increases due to contention, as the number of computational processors increases beyond the value of 10. This was observed for all test integrals reported in this section.

The rest of the test integrals were selected from general formulas where the integral is given in closed form and so the exact value could easily be calculated (Grandshteyn and Rzyhik, 1980). The knowledge of the exact value stands as a verification for the validity of the results obtained by the numerical integrator. For integration over the unit $n$-simplex the following results holds,
Figure 5. Execution Time vs. Number of Processes and Speedup vs. the Number of Processes For Integrand #1 On the Alliant FX/8.

Figure 6. Execution Time vs. the Number of Processes and Speedup vs. the Number of Processes For Integral #1 On the Sequent Symmetry.
Figure 7. Execution Time vs. the Number of Processes and Speed-up vs. the Number of Processes For Integral #1, On the Encore Multimax.

\[
\int \cdots \int _{\substack{x_1 \geq 0 \quad x_2 \geq 0 \quad \ldots \quad x_n \geq 0 \\ x_1 + x_2 + \cdots + x_n \leq 1}} \frac{x_1^{p_1-1} \cdots x_n^{p_n-1}}{(x_1 + \cdots + x_n)^\mu} dx_1 \cdots dx_n
\]

\[
= \frac{1}{(p_1 + \cdots + p_n - \mu)} \frac{\Gamma(p_1) \cdots \Gamma(p_n)}{\Gamma(p_1 + \cdots + p_n)}
\]

The above integral converges only if \((p_1 - 1) + (p_2 - 1) + \cdots + (p_n - 1) - \mu > -n\), that is, if \(\mu < p_1 + p_2 + \cdots + p_n\). We selected \(\mu = 2\), \(p_1 = 2\), \(p_2 = 2\) and \(p_3 = 3\) and thus the integrand function \#2 is: \(F(x, y, z) = \frac{xyz^2}{(x+y+z)^3}\), integrated over the unit tetrahedron. The input parameters of the integration routine were set as in the previous result, while an accuracy of \(10^{-5}\) was observed as the maximum number of function evaluations was reached. Tables 3 - 6 show the relations between the execution time (in hundredths of a second) versus the number of processes as well as the speedup versus the number of processes when the routine is executed on different SM machines.
Table 3
 Execution Time and Speedup as a Function of NP for Integrand Function #2
 On the Sequent Symmetry and Encore Multimax

<table>
<thead>
<tr>
<th>Machine</th>
<th>NP</th>
<th>Time</th>
<th>Speedup</th>
<th>Machine</th>
<th>NP</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>3442</td>
<td>1</td>
<td></td>
<td>1</td>
<td>4987</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2325</td>
<td>1.48</td>
<td></td>
<td>2</td>
<td>2958</td>
<td>1.58</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1205</td>
<td>2.85</td>
<td></td>
<td>3</td>
<td>1880</td>
<td>2.65</td>
</tr>
<tr>
<td>Sequent</td>
<td>4</td>
<td>900</td>
<td>3.82</td>
<td>Encore</td>
<td>4</td>
<td>1358</td>
<td>3.67</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>760</td>
<td>4.53</td>
<td></td>
<td>5</td>
<td>1180</td>
<td>4.22</td>
</tr>
<tr>
<td>Symmetry</td>
<td>6</td>
<td>635</td>
<td>5.42</td>
<td>Multimax</td>
<td>6</td>
<td>898</td>
<td>5.55</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>538</td>
<td>6.39</td>
<td></td>
<td>7</td>
<td>765</td>
<td>6.51</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>470</td>
<td>7.35</td>
<td></td>
<td>8</td>
<td>695</td>
<td>7.18</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>445</td>
<td>7.73</td>
<td></td>
<td>9</td>
<td>620</td>
<td>8.04</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>425</td>
<td>8.10</td>
<td></td>
<td>10</td>
<td>595</td>
<td>8.38</td>
</tr>
</tbody>
</table>

Test integral #3 was obtained from the following formula:

\[
\int \cdots \int \frac{x_1^{p_1-1} \cdots x_n^{p_n-1}}{x_1 + x_2 + \cdots + x_n} c dt \sqrt{\frac{1-x_1-x_2-\cdots-x_n}{1+x_1+x_2+\cdots+x_n}} dx_1 \cdots dx_n
\]

\[
= \frac{\sqrt{\pi}}{2} \frac{\Gamma(p_1) \cdots \Gamma(p_n)}{\Gamma(p_1 + \cdots + p_n)} \left[ \frac{\Gamma\left(\frac{m}{2}\right)}{\Gamma\left(\frac{m+1}{2}\right)} - \frac{\Gamma\left(\frac{m+1}{2}\right)}{\Gamma\left(\frac{m+2}{2}\right)} \right]
\]

where \( m = p_1 + p_2 + \cdots + p_n \). Selecting \( p_1 = 2, p_2 = 2, p_3 = 3, p_4 = 4 \), the integrand function becomes \( F(x, y, z, w) = x y z^2 w^3 \cdot \sqrt{\frac{1-x-y-z-w}{1+x+y+z+w}} \).

For this case the relative error tolerance was set to \( \varepsilon_r = 10^{-6} \) while the
absolute tolerance to $\varepsilon_a = 0$. A degree 9 basic quadrature rule is used while the number of function evaluations reaches the maximum set at 100,000. An accuracy of $10^{-4}$ was observed as the maximum number of function evaluations was reached.

Table 4

<table>
<thead>
<tr>
<th>Machine</th>
<th>NP</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alliant FX/8</td>
<td>1</td>
<td>3210</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1866</td>
<td>1.72</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1189</td>
<td>2.70</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>872</td>
<td>3.68</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>713</td>
<td>4.50</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>606</td>
<td>5.30</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>526</td>
<td>6.10</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>464</td>
<td>6.92</td>
</tr>
</tbody>
</table>

Test integral #4 was constructed using:

$$\int \cdots \int_{x_1 \geq 0, x_2 \geq 0, \ldots, x_n \geq 0}^{x_1 + x_2 + \cdots + x_n \leq 1} x_1^{p_1-1} x_2^{p_2-1} \cdots x_n^{p_n-1} e^{-g(x_1 + x_2 + \cdots + x_n)} dx_1 \cdots dx_n$$

$$= \frac{\Gamma(p_1) \cdots \Gamma(p_n)}{\Gamma(p_1 + \cdots + p_n)} \int_0^1 x^{p_1 + \cdots + p_n} e^{-gx} dx$$

where $p_1 > 0, \cdots, p_n > 0$. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
We selected $p_1 = 2$, $p_2 = 3$, $p_3 = \frac{5}{2}$, $p_4 = \frac{3}{2}$, $p_5 = \frac{9}{4}$, $p_6 = 4$ and $q = 2$. An estimate for the value of the one dimensional integral appearing in the right-hand side of the above formula, was obtained using a routine from the QUADPACK software package as in R. Piessens (1983). Specifically, the function $F(x, y, z, w, u, v) = xy^2 z^\frac{3}{2} w^\frac{1}{2} u^\frac{5}{4} v^3 e^{-2(x+y+z+w+u+v)}$ was integrated over the unit 6-simplex.

Table 5

Execution Time and Speedup as a Function of NP for Integrand #3
On the Sequent Symmetry and Encore Multimax

<table>
<thead>
<tr>
<th>Machine</th>
<th>NP</th>
<th>Time</th>
<th>Speedup</th>
<th>Machine</th>
<th>NP</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10934</td>
<td>1</td>
<td></td>
<td>1</td>
<td>12846</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6350</td>
<td>1.72</td>
<td></td>
<td>2</td>
<td>7340</td>
<td>1.75</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3757</td>
<td>2.91</td>
<td></td>
<td>3</td>
<td>4604</td>
<td>2.79</td>
<td></td>
</tr>
<tr>
<td>Sequent</td>
<td>4</td>
<td>2985</td>
<td>3.66</td>
<td>Encore</td>
<td>4</td>
<td>3519</td>
<td>3.65</td>
</tr>
<tr>
<td>5</td>
<td>2362</td>
<td>4.63</td>
<td></td>
<td>5</td>
<td>2854</td>
<td>4.50</td>
<td></td>
</tr>
<tr>
<td>Symmetry</td>
<td>6</td>
<td>1989</td>
<td>5.49</td>
<td>Multimax</td>
<td>6</td>
<td>2414</td>
<td>5.32</td>
</tr>
<tr>
<td>7</td>
<td>1750</td>
<td>6.25</td>
<td></td>
<td>7</td>
<td>2055</td>
<td>6.25</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1582</td>
<td>6.91</td>
<td></td>
<td>8</td>
<td>1917</td>
<td>6.70</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1425</td>
<td>7.67</td>
<td></td>
<td>9</td>
<td>1779</td>
<td>7.22</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1325</td>
<td>8.25</td>
<td></td>
<td>10</td>
<td>1636</td>
<td>7.85</td>
<td></td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
### Table 6
Execution Time and Speedup as a Function of NP for Integrand #3
On the Alliant FX/8

<table>
<thead>
<tr>
<th>Machine</th>
<th>NP</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alliant FX/8</td>
<td>1</td>
<td>10120</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>5953</td>
<td>1.70</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3601</td>
<td>2.81</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2811</td>
<td>3.60</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>2195</td>
<td>4.61</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>1931</td>
<td>5.24</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>1661</td>
<td>6.09</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1510</td>
<td>6.70</td>
</tr>
</tbody>
</table>

The relative error tolerance was set to $\varepsilon_r = 10^{-5}$ and the number of function evaluations reached the maximum set to 200,000. Tables 7 - 8 demonstrate the results obtained on the different MIMD-SM machines. The times are reported in hundredths of a second.
Table 7

Execution Time and Speedup as a Function of NP for Integrand #4
On the Sequent Symmetry and Encore Multimax

<table>
<thead>
<tr>
<th>Machine</th>
<th>NP</th>
<th>Time</th>
<th>Speedup</th>
<th>Machine</th>
<th>NP</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21345</td>
<td>1</td>
<td></td>
<td>1</td>
<td>24365</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>12125</td>
<td>1.76</td>
<td></td>
<td>2</td>
<td>14248</td>
<td>1.71</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>7925</td>
<td>2.69</td>
<td></td>
<td>3</td>
<td>8670</td>
<td>2.81</td>
<td></td>
</tr>
<tr>
<td>Sequent</td>
<td>4</td>
<td>6325</td>
<td>3.37</td>
<td>Encore</td>
<td>4</td>
<td>7166</td>
<td>3.40</td>
</tr>
<tr>
<td>5</td>
<td>4915</td>
<td>4.34</td>
<td></td>
<td>5</td>
<td>5801</td>
<td>4.20</td>
<td></td>
</tr>
<tr>
<td>Symmetry</td>
<td>6</td>
<td>3958</td>
<td>5.39</td>
<td>Multimax</td>
<td>6</td>
<td>4853</td>
<td>5.02</td>
</tr>
<tr>
<td>7</td>
<td>3314</td>
<td>6.44</td>
<td></td>
<td>7</td>
<td>3981</td>
<td>6.12</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2965</td>
<td>7.19</td>
<td></td>
<td>8</td>
<td>3556</td>
<td>6.85</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2712</td>
<td>7.87</td>
<td></td>
<td>9</td>
<td>3431</td>
<td>7.10</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2510</td>
<td>8.50</td>
<td></td>
<td>10</td>
<td>3360</td>
<td>7.25</td>
<td></td>
</tr>
</tbody>
</table>
Table 8

Execution Time and Speedup as a Function of NP for Integrand #4
On the Alliant FX/8

<table>
<thead>
<tr>
<th>Machine</th>
<th>NP</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>20780</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12906</td>
<td>1.61</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7668</td>
<td>2.71</td>
</tr>
<tr>
<td>Alliant FX/8</td>
<td>4</td>
<td>5772</td>
<td>3.60</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>4832</td>
<td>4.30</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>4066</td>
<td>5.11</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>3390</td>
<td>6.13</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>3298</td>
<td>6.30</td>
</tr>
</tbody>
</table>
CHAPTER IV

SPEEDUP ANALYSIS OF ADAPTIVE INTEGRATION ALGORITHMS ON MIMD-SM MACHINES

4.1 Introduction

This chapter presents a speedup analysis of certain types of global adaptive integration algorithms on MIMD-SM machines. The discussion is based on some results presented by Rice (1974, 1975) but the method is extended to handle multivariate integration of functions which may also exhibit some types of singular behavior. The analysis primarily applies to a specific type of global adaptive algorithm in contrast with the analysis by Rice which is intended for a local adaptive procedure.

Section 4.2 gives an overview of some types of adaptive integration algorithms. Representations of the error functional of a quadrature rule are presented in Section 4.3. We give those representations for quadrature rules for the unit square by means of Peano kernels and the unit $N$-cube by means of Sard kernels, assuming that the function has sufficient number of continuous derivatives. We also provide a list of theorems which show the representation of the error functional for a quadrature rule over a standard region ($N$-cube, $N$-simplex), when the integrand function exhibits certain types of singular behavior. Section 4.4 introduces the appropriate quantities needed for the
speedup analysis and through a series of Lemmas, presents the derivation of
the speedup function for a region-size global adaptive algorithm on MIMD-SM
machines. We give the theoretical speedup results for functions with sufficient
number of continuous derivatives over the integration region (N-cube, N-
simplex) as well as for those with some singular behavior.

4.2 Classification of Some Adaptive Quadrature Strategies

The local and the global adaptive strategies are two important types of
subdivision strategies that have been used in adaptive integration algorithms.
The following metalgorithm characterizes a general adaptive algorithm and
includes local and global adaptive schemes:

Initialize;
While (acceptance criterion is not satisfied) do
   Subdivide;
Enddo;

For a local adaptive algorithm:

1. Newly obtained regions are subject to local acceptance criterion and
   are classified as active or discarded.

2. For normal termination the set of active regions must be empty.

   This strategy fits the two-box algorithm considered by Rice (1974);
   consequently the while clause in the above metalgorithm could be replaced
   by: "While (active set is not empty) do".

For a global adaptive algorithm:
1. All regions remain active throughout the computations.

2. All regions are prioritized with respect to how desirable their further subdivision is.

3. The highest ranked region is selected of each subdivision step.

4. The criterion for normal termination depends on the estimated error over the entire set of regions.

For this strategy, the while clause in the above metalgorithm is translated into: “While (the global error estimate is larger than the required accuracy) do”.

Both of the above algorithms are characterized as adaptive since the decision to select the next region for subdivision is based on the information gathered about the behavior of the integrand function. This characterization does not exclude the possibility for a quadrature algorithm to be both global and local adaptive. Such algorithm would exhibit a global strategy in selecting the next subregion for subdivision and for checking the criterion for normal termination, while a local criterion is used to discard subregions with “small” local error. A routine with a global-local framework was given by Kahaner and Wyman (1983), for the evaluation of one dimensional integrals on microcomputers. In this routine, intervals are discarded if their local error estimate does not exceed a threshold determined by the relative machine accuracy.

By varying the method by which a region is selected for subdivision, global adaptive algorithms can be classified into a number of different categories as follows (Shapiro, 1984):
1. Absolute global adaptive strategy, where the priority of a region $R$ is given by $|\text{error estimate of } \int_R f|$. The logic behind the selection of the region with the largest absolute error is based upon the expectation that a large decrease in error will be achieved after the subdivision of such a region. This selection rule places a considerable emphasis on minimizing the number of function evaluations and is heavily dependent on the accuracy of the error estimating procedure. This strategy is also known as global adaptive.

2. Relative global adaptive, where the priority of a region $R$ is $\frac{|\text{error estimate of } \int_R f|}{\int_R |f|}$. This attempts to place an emphasis on the reliability with which we view the error estimate of a region.

3. Region-size global adaptive strategy. The priority of a region $R$, is expressed as $\frac{|\text{error estimate of } \int_R f|}{s(R)}$, where $s(R)$ is the “size” of the region. For the one dimensional case, “size” is length, for two dimension “size” represents area, while in higher dimensions it denotes the volume of the region. The goal of this scheme is to emphasize the region’s relative importance with respect to the entire integration region. We used this in a modified version of our algorithm, given in Chapter III, for integration over the $N$-simplex on MIMD- SM machines. The speedup analysis given in Section 4.4 is also based on this strategy.

4. Normalized global adaptive strategy, where the priority of a region is the relative error in the rule applied to $\int_R \hat{f}$. The function $\hat{f}$ is a normalized version of $f$ on $R$ such that $\min_{x \in R} \hat{f} = 0$ and $\max_{x \in R} \hat{f} = 1$.

5. Queued hybrid adaptive strategy. The priority of a region is its time
of generation. Even though the selection criterion for the next region to be subdivided is quite different from that in the previous approaches, this method has the advantage of a constant time data structure, which the other priority rules do not have.

Limited experimental analysis for comparing the above strategies is reported in Shapiro (1984). His test results were performed, using one dimensional integration algorithms. In regards to the accuracy obtained versus the number of function evaluations (which determines the overall execution time of the algorithm), Shapiro concluded that all strategies performed in a similar fashion, but their exact behavior depends on the integrand behavior; i.e., he did not determine an overall winner or loser.

4.3 Error Estimate of a Quadrature Rule

In this section we discuss the representation of the error functional $E_n f$ in an integration formula

$$I_{\Omega} f = \int_{\Omega} \cdots \int f(x_1, x_2, \ldots, x_N) dx_1 dx_2 \cdots dx_N$$

$$= \sum_{i=1}^{n} A_i f(v_{i,1}, v_{i,2}, \ldots, v_{i,N}) + E_n f$$

Expressions for the error estimate will be given for functions with a sufficiently large number of continuous derivatives and also for functions with particular types of singular behavior over a standard integration region ($N$-cube, $N$-simplex). With few exceptions, we adhere to the notation of Engels, (1980).
Let $\Omega$ represent an $N$-dimensional non-empty finite open set, simply connected, smooth or with piecewise smooth boundary. Let $\partial\Omega$ be the boundary of $\Omega$ and $\overline{\Omega} = \Omega \cup \partial\Omega$. Note that the definition of $\Omega$ is general enough to cover most of the integration regions that appear in practical applications. In equation (1) we denote $Q_nf = \sum_{i=1}^{n} A_i f(v_{i,1}, v_{i,2}, \ldots, v_{i,N})$ where the $A_i$ represent the weights and $(v_{i,1}, v_{i,2}, \ldots, v_{i,N})$, the nodes of the rule. Thus $E_nf = I_{\Omega} f - Q_nf$. Note that $Q_nf$ would be termed cubature formula by some authors (denoted by $C_n f$ by Engels), when it applies to multivariate integration.

Definition 1. Let $\{\phi_1, \phi_2, \ldots, \phi_\nu\}$ be a set of linearly independent functions, defined and integrable in $\Omega$. Then the requirement that

$$E_nf = 0 \quad \text{and} \quad f \in \{\phi_1, \phi_2, \ldots, \phi_\nu\}$$

is used to define the nodes or weights or both by solving a linear or a nonlinear system of equations. This procedure is called the method of undetermined coefficients.

Let $P_k$ denote the space of polynomials of degree $k$. The degree of a quadrature rule is defined as follows.

Definition 2. If the quadrature error $E_nf = 0$ for $f \in P_d$ and furthermore there exists a polynomial $g \in P_{d+1}$ such that $E_ng \neq 0$, then $Q_nf$ is said to have degree $d$. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
A desirable characteristic of a quadrature rule \( Q_n f \) is its symmetry which is defined as

**Definition 3.** A quadrature rule \( Q_n f \) for the unit hypercube \( (\Omega = [0, 1]^N) \) is symmetric if it is invariant under reflection about all the hyperplanes \( x_i = \frac{1}{2}, \ i = 1, 2, \ldots, N; \) i.e.,

\[
Q_n f_i = Q_n g, \quad f_i(x_1, \cdots, x_i, \cdots, x_N) = g(x_1, \cdots, 1 - x_i, \cdots, x_N)
\]

\( i = 1, 2, \ldots, N. \)

We now examine how the error of a quadrature rule is reduced if the region \( \Omega \) is subdivided into similar subdomains, such that the same rule \( Q_n f \), is applicable to every subregion. In the one dimensional case, \( \Omega = [0, 1] \), the error functional can be represented using Peano kernels (Engels, 1980).

**Theorem 1.** Let \( f \in C^{k+1}[0, 1] \) and let \( Q_n f \) be a quadrature rule of degree \( d \geq k \geq 0 \). Then, the error functional \( E_n f = I_\Omega f - Q_n f \) has the representation

\[
E_n f = \frac{1}{k!} \int_0^1 K(t) f^{(k+1)}(t) dt
\]

where \( K(t) = K_{n,k}(t) = E_n^x[(x-t)^k] \) is the Peano kernel of the quadrature rule \( Q_n f \) of degree \( d \geq k \). We denote

\[
(x - t)_+^k = \begin{cases} 
(x - t)^k & x \geq t \\
0 & x < t
\end{cases}
\]
Furthermore, the upper index \( x \) in \( E_n^x[(x - t)_+] \) indicates that the error functional is to be applied with respect to \( x \). Note that in order to allow for the case \( 0 \leq d < k \), (2) can be rewritten with \( k \) replaced by \( \delta = \min\{d, k\} \).

Choosing \( m+1 \) points \( t_j : 0 = t_0 < t_1 < \cdots < t_{m-1} < t_m = 1 \), let the interval \([0,1]\) be subdivided into \( m \) subintervals \([t_{j-1}, t_j], j = 1(1)m\) of length \( \Delta t_j \). Applying a scaled version of \( Q_n f \) to each subinterval and summing thus obtaining the composite rule \( Q_n^{(m)} f \), the following theorem gives a bound for \( E_n^{(m)} f \), assuming equal space \( \Delta t = \frac{1}{m} \) (Engels, 1980).

**Theorem 2.** If \( f \in C^{k+1}[0,1], k \geq 0 \) and \( Q_n f \) is of degree \( d \geq 0 \), \( E_n^{(m)} f \) is the error functional of \( Q_n^{(m)} f \) and \( \delta = \min\{d, k\} \), then

\[
|E_n^{(m)} f| \leq \frac{F_{\delta+1}}{m^{\delta+1}} \frac{1}{\delta!} \int_0^1 E_n^x [(x - t)_+] dt
\]

where \( F_{\delta+1} = \sup_{x \in [0,1]} |f^{(\delta+1)}(x)| \).

Note that the error bound for \( |E_n^{(m)} f| \) differs from the estimate for \( |E_n f| \), given by (2) by a factor \( m^{-(\delta+1)} \).

For higher dimensions, the error functional of a quadrature rule is represented by Sard kernels. We restrict our attention to functions of only two variables having the property that \( f \in C^{k+1}(\overline{\Omega}) \). Extensions to higher dimensions are straightforward. The following theorem gives the representation of the error functional \( E_n f \) of a quadrature rule \( Q_n f \) of degree \( k \). We assume that all points \((x, y) \in \overline{\Omega}\) have \( x \geq 0, y \geq 0 \), and \((0,0) \in \Omega\). Partial derivatives are denoted by \( f_{i,j} \) (Engels, 1980).
Theorem 3. The functional \( E_n f = I_n f - Q_n f \) has the representation

\[
E_n f = \frac{1}{k!} \sum_{\mu=0}^{\lambda} \binom{k}{\mu} \left\{ \int_0^\gamma K^\mu_1(u) f_{k+1-\mu,\mu}(u,0) du \\
+ \int_0^\sigma K^\mu_2(v) f_{\mu,k+1-\mu}(0,v) dv \right\}
\]

\[
(3)
\]

\[
= \sum_{k=0}^{\lambda} \left\{ \int_\Omega K_\lambda,\lambda(u,v) f_{\lambda+1,\lambda+1}(u,v) du dv, \quad k = 2\lambda + 1 \\
+ \int_\Omega K_{\lambda-1,\lambda}(u,v) f_{\lambda,\lambda+1}(u,v) du dv \right\}, \quad k = 2\lambda
\]

with \( \sum^* = \left\{ \begin{array}{ll} \sum & \text{for } k = 2\lambda + 1 \\
\sum & \text{for } k = 2\lambda \end{array} \right. \) and \( \sum_{n=1}^N a_n = \frac{1}{2} a_1 + a_2 = \cdots + a_N. \)

Here \( \tau = \tau(y) > 0, \sigma = \sigma(x) > 0 \) are such that \( (\tau, y) \in \partial \Omega \) and \( (x, \sigma) \in \partial \Omega. \) The functions \( K^\mu_1(u), K^\mu_2(v), K_{\lambda,\lambda}(u,v), K_{\lambda-1,\lambda}(u,v) \) and \( K_{\lambda-1,\lambda}(u,v) \) are the Sard kernel functions of \( E_n f \) and are defined as:

\[
K^\mu_1(u) \equiv E^x,y_\mu[(x-u)^{k-\mu}], \quad \mu = 0(1)\lambda, \quad (x,y) \in \bar{\Omega}
\]

\[
K^\mu_2(v) \equiv E^x,y_\mu[(y-v)^{k-\mu}], \quad \mu = 0(1)\lambda, \quad (x,y) \in \bar{\Omega}
\]

\[
K_{\lambda,\lambda}(u,v) \equiv E^x,y_\mu[(x-u)^{k+1}(y-v)^{k+1}], \quad k = 2\lambda + 1, \quad (x,y) \in \bar{\Omega}
\]

\[
K_{\lambda-1,\lambda}(u,v) \equiv E^x,y_\mu[(x-u)^{k+1}(y-v)^{k+1}], \quad k = 2\lambda, \quad (x,y) \in \bar{\Omega}
\]

\[
K_{\lambda-1,\lambda}(u,v) \equiv E^x,y_\mu[(x-u)^{k+1}(y-v)^{k+1}], \quad k = 2\lambda, \quad (x,y) \in \bar{\Omega}
\]
To develop a result for the two dimensional case, similar to that presented in Theorem 2, we choose $\Omega = [0, 1] \times [0, 1]$ as integration region and subdivide it along the $x$ and $y$ axis. We will give, for the following theorem, an outline of the proof as it is long and tedious.

Theorem 4. Let $\Omega = [0, 1]^2$ and $E_n f = I_n f - Q_n f$, where $E_n f$ is the error functional of the quadrature rule $Q_n f$ as it applies to $\Omega$. Consider partitions $\Pi_1 = \{0 = u_0 < u_1 < \cdots < u_{m_1-1} < u_{m_1} = 1\}$ along the $x$-axis and $\Pi_2 = \{0 = v_0 < v_1 < \cdots < v_{m_2-1} < v_{m_2} = 1\}$ along the $y$-axis. Let $(E_n f)_{ij}$ be the error obtained by applying a scaled version of $E_n f$ over the subrectangle $\Omega_{ij} = [u_{i-1}, u_i] \times [v_{j-1}, v_j]$ of $\Omega$. Then

$$|(E_n f)_{ij}| \leq \frac{1}{k!} \sum_{\mu=0}^{\lambda} \binom{k}{\mu} \left\{ (\Delta u_i)^{k-\mu+2}(\Delta v_j)^{\mu+1} e_1^\mu (F^1_{k+1-\mu,\mu})_{ij} + (\Delta u_i)^{\mu+1}(\Delta v_j)^{k-\mu+2} e_2^\mu (F^2_{k+1-\mu,\mu})_{ij} \right\} \frac{(\Delta u_i)^{\lambda+2}(\Delta v_j)^{\lambda+2}}{\lambda !} e_{\lambda,\lambda} (F_{\lambda+1,\lambda+1})_{ij}, \quad k = 2 \lambda + 1$$

$$+ \left\{ \frac{1}{2(\lambda-1)!\lambda!} \left\{ (\Delta u_i)^{\lambda+2}(\Delta v_j)^{\lambda+1} e_{\lambda,\lambda-1} (F_{\lambda+1,\lambda})_{ij} \right. \right. + (\Delta u_i)^{\lambda+1}(\Delta v_j)^{\lambda+2} e_{\lambda-1,\lambda} (F_{\lambda+1,\lambda})_{ij} \right\}, \quad k = 2 \lambda$$

where $e_i^\mu = \int_0^1 |K^\mu_i(u)| du$, $i = 1, 2$ and $e_{ij} = \int_\Omega |K_{i,j}(u,v)| dudv$ and

$$(F^1_{p,q})_{ij} = \sup_{x \in [u_{i-1}, u_i]} |f_{p,q}(x, 0)|,$$

$$(F^2_{p,q})_{ij} = \sup_{y \in [v_{j-1}, v_j]} |f_{p,q}(0, y)|,$$

$$(F_{p,q})_{ij} = \sup_{(x,y) \in \Omega_{ij}} |f_{p,q}(x, y)|.$$
Proof. Using Theorem 3, the representation of \((E_n f)_{ij}\) will be

\[
(E_n f)_{ij} = \frac{1}{k!} \sum_{\mu=0}^{\lambda} \binom{k}{\mu} \left\{ \int_{u_{i-1}}^{u_i} (K_1^n(u))_{ij} f_{k+1-\mu,\mu}(u,0) du + \int_{v_{j-1}}^{v_j} (K_2^n(v))_{ij} f_{\mu,k+1-\mu}(0,v) dv \right\}
\]

\[
+ \left\{ \begin{array}{l}
\frac{1}{\lambda!} \int_{u_{i-1}}^{u_i} du \int_{v_{j-1}}^{v_j} dv \ (K_{\lambda,\lambda}(u,v))_{ij} f_{\lambda+1,\lambda+1}(u,v), \quad k = 2\lambda + 1 \\
\frac{1}{2(\lambda-1)!\lambda!} \int_{u_{i-1}}^{u_i} du \int_{v_{j-1}}^{v_j} dv [(K_{\lambda-1,\lambda}(u,v))_{ij} f_{\lambda+1,\lambda}(u,v) + (K_{\lambda-1,\lambda}(u,v))_{ij} f_{\lambda,\lambda+1}(u,v)], \quad k = 2\lambda
\end{array} \right.
\]

But,

\[
(K_1^n(u))_{ij} = (E_n^{xy}[(y-v_{j-1})^{\mu}(x-u)^{k-\mu}])_{ij}
\]

\[
(4) \quad = \int_{u_{i-1}}^{u_i} dx \int_{v_{j-1}}^{v_j} dy \ (y-v_{j-1})^{\mu}(x-u)^{k-\mu} - (\Delta u_i)(\Delta v_j) \sum_{\ell=1}^{n} A\ell(v_{j-1} + \Delta v_j \ell - v_{j-1})^{\mu}(u_{i-1} + \Delta u_i \ell - u)^{k-\mu}
\]

We introduce the following transformations

\[
x = u_{i-1} + \Delta u_i \gamma
\]
\[
y = v_{j-1} + \Delta v_j \delta
\]
\[
u = u_{i-1} + \Delta u_i \alpha
\]
\[
v = v_{j-1} + \Delta v_j \beta
\]

Thus \(x - u = \Delta u_i(\gamma - \alpha)\) and \(y - v_{j-1} = \Delta v_j \delta\). Equation (4) gives
\[(K^\mu_1(u))_{ij} = (\Delta u_i)^{k-\mu+1}(\Delta v_j)^{\mu+1} \int_0^1 d\gamma (\gamma - \alpha)^{k-\mu} \int_0^1 d\delta \delta^\mu \]
\[- (\Delta u_i)^{k-\mu+1}(\Delta v_j)^{\mu+1} \sum_{l=1}^n A_l(x_l - \alpha)^{k-\mu} y_l^\mu \]
\[= (\Delta u_i)^{k-\mu+1}(\Delta v_j)^{\mu+1} \left[ \int_0^1 d\gamma (\gamma - \alpha)^{k-\mu} \int_0^1 d\delta \delta^\mu - \sum_{l=1}^n A_l(x_l - \alpha)^{k-\mu} y_l^\mu \right] \]
\[= (\Delta u_i)^{k-\mu+1}(\Delta v_j)^{\mu+1} \epsilon_n^\delta[\delta^\mu(\gamma - \alpha)^{k-\mu}] \]
\[= (\Delta u_i)^{k-\mu+1}(\Delta v_j)^{\mu+1} K^\mu_1(\alpha) \]

In a similar fashion we obtain

\[(K^\mu_1(v))_{ij} = (\Delta u_i)^{\mu+1}(\Delta v_j)^{k-\mu+1} K^\mu_2(\beta) \]

\[(K_{\lambda,\lambda}(u,v))_{ij} = (\Delta u_i)^{\lambda+1}(\Delta v_j)^{\lambda+1} K_{\lambda,\lambda}(\alpha, \beta) \]

\[(K_{\lambda,\lambda-1}(u,v))_{ij} = (\Delta u_i)^{\lambda+1}(\Delta v_j)^{\lambda} K_{\lambda,\lambda-1}(\alpha, \beta) \]

\[(K_{\lambda-1,\lambda}(u,v))_{ij} = (\Delta u_i)^{\lambda}(\Delta v_j)^{\lambda+1} K_{\lambda-1,\lambda}(\alpha, \beta) \]

Therefore,

\[(E_nf)_{ij} = \frac{1}{k!} \sum_{\mu=0}^\lambda \binom{k}{\mu} \left\{ (\Delta u_i)^{k-\mu+2}(\Delta v_j)^{\mu+1} \int_0^1 d\alpha K^\mu_1(\alpha)g_{k+1-\mu,\mu}(\alpha,0) \right. \]
\[+ (\Delta u_i)^{\mu+1}(\Delta v_j)^{k-\mu+2} \int_0^1 d\beta K^\mu_2(\beta)g_{\mu,k-\mu+1}(0,\beta) \]
\[\left. + \left\{ \begin{array}{c}
\frac{(\Delta u_i)^{\lambda+1}(\Delta v_j)^{\lambda+2}}{2(\lambda+1)! \lambda!} \int_0^1 d\alpha \int_0^1 d\beta K_{\lambda,\lambda}(\alpha,\beta)g_{\lambda+1,\lambda+1}(\alpha,\beta), \quad k = 2\lambda + 1 \\
\frac{(\Delta u_i)^{\lambda+1}(\Delta v_j)^{\lambda+1}}{2(\lambda+1)! \lambda!} \int_0^1 d\alpha \int_0^1 d\beta K_{\lambda-1,\lambda}(\alpha,\beta)g_{\lambda+1,\lambda}(\alpha,\beta), \quad k = 2\lambda \\
\end{array} \right. \right\} \right. \]

where \(g_{p,q}(\cdot,\cdot)\) is the transformed partial derivative of \(f_{p,q}(\cdot,\cdot)\) Q.E.D.
Using the notation of Theorem 4, we derive the following result for the case of equal space partitions.

Corollary 1. Assume that partitions $\Pi_1, \Pi_2$ induce the same spacing in each direction. Let $h = \Delta u_i = u_i - u_{i-1} = \Delta v_j = v_j - v_{j-1} = \frac{1}{m}$ for $i = 0(1)m, j = 0(1)m$. Then

$$|E_n f| \leq \frac{1}{m^{k+3}} (B_n f)_ij \leq \frac{1}{m^{k+3}} B_n f$$

where $(B_n f)_ij$ is the bound in $\Omega_{ij}$ as dictated from Theorem 4 and $B_n f$ is the bound valid in $\Omega$ as derived from (3):

$$B_n f = \frac{1}{k!} \sum_{\mu=0}^{\lambda} \binom{k}{\mu} \left[ e_1^\mu F_{k+1-\mu,\mu}^1 + e_2^\mu F_{k+1-\mu,\mu}^2 \right] + \left\{ \begin{array}{l} \frac{1}{\lambda+1} e_\lambda, \lambda F_{\lambda+1,\lambda+1}, \quad k = 2\lambda + 1 \\ \frac{1}{2(\lambda-1)!\lambda!} (e_\lambda, \lambda-1 F_{\lambda+1,\lambda} + e_{\lambda-1,\lambda} F_{\lambda+1,\lambda+1}), \quad k = 2\lambda. \end{array} \right.$$  

By summing over all subregions, we obtain a bound for the error $E_n^{(m^2)} f$ of the rule $Q_n^{(m^2)} f$ ($m^2$-copy rule).

Corollary 2. The error functional $E_n^{(m^2)} f$ of the $Q_n^{(m^2)} f$ rule is bounded as

$$|E_n^{(m^2)} f| \leq \frac{1}{m^{k+1}} B'_n f$$

Generalizing the results of Corollaries 1, 2 for the $N$-dimensional case we obtain the following (The proof is analogous to that of Theorem 4)
Theorem 5. Let $\Omega = [0,1]^N$ and $E_nf = I_\Omega f - Q_nf$. Consider partitions $\Pi_1, \Pi_2, \cdots, \Pi_N$ along each of the coordinate axes with equal spacing in each direction. Let $h$ be the common step size and denote $h = \frac{1}{m}$. Then the error functional $(E_nf)_i$ of the rule $Q_nf$ scaled to $\Omega_i$ satisfies

$$|E_nf|_i \leq \frac{1}{m^{k+1+N}} (\tilde{B}_n f)_i.$$

Summing over all subregions, we obtain a bound for $E_n^{(m^N)} f$ which represents the error functional of an $m^N$-copy rule $Q_n^{(m^N)} f$.

Corollary 3. The error functional $E_n^{(m^N)} f$ of the rule $Q_n^{(m^N)} f$ is bounded as

$$|E_n^{(m^N)} f| \leq \frac{1}{m^{k+1}} \tilde{B}_n f.$$

Note that many integration algorithms make use of the $m^N$-copy rule. For example, the algorithm presented in Chapter V, subdivides each triangle (2-simplex) into 4 similar triangles ($m = 2, N = 2$). Also in DCUTET (Berntsen et al., 1990), a tetrahedron (3-simplex) is partitioned into $2^3$ similar subsimplices ($m = 2, N = 3$) at each subdivision step. Some algorithms apply $m^N$-copy rules or approximations thereof in conjunction with a suitable extrapolation techniques. Extrapolation is performed by generating a sequence of quadrature approximations using a sequence of values of $m$. The values $\{\frac{1}{m}\}$ form what is called a "mesh sequence". Two commonly
used mesh sequences are the geometric \((G)\) and the harmonic \((H)\) defined as

\[
G = \{ m^{-1} | m = 2^k, \, k = 0,1,2, \cdots \} \\
H = \{ m^{-1} | m = k+1, \, k = 0,1,2, \cdots \}
\]

respectively. For example, classical Romberg integration is based on the endpoint trapezoidal rule and the \(G\) mesh sequence.

Lyness (1976b) gives a generalization of the classical Euler-Mclaurin formula making an expansion for the quadrature error functional for \(\Omega = [0,1]^N\).

**Theorem 6.** Let \(f(\vec{x})\) and all of its derivative functions \(f^{(r_1,r_2,\cdots,r_N)}(\vec{x})\) of total order \(r_1 + r_2 + \cdots + r_N \leq p\), be integrable over the unit hypercube. Then

\[
E^{(m^N)}_n f = Q^{(m^N)}_N f - I_n f = \sum_{s=1}^{t} \frac{B_s}{m^s} + R_t(Q^{(m^N)}_n; f) \, \, N \leq t \leq p
\]

where \(B_s\) depends on \(Q_n f\) and \(f(\vec{x})\) but not on \(m\), and \(R_t(Q^{(m^N)}_n; f) = O(m^{-t})\) as \(m \to \infty\). Furthermore if the rule \(Q_n f\) is symmetric, then \(B_s = 0, s \) odd. Also if \(Q_n f\) is of degree \(d\), then \(B_s = 0\) for \(s \leq d\).

Note that an expansion for the remainder term \(R_t\), using Sard kernels was derived for \(N = 2\) in Section 4.3 (under the consideration that the derivatives of \(f(\vec{x})\), of total order \(\ell\) are continuous). Similar results were obtained by Lyness and McHugh (1970). Asymptotic expansions have also been developed for the cases where \(f(\vec{x})\) exhibits certain types of singularities.
over a standard region \((N\text{-cube}, N\text{-simplex})\). We list some results that relate to the subsequent discussion in this chapter. Lyness (1976b) provides an expansion for \(Q^{(mN)}_n f - I_n f\), valid for a class of functions with an \(r^\alpha\)-type singularity, that is functions of the form

\[
f(\bar{x}) = r^\alpha \phi(\theta) h(r) g(\bar{x}) \quad \alpha > -N, \quad r = x_1^2 + \cdots + x_N^2\]

where

(i) \(f(\bar{x})\) is integrable over \(\Omega = [0, 1]^N\);

(ii) \(\phi(\theta)\) is analytic in \(\theta_2, \theta_3, \ldots, \theta_N\) for all values of these variables for which \((1, \theta)\) lies in \(\bar{\Omega}\);

(iii) \(h(r)\) is analytic for \(0 \leq r \leq N\);

(iv) \(g(\bar{x})\) is analytic in each variable \(x_i\) in \(\bar{\Omega}\).

Note that \((r, \theta)\) represents the point \(\bar{x}\) in a hyperspherical coordinate system.

Theorem 7. Let \(f(\bar{x})\) be function which satisfies the above conditions. Then

\[
E^{(mN)}_n f = Q^{(mN)}_n f - I_n f = \sum_{t=0}^{p-1} \frac{A_{\alpha+N+t}}{m^{\alpha+N+t}} + \sum_{s=1}^{t-1} \frac{B_s}{m^s} + O(m^{-t})
\]

\[
N \leq \ell \leq \alpha + N + p, \quad \alpha \neq \text{integer}
\]

and

\[
E^{(mN)}_n f = Q^{(mN)}_n f - I_n f = \sum_{s=1}^{\ell-1} \frac{A_s + B_s}{m^s} + \sum_{s=1}^{\ell-1} \frac{C_s}{m^s} \ell nm + O(m^{-\ell} \ell nm)
\]

\[
\ell \geq N, \quad \alpha = \text{integer}.
\]
The coefficients $A_{\alpha+N+t}$, $C_s$, and $B_s$ depend only on the quadature rule $Q_n f$ and the integrand function $f(x)$. If $Q_n f$ is symmetric then $B_s = C_s = 0$, $s$ odd. Furthermore, if $Q_n f$ is of degree $d$, then $B_s = C_s = 0$ for $s \leq d$.

The error expansion for $Q_n^{(m^N)} F$ when $F(x) = r^\alpha \ell \eta r \phi(\theta) h(r) g(x)$ can be derived from Theorem 7 and is also given in Lyness (1976b).

Theorem 8. Let $F(x)$ be defined as above. Then

$$E_n^{(m^N)} F = Q_n^{(m^N)} F - I_\Omega F = \sum_{t=0}^{p-1} \frac{A_{\alpha+N+t}}{m^{\alpha+N+t}} + \sum_{t=0}^{p-1} \frac{C_{\alpha+N+t} \ell n m}{m^{\alpha+N+t}}$$

$$+ \sum_{s=1}^{t-1} \frac{B_s}{m^s} + O(m^{-t}), \quad N \leq \ell \leq \alpha + N + p, \quad \alpha \neq \text{integer},$$

and

$$E_n^{(m^N)} F = Q_n^{(m^N)} F - I_\Omega F = \sum_{t=0}^{p-1} \frac{A_{\alpha+N+t}}{m^{\alpha+N+t}} + \sum_{t=0}^{p-1} \frac{C_{\alpha+N+t} \ell n m}{m^{\alpha+N+t}}$$

$$+ \sum_{t=0}^{p-1} \frac{D_{\alpha+N+t} (\ell n m)^2}{m^{\alpha+N+t}} + \sum_{s=1}^{t-1} \frac{B_s}{m^s} + O(m^{-t}), \quad N \leq \ell \leq \alpha + N + p, \quad \alpha = \text{integer}.$$

If the quadrature rule $Q_n F$ is symmetric, then $B_s = C_s = 0$, $s$ odd.

Furthermore, if $Q_n F$ is of degree $d$, then $B_s = C_s = 0$, for $s \leq d$.

Expansions similar to those of Theorems 6, 7, 8 are also valid when the integration region is the unit $N$-simplex. (Lyness and Puri, 1973; Lyness, 1978; Lyness and Montegato, 1980).
When the integrand function has line singularities over the unit \( N \)-cube, then an expansion of the error functional is given by Theorem 9 (de Doncker, 1987).

**Theorem 9.** Let \( f(x) \) be defined as \( f(x) = \prod_{i=1}^{k} x_{i}^{\lambda_{i}} g(x) \), where \( g(x) \) is analytic in its variables, \( k \leq N, \lambda_{i} > -1 \) and \( Qf \) is an one-point rule for \( \Omega = [0,1]^{N} \). Then

\[
E^{(m^{N})} f = Q^{(m^{N})} f - I_{\Omega} f \sim \sum_{s=1}^{k} \sum_{(i_{1}, \ldots, i_{s}) \in J_{s}} \sum_{t \geq 0} \frac{E_{s}^{\lambda_{1} + \lambda_{2} + \cdots + \lambda_{s} + t + s}}{m^{\lambda_{1} + \lambda_{2} + \cdots + \lambda_{s} + t + s}} + \sum_{s \geq 1} \frac{B_{s}}{m^{s}}
\]

where the coefficients are independent of \( m \) and \( J_{s} \) is the set of \( k \) integers taken \( s \) at a time.

Asymptotic expansions are also known for integrand functions with some other types of singularities (in one dimension or when the region is the unit square or triangle). We will restrict ourselves to the classes specified above.

### 4.4. Speedup Analysis

In this section we present a speedup analysis of a type of global adaptive algorithm used to approximate

\[
I_{\Omega} f = \int_{\Omega} f(x) \, d\bar{x}
\]
over an $N$-dimensional region such that subregions of the same nature, are the
original region, as generated at each subdivision step ($N$-cube, $N$-simplex).
We assume that the algorithm is implemented on a MIMD-SM machine. The
discussion is based on results presented by Rice (1974, 1975), but the method
is extended to handle multivariate integration of functions which may exhibit
some types of singular behavior. Also the analysis applies to certain global
adaptive quadrature algorithms. Note that the analysis by Rice (1974, 1975) is
intended for a local adaptive procedure. We start by giving some preliminary
information, assumptions and definitions that will be needed in the remainder
of this section.

We assume that there are $P$ independent processors on a MIMD-SM
machine, each running one slave process. Also the master process (defined in
Chapter III) is executed on a separate processor. The master process creates
the slave processes and all access the same global region collection in shared
memory. For the data structure of the region collection we chose to assume an
error-keyed heap. Such a structure was used in the algorithm of Chapter III.
Furthermore we assume that at each subdivision step the algorithm produces
$\Lambda = \nu^N$ subregions of equal size. We will use $\Lambda^d = m_d^N$ to apply with
$\nu^N$-copy rules, where $\{m\}$ is a sequence, $m_i = \nu^i$. The following four
times will emerge in the analysis below.

1. Processing-time: Time for a slave process to perform integration and
estimate the error over a single subregion of $\Omega$. Assuming that a fixed number
of function evaluations are used, the processing-time is bounded by a constant
2. Acquisition-of-region-time: Time to extract a region from the global data structure, binary heap, and reheap the resulting tree. Let MAXSUB be the maximum allowed number of subregions in the heap. Then the above time will be \( O(\log \text{MAXSUB}) \). In most algorithms, MAXSUB is determined as a linear function of \( M \), which denotes the overall maximum number of function evaluations used by the algorithm. For example, in the algorithm of Chapter 3, \( \text{MAXSUB} = \frac{M \cdot \text{NUM}}{2 \cdot \text{NUM}} + 1 \), where \( \text{NUM} \) is the fixed number of function evaluations used over each subregion. Therefore, the acquisition-of-region-time is of order \( O(\log M) \).

3. Insertion-of-region-time: Time to insert subregions to the global binary heap data structure and reheap the tree. Assuming that a fixed number of subregions are generated at each subdivision step, by a similar argument as in (b), we conclude that the insertion-of-region-time is of order \( O(\log M) \).

4. Update-time: Time required by the slave processes to add their contribution and update the estimates for the integral and error value. Since the updates are performed within a monitor (critical code), the update-time could be bounded by a function \( T_u(P) \). In this context, Rice (1974) introduces the return-time as the time from the completion of the processing of a region to the acquisition of the results by the collection manager and algorithm controller. He models the return-time as a linear function of the participating number of processors.

Let \( T_c \) denote the cycle-time, which is the overall time elapsed from
the moment a region is retrieved from the region collection until its subregions are inserted into the global task pool. Summing up the above times we obtain

\[ T_c \leq c_1 + c_2 \log M + T_u(P). \]

In the remaining part of this section we present a series of lemmas which lead to the derivation of the speedup function for a region-size global adaptive algorithm on MIMD-SM machines. The speedup results are given for functions with sufficient large number of continuous derivatives and also for those with some singular behavior in the integration region. Following the presentation of some basic concepts which are used in the sequel, Lemma 1, gives the relation between the depths of the region trees generated by a sequential (a) local and (b) region-size adaptive strategy. The discussion following Lemma 1 demonstrates under what conditions and assumptions, the result of Lemma 1 is valid for a parallel implementation of the above strategies on MIMD-SM machines. Lemma 2 presents a derivation of the order of the set of discarded regions by a local adaptive strategy. Lemma 3 estimates the overall number of function evaluations needed by a quadrature rule of a local adaptive strategy, for integration over a standard \( N \) - dimensional region. Lemma 4 gives a bound for the depth of the region tree generated by a local adaptive algorithm. Results are obtained for functions which may exhibit some singular behavior and for those having sufficient large number of continuous derivatives. Using the results obtained by the four lemmas, the speedup function for a region-size global adaptive strategy is derived.
Every adaptive quadrature algorithm can be thought of as an algorithm for generating and searching a tree of regions. Each node of the tree corresponds with a region which is stored at the node together with additional information including an estimate for the integral and error. The root of the tree corresponds with the original region of integration, while the children of a node represent subregions into which the parent is partitioned. Lemma 1 gives a relation between the depth of tree of regions generated by a region-size global adaptive algorithm (defined in Section 4.2) and that of a local adaptive strategy in a sequential context. We assume that \( \Lambda \) subregions of equal size are generated at each subdivision step. The property was stated for \( \Lambda = 2 \) and for one dimensional integration by Shapiro (1984).

Consider the following local adaptive strategy, which involves an iteration of the form: (a) select a particular region \( R \) with associated error \( E(R) \); (b) subdivide \( R \) into \( \Lambda \) parts and compute new integral and error estimates for each part; (c) discard those regions where the error estimate is acceptable and add the others to the collection for further processing.

Let \( R_0 = \Omega \) denote the original integration region and let the local tolerated error acceptance criterion of (c) above be defined as \( E(R_i) \leq \varepsilon_a \frac{s(R_i)}{s(R_0)} \), where \( \varepsilon_a \) is the global absolute tolerance and \( s(R) \) measures the size of the region \( R \). Let \( S, T \) denote the active and discard sets of regions. A region will be moved from \( S \) to \( T \) if it satisfies the local acceptance criterion.

Lemma 1. The depth of the tree of regions generated by a region-size global
adaptive algorithm does not exceed the depth of the tree generated by the above defined local adaptive strategy.

Proof. Let $R_0$ denote the initial integration region. Assume that the region-size global adaptive algorithm is not terminating. Let a region $R_i$ of depth $i$, in the tree of regions, be selected for subdivision by the algorithm and let $E(R)$ denote the error estimate for a region $R$. We will show that $R_i$ cannot be a leaf node in the tree generated by the local adaptive algorithm.

By definition of the selection procedure of the region-size global adaptive strategy, we must have

$$\frac{E(R_j)}{s(R_j)} \leq \frac{E(R_i)}{s(R_i)} \quad \forall R_j \in S,$$

where $S$ is the current set of regions. Note that all regions are characterized as active. Since

$$s(R_j) = \frac{s(R_0)}{\Lambda^j} \quad \text{and} \quad s(R_i) = \frac{s(R_0)}{\Lambda^i},$$

we have

$$\frac{E(R_j)}{s(R_0)} \leq \frac{E(R_i)}{s(R_0)} \quad \forall R_j \in S. \quad \text{Thus,}$$

$$(5) \quad \Lambda^j E(R_j) \leq \Lambda^i E(R_i) \quad \forall R_j \in S.$$
If $R_i$ was a leaf node of the tree generated by the local adaptive strategy, then it would satisfy the local acceptance criterion

\begin{equation}
E(R_i) \leq \varepsilon_a \frac{s(R_i)}{s(R_0)}.
\end{equation}

Consequently, $E(R_i) \leq \varepsilon_a \Lambda^{-i}$ and so $\Lambda^i E(R_i) \leq \varepsilon_a$. Thus, in view of (5) we conclude

\begin{equation}
\Lambda^j E(R_j) \leq \varepsilon_a \forall R_j \in S.
\end{equation}

Summing $E(R_j) \leq \varepsilon_a \Lambda^{-j}$ over $R_j \in S$ gives

\[
\sum_{R_j \in S} E(R_j) \leq \varepsilon_a \frac{\sum_{R_j \in S} \Lambda^{-j} s(R_0)}{s(R_0)} = \varepsilon_a \frac{\sum_{R_j \in S} s(R_j)}{s(R_0)} = \varepsilon_a.
\]

Thus

\begin{equation}
\sum_{R_j \in S} E(R_j) \leq \varepsilon_a.
\end{equation}

Since with (7), the termination condition of the region-size global adaptive algorithm is satisfied, $R_i$ is not to be subdivided in the region-size adaptive strategy.

Therefore, the depth of tree generated by a region-size global adaptive procedure does not exceed the depth of the tree generated by a local adaptive algorithm. Q.E.D.
For the remainder of the discussion, we assume that the error associated
with a region reduces at a certain rate after the subdivision of a region. From
the proof of Lemma 1 we can also conclude that in a sequential context,
the tree generated by a region-size global adaptive algorithm is a subtree of
that generated by a local adaptive strategy. However, in a parallel context,
assuming \( P \) processors which are part of a MIMD-SM machine, the tree
generated by a parallel region-size global adaptive algorithm may differ from
the one generated sequentially by the same strategy. This is caused by

1. Singularity loss. Let \( R_i \) be a region selected for subdivision by a
sequential version of the region-size global algorithm. Let \( \{R_{i_1}, R_{i_2}, \ldots, R_{i_k}\} \)
correspond to the leaf nodes of the current region tree. Let us also assume that
the errors associated with the \( R_{ij}, j = 1, \ldots, k \) satisfy the local acceptance
criterion of the form (6). It is evident that for a local adaptive algorithm,
none of the \( R_{ij}, j = 1, \ldots, k \) will be selected for further subdivision. In
the corresponding parallel region-size algorithm, recall that only one process
at a time can access the region collection. However, after \( R_i \) is obtained by
one process, the remaining processes can acquire regions from the \( R_{ij}, j = 1, \ldots, k \) for further subdivision. The concept of the singularity loss was
defined by Kapenga and de Doncker (1988a) as the loss incurred from processes
performing unneeded work, caused by selecting an unimportant region from
the pool before some other process has partitioned its problem and returned
an important task. In practice, one case where this occurs is in problems
which have a singularity such that the region tree the serial algorithm would
produce has one long narrow subtree resulting from the subdivision of regions in the vicinity of the singularity. As a result, *singularity loss* will tend to increase the width of the tree generated by the parallel region-size global adaptive algorithm rather than the depth. We are assuming that the time needed to process each region (its subdivision and integration over its subregions) is approximately constant.

2. *Breaking loss.* This loss is caused by processes which continue to execute after their work is no longer needed. It may occur near termination of the algorithm between the time that compliance with the termination condition is detected by a process, and the time that all other processes are notified.

Possible modulo the subdivisions caused by the *breaking loss* at the end of the computations, the result obtained in Lemma 1 remains valid for a parallel region-size strategy. One may also note the following. Use of a local discard criterion (6) will not have an effect in a serial region-size algorithm (since if a region is discarded in a local adaptive algorithm, it would not be selected by the region-size global adaptive strategy). However, use of the local discard criterion in the parallel region-size algorithm would avoid *singularity* and *breaking loss* to the extend that no regions will be subdivided which would not have been subdivided by the local adaptive strategy. At the same time this enforces the validity of Lemma 1 as for the serial algorithm.

The following lemma gives an estimate of the cardinality of the *discard set* for a local adaptive algorithm, i.e., the number of regions which will be generated eventually by the subdivision procedure. Assume that the subdii-
sion of a region $R$ produce $\Lambda$ subregions $R_\ell$, $\ell = 1, \cdots, \Lambda$. Let $0 < \gamma, \beta < \frac{1}{\Lambda}$. Consider the following model for the rate by which the error associated with the region $R$ decreases at subdivision. $R$ is termed a regular region if it does not contain a singularity, and a distinguished region if it has an integrand singularity at a corner.

(a) $E(R_\ell) = \gamma E(R), \ell = 1, \cdots, \Lambda$, if $R$ is a regular region,

(b) $E(R_\ell) = \beta E(R), \ell = 1, \cdots, \Lambda$, if $R$ is a distinguished region.

The following lemma is an adaptation of Theorem 1 in Rice (1975). We use a modified local acceptance criterion from the one used in Lemma 1.

Lemma 2. For a local adaptive algorithm satisfying the assumptions above, let $F(\gamma, \varepsilon_\alpha)$ be the size of the discard set $T$ when the algorithm terminates, where $\varepsilon_\alpha$ is the tolerated absolute error. Then

$$F(\gamma, \varepsilon_\alpha) = O(\varepsilon_\alpha^{\frac{1}{\log_\Lambda(1/\gamma)}}).$$

Proof. The proof proceeds by examining two cases.

Case 1. Consider a regular region $R$ with associated error $E(R)$. The error $E(R')$ associated with a region $R'$ resulting after $d$ subdivisions is requested to satisfy
in order that $R'$ be put into the discard set $T$. For reasons of simplicity, a modified local acceptance criterion was used in (9). This modification leads to the sum of the errors of regions in $T$ being no greater than $\varepsilon_\alpha$ times the size of the original region (as opposed to being no greater than $\varepsilon_\alpha$).

Since $R'$ was produced after $d$ subdivisions, $s(R') = \frac{s(R)}{\Lambda^d}$. From (9) we obtain

$$(10) \quad (\Lambda \gamma)^d \leq \frac{\varepsilon_\alpha s(R)}{E(R)}.$$

The smallest possible value $d'$ of $d$ for which (10) is valid, satisfies

$$d' \geq \frac{\log_\Lambda \left( \frac{\varepsilon_\alpha s(R)}{E(R)} \right)}{\log_\Lambda (\Lambda \gamma)} > d' - 1.$$

Note that $d$ and $d'$ represent integers, and it is assumed that $\Lambda \gamma < 1$.

Since $\Lambda$ subregions are generated at each subdivision step, the number of regions placed in $T$ will satisfy

$$\Lambda^{d'} = O(\varepsilon_\alpha^{\frac{1}{\log_\Lambda (\Lambda \gamma)}}).$$

**Case 2.** Consider a *distinguished* region $R^*$. After $d$ subdivisions, the error of a *distinguished* descendant $R'^*$ of $R^*$ needs to satisfy

$$(11) \quad E(R'^*) = \beta^d E(R^*) \leq \varepsilon_\alpha s(R'^*)$$
in order to be placed in \( T \). Denoting by \( d'' \) the smallest possible \( d \) for which (11) is valid, and taking into account that \( s(R^*) = \frac{s(R^*)}{\Lambda^d} \), we have

\[
d'' \geq \frac{\log_{\Lambda} \left( \frac{s(R^*)}{E(R^*)} \right)}{\log_{\Lambda} (\gamma \beta)} \geq d'' - 1.
\]

Subdivision of a distinguished region provides, on the next level (call this level \( k \)), a distinguished region \( R^{'*} \) and \( \Lambda - 1 \) regions \( R_t \) with their associated errors \( \beta^k E(R^*) \); and this happens for \( k = 1, 2, \ldots, d'' - 1 \). A descendant \( R'_t \) of each of the regions \( R_t \) will be put in \( T \) after \( d_k \) subdivisions when

\[
(12) \quad E(R'_t) = \gamma^d E(R_t) = \gamma^d \beta^k E(R^*) \leq \varepsilon \alpha s(R'_t).
\]

Since \( s(R'_t) = \frac{s(R^*)}{\Lambda^k + d_k} \), equation (12) gives

\[
d_k \geq \frac{\log_{\Lambda} \left( \frac{\varepsilon \alpha s(R^*)}{(\Lambda \beta)^k E(R^*)} \right)}{\log_{\Lambda} (\gamma \beta)} > d_k - 1.
\]

The total number of regions descendant from \( R^* \) that will be put in \( T \) is bounded by

\[
(13) \quad \Lambda + \sum_{k=1}^{d''-1} \Lambda^d s < \Lambda + \Lambda \left( \frac{\varepsilon \alpha s(R^*)}{E(R^*)} \right)^{-1} \log_{\Lambda} (\Lambda \gamma) \sum_{k=1}^{d''-1} \left( (\Lambda \beta)^{-1} \right)^k.
\]

The summation in the right hand side of (13) can be replaced by a series summation, i.e., the upper limit \( d'' - 1 \) can be replaced by \( \infty \), in view of the fact that the geometric series converges.
Consequently, the right hand side of (13) is $O(\varepsilon^{\frac{1}{\log \lambda (\lambda \gamma)}})$, which concludes the proof. Q.E.D. Slight generalizations of Lemma 2 are given in the Corollaries following.

Corollary 4. If the assumptions (a) and (b) used with the "=" signs are replaced by inequalities "\leq", then the result of Lemma 2 holds.

Corollary 5. Under the conditions of Lemma 2 or Corollary 4 we also have that

$$F(\gamma, \varepsilon_\alpha) = \Theta(\varepsilon^{\frac{1}{\log \lambda (\lambda \gamma)}}),$$

i.e., there are constants $c_1$ and $c_2$ such that (for small enough $\varepsilon_\alpha$)

$$c_1 \varepsilon^{\frac{1}{\log \lambda (\lambda \gamma)}} \leq F(\gamma, \varepsilon_\alpha) \leq c_2 \varepsilon^{\frac{1}{\log \lambda (\lambda \gamma)}}.$$

Corollary 6. Let $g_1$ and $g_2$ be positive functions, $g_1$ defined on the set of regular regions and $g_2$ on the set of distinguished regions, with the property

$$g_r(R_i) \leq g_r(R_j), \text{ if } R_i \subseteq R_j,$$

where $r = 1$ or $2$ if $R_i$ and $R_j$ are regular or distinguished regions respectively. Let $\overline{g}_1 = \max\{g_1(R_i)\}$ for all regular regions $R_i$ and $\overline{g}_2 = \max\{g_2(R_i)\}$ for all distinguished regions $R_j$. If in the assumptions of (a) and (b), or in their modifications specified in Corollary 4 the quantities $\gamma$ and $\beta$ are replaced by $\gamma g_1(R_i)$ and $\beta g_2(R_i)$, then then conclusion of
Lemma 2 reads

\begin{equation}
F(\gamma, \varepsilon_a) = O(\varepsilon_a^{\mu_1} (\Lambda \gamma \bar{\gamma}_1)).
\end{equation}

Note that the quantity $\Lambda \gamma \bar{\gamma}_1$ in (14) is assumed smaller than 1.

Corollary 7. If in addition to the conditions of Corollary 6 we have that $\Lambda \gamma < 1$, $\bar{\gamma}_1 < 1$ and $\varepsilon_a \leq 1$ then the conclusion of Lemma 2 holds as stated in (8).

Lemma 3 will provide a bound on the number of function evaluations $M$ used by a local adaptive algorithm. Consider an integrand function $f$ such that $f \in C^{k+1}(R)$ for each regular region $R$ generated by subdivision process. Without loss of generality we can assume that the local integration rule is of degree at least $k$ (if it was not, we could replace $k$ by the minimum of $k$ and the degree $\tau$ of the rule).

On a regular region which corresponds to a "panel" (gridcell) of an $m^N$-subdivision of the original integration region $\Omega$ (the unit $N$-cube or simplex), the error can be bounded as suggested by Corollary 1 in this chapter (for $N = 2$) or its $N$-dimensional analogue (see also Theorem 5):

\begin{equation}
|(E_n f)|_r \leq \frac{1}{m^{k+1+N}} (B_n f)_r \leq \frac{1}{m^{k+1+N}} (B_n f).
\end{equation}

An example of a distinguished region would be a cell which has the origin as a vertex and the integration error of order $\frac{1}{m^{a+N}}$ for some $\alpha$ such that
\[ \alpha + N > 0. \] If, in the first part of Theorem 7, \( \alpha + N < \tau + 1 \leq \ell \), then

\[ \frac{a}{M^{\alpha+N}} \] would determine the order of the leading term in the error expansion of the \( m^N \)-copy rule. In the present context we assume \( m \) of the form

\[ m = m_i = \nu^i. \] Furthermore, a subdivision step of the quadrature algorithm subdivides each region into \( \Lambda = \nu^N \) (similar) subregions of equal size. Consequently a region on level \( d \) of the algorithm's region tree corresponds to a panel of the \( m_d^N = \nu^{Nd} \)-copy rule.

This leads us to estimate the ratios \( \gamma \) and \( \beta \) of Lemma 2 and its Corollaries as

\begin{equation}
\gamma = \nu^{-(k+1+N)} \quad \text{and} \quad \beta = \nu^{-\alpha+N}.
\end{equation}

Note, however, that since we required \( \Lambda \beta < 1 \), we need \( \alpha > 0 \).

**Lemma 3.** Under the above conditions on the integrand and subdivision process, the overall number of function evaluations \( M \) used by the local adaptive algorithm satisfies

\begin{equation}
M = \Theta(\varepsilon^{-\frac{N}{\alpha+1}}).
\end{equation}

**Proof.** From Lemma 2, Corollary 5 and (15), we have that

\[ F(\gamma, \varepsilon_\alpha) = \Theta(\varepsilon^{-\frac{N}{\alpha+1}}). \]
In the local adaptive algorithm, the total number of regions generated is less than \( A \) times the number of regions accepted. The bound given in (16) follows since a fixed number of function evaluations is used per region. Q.E.D.

Note that the result of Lemma 3 holds for a region-size global adaptive algorithm as long as the tree generated by the latter is a subtree of that of the local adaptive algorithms, such as for the serial algorithms, or when the region-size global adaptive algorithm is run in parallel with a "forced" local discard criterion (see the remarks after Lemma 1).

Lemma 4 will give a bound for the length \( \hat{d} \) of the longest path in the tree generated by the local adaptive algorithm.

Lemma 4. Let \( \zeta = \min\{k + 1, \alpha\} \) where \( k + 1 \) and \( \alpha \) are as in (15) and denote the original integration region by \( R_0 \). Then

\[
(17) \quad \hat{d} \leq \frac{k + 1}{N\zeta} \log_{\nu} \left( \frac{MK^{1/N}}{C} \right) + 1.
\]

Proof. Let \( Z = \nu^{-\zeta - N} \), and \( K = \frac{E(R_0)}{s(R_0)} \) where \( \Omega = R_0 \) denotes the original integration region. We derive a bound for the longest path in the tree by requiring, for a region on level \( d \),

\[
E(R) \leq Z^d E(R_0) \leq \varepsilon_{\alpha} s(R) = \frac{\varepsilon_{\alpha} s(R_0)}{\nu N^d},
\]

so that

\[
d \geq -\frac{1}{\zeta} \log_{\nu} \left( \frac{\varepsilon_{\alpha}}{K} \right),
\]
and solving for the smallest possible value \( (\hat{d}) \) of \( d \) which satisfies the above inequality. Taking into account that

\[
\hat{d} \geq -\frac{1}{\zeta} \log_{\nu} \left( \frac{\varepsilon \alpha}{K} \right) > \hat{d} - 1,
\]

and using Lemma 3, we obtain (17) as stated Q.E.D.

As in Rice (1974), we now consider the parallel processing of the region tree, as it proceeds from the root (the original region) toward the leaf nodes (the accepted subregions). Envision the complete tree as it exists at termination, with total number of nodes \( D \) and the longest path of length \( \hat{d} \).

The tree processing algorithm is characterized by the partitioning of the set of \( D \) nodes, at each point in time, into three groups: the leaf nodes which have been reached, the active nodes (which still need to be branched from), and all remaining nodes (which have not yet been reached). In regard to the parallel system with \( P \) processors we assume furthermore: a) if at any time \( t \) there are \( \rho < P \) active nodes then by time \( t + T_c \) (where \( T_c \) is the cycle time) all \( \rho \) of these nodes have been processed; b) if at any time \( t \) there are at least \( P \) active nodes, then \( P \) of these are processed from time \( t \) to \( t + T_c \).

The following Lemma can be proved as Theorem 2 in Rice (1974).

**Lemma 5.** The parallel tree processing time is bounded by

\[
\left( \frac{D}{P} + \hat{d} \right) T_c.
\]
Using Lemma 5 and the bound established for \( \hat{d} \) in Lemma 4, we now establish the final Theorem indicating the performance of the integration algorithm executed in parallel.

**Theorem 10.** The time \( T_{M,P} f \) taken by the parallel local adaptive algorithm using \( P \) processors and evaluating the integrand at \( M \) points is of the order

\[
T_{M,P} f = O \left( \frac{MT_c}{P} \right).
\]

**Proof.** According to Lemma 4 and in view of \( M = qD \) we have that

\[
\frac{D}{P} + \hat{d} < \frac{M}{Pq} + \frac{k + 1}{\zeta N} \log \left( \frac{MK^{\frac{x}{1-x}}}{C} \right) + 1.
\]

Thus

\[
\frac{D}{P} + \hat{d} = O \left( \frac{M}{Pq} + \frac{k + 1}{\zeta N} \log M \right)
\]

\[
= O \left( \frac{M}{Pq} \left( 1 + \frac{k + 1}{\zeta N} \frac{Pq}{M} \log M \right) \right).
\]

In a realistic situation, the number of function evaluations will be considerably larger than the number of processors \( P \). If in (19) we have \( M > P^x \) for \( x > 1 \), then the function \( \frac{P}{M} \log M \) is bounded, i.e.,

\[
\frac{P}{M} \log M < \frac{\log M}{M^{1-\frac{1}{x}}} < K_1
\]
for some constant $K_1$. Application of Lemma 5 now yields

$$T_{M,P} = \left( \frac{D}{P} + \hat{d} \right) T_c = O\left( \frac{MT_c}{P} \right).$$

Q.E.D.

Corollary 8. The result of Theorem 10 holds for a region-size adaptive algorithm as long as the region tree is a subtree of that associated with the local adaptive algorithm.

Recall that the cycle time $T_c$ in the bound of (18) satisfies

$$T_c \leq C_1 + C_2 \log M + T_u(P)$$

for some constants $C_1$ and $C_2$, and where $T_u(P)$ represents the update time. Note that the actual dependence of $T_u$ on $P$ is determined by the time needed by the (parallel) processors to update shared memory, which depends on the system and the implementation of synchronization.
CHAPTER V

A PARALLEL GLOBAL ADAPTIVE ALGORITHM
FOR INTEGRATION OVER A SET OF TRIANGLES
ON LOOSELY COUPLED SYSTEMS

5.1 Introduction

Let \( D = \bigcup_{i=1}^{M} \Delta_i \) where \( \{ \Delta_i \}^M_1 \) is a given set of non-overlapping triangles, i.e., the intersections of their interiors are pairwise disjoint. The problem for an automatic integrator is the calculation of an approximation \( Q(f) \) to the integral:

\[
I(f) = \int_D f(x,y) \, dx \, dy
\]

which will hopefully satisfy \( |I(f) - Q(f)| \leq \max\{ \varepsilon_a, \varepsilon_r |I(f)| \} \). The \( \varepsilon_a \) and \( \varepsilon_r \) are the user prescribed absolute and relative tolerances respectively.

One of the most natural ways to partition a two-dimensional region of integration is into a set of non-overlapping triangles. Applying an integration routine to each triangle in the partition will not always be effective. For example, setting the accuracy requirements on the separate triangles is hard if a relative error must be satisfied on the entire region. Furthermore, imposing a too restrictive tolerance on triangles which have a small contribution to the overall estimate of the integral will lead to unnecessary work by the integration algorithm. Problems may also arise in performing the integrations...
over each triangle to sufficient accuracy, especially in cases when the integral over the entire region is small while the contributions over some triangles are large in absolute value. To overcome these problems, TRIPET (de Doncker et al., 1990) was developed as an adaptive algorithm for integration over a triangulated two dimensional region. It also incorporates an extrapolation procedure by means of the $\varepsilon$-algorithm (de Doncker et al., 1990). A parallel global adaptive algorithm for integration over a set of triangles on a loosely coupled system, without using extrapolation, was developed by de Doncker and Kapenga (1992). A dynamic load balancing procedure was incorporated in the integrator. In this chapter we present a modified version of the parallel integration routine, where new load balancing schemes have been incorporated to enhance the performance of the algorithm. Section 5.2 gives some background information primarily about the node communication system that has been used and which allows the algorithm to be portable across a range of loosely coupled systems. The parallel algorithm is presented in 5.3 while the incorporated load balancing scheme is explained in section 5.4. A variety of test results demonstrating the performance of the algorithm are given in the last section of this chapter.

5.2 Background

Our main goals in constructing a parallel global adaptive numerical integrator were to ensure the portability of the routine across a range of loosely
coupled systems as well as the reliability and efficiency of the algorithm. Software portability has always been an important issue. In the past when all computers were of von Neumann type, portability could be achieved by establishing standards for programming languages. As long as the programmer did not use vendor-specific extensions, the code was portable, assuming that the appropriate compiler was available. With the emergence of parallel machines, the portability problem became much more severe, due to the fact that a parallel program relies on mechanisms for coordinating tasks on the different processors which have been assigned a common problem. These mechanisms are architecture dependent. In the last few years great efforts have been expended towards the development of autoparallelizing compilers. Successful progress towards solutions in this endeavor was reported for the case of shared memory machines, whereas for local memory systems, efforts are still in their infancy. Therefore, in order to avoid the situation where a large portion of code must be rewritten in porting an application program from one multicomputer (local memory) to another, we resort to the use of a parallel environment which is available on a large class of machines.

One such environment, which is widely used, is the Reactive Kernel/Cosmic Environment (RK/CE) system by Seitz et al., (1988). The Cosmic Environment (CE) and Reactive Kernel (RK) were developed to support a message-passing programming environment on multicomputer nodes. The CE system consists of a set of daemon processes, utility programs and libraries. It
can be used as a stand-alone system for running message-passing programs on network connected UNIX hosts (workstations) and can also support uniform communication between host and node processes on a multicomputer. RK is a node operating system developed by Seizovic (1988). The scheduling strategy that has been implemented in the RK operating system is characterized as reactive. With this type of strategy, a process will be scheduled to run only when there is a message for it. The process runs as long as it is making progress. The layered structure of RK with the well-defined interfaces between layers, provides ease in debugging, modifications, and extensions. The only difference between message-passing programs on UNIX hosts under the CE system and multiprocessors under the RK system is in their process creation functions. To be able to simulate multiprocessors, the CE system includes a feature called ghost cubes. This allows a collection of networked hosts (workstations) to behave like a (local memory) multiprocessor. Therefore, multicomputer programs written for this environment can run on a wide range of systems (iPSC1/2/860, Symult-2010, nCUBE-2, ghost cubes). In this work, most of the code development and initial debugging of the parallel integrator was performed on a ghost cube on the Sun network of the Department of Computer Science at Western Michigan University.

For the current implementation of our algorithm we use a setup with a node program and a host program. The first runs on the nodes of the multicomputer while the latter runs on the host processor. The host program
takes care of I/O, spawns the node processes, sends data to the nodes, and receives the final result and error estimate to the integral.

The entire set of processes involved in a computation is defined as a process group. One way that such a group, along with a (possibly empty) set of multicomputer nodes could be allocated by the user with the host utility `getcube`. Every process within a process group has a unique identification which is represented as an ordered pair: \((\text{node}, \text{pid})\). For a node process, \text{node} is a C integer in the range \(0 \cdots N - 1\), while \text{pid}, which represents a user process within a node, is a C integer in the range \(0 \cdots \text{MAXPID}\). The system’s process spawning mechanisms give the programmer explicit control of process placement through the arguments of dynamic process creation functions. The `spawn` function with syntax `spawn("filename", \text{node}, \text{pid}, "mode")` provides one way of creating node processes. We use the following piece of C code in the current implementation of the host program to spawn the node process (mediator program) which controls the new load balancing scheme. It is assumed that the user had acquired \(N\) nodes.

```c
/*host program */
#define APID 0
#define mediatorid 1
/* spawn the mediator program on node 1 of the multicomputer.
The object code of this program is stored in the file named med 1 */
if (ispawn=spawn("med 1", mediatorid, APID,"")){
    printf("failed spawn in node %d, with errorcode %d", 1, ispawn);
    exit();
}
```

The `spawn()` function returns 0 if it is successful or a non-zero error code if it
is unsuccessful. The process of spawning will fail if the specified file does not exist, if the node does not exist, or has insufficient memory, or if the pid is already in use, or outside of the allowed range. Thus the possibility of failing can be captured and an appropriate error message is displayed. The spawn() function returns only after the process exists or the operation has failed. When a process is successfully spawned, a startup routine calls main(), just as in other C environments. The following segment of code was used to spawn the node program. Copies are created on \( N - 1 \) nodes of the machine since node 1 is reserved to run the program for controlling the load balancing.

```c
/* host program */
#define APID 0
#define mediatorid 1
/* spawn the node program on \( N - 1 \) nodes of the machine.
The object code of the program is stored in comp_node file */
i = 0;
j = 0;
while (i < icnt) {
    if (i != mediatorid) {
        if (ispawn = spawn("comp_node", i, APID, "") ) {
            printf("failed spawn in node \%d, with error code \%d", i, ispawn);
            exit();
        } else {
            dest[2*j]=i;
            dest[2*j+1]=APID;
            j++;
        }
    } else {
        i++;
    }
} /* end of while */
```

The variable icnt holds the value of \( N \) obtained by a call to the function nnodes(), where \( N \) represents the number of participating processors. The
dest array will be used later on in the broadcasting of information from the host to the node processors.

Messages are logical units of information exchanged between processors. Message space is organized as a heap which is referenced to as the message heap. The latter is similar to the heap that C programmers access with the malloc() and free() commands. Message space can be allocated by:

\[ p = \text{xmalloc(length)}; \]

where length is specified in bytes and deallocated by:

\[ \text{xfree(p);} \]

Note that xmalloc() returns a pointer to a message buffer. A message that has been built in a block which has been allocated from the message heap can be sent using the function xsend():

\[ \text{xsend(p, node, pid);} \]

The xsend() function also deallocates the space in the message heap; that is, xsend(p, ...) is like xfree(p), except that it also sends a message. When xsend() returns, the message heap block is gone. A message can be received using a blocking receive function as follows:

\[ p = \text{xrecvb();} \]

which does not return until a message has arrived for the process. On the other hand, the use of the non-blocking receive function xrecv() will allow a process to probe for a message without giving up the right to continue execution. The xrecv() function may return a NULL pointer if there is no message queued for this process. In the current implementation of the algorithm the non-
blocking receive function was used by the control and computational nodes for processing their respective message queues. This mechanism will be explained further in the next section of this chapter. Another feature that is supported by the RK/CE system is multicast communication. This refers to the delivery of the same message from a source node to an arbitrary number of destination nodes. Multicast is supported by the invocation of the function \texttt{xmsend()}: \[
\texttt{xmsend}(p, cnt, dest);
\]

where \texttt{p} points to the message being sent, \texttt{cnt} is an integer specifying the number of messages to be sent and \texttt{dest} is an array of \(2^{*\texttt{cnt}}\) integers of the form \{node1, pid1, node2, pid2, \ldots\}. In our case, \texttt{dest} was constructed after the spawning of the node processes from the host program. For example, in the current implementation of the algorithm, the initial value of the average work load is sent to all computational nodes using the \texttt{xmsend()} function:

\[
\texttt{xmsend}((\texttt{char} *)\texttt{plav},\texttt{icnt2},\texttt{dest});
\]

The parameter \texttt{plav} is a pointer to a structure holding the information for the average load while \texttt{icnt2} stores the value of \(N - 2\). It must be mentioned that the \texttt{xmsend()} function is implemented to exploit whatever efficient hardware may exist in a given system. Lacking any such hardware, it is implemented as a library function that performs the necessary copying and multiple calls to the \texttt{xsend()} function. This imposes limitations which effect running our parallel algorithm successfully with large number of computational processors. Recently, Li and Ni (1989) have developed faster multicast algorithms.
5.3 Parallel Algorithm and Implementation

In this section we present the overall structure of the parallel numerical integration routine and provide implementation details which are essential for the understanding of the algorithm. The structure of the algorithm, with the exception of the load balancing scheme, is based on the one given by de Doncker and Kapenga (1992).

The algorithm has been implemented on a range of MIMD-DM machines as well as on a network of workstations. Portability was insured by the use of the RK/CE system (Seitz et al., 1985) as a node communication system which is supported by a variety of loosely coupled systems.

A host program and three node programs were used in the current implementation of the algorithm. The host program runs on the host machine whereas the node programs run on the node processors of the concurrent system. Note that, depending on the application, the host-node approach is not the only approach possible, since node programs can be spawned directly into the nodes. As node programs we use: (a) a program running on node 0 of the machine which acts as the algorithmic controller, (b) a program designated as the mediator which runs on node 1 and which controls the load balancing scheme, and (c) a program that performs the actual integration. Copies of the latter run on the remaining nodes acquired by the user for the execution of the algorithm. Node 0 and node 1 are the logical names assigned to two computational node processors and their physical location within the machine is not important to our application programs. The entire code is written in
C which offers more flexibility over FORTRAN for implementing message-passing algorithms. Also C is generally supported by the vast majority of loosely coupled systems.

Figure 8 depicts a user’s point of view of the machine upon which the algorithm is executed. It indicates the different types of nodes as they relate to the routine, and the flow of control and data required by the algorithm. A meta-algorithm for global adaptive integration could be represented as:

```c
integrate () {
    initialize;
    while (the global acceptance criterion is not satisfied)
        partition();
}
```

The acceptance criterion is based on the size of the total estimated error and the tolerated error over the entire domain. The selection criterion for the partitioning of the next region (triangle) is determined according to a priority queue of the local error estimates. On a MIMD system the above algorithm, conceptualized as task partitioning algorithm (de Doncker and Kapenga, 1991), is mapped to each of a number of designated processes. The mapping of processes to processors is left unspecified even though in general we associate each process with a different processor (node). In loosely coupled systems where every node has its own memory and the nodes are connected by a message-passing communication network, the pool of regions is distributed over the local memories of the nodes. This is in contrast with the shared memory architecture where the pool is global to the participating processes and is accessed via monitor operations. The “generic” global adaptive
LEGEND

- exchange of information between control and computational nodes (initial broadcasting, update of results)
- transfer of tasks (triangles)
- problem parameters are sent and final results are received
- exchange of information between mediator and computational nodes

Figure 8. Flow of Data and Control.
integration algorithm, mentioned above, runs on all computational nodes involved in the adaptive partitioning of regions. One of the nodes, designated as the control, is responsible for global initializations, monitoring updates of global quantities from the computational nodes, testing termination criteria and flagging termination conditions to the computational processors. Since these operations can be performed asynchronously with region partitioning, the designation of a single control presents no serial bottleneck.

Before we go into more details about the algorithm, we give a pseudocode version of the routines running in parallel on the host processor, the control node (node 0) and the participating computational nodes. The pseudocode reflects one of the proposed load balancing schemes:

/*Host Program */
main() {
    input problem parameters;
    initialize the environment;
    spawn all node processes;
    broadcast problem parameters to all node processors;
    print results;
}

/*Control Program on Node 0 */
main() {
    initialize global variables;
    while (the global acceptance criterion is not satisfied and no abnormal termination conditions have occurred) {
        check the message queue and process the messages according to their type;
    }
    send final integral and error estimates to host;
    send stop signal to all other nodes;
}

/*Node Program on Nodes 2, 3, · · · , N − 1 */
main (){
    receive problem parameters from host;
    set local error tolerances;
    perform integration over the initial set of regions;
    send initial results and error estimates to control node (node 0);
    calculate initial load value;
    while (no abnormal termination occurs or a stop signal has not
        been received) {
        check message queue and process messages according to their type;
        if (local error estimate > local tolerance)
            for (i = 0; i < maximum number of regions per partitioning set; i ++){
                select region with maximum error from local pool;
                subdivide region and compute integral and error estimate over
                each subregion;
                insert new regions into local pool;
            }
        determine new load value;
        receive incoming load from one or more computational nodes and link
        new regions to local pool;
        update current load value;
        if (node is currently overloaded) {
            send regions to underloaded nodes, as instructed by the mediator;
            update current load value;
        }
        if (node is currently overloaded or underloaded)
            send message to mediator;
            if (load value has changed significantly)
                send new load value to control node;
        }
    if (abnormal conditions are met)
        send stop signal to control node;
}

The mediator algorithm and the newly developed load balancing schemes will
be described in the next section.

In most cases the host processor does not take part in the actual com­
putations due to the low speed of host-node communication, but acts as an
interface between the user and the parallel system and is used primarily for
I/O. A complication in the communication between host and node processors may arise due to the fact that their byte-order convention is not necessarily the same, since different processors use different conventions for packing and unpacking types other than character into byte streams. To ensure the portability of the code, RK/CE provides a set of functions for converting the host’s data representation into the multicomputer processors’ data representation and vice versa.

After the initializations of the global variables, the control node checks its message queue. This is achieved by means of the RK/CE non-blocking receive function (recv()). A message is processed upon arrival according to its type. This is implemented as a switch (C statement) on the message type value which is stored as the first element in the message structure. Initial contributions towards the evaluation of the integral and the estimation of the error are sent to node 0 in a message of initype. Later updates are sent in a message of type deltatype (de Doncker and Kapenga, 1992). We also incorporated a message of type initavtype for the calculation of the initial load average of the system as well as a message of type deltavtype for later updates of the value of the average load. When node 0 receives the last initype message, it calculates the global requested accuracy as: max{\(\varepsilon_a\), \(\varepsilon_r\)|result |} where \(\varepsilon_a\) and \(\varepsilon_r\) are the prespecified absolute and relative tolerance respectively. The initial error tolerance as well as its subsequent updates, if significant, are broadcast to all participating nodes via the RK/CE xmsend() command. The computational nodes execute a message polling loop similar to that of the...
control node. In Appendix II we give the C code of that loop. The local absolute error tolerance, used by each computational processor, is calculated by dividing the global absolute tolerance by the number of participating computational nodes. Other definitions are possible; for example, the local absolute tolerance could be calculated as \[ \varepsilon_a \sum \frac{(\text{areas of regions over a local pool})}{\text{area of initial integration region}} \]. The actual subdivision of each triangle and the integration as well as the error estimate over each subtriangle is carried as in the algorithm TRIEX (de Doncker and Robinson, 1984). Lyness and Jespersen (1975) formulas of degree 9 with 19 points and of degree 11 with 28 points are used as the basic integration rules. These are rules of relatively high degree which is achieved with a small number of points. They also have the desired properties that all nodes of the integration rule lie within the integrated triangle and all weights are positive. Also for the 4 triangles resulting from a subdivision step, six nodes of the 11 degree rule occur on the boundaries of the middle triangle and are shared among the 4 triangles without recalculation. A linked list data structure is used for the priority queue on the local pool of regions. Furthermore, only updates, i.e., differences between current and previous values are sent to the control node, so no actual integral and error values over the local triangle sets need to be communicated. This is done in order to allow triangles to move to other processors as a result of load balancing, so there is no need to keep track of where they are physically located. Before a computational node exits the main loop (while loop), it checks if, at that stage, the node is characterized as overloaded or underloaded.
This is achieved by comparing the difference (myload - averageload) with an overload threshold and comparing the difference (averageload - myload) with an underload threshold value. The threshold values are calculated as a fraction of the current value of the average load and thus change in value dynamically in the course of the computations. This idea is elaborated in the next section where the load balancing schemes are described.

5.4 Dynamic Load Balancing

There are numerous load balancing strategies which are either application independent or tailored to enhance the performance of an algorithm for solving a specific type of problem. The various strategies can be classified into different categories depending on the properties of their components. In this section we describe a load balancing scheme which uses aspects of centralized and distributed strategies for implementing its scheduling policy. Non-local information is used for the task transfer and task location policies. The first policy deals with the decision as to when a region (triangle) should be moved from one processor to another, while the other determines which processor(s) should send regions to other regions from other processors. The load balancing scheme incorporated into the parallel integrator is characterized as dynamic, so the decision making adapts with the current state of the system. The method combines the sender- and receiver-initiated aspects of a load balancing strategy. A specific node, called the mediator (node 1), is designated to store information indicating the overload of any node and the
work requests of any underloaded node. The mediator attempts to minimize the time that the computational processors will be idle during the computations, by controlling the flow of regions from overloaded to underloaded nodes.

To our knowledge, there are only two other routines for multivariate integration which incorporate any type of load balancing. De Doncker and Kapenga (1992) have developed a sender-initiated strategy for the load balancing. It is used by their parallel integration algorithm for integrating over a collection of triangles. Carino (1992) has used a receiver-initiated strategy for transferring regions among the processors of a multicomputer. The algorithm was designed for integration over a set of triangles and furthermore applies extrapolation by the use of the d-transformation. (Levin and Sidi, 1981). Load balancing is an essential component of any general purpose adaptive parallel integrator on distributed memory machines. In practice, integration regions where some type of irregular integrand behavior arises, may be assigned to a very limited number of processors. Therefore, to achieve a meaningful speedup on a DM machine, load balancing must be used to equally redistribute as much as possible the load among the participating computational nodes. For the load balancing, each processor must calculate its load. This is defined as the number of triangles with an error estimate that exceeds the maximum of the local tolerance and a fraction of the local estimated error. The following piece of code demonstrates the calculation of the initial load value on each computational processor.
/* Calculation of initial load after the first integration over
   the local pool of triangles */
lp = moll[1];
myload = 0;
for (i= 0; i< ind; i++){
    maxerr = moll [ l p + 1];
    if (elist[maxerr] <= dmaxl(epsa, 0.01*newabserr))
        break;
    myload++;
    lp = moll[lp];
}

The variable \texttt{ind} denotes the current number of triangles in the pool while \texttt{epsa} and \texttt{newabserr} are the local tolerance and the local estimated error respectively. The array \texttt{elist} holds the error estimates for each triangle in the local pool while \texttt{maxerr} is a pointer indicating which entry of \texttt{elist} holds the maximum error estimate. Every computational node after the first integration (single integration over the initial local pool) calculates the load and sends this information to node 0 (control node) to be used for the calculation of the current value of the average load of the system. The load value on each node is updated as needed (after subsequent integrations, receiving/sending triangles to/from other nodes). An alternative definition for the load may use the actual sizes of error present in the pool. Also the error distribution could be used to predict future loads. A model for that is given in de Doncker and Kapenga (1992).

A computational node is characterized as overloaded if the difference of its current load and the current value of the average load exceeds a threshold value. The latter is calculated as \( \max (\text{int}(FR* \text{averageload}), 6) \). The
value of $FR$ was set experimentally. Consequently the threshold value changes dynamically to reflect the fluctuations of the overall load. A similar criterion with a dynamic threshold was used to characterize a node as underloaded. Nodes classified in either category send and receive messages accordingly, to and from the mediator (node 1).

The process which controls the load balancing strategy is spawned by the host program on node 1 using the \texttt{spawn} function. The code was given in Section 5.2. Two priority queues which are implemented as linked lists are constructed on the mediator node. The first, called \textit{worklist} contains information sent form overloaded nodes, while the other called \textit{reqlist} stores requests for more work from underloaded nodes. In the remainder of the section we describe two load balancing strategies which allow regions to be sent from one to any other of the participating computational nodes. This is in contrast with the previously developed algorithm by de Doncker and Kapenga (1992), where regions flow from a processor to a specified set of processors (referred to as its load balancing "neighbors", although they do not actually need to be its physical neighbors).

Figure 9 depicts the communication patterns between the mediator and a set of participating nodes, for the first version of our load balancing scheme. In the sequel this will be referred as \textit{scheme #1}. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Request messages from underloaded nodes are implemented as C structures containing the requested amount and the identification number (node id) of the node that has issued the request. These messages are stored in the `reqlist` data structure as soon as they are received by the mediator node. Overloaded nodes send their excess load (triangles) to node 1 via work type messages.
The incoming triangles are stored in the *worklist* data structure (linked list) in descending order of their error estimates. Work messages are implemented as C structures and contain all necessary information on the triangles being transferred (estimate of the integral and error over each triangle, coordinates of the vertices, etc.) as well as the node number of the processor that has issued such a message. The mediator tries to "match" requests from underloaded nodes with tasks present in the *worklist* in order to minimize the time that nodes remain idle. Depending on the availability of work (triangles) in the mediator *worklist*, a load request may be fulfilled partially or completely. Regions in the local pools of the computational nodes are prioritized (as linked lists) with respect to their error estimates. In transferring load via a work type message from a node to the mediator, every other triangle, excluding the two with the biggest error, is sent to the mediator. Without this approach, an overloaded node could become underloaded in the immediate future, which would result in more message passing. Note that unnecessary message traffic may degrade the performance of the algorithm or lead to contention. The same transferring scheme has been used in passing triangles from the *worklist* (error keyed priority queue) of the mediator to the local pools of the underloaded nodes. In the systems where the algorithm has been implemented, the average time for sending a message between two nodes is a linear function of the size of the message (measured in bytes) (Seitz et al, 1988). In scheme #1, triangles are transferred via the mediator, in order to reach an underloaded node. Such work type messages are long and hence time consuming. We de-
developed a second approach where regions are moved directly from overloaded to underloaded nodes.

Figure 10 captures the new load balancing scheme. In this approach, an overloaded node sends a work load.
value type message to the mediator with the number of triangles that constitute the overload. An underloaded node sends request messages as in scheme #1. If messages of either type exist on the linked list structures of the mediator, they are replaced by subsequent ones, so that the more current load state of a node is reflected on node 1. Provided with load value messages on its worklist (indicating overloads), and load requests on reqlist, the mediator "matches" available tasks with work requests. In this way, successive overloads are passed out to satisfy successive requests. The last request dealt with may be partially satisfied. With this load balancing scheme, an underloaded node may be served directly by one or more overloaded nodes, while an overloaded node may transfer triangles to one or more underloaded nodes. Instruction type messages are sent to the overloaded nodes, containing the node identification numbers and the requested amounts from the "matched" underloaded nodes. Upon the creation of an instruction type message, the thus processed requests and load value type messages are erased from the corresponding queues on the mediator. Note that load requests may be partially satisfied also because the load of an overloaded node, at the time it processes the instruction type message, may have become less than that when was posted in the mediator worklist. Partially satisfied requests will be taken care of by subsequent load requests, if the node is still underloaded.

Comparing the two load balancing schemes outlined above in the examples of Section 5.5, higher speedups could be achieved by using scheme #2. Most of the test results obtained on the different types of loosely
coupled systems, as presented in the next section, are generated using the latter approach.

The mediator, as well as the other participating nodes, processes the different type of received messages in a message polling loop. This is implemented by means of a switch statement on the message type value stored as the first element in each message structure. Appendix C provides the complete C code of the mediator algorithm implementing the load balancing scheme #2.

5.5. Some Test Results

This section presents test results which demonstrate the speedup of the algorithm as it runs on an iPSC/2 hypercube and a Symult-2010. The two systems are at the Caltech Concurrent Supercomputing Facility. Table 9 lists the integrand functions used in the tests and provides a characterization for each one of them (Van Dooren and de Ridder, 1976; Lyness, 1976a). As the integration region, we have selected the unit square subdivided into a collection of non-overlapping triangles. This set provides the initial region of integration. The number of triangles, in the initial set is controlled (as input parameter) by the user. At initialization, the initial triangles are assigned to the computational processors in a round robin fashion. Apart from Problem #3, the behavior of the functions in Table 9 is such that the processor work load evolves differently through the course of the computations, so that the work is spread around adaptively via the proposed dynamic load balancing schemes.
### Table 9
Test Integrand Functions

<table>
<thead>
<tr>
<th>Integral #</th>
<th>Integrand Function</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\frac{1}{\sqrt{2}}$</td>
<td>Edge singularity along $y$-axis</td>
</tr>
<tr>
<td>2</td>
<td>$(x^2 + y^2)^{\frac{3}{2}} e^{-(2x^2+y^2)}$</td>
<td>Singularity at origin</td>
</tr>
<tr>
<td>3</td>
<td>$\cos(30 \pi x)\sin(60y)$</td>
<td>Oscillating</td>
</tr>
<tr>
<td>4</td>
<td>$\frac{[(x^2 + 0.0001)((y + 0.25)^2 + 0.001)]^{-1}}{4}$</td>
<td>Peak</td>
</tr>
<tr>
<td>5</td>
<td>$e^{</td>
<td>x+y-l</td>
</tr>
</tbody>
</table>

For all test results, on the different multicomputers, timings were performed in the host program starting before the problem information is broadcast to the nodes, but after the node processes are spawned, and ending after the final results have been received from node 0. For all runs, the maximum number of regions per partitioning step (given in Section 5.3) was set to 5.

Integrand function #1, was integrated over 8 triangles. The initial region of integration is given in Figure 11. The assignment of the original triangles to the processors was such that computational processor 1 (node 2 of the machine) receives the triangle with singularity along $x = 0$, which constitutes the “hot-spot” of the integration problem. Computational processor 2 is assigned a triangle with a point singularity at $x = 0$. For the integrator,
the latter is much easier to deal with than the line singularity.

![Figure 11. Sample Triangles and Processor Assignment With 2 Computational Processors.](image)

Test results were obtained on an 8-node iPSC/2 hypercube at California Institute of Technology. We use 2 and 6 computational nodes since node 0 acts as the control node and node 1 as the mediator. Table 10 shows the relation between the execution time and the number of participating computational nodes. We let the routine terminate after about 50,000, 100,000 and 200,000 integrand evaluations, imposing a very small error requirement (which cannot normally be achieved). For each case we ran the algorithm using the load balancing schemes #1 and #2 given in the previous section. All times are reported in seconds.
### Table 10

**iPSC/2 Host Times**

<table>
<thead>
<tr>
<th>Computational Nodes</th>
<th>Scheme #1 Time</th>
<th>50,000</th>
<th>100,000</th>
<th>200,000</th>
<th>50,000</th>
<th>100,000</th>
<th>200,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>16.10</td>
<td>36.2</td>
<td>65.4</td>
<td>13.1</td>
<td>29.5</td>
<td>52.3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>9.50</td>
<td>18.5</td>
<td>29.3</td>
<td>5.1</td>
<td>12.4</td>
<td>19.8</td>
<td></td>
</tr>
</tbody>
</table>

It can be seen that *scheme #2* is superior to *scheme #1*. This is due to the reduction in message volume for *scheme #2*, since the extra work (triangles) is transferred directly from overloaded to underloaded nodes. The message traffic with the mediator is restricted to short request and instruction type messages. Recall that *scheme #1* requires the extra load to be collected and re-distributed by the mediator node.

Table 11 provides timing results and Figure 12 the corresponding speed-up curves for the integrand function #1, as the algorithm runs on the Symult-2010. This machine has a mesh architecture, and the number of processors allotted does not need to be a power of 2. Note that the physical location of the reserved set of nodes in the overall 2-D mesh configuration of the Symult-2010 imposes no constraints on the successful execution of the algorithm. We let the routine terminate after about 200,000 integrand evaluations. The initial domain of integration was split up into 64 triangles. The speedup function...
s(NP) was calculated as \( s(NP) = \frac{T(1)}{T(NP)} \), where \( T(1) \) is the time spend with one computational processor and \( T(NP) \) is the corresponding time with \( NP \) computational processors. All times are measured in seconds.

Table 11
Symult-2010 Host Times and Speedup for Integrand Function #1

<table>
<thead>
<tr>
<th>Computational Nodes</th>
<th>Scheme #1 Time</th>
<th>Speedup</th>
<th>Scheme #2 Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>90.0</td>
<td>1</td>
<td>85.0</td>
<td>1.0</td>
</tr>
<tr>
<td>6</td>
<td>17.3</td>
<td>5.2</td>
<td>15.5</td>
<td>5.5</td>
</tr>
<tr>
<td>11</td>
<td>9.9</td>
<td>9.1</td>
<td>8.3</td>
<td>10.3</td>
</tr>
<tr>
<td>16</td>
<td>7.4</td>
<td>12.2</td>
<td>6.0</td>
<td>14.2</td>
</tr>
<tr>
<td>21</td>
<td>5.6</td>
<td>16.0</td>
<td>4.7</td>
<td>18.2</td>
</tr>
<tr>
<td>26</td>
<td>5.0</td>
<td>18.1</td>
<td>4.2</td>
<td>20.4</td>
</tr>
<tr>
<td>31</td>
<td>4.8</td>
<td>18.9</td>
<td>3.9</td>
<td>22.2</td>
</tr>
</tbody>
</table>

We noted again that Scheme #2 outperforms Scheme #1. If we let the number of participating computational nodes increase beyond 32, the execution time increases as we increase the number of participating nodes, as a result of contentional. Note that this example demonstrates one of the worst possible cases in terms of adaptive algorithm on a local memory machine. Load balancing is required to a major extent in order to achieve any speedup at all and put the parallel system in use.
Subdividing the original integration region into 8 triangles, we run the algorithm using integrand function #1, on the Symult-2010, with 1, 2, 4, 6 and 8 computational processors. Figure 13 depicts timings when the load balancing is disabled. It is evident that the work generated remains in mainly one computational node and has to be carried out by this processor sequentially.

Figure 12. Speedup Curves For Integrand Function #1 on Symult-2010.

Integrand function #2 exhibits a singularity at the origin. Using the same assignment of the initial regions as shown in Figure 11, computational processors 1 and 2 will be provided with troublesome triangles. Using a partition into 64 triangles and incorporating the load balancing Scheme #2 we let the routine perform 200,000 integrand evaluations. Table 12 and Figure 14 show the timings and the speedup curve for integrand function #2.
Figure 13. Symult-2010 Host Times Without Load Balancing.

Table 12

Symult-2010 Host Times and Speedup for Integrand Function #2

<table>
<thead>
<tr>
<th>Computational Nodes</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>80</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>14.5</td>
<td>5.5</td>
</tr>
<tr>
<td>11</td>
<td>7.5</td>
<td>10.6</td>
</tr>
<tr>
<td>16</td>
<td>5.4</td>
<td>14.9</td>
</tr>
<tr>
<td>21</td>
<td>4.3</td>
<td>18.8</td>
</tr>
<tr>
<td>26</td>
<td>3.9</td>
<td>20.6</td>
</tr>
<tr>
<td>31</td>
<td>3.6</td>
<td>22.5</td>
</tr>
</tbody>
</table>
Table 13 provides timing and speedup results while Figure 15 shows the relation between the execution time and the number of participating computational processors for problem #3 with an oscillating integrand behavior. Since the integrand function is uniformly behaved across the integration region, load balancing is not essential. Very light traffic is observed for the problem thus allowing the algorithm to achieve high speedups. As in previous test runs, the times are reported in seconds. The number of initial triangles was set to 64.
Table 13
Symult-2010 Host Times and Speedup for Integrand Function #3

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Computational Time</th>
<th>Speedup</th>
<th>Computational Time</th>
<th>Speedup</th>
<th>Computational Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>45.1</td>
<td>1</td>
<td>1</td>
<td>96.1</td>
<td>1</td>
<td>219.6</td>
</tr>
<tr>
<td>6</td>
<td>8.1</td>
<td>5.6</td>
<td>16.9</td>
<td>5.7</td>
<td>37.9</td>
<td>5.8</td>
</tr>
<tr>
<td>11</td>
<td>4.7</td>
<td>9.5</td>
<td>10.2</td>
<td>9.4</td>
<td>22.9</td>
<td>9.6</td>
</tr>
<tr>
<td>16</td>
<td>3.4</td>
<td>13.1</td>
<td>7.1</td>
<td>13.6</td>
<td>15.6</td>
<td>14.1</td>
</tr>
<tr>
<td>21</td>
<td>2.3</td>
<td>19.2</td>
<td>5.0</td>
<td>19.4</td>
<td>11.2</td>
<td>19.6</td>
</tr>
<tr>
<td>26</td>
<td>2.0</td>
<td>23.0</td>
<td>4.1</td>
<td>23.5</td>
<td>9.2</td>
<td>24.0</td>
</tr>
<tr>
<td>31</td>
<td>1.7</td>
<td>25.2</td>
<td>3.6</td>
<td>26.5</td>
<td>8.3</td>
<td>26.9</td>
</tr>
<tr>
<td>36</td>
<td>1.6</td>
<td>28.4</td>
<td>3.4</td>
<td>28.6</td>
<td>7.6</td>
<td>29.1</td>
</tr>
<tr>
<td>41</td>
<td>1.5</td>
<td>29.1</td>
<td>3.2</td>
<td>29.6</td>
<td>7.3</td>
<td>30.1</td>
</tr>
</tbody>
</table>

Figure 15. Symult-2010 Host Times For Oscillating Integral.
Table 14 gives timing results and Figure 16 the corresponding speedup curves for integrand function #4, as the algorithm runs on the Symult-2010. We let the routine terminate after about 200,000 integrand evaluations for both schemes of the load balancing strategy. The initial domain of integration was split up into 64 triangles.

Table 14
Symult-2010 Host Times and Speedup for Integrand Function #4

<table>
<thead>
<tr>
<th>Computational Nodes</th>
<th>Scheme #1</th>
<th>Scheme #2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Speedup</td>
</tr>
<tr>
<td>1</td>
<td>92.3</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>17.1</td>
<td>5.4</td>
</tr>
<tr>
<td>11</td>
<td>9.9</td>
<td>9.3</td>
</tr>
<tr>
<td>16</td>
<td>7.5</td>
<td>12.3</td>
</tr>
<tr>
<td>21</td>
<td>5.7</td>
<td>16.2</td>
</tr>
<tr>
<td>26</td>
<td>5.0</td>
<td>18.4</td>
</tr>
<tr>
<td>31</td>
<td>4.8</td>
<td>19.3</td>
</tr>
</tbody>
</table>

For integrand Function #5, we use a partition into 32 triangles and incorporate the load balancing Scheme #2. We let the routine perform 100,000 evaluations. Table 15 shows timing and speedup results for integrand Function #5. The times are reported in seconds.
Figure 16. Speedup Curves for Integrand Function #4 on Symult-2010.

Table 15

Symult-2010 Host Times and Speedup for Integrand Function #5

<table>
<thead>
<tr>
<th>Computational Nodes</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>39.5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>6.8</td>
<td>5.8</td>
</tr>
<tr>
<td>11</td>
<td>3.7</td>
<td>10.7</td>
</tr>
<tr>
<td>16</td>
<td>2.6</td>
<td>14.9</td>
</tr>
<tr>
<td>21</td>
<td>2.1</td>
<td>18.9</td>
</tr>
<tr>
<td>26</td>
<td>1.9</td>
<td>21.0</td>
</tr>
<tr>
<td>31</td>
<td>1.7</td>
<td>22.9</td>
</tr>
</tbody>
</table>
CHAPTER VI

SUMMARY AND FUTURE WORK

The first part of this chapter reviews the results obtained in this work. Section 6.2 explores some avenues for further research and provides ideas and suggestions for improvements of the algorithms given earlier.

6.1 Summary

In this work we have investigated the design, implementation and performance of global adaptive integration algorithms on parallel (MIMD) machines with shared or distributed memory. A speedup analysis for a class of global adaptive strategies on MIMD-SM machines has also been presented.

A parallel global adaptive algorithm for integration over the $N$-simplex has been implemented on a range of MIMD-SM machines. The portability of the algorithm is achieved through the use of the Argonne monitor macros. The macro package was employed as a bottom layer in the task pool management scheme and new primitives were developed to support a heap data structure on the pool of regions. Using a variety of integrand functions, test results were obtained which demonstrate a good speedup of the algorithm as it runs on an Alliant FX/8, a Sequent Symmetry and an Encore Multimax.

A theoretical derivation of the speedup for certain types of global adap-
tive integration algorithms on MIMD-SM machines was given in Chapter IV. This includes a study of the behavior of the adaptive integration process for specific types of integrands. Based on results by Rice (1974, 1975), the method has been extended to handle multivariate integration over the $N$-cube and the $N$-simplex of functions which may have some type of singular behavior. The analysis is valid for a region-size global adaptive strategy. Note that Rice's results apply to a local adaptive procedure.

A portable parallel multivariate integration algorithm on loosely coupled systems was given in Chapter V. The algorithm is characterized as global adaptive and is used for integration over a set of triangles. The portability is achieved by means of the Reactive Kernel/Cosmic Environment node communication system. Two new load balancing schemes were designed and compared. The new parallel integrator improves on the performance of its predecessor by de Doncker and Kapenga (1992). Test results were obtained on a iPSC/2 and a Symult-2010.

6.2. Future Work

Using monitors to "protect" an entire global data structure, causes that only one process at a time can modify the structure. A typical global adaptive numerical integration routine will perform at least twice as many insertions into the global task pool as deletions. This suggests that efficient data structures with respect to the insertion operation should be used. Recently, the
binary heap data structure has been used by several authors of numerical integration routines, due to the efficiency of the heap and the simplicity of its implementation. However, there are data structures with constant insertion time ($O(1)$). For example, the relaxed heap (Driscoll, Gabow, Shrairman and Tarjan, 1988) and the implicit binomial queue (Carlsson, Munro and Poblete, 1988) both achieve $O(1)$ insertion time. Unfortunately their implementation is considerably more complicated than that of the binary heap. It would be worthwhile to incorporate implementations of these (and of the alternative data structures mentioned in Chapter III) for use with region partitioning algorithms. That would enable the user to choose a suitable data structure for a specific problem. An endeavor of this nature would join with efforts which have been proposed in the framework of the construction of a parallel task partitioning system at the Concurrent Computation Research Center (CCRC) of the College of Arts and Sciences at Western Michigan University.

It is usually assumed that the most expensive part of a quadrature routine is the time spent on function evaluations. If the subdivision strategy provides each processor with useful work at each step, then good speedups can be achieved with respect to the function evaluations especially on MIMD-SM machines. However, if the priority queue management is not done effectively in parallel, this may affect the performance of the parallel algorithm considerably. Therefore, investigation on new data structures and their parallel manipulation are of interest in numerical integration on MIMD-SM architec-
tures. Rao and Kumar (1988) presented an algorithm that allows insertions into and deletions from a heap by more than one process at a time. They achieve a theoretical $O(\log N)$ speedup, where $N$ is the number of elements in the heap, but their implementation reveals a significant contention when more than 15 processors are used. Deo and Prasad (1990) constructed a parallel heap data structures which allows $\Theta(P)$ inserts and or deletes in $O(\log N)$ time, where $P$ is the number of processors accessing the heap and $N$ represents the number of the elements in the structures. Das and Horng (1991) have improved this algorithm, mainly with respect to space complexity. It may be mentioned that both of these studies were purely theoretical, and these algorithms have not been implemented at the time of their publication.

Further modifications and extensions of the algorithm presented for integration on distributed multiprocessors focus on various aspects that would improve the performance of the routine. Currently the local pool of regions on each processor is implemented as a linked list. The heap data structure variations are promising alternatives. Use of the Distributive Array Queue (Kapenga and de Doncker, 1988b) would result in a constant insertion and removal time. Furthermore, the role of the mediator node in the algorithm, could be given to the control node. This approach may be beneficial especially in the case of load balancing Scheme #2, where the mediator node is under-utilized in comparison with the computational processors. This is exemplified for problems where the function evaluations are time consuming. Alternatively
the control/mediator node could take part in the integration when it is idle.

To address the scaling problem (achieving good speedups with high number of processors), the given load balancing schemes can be extended by following a hierarchical approach. Such a scheme would group the computational nodes (involved in the integration) into clusters and designate a specific node, in each cluster, as the cluster mediator. Thus each cluster would resemble the configurations of Schemes #1 or #2 given in Chapter V. A two level mediation strategy would mediate processors within clusters at the first level, and then resolve imbalances between clusters at the second level. Some open issues involved in this type of approach are: (a) which data structures should be used in each mediator node; (b) what should be the formation of the clusters; (c) what type of messages should be sent among and within clusters.

Alternative definitions of load can be used in the algorithm. Furthermore, the load balancing can be based on predicted future loads derived from a model for the evolution of error distribution. One such theoretical model is given by de Doncker and Kapenga (1992). It would also be beneficial to spawn the node processes recursively. Since in the current implementation, node processes are spawned by the host program one at a time, there may be a considerable difference in the starting times of the first and the last spawned process.

A speedup analysis of global adaptive algorithms on DM machines using the proposed load balancing schemes remains for further research. The
analysis would resemble that of Chapter III, but has to take into account the load balancing strategy.

Another area of possible interest is the exploration of alternative integration strategies. A global staged adaptive strategy which has been successfully used in CUBEX (de Doncker and Kapenga, 1988), uses extrapolation to the limit by means of the $s$-algorithm. CUBEX performs numerical integration over the unit $N$-cube and is implemented on MIMD-SM machines. Exploring the incorporation of an extrapolation procedure to a parallel integrator on MIMD-DM machines would enhance the performance of the algorithm and expand its applicability to a variety of integrands with singularities. Investigating possible methods for manipulating the stages as they evolve throughout the computation, and the maintenance of the local pools with regions of different stages leads to further research.

For parallel integration routines, on shared or distributive memory, the method of double adaptivity could also be explored. Using this approach, the algorithm can make a choice, based on arguments of local smoothness of the integrand, to either subdivide the region(s) into two (or more) subregions or to integrate over the same region with a formula of higher degree. A similar method which uses extrapolation (as opposed to higher degree rules) was developed by Cariño (1992). The incorporation of alternative subdivision strategies, integration rules (for higher dimensional integration regions), and more reliable error estimates may also be beneficial.
To our knowledge there are no adaptive multivariate integration routines implemented on Single Instruction Multiple Data (SIMD) machines (which use more parallelism than of a parallel evaluation of the function values). For example, the Connection Machine is such an architecture. An investigation of the feasibility of parallel adaptive strategies on these machines would be of interest.

As a last note, a natural extension of this work would be in the design of parallel algorithms for integration over a collection of higher dimensional regions.
Appendix A

List of User-Defined Macros
C List of user-defined macros for maintenance of the heap data structure
C and update of the global variables.

C Written: March 1991
C Last modified: May 1992

changequote((,)) dnl
define(HEAPDEC,
  \{INTEGER $1$IMAX, $1$MIN
  INTEGER I, J
  REAL $1$ERROR, $1$ERROR
  REAL $1$VALUE, $1$VALUE
  REAL $1$GREAT, $1$GREAT
  REAL $1$VOLUME, $1$VOLUME
  REAL $1$VERTEX(NDIM,0:NDIM),$1$VERTEX(NDIM,0:NDIM)
C end Macro-heapdec)
define(HEAPBOUNDS,
  \{$1$IMAX = $2
  $1$MIN = $3\}
C end Macro-heapbounds)
define(GETSPACE,
  \{MEXIT($1)
  IF (1 + $2.GT.$1IMAX) THEN
    $3 = 1
  ELSE
    $3 = 0
  ENDIF
  MEXIT($1)}
C end Macro-getspace)
define(ADDETHREE,
  \{IF(((\$2.LT.$1IMAX).AND.($2.GT.0)).OR.(INTGRFLAG.EQ.1)) THEN
    IF (INTGRFLAG.EQ.1) THEN
      CALL ADDREHEAP($1,$2+1,$1$ERROR,$1$VALUE,$1$VERTEX,$1$GREAT,$1$VOLUME)
      CALL ADDREHEAP($1,$2+2,$1$ERROR,$1$VALUE,$1$VERTEX,$1$GREAT,$1$VOLUME)
      $2 = $2 + 2
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
ENDIF
IF($2.GT.0) THEN
  $3 = 0
  CALL DELREALHEAP($1,$2,$1IMAX)
  LOCSBR = $1
  CALL SHALVCOPLOC(IP,LOCVERTCS,LOCVOLUMS,ENDA,ENDB)
ELSE
  $3 = -1
ENDIF
C end Macro-addgetomark)

define(ADDTOTEMP,
{IF($1.LT.MOIMAX) THEN
  $3 = 0
  $4 = LOCERRORS($2)
  $5 = LOCVALUES($2)
  IF($3.LT.4) $3 = $4
  $6 = LOCVOLUMS($2)
  DO $9 J = 0,NDIM
  DO $9 I = 1,NDIM
    $7(I,J) = LOCVERTCS(I,J,$2)
  $9 CONTINUE
ENDIF}
C end Macro-addtotemmark)

cchangequote([,]) dnl
define(UPDGVAR,
{MENTER($1)
  VALUE = VALUE + LOCVALUES(LOCSBR+1) + LOCVALUES(LOCSBR+2)
  ERROR = ERROR + LOCERRORS(LOCSBR+1) + LOCERRORS(LOCSBR+2)
  MEXIT($1)}
C end Macro-updgvar)
Appendix B

Message Polling Loop on a Computational Node
/ * Message polling loop as executed by each computational processor
 * to check its message queue. It has been implemented by means of a
 * C switch statement on the message type value which was stored as
 * the first element in each message structure.
 */

/* check the message queue */

while(msgptr = (unsigned long *) xrecv()) {  /* msg is present */
    switch(msgptr) {
        case stotype:  /* termination msg */
            ps = (struct msgtyp7 *) msgptr;
            pt_term = 1;
            xfree((char *)msgptr);
            break;  /* out of switch */
        case totype:   /* tolerance update from node 0 */
            toflag = 1;
            pt = (struct msgtyp3 *) msgptr;
            if(pt->term)
                pt_term = pt->term;
            else
                epsa = pt->tol/(double)icont2;
            xfree((char *)msgptr);
            break;  /* out of switch */
        case loadavertype:  /* load average update from node0 */
            plav = (struct msgtyp9 *) msgptr;
            loadavflag = 1;
            average_load = plav-> averload;
            xfree((char *)msgptr);
            break;  /* out of switch */
        case sendtoonestype:  /* receive from mediator the node id and the */
                              /* load amount to be send to the underload node */
            psonet = (struct msgtyp10 *) msgptr;
            soneflag = 1;
            numtri = psonet->numtriangles;
            destnode = psonet->node;
            xfree((char *)msgptr);
            break;  /* out of switch */
        case sendtomanytype: /* receive from mediator the node ids and the */
/* load amounts to be send to each underload node*/
psmany = (struct msgtyp6 *) msgptr;
smanyflag = 1;
totalamount = 0;
for (i = 0; i < maxnumber; i++)
destnodes[i] = 0;
for (i = 0; i < icant; i++)
lowamount[i] = 0;
for (i = 0; i < psmany->nlowloadnodes; i++) { /* store in an array the */
    /* the node ids */
destnodes[i] = psmany->lowloadnode[i];
}
for (i = 0; i < icant; i++) { /* calculate the total load amount*/
    /* needed and store in an array the */
    /* requested amount from each underload node*/
totalamount = totalamount + psmany->loadamount[i];
lowamount[i] = psmany->loadamount[i];
}
nnumlovenodes = psmany->nlowloadnodes;
xfree((char *) msgptr);
break; /* out of switch */

case worktype: /* work from computational nodes */
    if (i++) { /* next node in the work chain */
pwork->next = (struct msgtyp6 *) msgptr;
pwork = pwork->next;
    }
else { /* first node in the work list */
worklist = (struct msgtyp6 *) msgptr;
pwork = worklist;
    }
break; /* out of switch */

} /* of while(msgptr =/
Appendix C

The Mediator Program
/*
 * The Mediator Program
 */

/*
 * This code runs on the mediator node (node 1) in parallel with the
 * host and the control program on a loosely coupled multicomputer.
 * It implements the approach #2 of the load balancing scheme as
 * presented in chapter 5.
 */

/*
 * Last modified: June 7, 1992.
 */

#include <sys/time.h>
#include <cube/cubedef.h>
#include <stdio.h>
#include "math.h"
#include "defs.h"
#define APID 0
#define NULL 0

/* define the work queue link list on the mediator node */
struct work_element {
    int nodeid;
    int overload;
    struct work_element *next;
};

/* define the request link list on the mediator node */
struct req_element {
    int nodeid;
    int underload;
    struct req_element *next;
};

/* declare variables used in this program */
unsigned long *msgptr; /* generic pointer */
int nodefound,
    insflag,
    loopbreak,
    totworload,
    totrqload,
i,
icnt,
id,
flaglwnd,
tmp;
/* pointers to the structures */
struct work_element *pwel;
struct work_element *tempwk;
struct work_element *startworkvolumelist;
struct work_element *prevwel;
struct req_element *prel;
struct req_element *temprel;
struct req_element *startreqlist;
struct req_element *prevrel;
struct msgtyp7 *psmed;
struct msgtyp7 *ps;
struct msgtyp10 *poverload;
struct msgtyp10 *punderload;
struct msgtyp10 *psone;
struct msgtyp11 *psmany;

main() {
    id = mynode();
cosmic_init(id,APID);
icnt = nnodes();

    /* initialize the pointers to link list to NULL */
    startworkvolumelist = NULL;
    startreqlist = NULL;

    /*initialize loopbreak variable */
    loopbreak = 0;

    /* Check the queue for messages from the nodes using a message
    * polling loop. An overload node will send a work volume type message
    * while an underload node will send a request type message.
    */
    while(1) {
        if(msgptr = (unsigned long *) xrecv()) {
            switch(msgptr) {
                case workvolumetype:
                    /* Process a work volume type message */
                    /* and link it to the work queue */
                    poverload = (struct msgtyp10 *) msgptr;
                    /* get space for the new incoming work node */
                    tempwk = (struct work_element *) malloc(sizeof(struct work_element));
            }
        }
    }
}
tempwk->nodeid = overload->node;  
tempwk->overload = overload->numtriangles;  
tempwk->next = NULL;  
if(startworkvolumelist == NULL) { /*if empty list, just insert*/  
    startworkvolumelist = tempwk;  
    pwe1 = startworkvolumelist;  
} else { /*delete node from work list with*/  
    if(startworkvolumelist == NULL) { /*same nodeid as incoming message.*/  
        pwe1 = NULL;  
        pwe1 = startworkvolumelist;  
        nodefound = 0;  /*flag to signal if the node exist (or not)*/  
        while(nodefound != 1) {  
            if(pwe1 == NULL)  
                nodefound = 1;  
            else if(tempwk->nodeid == pwe1->nodeid)  
                nodefound = 1;  
            else {  
                prevwe1 = pwe1;  
                pwe1 = pwe1->next;  
            }  
        }  
        if(pwe1 != NULL) {  
            if(prevwe1 == NULL) /*check for deletion at the top of list*/  
                startworkvolumelist = pwe1->next;  
            else {  
                prevwe1->next = pwe1->next;  
                xfree((char *) pwe1);  
            }  
        }  
    }  
}  
/* Code for inserting the new work volume type message in the  
* correct place in the work list. The list is maintained in descending  
* order of the magnitude of the overload.  
*/  
prevwe1 = NULL;  
pwe1 = startworkvolumelist;  
insflag = 0;  
while(insflag != 1) {  
    if(pwe1 == NULL)  
        insflag = 1;  
    else if(tempwk->overload > pwe1->overload)  
        insflag = 1;  
    else {  
        prevwe1 = pwe1;  
        pwe1 = pwe1->next;  
    }
if(prevwel == NULL) { /* test for insertion at top of the list */
    tempwrk->next = startworkvolumelist;
    startworkvolumelist = tempwrk;
} else {
    tempwrk->next = pwe1;
    prevwel->next = tempwrk;
}
/*end of else*/
xfree((char *) tempwrk);
break;

case requesttype: /* Process a request type message */
    /* and link it to the request queue */
    punderload = (struct msgtyp10 *) msgptr;
    tempreq = (struct req_element *) xmalloc(sizeof(struct req_element));
    tempreq->nodeid = punderload->node;
    tempreq->underload = punderload->numtriangles;
    tempreq->next = NULL;
    if(startreqlist == NULL) { /* if request list empty, just insert */
        startreqlist = tempreq;
        pr el = startreqlist;
    } else { /* delete possible existing node with same id, then insert*/
        prevrel = NULL;
        pr el = startreqlist;
        nodefound = 0;
        while(nodefound != 1) {
            if(prevrel == NULL)
                nodefound = 1;
            else if(tempreq->nodeid == pr el->nodeid)
                nodefound = 1;
            else {
                prevrel = pr el;
                pr el = pr el->next;
            }
        }
        if(prevrel != NULL) { /* check the top of the list */
            if(prevrel == NULL)
                startreqlist = pr el->next;
            else {
                prevrel->next = pr el->next;
                xfree((char *) pr el);
            }
        }
    }
}  
*/
 /* Code to insert (in right place) the new arrived request from
 * an underloaded node.
 */

 prevrel = NULL;
 prel = startreqlist;
 insflag = 0;
 while(insflag != 1) {
   if(prel == NULL)
     insflag = 1;
   else if(tempreq->underload > prel->underload)
     insflag = 1;
   else {
     prevrel = prel;
     prel = prel->next;
   }
 }

 if(prevrel == NULL) { /* test for insertion at top of the list */
   tempreq = startreqlist;
   startreqlist = tempreq;
 }
 else {
   tempreq->next = prel;
   prevrel->next = tempreq;
 }

 xfree((char *) tempreq);
 break;

 case stotype : /* if a stop signal is received from node 0 */
   /* execution terminates. */
   psmem = (struct msgtyp? *) msgptr;
   loopbreak = 1;
   break;
 }  /* end of switch */
 xfree((char *) msgptr);
 if(loopbreak) break; /* stops the while(1) loop, exits program */

} /* end of if(msgptr .... */
if((startworkvolumelist != NULL) && (startreqlist != NULL)) {
  /*
   * case where a request will be "satisfied" with more than one messages of
   * work volume type.
   */
  if(startreqlist->underload >= startworkvolumelist->overload) {
    prel = startreqlist;
    loopbreak = 0;
    totworkload = startworkvolumelist->overload;
    while((startreqlist->underload > totworkload)
      && (startworkvolumelist != NULL)) {
      pwel = startworkvolumelist;
      psone = (struct msgtyp10 *) xmalloc(sizeof(struct msgtyp10));
      psone->type = sendtoonestype;
      psone->node = startreqlist->nodeid;
      psone->numtriangles = pwel->overload;
      xsend((char *)psone, pwel->nodeid, APIID);
      if(pwel != NULL) {
        startworkvolumelist = pwel->next;
        xfree((char *) pwel);
      } else loopbreak = 1;
      if(loopbreak == 1)
        break;  /* breaks out of this local while loop */
    if(startworkvolumelist != NULL)
      totworkload = totworkload + startworklist->overload;
   } /* end of while((startreqlist..... */
    startreqlist = prel->next;
    xfree((char *) prel);
  } /* end of if(startreqlist->underload ... */

  /*
   * case where a work volume type message tries to "satisfy" many requests
   * from underloaded nodes.
   */
  else if(startworkvolumelist->overload > startreqlist->underload) {
    flaglownd = 0;
    loopbreak = 0;
    pwel = startworkvolumelist;
    totreqload = startreqlist->underload;
    psmany = (struct msgtyp11 *) xmalloc(sizeof(struct msgtyp11));
    for(i=0;i<maxregperstep;i++)
      psmany->lowloadnode[i] = 0;
    for(i=0;i<cnt;i++)
      printf("%d %d", psmany->lowloadnode[i], psmany->lowloadnode[i]);
  }
}
psmany->loadamount[i]=0;
psmany->nlowloadnodes =0;
while((startworkvolumelist->overload >= totreqload)
  && (startreqlist != NULL)) {
  prel = startreqlist;
  psmany->nlowloadnodes = psmany->nlowloadnodes + 1;
  if(psmany->nlowloadnodes <= maxregperstep) {
    tmp = psmany->nlowloadnodes -1;
    psmany->nlowloadnode[tmp] = startreqlist->nodeid;
    psmany->loadamount[startreqlist->nodeid] = startreqlist->underload;
  }
  else {
    flagnond = 1;
    break;
  }
  if(prel !=NULL) {
    startreqlist = prel->next;
    xfree((char *) prel);
  }
  else loopbreak = 1;
  if (loopbreak == 1)
    break; /*breaks out of the local while loop*/
  if(startreqlist != NULL)
    totreqload = totreqload + startreqlist->underload;
}
if(flagnond == 1) {
  ps = (struct msgtyp7 *) xmalloc(sizeof(struct msgtyp7));
  ps->type = stotype;
  ps->node = id;
  ps->ictnt = 50; /* special error code flag */
  xsend((char *)ps,0, APID);
  xfree((char *) psmany);
  break; /* break out of the while(1) loop */
}
  xsend((char *)psmany, pwe1->nodeid, APID);
startworkvolumelist = pwe1->next;
xfree((char *) pwe1);
} /* end of if((startworkvolumelist != NULL & & ... */
} /* end of the while(1) loop */
BIBLIOGRAPHY


137

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


