Efficiency of Distributed Priority Queues in Parallel Adaptive Integration

Rodger Zanny
Western Michigan University

Follow this and additional works at: https://scholarworks.wmich.edu/masters_theses
Part of the Computer Sciences Commons

Recommended Citation

This Masters Thesis-Open Access is brought to you for free and open access by the Graduate College at ScholarWorks at WMU. It has been accepted for inclusion in Master's Theses by an authorized administrator of ScholarWorks at WMU. For more information, please contact maira.bundza@wmich.edu.
EFFICIENCY OF DISTRIBUTED PRIORITY QUEUES IN PARALLEL
ADAPTIVE INTEGRATION

by

Rodger Zanny

A Thesis
Submitted to the
Faculty of The Graduate College
in partial fulfillment of the
requirements for the
Degree of Master of Science
Department of Computer Science

Western Michigan University
Kalamazoo, Michigan
April 1999
ACKNOWLEDGEMENTS

In no order of importance, I would like to thank the people that have helped me, directly or indirectly, with this thesis.

I would like to express my appreciation to my thesis committee. Dr. Ajay Gupta, who found time to help me out despite his many new responsibilities, Dr. Elise de Doncker, who put up with my apparent inability to understand anything that might have come from a math textbook, and Dr. Don Nelson, who took time out from his sabbatical to help with my thesis. In addition to helping me with my thesis, they have guided me through the beginning of the process of becoming a computer science researcher.

I must thank Sue Moorian for her help on many administrative snafus, and Ken Thies and Orlen Vandenbergh for their help in solving many system related problems that cropped up while working on PARINT these last few years.

Without the help of all the members of the PARINT research group, past and present, the research environment that made this thesis possible would not have existed. And as always, credit is due to all of the scientists upon whose work this thesis is based.

I must thank my parents for raising me as they did and for instilling in me a sense of curiosity.

And I of course must thank my wife Brenda, who put up with my many
late nights, my swings between excited optimism and basement level pessimism concerning my thesis, and my continual habit of working on my thesis while finding and listening to some new source of music that surely must have seemed designed just to annoy her.

Rodger Zanny
EFFICIENCY OF DISTRIBUTED PRIORITY QUEUES IN PARALLEL ADAPTIVE INTEGRATION

Rodger Zanny, M.S.

Western Michigan University, 1999

The adaptive integration algorithm is effective in numerically solving integration problems. It is able to focus the application of integration rules on the portion of the integration region where the integrand is the least well-behaved. Parallel implementations must use dynamic load balancing or performance suffers.

Dynamic local load-balancing techniques allow each processor to maintain its own pool of work in a local priority queue and balance the workload based on local criteria. However, the use of locally controlled priority queues is known to be inefficient (in terms of the number of integration rule applications needed to reach an answer) as the number of processors increases.

This thesis implements and analyzes the use of a distributed priority queue for managing the current pool of work and performing global load balancing in the parallel adaptive integration algorithm. Initial experimental results show that the use of a distributed priority queue does not provide any clear benefits over the use of multiple local priority queues. However, analyzing the circumstances where the results are better has led to greater understanding of the basic algorithm.
TABLE OF CONTENTS

ACKNOWLEDGEMENTS .................................................. ii
LIST OF TABLES ......................................................... x
LIST OF FIGURES ......................................................... xi

CHAPTER

I. INTRODUCTION ..................................................... 1
   Introduction to the Research ..................................... 1
   Thesis Contents .................................................. 4
   Thesis Composition .............................................. 4

II. PROBLEM TERMINOLOGY ........................................... 5
   Single Integrand Functions ..................................... 5
   Vector Integrand Functions ................................... 6

III. INTRODUCTION TO THE META-ALGORITHM ....................... 8
   Integration Rules ............................................... 8
   Sequential Algorithm .......................................... 9
      Non-adaptive Integration .................................... 9
      Adaptive Integration ....................................... 10
      Splitting the Regions ...................................... 11

IV. LOCAL HEAP ALGORITHM .......................................... 13
TABLE OF CONTENTS – Continued

CHAPTER

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel Processing Model</td>
<td>13</td>
</tr>
<tr>
<td>Local Heap Version</td>
<td>15</td>
</tr>
<tr>
<td>Controller as Worker</td>
<td>18</td>
</tr>
<tr>
<td>Priority Queues</td>
<td>19</td>
</tr>
<tr>
<td>Load Balancing</td>
<td>19</td>
</tr>
<tr>
<td>PARINT Environment</td>
<td>21</td>
</tr>
<tr>
<td>Recent Modifications</td>
<td>21</td>
</tr>
<tr>
<td>Round-Robin Idle Worker Selection</td>
<td>22</td>
</tr>
<tr>
<td>Handling of Local Heap During Load Balancing</td>
<td>22</td>
</tr>
<tr>
<td>V. INEFFICIENCIES IN LLB VERSION</td>
<td>24</td>
</tr>
<tr>
<td>Efficiency in Terms of Regions Evaluated</td>
<td>24</td>
</tr>
<tr>
<td>Function Evaluations vs. Region Evaluations</td>
<td>26</td>
</tr>
<tr>
<td>Evaluated Regions Viewed as a Binary Tree</td>
<td>27</td>
</tr>
<tr>
<td>Regions Evaluated by the Sequential Algorithm</td>
<td>28</td>
</tr>
<tr>
<td>Assumptions in the Adaptive Integration Algorithm</td>
<td>29</td>
</tr>
<tr>
<td>Regions Evaluated by the Parallel Algorithm</td>
<td>30</td>
</tr>
<tr>
<td>Experimental Environment</td>
<td>32</td>
</tr>
<tr>
<td>Initial Efficiency Results</td>
<td>34</td>
</tr>
<tr>
<td>Other Sources of Lost Efficiency</td>
<td>36</td>
</tr>
</tbody>
</table>
# Table of Contents – Continued

## CHAPTER

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Braking Loss</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>Iteration Loss</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>Reducing Inefficiencies</td>
<td>38</td>
</tr>
<tr>
<td>VI.</td>
<td>INTRODUCTION TO THE GLOBAL HEAP</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>Motivation for a Global Heap</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>Needed Operations</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>Simple One-Node Global Heap</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>Multi-Node Global Heap</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>Overview of Storing Distributed Data</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>Previous PARINT Global Heap Research</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>Horizontally Sliced Global Heap Design</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>Current Related Research</td>
<td>47</td>
</tr>
<tr>
<td>VII.</td>
<td>CURRENT DESIGN FOR THE GLOBAL HEAP</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>Design Overview</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>Design of the GHC</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>Global Heap Adjustments</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>Pattern of Errors Over Time</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>Initial Handling of Heap</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>Single-Node Version</td>
<td>59</td>
</tr>
</tbody>
</table>
# Table of Contents - Continued

## CHAPTER

Details of the Global Heap Algorithm .................................. 59

Message Types for the Global Heap .................................. 60

State Flow for the GHW ........................................... 61

State Flow for the GHC ........................................... 63

Integration of the Global Heap With PARINT ......................... 66

### VIII. GLOBAL HEAP IMPLEMENTATION DETAILS ..................... 68

Overview of the Implementation .................................... 68

Terminology Used in This Chapter ................................ 69

Rationale for the Single Pass Message Loop ......................... 70

Process Spawning Drawbacks ........................................ 70

Control Over Interleaving of Execution ............................. 71

Speed of Message Passing ......................................... 71

Terminology Used Within the Pseudocode ........................... 73

Single Pass Message Loop ......................................... 74

Simple Example .................................................... 75

Mixing Message Receiving With Local Work ....................... 76

State Information ................................................ 80

Multiple Functionalities in a Single Process ....................... 81

Functionality States and Handling of Preferred Messages ....... 84
CHAPETR

Use of a SPML in the LLB Algorithm ............. 86

IX. EXPERIMENTAL RESULTS ...................... 89

Hypothesis .......................................... 89

Initial Results ...................................... 89

Poor Results ....................................... 90

Explanation of Initial Results .................... 92

Trail Ends of the Error Curve ..................... 99

Lower Error Tolerances ............................ 103

Revisiting the Hypothesis .......................... 107

Non-Ideal Regions From the Global Heap ........ 109

Effects of Singularities on Region Retrieval .... 111

Domino Effect of Evaluating Incorrect Regions .. 112

Analysis of the Domino Effect .................... 113

Benefit of the Global Heap ........................ 114

Lowering the Function Count Limit ............... 114

Initial Timing Results ............................. 115

Expected Timing Results ........................... 116

Timing and Efficiency Results .................... 117

Multi-Node Global Heap ............................ 118
Table of Contents – Continued

CHAPTER

X. FUTURE WORK .............................................. 120

Improvements to the Global Heap ..................... 120

Modify the Number of Regions Per Request ........ 120

Increase Work Between Requests ..................... 121

Increase Work Between Updates ...................... 122

Continual Working ......................................... 123

Dynamic Allocation of Arrays ......................... 123

Attempt to Keep Regions Local in the Global Heap .. 125

Increasing the Global Heap Size ...................... 126

Optimizing Inter-Heap-Node Message Sizes .......... 127

Future Research Directions .......................... 128

XI. CONCLUSIONS ........................................... 130

BIBLIOGRAPHY .............................................. 130
LIST OF TABLES

1. Integrals Used for Testing ........................................ 33
LIST OF FIGURES

1. Schematic of a Simple Integral ........................................... 2
2. Adaptive Integration Algorithm ............................................. 11
3. Local Heap IW Algorithm .................................................... 17
4. Local Heap IC Algorithm .................................................... 17
5. Efficiency (in # Regions) for All Functions ............................ 35
6. State Diagram for a Non-leaf GHW ......................................... 62
7. State Diagram for the GHC .................................................. 65
8. Simple Message Loop ......................................................... 75
9. Initial Single Pass Message Loop ......................................... 76
10. Single Pass Message Loop With Local Work ............................ 79
13. GHEAP vs. LLB Data for fcn1 ............................................... 90
14. GHEAP vs. LLB Data for fcn2 ............................................... 91
15. GHEAP vs. LLB Data for fcn4 ............................................... 92
16. GHEAP vs. LLB Data for fcn7 ............................................... 93
17. GHEAP vs. LLB Data for fcn10 ............................................. 94
18. GHEAP vs. LLB Data for fcn14 ............................................. 95
List of Figures - Continued

19. GHEAP vs. LLB Data for \textit{fcn18} .............................................. 96
20. GHEAP vs. LLB Data for \textit{fcn19} .............................................. 97
21. GHEAP vs. LLB Data for \textit{fcn21} .............................................. 98
22. GHEAP vs. LLB Data for \textit{fcn22} .............................................. 99
23. Single Run Error Ratio Curve for \textit{fcn22} and $p = 4$ ......................... 100
24. Single Run Error Ratio Curve for \textit{fcn22} and $p = 16$ ......................... 101
25. Single Run Error Ratio Curve for \textit{fcn22} and $p = 24$ ......................... 102
26. Single Run Error Ratio Curve for \textit{fcn18} and $p = 4$ ......................... 103
27. Single Run Error Ratio Curve for \textit{fcn18} and $p = 16$ ......................... 104
28. Single Run Error Ratio Curve for \textit{fcn18} and $p = 24$ ......................... 105
29. Single Run Error Ratio Curve for \textit{fcn22}, $p = 16$, and $\varepsilon_r = 10^{-6}$ 106
30. GHEAP vs. LLB Data for \textit{fcn7}; Multiple $\varepsilon_r$ Values .................. 108
31. GHEAP vs. LLB Data for \textit{fcn18}; Multiple $\varepsilon_r$ Values .................. 109
32. GHEAP vs. LLB Data for \textit{fcn22}; Multiple $\varepsilon_r$ Values .................. 110
33. GHEAP vs. LLB Timing Data for \textit{fcn22} ........................................ 118
34. GHEAP vs. LLB Efficiency Data for \textit{fcn22} ........................................ 119
CHAPTER I

INTRODUCTION

Integration is a fundamental problem in mathematics, and the efficient solving of integral problems using computers is an important problem in computer science.

Given some function $f(x)$ (called the integrand) and two $x$ coordinates $a$ and $b$, with $a \leq b$, the integral of that function over the region defined by $a$ and $b$ can be intuitively understood as the area bounded by the function, the $x$-axis, and the boundaries $a$ and $b$, as shown in Figure 1. In three dimensions, the boundaries are defined by a region in the $xy$-plane, the function $f(x, y)$ is a surface, and the integral is the bounded volume. The problem definition is analogous through any number of dimensions (in which case the function variables are specified as a vector $\vec{x}$).

Introduction to the Research

Some integrands can be integrated symbolically, i.e., they can be analyzed and a closed-form equation of the integral result can be derived. This closed-form equation can be numerically evaluated to provide a quick and accurate answer to
Figure 1. Schematic of a Simple Integral.

Other integrals are not able to be symbolically integrated and numerical techniques must be used to evaluate the integral.

Generally, the integral method is viewed as a black-box. While there are certain numerical techniques that work especially well with certain kinds of integrand functions (singular, peak, oscillatory, etc.), a general approach will make few assumptions about the function to be integrated. The only thing known to the computer about the function is how to compute the function's value for each applied input.

To solve the integration problem over a particular region numerically, the function is evaluated many times within that region, for particular values of \( \tilde{x} \), and the results are used to provide an integral answer and an estimate of the error.
in the result. This functionality is provided by an integration rule. A rule can be applied just once or many times (with combined results) to solve a particular integration problem. (For further information on symbolic and numerical integration, the reader is referred to [25, 38].)

This method is very computationally intensive. As with many such problems, recent research has focussed on deriving numerical integration techniques that can be parallelized. I.e., multiple processing elements (CPU's, workstations, etc.) are utilized together to reach a suitable answer faster than a single processing element could by itself.

And as is the case with many applications, once parallel techniques are investigated, many interesting problems arise that need to be solved in order to have an efficient algorithm. One problem that arises in the area of parallel numerical integration (again, as in many other problems) is that as the number of processing elements used increases the amount of work (in terms of integration rule applications) that must be done to solve the problem increases. This increase in work is considered an inefficiency, and it slows down the algorithm.

A research goal is therefore to devise parallel techniques that reduce these inefficiencies. This thesis focuses on such a technique, developed within the context of the PARINT parallel numerical integration research project.
Thesis Contents

Chapter II defines the terminology used to specify the integration problems that will be solved. An introduction to current parallel techniques appears in Chapter III and Chapter IV. Chapter V discusses current problems with general and specific parallel techniques, and proposes the solution which forms the heart of this thesis. The background of this solution is provided in Chapter VI, while the current design is discussed in general in Chapter VII. The details of the implementation, within the context of a very general implementation technique (which can be applied to many algorithms), are in Chapter VIII. The experimental results comparing the old and new techniques appear in Chapter IX, with Chapter X discussing future research directions, and Chapter XI summarizing and concluding the thesis.

Thesis Composition

This thesis was prepared under the formatting guidelines [37] required of theses and dissertations prepared at Western Michigan University.

Formatting was performed using \LaTeX{} (version \LaTeX{} 2.0) [29, 21] using \TeX{} templates written by Jay Ball for his thesis [3]. All editing of the thesis and code was done in Emacs. Figures and graphs were completed on a PC using Visio Express and Microsoft Excel and transferred and included as postscript files.
CHAPTER II

PROBLEM TERMINOLOGY

This chapter introduces the terminology used for the parameters of PARINT integration problems.

Single Integrand Functions

Let $D$ be a hyper-rectangular region in $\mathbb{R}^N$. Let $f(\vec{x})$ be the function to integrate over $D$. PARINT will attempt to calculate a numerical approximation $Q$ and an error estimate $E_a$ for the integral

$$I = \int_D f(\vec{x}) d\vec{x},$$

where the error estimate should satisfy $|I - Q| \leq E_a$. PARINT will attempt to find an answer within a user specified maximum allowed error, or, maximum allowed error tolerance. (This characterizes PARINT as having an "automatic" integration algorithm, for it proceeds until a satisfactory answer is reached and then stops automatically.) There are two different parameters that specify this tolerance. The parameter $\varepsilon_a$ is the absolute error tolerance and $\varepsilon_r$ is the relative error tolerance. As PARINT solves the integration, it will attempt to satisfy the
least strict of these two tolerances, trying to ensure that:

$$|I - Q| \leq E_a \leq \max\{\varepsilon_a, \varepsilon_r |I|\}.$$

Note that the value of $\max\{\varepsilon_a, \varepsilon_r |I|\}$ is approximated by $\max\{\varepsilon_a, \varepsilon_r |Q|\}$ as the algorithm proceeds.

As PARINT proceeds in its calculations, it will need to evaluate the function $f(x)$ for many values of $x$. The number of function evaluations has traditionally been used as a measure of the amount of effort spent in the computation of the integral. The user can set a limit $C$ on the number of function evaluations performed, ensuring that the calculations will not go on without stopping. Note that it is quite possible that for a given integrand function, the results of an integration problem may not be able to be achieved within the given error tolerances due to the nature of the integrand function, the effects of round-off errors in the computation and the limits on machine precision. If the function count limit $C$ is reached, then the required accuracy is generally not believed to have been achieved.

Vector Integrand Functions

The terminology presented in the previous section actually represents a simplification of the problems capable of being solved. If there are several functions to be integrated over the same region, and they behave similarly over that
region, then PARINT can integrate them together as a vector function.

The values $\varepsilon_a$, $\varepsilon_r$, and $L$ are as before. With the integrand function specified as $\tilde{f}(\tilde{x})$, PARINT will calculate a numerical approximation $\tilde{Q}$ and an error estimate $\tilde{E}_a$ to the integral

$$\tilde{I} = \int_D \tilde{f}(\tilde{x}) d\tilde{x},$$

while attempting to satisfy

$$||\tilde{I} - \tilde{Q}|| \leq ||\tilde{E}_a|| \leq \max\{\varepsilon_a, \varepsilon_r ||\tilde{I}||\},$$

where the infinity norm is used. This terminology subsumes that used earlier, as a single function can be thought of as a vector function containing a single component function. However, as throughout this thesis only vector integrand functions of a single function will be integrated, the simpler terminology will be used.
CHAPTER III

INTRODUCTION TO THE META-ALGORITHM

This chapter introduces the basic adaptive partitioning algorithm used to solve integrals.

Integration Rules

Given a region and a function to integrate, the integration rules used provide an estimate of the integral over that region and an estimate of the error in the result. Generally, quadrature rules are used to calculate results and error estimates, though the integration rules are viewed as a "black box" throughout this thesis; no concern is taken with their internal workings. The version of the PARINT software used provides four different rules from Genz and Malik [19, 20], each best suited for different kinds of functions or for functions of different dimensionality. For this thesis, a single integration rule was chosen for each integrand function. This ensures consistent experimental results: experiments performed in this thesis will examine the behavior of different integration algorithms on the evaluation of integrals. Changing the integration rule for a function across these experimental runs would cloud the results.
Sequential Algorithm

This section explains the sequential (single processor) integration algorithm.

Given an initial region and a function to integrate over that region, it is possible to evaluate an integration rule a single time over that region to get the result. In general, the estimated error may be too large.

Non-adaptive Integration

The next approach may be to divide the initial region into $n$ equally sized subregions using a fixed a priori dividing technique. The integration rule can be called once for each subregion, and the final result and error estimate are the sum of the results and errors over all subregions. This technique is classified as non-adaptive. A typical form of non-adaptive integration evaluates a fixed sequence of rules over the given region. If the sequence is extended to stop when a prescribed accuracy is reached (or it is determined that the error requirement cannot be achieved) then this is classified as an automatic non-adaptive algorithm [34]. The problem with this approach is that the integrand will generally have different characteristics over the region to integrate. Where the integrand behaves poorly (e.g., where there are peaks or singularities in the function), the quadrature rules will not perform as well. The subregions chosen should be smaller in these areas.
of the initial region. Where the integrand function is smoother, the quadrature rules will perform well, and the subregions will not need to be as small. In fact, given subregions of identical size, the integration rule may get an answer that is accurate well within the given tolerance where the function is smooth, resulting in wasted effort. This need to vary the size of the subregions leads us to the adaptive integration algorithm.

Adaptive Integration

In this approach, the integration rule is evaluated once on the entire initial region. If the result does not appear to be accurate enough, the initial region is split into two. The integration rule is evaluated over each subregion. The two subregions are then placed on a priority queue which orders the previously evaluated regions on the queue by their estimated error. The subregion with the largest estimated error is taken off the priority queue and divided in half. The integration rule is applied to each of these sub-subregions, and they are again placed back on the priority queue. The algorithm continues in this fashion until the function count limit has been reached, or, until the estimated error is below the tolerated error. This algorithm classified as automatic adaptive integration [34], and is presented in Figure 2. The algorithm is further classified as being a globally adaptive algorithm, as the error value used to control termination is the sum of the estimated errors over all regions. (In a locally adaptive algorithm an
individual region is continually divided and evaluated until the error over that region is small enough.

```adapтивe integration algorithm:
initialize;
while (function limit not reached and estimated error too large)
    retrieve region from priority queue
    split region
    evaluate new sub regions and update results
    place sub regions back onto priority queue
```

Figure 2. Adaptive Integration Algorithm.

Splitting the Regions

Regions are generally split in half. When the integrand is of multiple dimensions, it will not be clear along which dimension each region should be split. Therefore, an additional piece of information that the integrand rule must provide when evaluating a region is the direction along which the region should be split next time. Generally, this is the direction in which the function appears to be changing most rapidly (in PARINT this is estimated from 4th order differences of integrand values obtained for each region in each coordinate direction [19]). This information is stored along with the region on the priority queue and is used to split the region when it is taken off the queue.

Note that work is wasted in this approach. As the algorithm progresses, a small piece of the initial region may have the integration rule evaluated over
it many times as that portion of the initial region is split finer and finer. However, for functions that are difficult, of low dimensionality, and about which the behavior of the function is not known a priori, the adaptive approach is best. (At higher dimensions, e.g., > 15 dimensions, non-adaptive Monte Carlo approaches are usually best.)

This algorithm, as presented, is sequential; i.e., it is executed on a single processor. The rest of this thesis concerns the modification of this algorithm for a parallel environment. In such an environment issues arise concerning the division of the work, the storing of the priority queue, and the updating of results and error estimates. Chapter IV presents the Local Heap version, in which each processor works mostly independently, managing its own piece of the initial subregion and storing its own priority queue. In Chapter VI, the Global Heap approach, the focus of this thesis, is presented. In this approach, a single global priority queue, managed by one or more processors, is used to store all subregions for all processors. Each processor requests from the global heap the region with the largest error estimate and returns the new subregions to the global heap after performing a round of work.
CHAPTER IV

LOCAL HEAP ALGORITHM

This chapter presents the first parallel version of the adaptive integration algorithm, the local heap version. First, the parallel processing model used is presented.

Parallel Processing Model

When discussing a parallel algorithm, it is necessary to explain the underlying computational model used. In this thesis, the model is that of independent processing elements, each with a local memory, communicating with other processing elements via a message-passing interface. Within Flynn's classification [17], this falls within the MIMD (Multiple Instruction Multiple Data) model.

The network over which the processing elements communicate is further assumed to be a bus network, in which each pair of processing elements are considered to be an equal distance apart. (So that communication between any two processing elements requires a constant time.) The corresponding hardware model is that of a multicomputer, or more specifically, of a network of workstations (NOW) communicating over a Local Area Network (LAN).
This is to be contrasted with a fixed-interconnection network [2], in which communication channels connect individual processing elements in a specific network topology (e.g., a hypercube, butterfly, mesh, etc.). This is also to be contrasted with the Parallel Random Access Model [2], in which processing elements communicate using shared memory. Most Exclusive Read Exclusive Write (EREW) PRAM algorithms map easily to message passing algorithms where each memory read or write becomes a message received or sent.

The NOW model is a loosely-coupled parallel model, in which communication costs are relatively high, and in which coarse-grained algorithms perform better than fine-grained algorithms [27].

While most PARINT work has been on NOW's, it should be noted that previous versions of PARINT [10, 8, 9, 7] have been implemented and analyzed on a hypercube network architecture, specifically the nCUBE-2 [33].

Throughout this thesis, a Processing Element will be abbreviated as a 'PE', or referred to as a 'processor' or a 'process'. Within this thesis, these terms are generally synonymous, as there will be a single process per processor, and it is assumed that there is a single processor per workstation, though use of the term 'processing element' provides generality. Specific PE's may be referred to as controllers, workers, etc., as the situation demands.
The algorithm presented in this section is the primary algorithm used within the PARINT parallel adaptive integration research project [11]. (The PARINT work is based on previous work in [14, 4, 26] and others.)

In this version, each of the $p$ processes is assigned a piece of the initial region. Each process then begins to execute the adaptive integration algorithm over its initial subregion. Each process maintains its own local priority queue consisting of the smaller subregions generated from its initial subregion.

One process is chosen to be the Integral Controller (IC) process. The other processes are called Integral Worker (IW) processes. The IC's primary responsibility is to keep track of the global result and global error estimate. It accomplishes this by receiving local update messages from the IW's, from which it maintains the global result and error estimates. These allow it to determine when the result is within the desired tolerances, or, when the function count limit has been reached, at which point it will broadcast a STOP message to all the IW's.

It is possible that one worker may finish before another worker, either because the worker is executing on a faster processor, or, because the integrand was smoother within its initial subregion. Therefore, there must be a way for each worker to know when it has satisfactorily completed work on its initial subregion. There are a variety of specific techniques for accomplishing this and various criteria
are used. The method used in PARINT is for the IC to broadcast the current global \( \varepsilon \) value, calculated as:

\[
\varepsilon_{\text{global}} = \max\{\varepsilon_a, \varepsilon_r | Q| \}
\]

where \( Q \) is the current result held at the IC. The IW \( i \) takes the \( \varepsilon_{\text{global}} \) and calculates from it its local \( \varepsilon \) value:

\[
\varepsilon_{\text{local}} = \varepsilon_{\text{global}} \frac{\text{vol}(D_i)}{\text{vol}(D)}
\]

where \( D_i \) is the initial subregion assigned to worker \( i \).

A worker knows that it has satisfactorily completed work over its initial subregion when its local error estimate (the sum of the error estimates over the subregions it has generated) is less than its \( \varepsilon_{\text{local}} \) value.

Note that since the IC’s value of \( Q \) changes as it receives updates, it must periodically send down to all the workers a fresh \( \varepsilon_{\text{global}} \) value. The IW to IC result updates and the IC to IW \( \varepsilon_{\text{global}} \) updates constitute the bulk of the communication costs in the algorithm.

An additional algorithm parameter is the value \( ns \). This algorithm parameter controls the communication granularity at the IW by allowing the IW to accumulate result update information for \( ns \) basic iterations before sending an update message to the IC. A smaller value of \( ns \) will result in more messages, while
a larger value will result in fewer communications, but possibly wasted effort as an additional unneeded iteration or two is executed.

The basic algorithm at the IW is presented in Figure 3, the IC's algorithm is in Figure 4.

```
Integral Worker:
    initialize;
    while (global stop message not received)
        if (not locally-done)
            Check for updated eps_global
            for (i = 0; i < ns; ++i)
                Retrieve region from priority queue
                Split region
                Evaluate new subregions and update local results
                Place subregions back onto priority queue
            Send update msg to IC
            Update locally-done information

Figure 3. Local Heap IW Algorithm.
```

```
Integral Controller:
    initialize;
    while (function limit not reached and estimated error too large)
        Receive update message from an IW
        Update global result, error estimate, and eps_global
        if (time to send eps message)
            broadcast new eps_global value to all IW's
        Update globally done information
    Send STOP message to all IW's

Figure 4. Local Heap IC Algorithm.
```

Note that the IC does not send down $\epsilon_{global}$ information after every update,
but rather after it has changed by a significant percentage or after some set number of iterations, in order to reduce the communication costs. The $\varepsilon_{global}$ value held by a worker may be slightly out of date as a result.

This algorithm is asynchronous, in that the processes never synchronize during the execution of an algorithm. Each processor iterates through its loop at its own pace, performing integral work, sending messages as needed, and receiving messages as they arrive.

**Controller as Worker**

Previous experiments show that the IC is not kept busy in its rather simple job. As a result, the IC was modified to also act as one of the integral workers (this was done for the first time within the PARINT project in [12]). This functionality is termed *controller-as-worker* (CAW). The IC calls a function when there are no available updates to collect from the IW's. This function performs a single round of work (i.e., it splits a single region and calls the integration rule twice) and then returns the result to the IC portion of the code. The CAW code is assigned a portion of the initial region along with the other workers, and maintains its own priority queue.
Priority Queues

Only very basic and common operations are required for the priority queues; accordingly, a simple data structure like a binary heap can be used. For a general reference on priority queues and heaps, see [5, 1].

Specifically, only operations to initialize the heap, to delete and return the max element, and to insert a new element need to be supported. Therefore, a simple pointer-based binary heap implementation of the priority queue is used. From this point on, the term “local heap” will be synonymous with the term “local priority queue”.

Load Balancing

It is quite possible that one of the workers will complete the work needed over its initial subregion very early, while other workers are still processing their subregions. This processor will then be idle. In the worst case, one processor will end up with nearly all the work to be performed. For this reason, a dynamic load balancing technique is used to attempt to keep all of the workers busy at all times.

Note that the division of the initial region can be considered to be an instance of load balancing, specifically, static load balancing, i.e., load balancing done before work has commenced on the problem. Dynamic load balancing involves the transferring of work while the problem is being solved. See [27, 28] for
an overview of various dynamic load balancing schemes.

In PARINT, a scheduler-based receiver initiated load balancing scheme is used as follows: A worker knows when it is locally idle. At its next update, it informs the controller (the regular update message contains idle information). The next non-idle worker $i$ to send an update to the controller is sent a HELPER_ID message from the controller informing them of the id of an idle worker. If worker $i$ is still not idle by the time it receives the HELPER_ID message, and is not “close” to being idle (based on an algorithm parameter), then it selects the region on the top of its local heap and sends it to the idle worker in a SEND_RGN message. Otherwise, if worker $i$ is idle or close to being idle, it sends a dummy region. The recipient of the SEND_RGN message either receives the dummy region and informs the IC that it is still idle, or, receives the actual region, places it on top of its local heap, and resumes working. The local heap, local load-balancing version will be abbreviated LLB throughout this thesis.

A worker is considered to be close to idle if the ratio between its local estimated error and its $\varepsilon_{local}$ is less than a specific ratio, the LB_HELP_RATIO, another algorithm parameter. Note that it is completely idle when that ratio reaches 1.0.
PARINT Environment

The PARINT research group has released the current version of its integration tools, PARINT1.0. This release implements the LLB algorithm within the MPI [22] message-passing environment for use primarily with UNIX systems. Users can solve integration problems by calling the PARINT executable on the UNIX command line, by writing their own applications via the PARINT Application Programmer Interface (API), or through an integrated Java based graphical user interface. The release is available at [11]; the PARINT1.0 User Manual is available at [40].

The PARINT environment allows for the easy manipulation of integration parameters ($\epsilon_a, \epsilon_r, \mathcal{L}$, etc.) and algorithm parameters (ns, LB_HELP_RATIO). The algorithm can be modified via other run-time parameters, including the turning on and off of load-balancing.

Recent Modifications

Two modifications were made to the PARINT1.0 release code after the official beta release date, but before experiments were performed for this thesis. The benefits of these modifications were known, and it was felt that they should be added to the LLB version in order to compare the best LLB implementation with the modified design proposed in this thesis.
Round-Robin Idle Worker Selection

The first modification was first made in the experimental PARINT version of [13]. It was noticed that the higher ranked workers generally performed less work than the lower ranked workers. The problem was in the selection of idle workers. Whenever the id of an idle worker needed to be sent to a non-idle worker, the selection routine always sent the lowest-ranked idle worker's id. The result was that lower-ranked workers were more likely to have their id sent, and higher ranked workers were more likely to stay idle, resulting in an unbalanced load across the workers. The solution was to have the selection algorithm proceed in a round-robin fashion: the selected idle worker is the lowest-ranked worker with a rank higher than the last worker selected, wrapping around back to the lowest ranked worker as needed.

Handling of Local Heap During Load Balancing

The second modification involved the handling of a local heap when a worker received a new region as part of load balancing. Traditionally, when that new region arrived at an idle worker, the worker's old heap was completely deleted and a new heap consisting of the new single region was initialized. While investigating some difficulties with a particular function [39], it was realized that some of the regions that were being thrown out might still need to be evaluated in order
to reach an answer to the desired tolerances. The change was to have the workers keep their old heaps when doing load balancing and merely insert the new region onto the old heap. It is expected that the new region has the worst error, but it is possible that the worker will end up at some point evaluating some regions that in the old version would have been thrown out. The old technique was called *kill-heap*, the new technique *keep-heap*. For some functions it was discovered that keep-heap led to significant reductions in the amount of work required to reach an answer; keep-heap is also necessary for the extrapolation method reported on in [13]. The keep-heap technique is implemented as part of the PARINT1.0 release; the next release will allow for switching between the techniques as a run-time option for additional experimentation.

As the next chapter will discuss, there are various inefficiencies with the local load balancing version, and, with any adaptive integration algorithm.
CHAPTER V

INEFFICIENCIES IN LLB VERSION

This chapter discusses inefficiencies that can result in any adaptive integration algorithm, and, specifically, the LLB algorithm.

The term efficiency, when applied to parallel algorithms, generally refers to the ratio of work performed by a sequential algorithm to the work performed by the corresponding parallel algorithm [2]. Specifically, it is defined as $t_1/(t_p p)$ (where $t_1$ is the sequential time and $t_p$ is the time with $p$ processors), or, the ratio of speedup to $p$. Ideally, this ratio would be 1.0, indicating that there is no additional work performed. The additional work includes all interprocess communication, as the sequential algorithm needs no communication to complete its work. Since the parallel algorithm must perform at least some communication, the efficiency will always be less than 1.0.

Efficiency in Terms of Regions Evaluated

This thesis will focus on specific kinds of work performed by the adaptive integration algorithm, and modify the definition of efficiency accordingly. Specifically, the focus will be on the number of regions evaluated by the adaptive
integration algorithm. Therefore, the measure of the amount of work completed by the algorithm will be the number of regions evaluated.

The rationale for this is threefold. First, in these algorithms the number of regions evaluated to reach an answer can vary greatly as the number of processes solving the problem varies, as the parameters of the algorithm vary, and as the algorithm is modified. A meaningful goal therefore is to design a parallel algorithm that minimizes the number of regions evaluated.

Second, the time spent evaluating the integration rule is the most time consuming of the non-communication activities of a process. The best measure of this time is the number of regions evaluated.

Last, one must consider the relative costs of communication and the cost of evaluating a single region. As the communication costs increase, it will be more desirable to reduce communications, even at the cost of evaluating more regions (or, in general, doing more local computation).

The cost of evaluating a single region can vary greatly, though. A single function evaluation can take a few floating point operations, or, up to tens of thousands. And an integration rule can require a few dozen or a few thousand function evaluations to be performed. Therefore, given any level of communication cost, there will always be a function/integration rule combination where the cost of evaluating a region dwarfs the communication costs. And, it will be seen that the number of region evaluations needed to get an adequate answer tends to increase
as the number of PE's increases. In this situation, the best algorithm will work to reduce the number of regions evaluated at the expense of more communication (for a single given integration rule).

To consider a real-world analogy, consider the situation where the regions are evaluated by human beings, by hand, and where a manager assigns regions to be evaluated. It will take only a second or two for each region to be assigned, but it will require a human many hours to evaluate even a single region. In this analogy, every unnecessary region evaluation greatly increases the time to get a solution.

Function Evaluations vs. Region Evaluations

In the numerical integration community, it is traditional to measure the work performed by an integration algorithm in terms of the number of function evaluations. This thesis instead focuses on the number of regions evaluated. These measures are largely able to be interchanged; for any given integration rule and function dimensionality, the number of function evaluations will be a constant multiple of the number of regions evaluated.

The number of regions evaluated, however, is a more intuitive number. For example, after a worker in the LLB version has performed \( i \) region evaluations, there will be \( i + 1 \) regions in its local heap. And the number of basic iterations a worker has performed after the initial round of work is equal to the ratio of \( i - 1 \)
and \( ns \) (recall that \( ns \) is the integration parameter specifying how many rounds of work should be done between sending update messages; see page 16).

Note that if experiments were being performed to evaluate different integration rules for a single function, then it would be important to use function evaluations as a measure, to weigh the benefits of each additional function evaluation performed within a single rule evaluation. (Especially when considering the accuracy achieved, as this generally is affected by the number of function points taken by each rule application.) However, as mentioned, this thesis will only ever use a single integration rule per function.

Evaluated Regions Viewed as a Binary Tree

Consider first the sequential algorithm. The pattern of subregions produced by the adaptive algorithm can be viewed as a binary tree [7]. Each node of the tree corresponds to a region with the initial region as the root. Every division of a region results in the creation of two child nodes. (Though, it should be noted that the tree is a binary tree merely due to the choice of each region being divided in two; there is no inherent reason why regions could not be divided into more pieces.) At any point in the algorithm, the regions in the priority queue (the active regions) correspond to the leaf nodes of the tree. The internal nodes represent regions that have been previously evaluated and divided and are no longer managed by the algorithm. The number of nodes, both internal and leaf nodes,
represent the total number of regions evaluated in the course of the algorithm.

Note also that in the parallel LLB algorithm, the initial region is divided a priori into $p$ pieces; so the binary tree is actually a forest of trees, one for each worker. Also, the transferring of a region from one worker to another distorts the tree view.

We can also view the nodes evaluated as a set of nodes. Given the initial region, all possible subregions able to be generated from that initial subregion can be enumerated. A specific set of these generated subregions can correspond to the nodes evaluated by the execution of an algorithm. (Though many sets do not correspond to any execution of an algorithm; a valid pattern of subdivisions must be followed.)

Viewing the nodes evaluated by the algorithm as a tree or a set aids in comparing different algorithms, as will be done in Chapter IX.

Regions Evaluated by the Sequential Algorithm

The sequential algorithm is a deterministic algorithm, in that if it is run many times over a single set of inputs, the outputs will be the same every time, including the set of regions evaluated. The parallel message-passing algorithm (an asynchronous algorithm) has an element of nondeterminism; due to the variations in the speed of messages being passed around, the algorithm will generally proceed differently on each execution. Accordingly, the set of regions and the number of
regions evaluated will be different across different runs.

Assumptions in the Adaptive Integration Algorithm

The adaptive integration algorithm is built upon several assumptions concerning the behavior of the error estimates of the regions encountered. These assumptions must hold for the majority of the regions evaluated. If they do not, the algorithm's performance may suffer or the algorithm may fail, as explained below.

First, it is assumed that when a region is evaluated, the total error estimate will be reduced, as the evaluation of that region represents a refinement of the result as compared to when the parent region was evaluated. If a singularity or peak is "discovered" as the result of evaluating a region, then the total error estimate can temporarily increase, but this is assumed not to be the usual case. (Indeed, the total error estimate must decrease as regions are evaluated, or the algorithm will not converge upon an answer. Again, this is possible, but is not the usual case.)

The ordering of the regions by the priority queue (by estimated error over each region) reflects a second assumption, that splitting the region with the highest estimated error and evaluating the resulting subregions should reduce the total error by the greatest amount. This is also an assumption that is not always true.

A third assumption is that changes in the estimated error will reflect
changes in the actual error. In other words, seeking to quickly reduce the esti-
mated error should also result in quickly reducing the actual error. Again, there
is no guarantee that this will be the case.

However, if we grant these assumptions, then it is clear that the sequential
algorithm will, within the limits of the algorithm, converge on the true result in
the fewest number of region evaluations.

Regions Evaluated by the Parallel Algorithm

The LLB parallel algorithm does not have access to a single priority queue
of regions. Each worker has its own queue. It cannot be assumed that as each
worker takes a region off its local queue that it is getting a region with an error that
is large when compared with all regions. It is possible that one worker has a queue
full of generally low-error regions when compared with other workers. Therefore,
by the second assumption above, we can conclude that the parallel LLB algorithm
should reduce the global error estimate more slowly than the sequential algorithm,
i.e., that after a set number of region evaluations, the sequential algorithm would
have a lower error estimate than the parallel algorithm.

A related conclusion is that the parallel algorithm will require more regions
to get the error estimate below the tolerated error than will the sequential algo-

rithm. This is an analytical justification for the reduced efficiency of the parallel
algorithm.
This particular loss in efficiency is termed "singularity loss" [26, 7]. This term is derived from the specific case where a significant singularity is discovered by a worker. The regions containing that singularity will have relatively large error estimates, and the workers' queues containing those regions will therefore include regions with large errors. A worker whose initial region does not contain the singularity will have a queue consisting of regions with smaller errors.

The worker with the smaller error regions may divide and evaluate regions that do not need to be evaluated to get an answer to the desired tolerances. Even then, the loss will be a partial loss, as evaluating any region, even a relatively unimportant one, can reduce the global error estimate. Also, this problem will be reduced by dynamic load balancing, as this allows regions to flow to workers with generally low error regions (depending upon the load balancing strategy used).

The phenomenon of singularity loss can occur when there are no singularities, though. It will occur any time the integrand function behaves poorly over some portion of the initial region and is well behaved over other portions.

Note that the problem of singularity loss increases as the number of workers increases. As the number of local queues increases, it becomes more likely that workers are taking regions off their queue whose error estimate is locally worse, but is not that large globally. This may give a justification for the increasingly poor performance, in terms of efficiency, as the number of workers increases.

Some initial efficiency results for the LLB algorithm can now be presented.
First the general experimental environment must be introduced.

Experimental Environment

All program executions completed for the experiments in this thesis were run on a 10Mbs Ethernet network of UNIX Sparc workstations. The workstations on the network include Sparc 1+, Sparc IPC, and Sparc 5 machines; however, the actual runs were performed on a set of homogeneous Sparc Ultra 10's.

The PARINT release PARINT1.0 was used for the LLB runs. The runs generally included from 1 to 24 workstations. The set of functions did not include any vector functions, so the max vector length constant was set to 1 to reduce memory requirements. The max dimension value was also reduced to 6 to reflect the functions used. (Note that these are compile-time constants within PARINT1.0, and are used to declare vectors of values that are ultimately stored within the local heaps. It is desirable to reduce their values as much as possible to save memory.) All runs in which the results could vary (due to algorithm nondeterminism or varying timing results) were run at least 5 times and averaged.

Table 1 shows the functions used in the experiments. While it is not possible to come up with a set of "average" functions, those used contain a wide range of function behavior.

Generally, the $\varepsilon_a$ was set to zero so that only the $\varepsilon_r$ value was used to calculate the desired tolerance. The $\varepsilon_r$ value for each function was set to as small
Table 1

Integrals Used for Testing

<table>
<thead>
<tr>
<th>Name</th>
<th>Integral</th>
<th>$\varepsilon_d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smooth integrands</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fcn1</td>
<td>$\int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} \int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} \int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} \int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} \int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} \frac{x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2}{(x_1 + x_2 + x_3 + x_4 + x_5)^2} \ dx_1 \ dx_2 \ dx_3 \ dx_4 \ dx_5 \ dx_6$</td>
<td>6</td>
</tr>
<tr>
<td>fcn2</td>
<td>$\int_{0}^{1} \int_{0}^{1} \int_{0}^{1} \int_{0}^{1} \frac{x_1^2}{x_1 + x_2 + x_3 + x_4 + x_5} \ dx_1 \ dx_2 \ dx_3 \ dx_4$</td>
<td>7</td>
</tr>
<tr>
<td>fcn3</td>
<td>$\int_{0}^{1} \int_{0}^{1} \int_{0}^{1} \int_{0}^{1} \frac{x_1 x_2 x_3}{(x_1 + x_2 + x_3 + x_4 + x_5)^2} \ dx_1 \ dx_2 \ dx_3 \ dx_4$</td>
<td>9</td>
</tr>
<tr>
<td>fcn16</td>
<td>$\int_{0}^{1} \int_{0}^{1} \exp(x + y) \ dx \ dy$</td>
<td>9</td>
</tr>
<tr>
<td>Oscillating integrands</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fcn4</td>
<td>$\int_{0}^{1} \int_{0}^{1} \int_{0}^{1} \int_{0}^{1} \cos(x_1 + x_2 + x_3 + x_4 + x_5) dx_1 \ dx_2 \ dx_3 \ dx_4 \ dx_5$</td>
<td>7</td>
</tr>
<tr>
<td>fcn5</td>
<td>$\int_{0}^{1} \int_{0}^{1} \int_{0}^{1} \int_{0}^{1} \sin(10x_1) dx_1 \ dx_2 \ dx_3 \ dx_4$</td>
<td>9</td>
</tr>
<tr>
<td>fcn6</td>
<td>$\int_{0}^{1} \int_{0}^{1} \cos(x + y) dx_1 \ dx_2 \ dx_3 \ dx_4 \ dx_5$</td>
<td>9</td>
</tr>
<tr>
<td>Integrand with singularity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fcn7</td>
<td>$\int_{0}^{1} \int_{0}^{1} \int_{0}^{1} \int_{0}^{1} \int_{0}^{1} \frac{1}{(x + y + z)^2} dx_1 \ dx_2 \ dx_3 \ dx_4 \ dx_5$</td>
<td>7</td>
</tr>
<tr>
<td>fcn14</td>
<td>$\int_{0}^{1} \int_{0}^{1} \int_{0}^{1} \int_{0}^{1} \frac{1}{x} dx_1 \ dx_2 \ dx_3 \ dx_4$</td>
<td>10</td>
</tr>
<tr>
<td>fcn18</td>
<td>$\int_{0}^{1} \int_{0}^{1} \int_{0}^{1} \int_{0}^{1} \frac{1}{x^2} dx_1 \ dx_2 \ dx_3 \ dx_4$</td>
<td>6</td>
</tr>
<tr>
<td>fcn19</td>
<td>$\int_{0}^{1} \int_{0}^{1} \int_{0}^{1} \int_{0}^{1} \int_{0}^{1} \frac{1}{x^2} dx_1 \ dx_2 \ dx_3 \ dx_4 \ dx_5$</td>
<td>2</td>
</tr>
<tr>
<td>fcn21</td>
<td>$\int_{0}^{1} \int_{0}^{1} \int_{0}^{1} \int_{0}^{1} \int_{0}^{1} \frac{1}{x^2} dx_1 \ dx_2 \ dx_3 \ dx_4 \ dx_5$</td>
<td>2</td>
</tr>
<tr>
<td>fcn22</td>
<td>$\int_{0}^{1} \int_{0}^{1} \frac{1}{x^2} dx_1 \ dx_2 \ dx_3 \ dx_4 \ dx_5$</td>
<td>11</td>
</tr>
<tr>
<td>Integrands with a peak</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fcn8</td>
<td>$\int_{0}^{1} \int_{0}^{1} \frac{605y}{(1 + 120(1 - y))(1 + 120(1 - y))^2 + 25z^2 + y^2) dx_1 \ dx_2 \ dx_3 \ dx_4 \ dx_5$</td>
<td>9</td>
</tr>
<tr>
<td>fcn9</td>
<td>$\int_{0}^{1} \int_{0}^{1} \int_{0}^{1} \int_{0}^{1} \frac{1}{(y + 0.0001)(y + 0.0001)^2 + 0.0001) dx_1 \ dx_2 \ dx_3 \ dx_4 \ dx_5$</td>
<td>9</td>
</tr>
<tr>
<td>Integrand with discontinuous derivative</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fcn10</td>
<td>$\int_{0}^{1} \int_{0}^{1} \exp(</td>
<td>x + y - 1</td>
</tr>
</tbody>
</table>

A value as was generally possible given the inherent limits on accuracy due to limits on machine precision, data types used (8 byte double's were used for all relevant variables in most runs), and round-off errors that occur after too many iterations.

Setting the error tolerances too loose (i.e., setting the $\varepsilon_r$ or $\varepsilon_a$ too large)
results in runs of so few iterations that the total times are so low (e.g., < 0.05 seconds) that accurate timing results can not be achieved, as any variation in the network load during the short run can greatly affect the time. Table 1 also shows the default $\epsilon_r$ value used for each function, shown as $\epsilon_d$ (defined as the number of digits of accuracy requested by the $\epsilon_r$ value, so $\epsilon_d = -\log_{10}\epsilon_r$). Even though high tolerances were requested, some problems were inherently too easy, so a limited set of functions were used on some experiments.

Unless otherwise noted, the function count limit was set high enough that it was not reached on any runs. Allowing the function count to be reached does not allow for a measurement of the total region evaluations to get an answer to the desired accuracy.

Initial Efficiency Results

Figure 5 shows efficiency results for all of the functions. The efficiency measure graphed is the number of regions evaluated by the sequential run, divided by the number of regions evaluated by a particular run at some number of workers. The graph is not intended to be viewed on a function-by-function basis, but rather to show the trend over all functions of falling efficiency as the number of workers increases.

The efficiencies at 24 processors range from 0.8 to less than 0.1. Consider that at an efficiency of 0.25, four times as many regions are evaluated than in
the sequential version. Given that the goal of any parallel algorithm is linear speedup of time, having to perform 4 times as many region evaluations is a large obstacle. For example, the efficiency of \textit{fcn10} at \( p = 10 \) is 0.44. Consequently \( \frac{1.0}{0.44} \approx 2.27 \) times as many regions must be evaluated at \( p = 10 \) to get an answer than when \( p = 1 \). Since we cannot expect more than a 10-fold speedup in the time per function evaluation with \( p = 10 \), having to perform 2.27 times as many region evaluations will likely limit the overall speedup to \( \frac{10}{2.27} \approx 4.4 \). (Though, other sources of inefficiency must also be considered.)

The efficiency does not decrease monotonically as \( p \) increases. Generally
this is due to the different arrangement of the $p$ initial regions. Some combinations of initial subregions and functions just perform better, others perform worse. In PARINT1.0 and previous PARINT code, the number of integral workers had to be $2^k$ or $2^k - 1$ for some integer $k$. A change was made to the PARINT code allowing any number of workers. The previous power-of-two algorithm for finding the initial subregions attempted to preserve in each subregion the shape of the initial region (e.g., dividing each dimension an identical number of times, etc.). The new simplified algorithm simply finds the longest dimensional length of the initial region and divides it $p - 1$ times to form $p$ slices of the initial region. This method was used at all values of $p$, even powers of two, to reduce the variation between the forms of the initial subregions as $p$ increased.

Other Sources of Lost Efficiency

There are two other sources of lost efficiency in the parallel LLB algorithm, as identified in [26, 7]. (Other losses can exist in other kinds of algorithms and in other parallel architectures; see [32].)

Braking Loss

When the controller determines that the global result is within the desired tolerances (or that the function count limit is reached), it will send out a STOP message to all the workers. The term braking loss is applied to the loss of efficiency
due to the workers performing some additional rounds of work after the controller has determined that the results are close enough, but before the workers get the STOP message.

The braking loss can easily be one round of work per worker; for the PARINT algorithm this could result in $2 \times \text{ns}$ additional region evaluations per worker (since each iteration of the \text{ns} loop divides a region and evaluates the two resulting subregions). A detail of the implementation, though, is that a worker will not send up an update if it does not have a "recent" $\varepsilon_{global}$ value on which to base its decision to go locally idle. Therefore, it is possible for the braking loss at a single worker to be a larger multiple of \text{ns}. (A method for improving the update and $\varepsilon_{global}$ message passing has been devised, and will likely be implemented in future releases of PARINT.)

As the number of workers increases, it is likely that it will take longer for the STOP message to reach all the workers. The workers to receive the message last may have a higher braking loss. Note that if a true broadcast capability is supported by the communication network, then this may not be the case. Within the MPI communication system, though, using the broadcast message primitive (which may or not be a true broadcast depending upon the underlying network) requires synchronization of the processes. Accordingly, the STOP message is sent as a separate message to each process, from the lowest ranked through the highest ranked worker.
In the PARINT1.0 implementation, the IC does collect all of the extra region evaluations and applies them to the global results before terminating. The rationale is that since the work has been done to get these additional updates, the controller might as well perform a small amount of extra work and include their contribution in the final answer. Ideally, though, the workers would not perform this extra work.

**Iteration Loss**

Another kind of loss is termed *iteration loss*. This loss stems from the static initial division of the given region. The initial region is broken into $p$ pieces, but, the sequential algorithm may be able to solve the integration problem in fewer than $p$ region evaluations. The additional regions created at the initial division already represent an inefficiency, especially as the number of workers increases. Of the functions and $\varepsilon_r$ values in Table 1, $fcn5$ requires fewer than 24 region evaluations (specifically, 17) to be completed with a single processor.

**Reducing Inefficiencies**

By far, the largest loss of efficiency is singularity loss. And the primary cause of this loss is the division of what is a single priority queue in the sequential version with $p$ priority queues in the parallel version. The next chapter presents a solution to this problem, using a single priority queue over all workers.
CHAPTER VI

INTRODUCTION TO THE GLOBAL HEAP

This chapter describes a previous global heap version of the adaptive partitioning integration algorithm. It also presents background information and discusses related work and the research that led to the current implementation.

Motivation for a Global Heap

As discussed in Chapter V, one idea for increasing the efficiency of the parallel adaptive partitioning algorithm is to have a single priority queue of all the regions. This single priority queue is called the global priority queue, or, global heap. There are still multiple workers, but each gets its regions from the global heap, divides and processes the region, and then places the new regions back on the global heap. (Note that technically it is premature to refer to this data structure as a “heap”, for that implies a specific implementation, while at this point we are considering the data structure as an abstract data type divorced from any implementation. However, this data structure will end up being implemented as a heap, so its final name will be used for simplicity.)

This approach ensures that workers will be working on the $p$ regions that
have the largest error over all regions. From the assumptions in the previous chapter, this should lead to an algorithm that can find a result to the desired tolerance in fewer iterations than when multiple local heaps are used.

**Needed Operations**

Before discussing the design of any implementation, it is instructive to consider the operations needed for this global heap. Each worker will make two requests of the global heap: they will request the deletion and retrieval of the maximum region from the heap (referred to as a `DELETEMAX` operation), and, they will request the insertion of a new region onto the heap (referred to as an `INSERT` operation).

There will need to be some sort of initialization operation, to initialize a new heap. In the course of executing the adaptive partitioning algorithm, we only need to initialize a heap at the beginning of the algorithm; after that point, the heap grows monotonically. Since this operation is only used once, this thesis will not explicitly discuss the initialization operation.

**Simple One-Node Global Heap**

The parallel model used is that of processors communicating via messages over a NOW. The simplest corresponding global heap consists of a single processor, termed the *Global Heap Controller* (GHC), that stores all the regions in the global
heap within its local memory space.

Whenever any worker needs a new region, it forms a message requesting a region and sends that message to the GHC. The GHC receives the request, extracts the maximum error region from the heap, and sends it back to the requesting worker. When the worker has completed a round of work, it sends the two new regions back to the GHC, requesting that they be inserted into the global heap. Since the latter message is always followed by a new request for regions, a simplification is to have a single message combining the requests for insertion and deletion.

In this design, called the Single Node Global Heap, the GHC can simply store the regions in a normal binary (or d-ary) heap, and perform DELETEMAX and INSERT operations on the heap as in any sequential heap implementation. The GHC is acting as a region server, serving regions to the workers, who are acting as region clients.

The clients do not care how the global heap stores the data items, so long as the DELETEMAX and INSERT operations are supported efficiently.

Multi-Node Global Heap

In the Single Node Global Heap, the GHC will have to store all the regions that used to be stored by all the workers individually. This could be a very large number of regions. It is instructive to consider an alternate design for the global
heap, a *Multi-Node Global Heap*. In this design, multiple PE’s work together to store the regions in the global heap, and, work together to implement the required heap operations. The actual regions are stored in the local memory of multiple PE’s, yet in a fashion that allows for the quick retrieval of the maximum error region over all regions in the global heap, and, for the quick insertion of new regions.

Such an implementation has been previously done in the PARINT project and elsewhere. The next few sections describe previous PARINT global heap implementations and related work.

**Overview of Storing Distributed Data**

The basic idea behind the multi-node global heap is to spread the data to be stored across multiple PE’s. This is a *distributed data structure* (DDS) [31], a data structure which is distributed across multiple sites of a communication network and may be accessed by multiple PE’s concurrently. The aspects of a distributed data structure include: an organization or topology of PE’s, each of which stores a portion of the data, local data structures to be used at the PE’s for the storing of data, and, a defined collection of operations and protocols for retrieving data, inserting data, and modifying data in the distributed data structure.

The goals of a DDS mirror the goals of any parallel algorithm: faster access to data, greater concurrency in the accessing of data, the ability to store greater
amounts of data, etc.

Previous PARINT Global Heap Research

The previous PARINT global heap design was derived from Gupta and Photiou in [23], based on previous work in [16, 35, 6]. The Gupta and Photiou design was implemented and tested within the context of the PARINT project in [9]. This implementation was done on the nCUBE-2 [33] architecture using native message passing primitives. Further analysis of the global heap was done in [7].

In this design the global heap is broken into a tree of nodes. Each node contains regions; the regions in a node are guaranteed to meet the global max-heap condition: that any region in a node is guaranteed to have an error estimate larger than or equal to that of any region in the child nodes, and, less than or equal to that of any region in the parent node.

Two specific schemes were developed for distributing the nodes over a collection of processors, the *vertically sliced global heap* and the *horizontally sliced global heap*. In the vertically sliced heap, each PE maintains a portion of every node in the heap, i.e., each PE stores regions of low, high, and medium priority. In the horizontally sliced heap, each PE manages a single node (i.e., each PE stores regions within a range of priorities).

In either approach, global inserts and deletes are initiated at the root node with the effects rippling downward through the tree of nodes. In addition, the
tree is guaranteed in these designs to be balanced; if there are a total of \( p \) PE's in
the tree, and a total of \( n \) regions to be stored, then each PE stores approximately
\( n/p \) regions (specifically, either \( \lfloor n/p \rfloor \) or \( \lceil n/p \rceil \) regions).

As Gupta and Photiou concluded in [23], the horizontally sliced heap re-
sults in less communication overhead, so that design was used as the basis for the
modified global heap design of this thesis.

Horizontally Sliced Global Heap Design

The Horizontally Sliced Global Heap consists of a complete binary tree
of processors in which each processor stores all the regions for a single node (a
complete tree implying that \( p \) is \( 2^k - 1 \) for some \( k \geq 1 \)). Therefore, each processor
stores regions whose error estimates are less than or equal to any region at the
parent processor, and are greater than or equal to any region at the child processors
(regions of equal error are prioritized arbitrarily). The value of \( p \) can be equal to,
or less than, the total number of PE's participating in the integration.

Consider the insertion of \( m \) items into the global heap currently containing
\( n \) regions over \( p \) nodes. The \( m \) items arrive at the root node. Currently each node
has either \( \lfloor n/p \rfloor \) or \( \lceil n/p \rceil \) regions and the max-heap condition applies. After the
insertion is completed, each node will have either \( \lfloor n+m/p \rfloor \) or \( \lceil n+m/p \rceil \) regions, and
the max-heap property must still hold.

Consider the “thinking” of the root node. It knows (or can easily find
out) the range of errors in the regions that it stores. It does not know the range of errors for the regions stored at either of its children. It can also examine the errors in the incoming regions. In the worse case, the regions it must insert will have low errors, meaning that there are regions at the root's children nodes that have larger error than some of those to be inserted. In this case, the new regions must be sent down to the children (to be stored at the children or even lower-level nodes), and, nodes from the children must be sent to the root node for storage.

The insertion proceeds as follows: the regions to be inserted are sorted into a list \( L \). The root requests some calculated number of regions (for details see [23]) from each child node, receives them, and sorts them into \( L \). The largest \( \varepsilon_{\text{rem}} \) regions (also calculated) in \( L \) are added to the root's collection of regions. Then, over the remaining items in \( L \) and the root node's regions, the smallest \( \varepsilon_l \) and \( \varepsilon_r \) regions must be sent to the left and right children, respectively. (The root's regions must be considered because they may have a lower error than the new regions in \( L \).) The root node must therefore be able to both insert new items into its local data structure, delete max items (to support the \textsc{DeleteMax} global heap operation), and, delete minimum items. A convenient data structure that can handle these operations is the \textit{deap} [24] (these data structures will be termed the nodes' local deaps).

The children then receive the regions from the root node and each recursively initiates its own heap insertion at the subtree rooted at its node (taking into
account that each has already sent some of its regions to the root node). Once the root node has requested regions, shuffled its data structures, and then sent regions to its children, it is finished with that global insert.

Consider now the DELETEMAX operation. On a deletion of \( m \) nodes, each node must finish with \( \left\lceil \frac{n-m}{p} \right\rceil \) or \( \left\lfloor \frac{n-m}{p} \right\rfloor \) regions.

Assuming that \( m \leq \left\lfloor \frac{n}{p} \right\rfloor \) (which it must, as will be discussed below), the root node holds the \( m \) regions that must be extracted. It can remove them from its local deep. At that point, the global heap is unbalanced; the global heap must request regions from its children. As the root node does not have perfect information about the regions stored by its children, it must request the worst-case number of regions, shuffle regions between what it receives and what it stores, and then send regions back down to the children.

The root node requests the regions and receives them into a sorted list \( L \). At this point, the root node has some regions in its local deep, some regions to insert from a sorted list \( L \), and the children are expecting to receive regions from their parent. The global deletion routine at this point can proceed exactly like the insertion routine: the root node keeps the appropriate number of highest-error regions, sends the remaining lowest-error regions to its children, who receive them and initiate their own heap adjustment on their own subtrees.

Note that while the algorithm in Figure 2 was presented as the local heap algorithm, it is also the algorithm used with the global heap with the exception
that the maintenance of the priority queue occurs on a global, cooperative level rather than a local level. In the previous implementation of the global heap, this meant that all workers were synchronized, each doing its portion of the global heap maintenance in a synchronized fashion between each round of integration work. This thesis implements an asynchronous version. The former algorithm will be termed the *Synchronized Global Heap Algorithm*, to distinguish it from the current asynchronous design.

The design restricts global deletions to only retrieve regions stored at the root node. As noted, this restricts deletions to $m \leq \lfloor \frac{n}{p} \rfloor$ nodes; this implies that the global heap must contain $n \geq mp$ regions before any deletions can be done. The implementation done in [9] required that all integral workers initially maintain a local heap across multiple iterations of the adaptive integration algorithm until a threshold number of regions had been generated. At this point, the first global insertion was done, and after that point the global heap was used. This transition from local heaps to a global heap incurred a large cost. The design used in this thesis eliminates this problem.

**Current Related Research**

Mans has developed a related global heap design and implementation [30, 31]. The preferred term used here is *distributed priority queue* (DPQ) (which avoids confusion with the term “global heap” which can apply to a heap of local memory
space as allocated and managed by an operating system).

Mans' implementation used MPI and was tested on Cray-T3D and Meiko-T800 machines. His design included DPQ’s implemented in d-ary heaps and binomial heap structures [5], where the processing nodes were linked in a corresponding tree structure. As with the synchronized global heap algorithm, the root of the tree of nodes contained the highest priority items; the leaves of the tree the lowest priority items.

They are load balanced designs, in that the items stored are balanced across the processors. The technique used to do this (in the context of a global insertion) is as follows: The insertion of each item is considered separately. For each item, a target leaf node is selected in a round-robin fashion using an algorithm that guarantees a balanced selection and reduces congestion of messages up and down the tree. After the insertion is complete, this leaf node will have an additional data item. The insertion proceeds from the root node down the tree towards the destination leaf; at each step, the item is sent downward until its priority matches the range of priorities of a node.

After this downward shuffling, the participating nodes may also have to re-heapify, i.e., they may have to exchange data items with their children nodes until the global heap condition is re-satisfied. This is necessary to ensure a balanced heap.

It is interesting to examine the method used in this reheapification. The
Mans algorithm exchanges data items between a parent node and its children nodes one at a time until the heap condition is satisfied. This results in a possibly large number of small (single item) messages. The total number of items needed to be sent is not known a priori due to the nature of the local data structures (splay trees [36] are used). However, this technique does result in the minimum number of data items sent. The cost, however, is the large number of messages sent.

In the synchronized global heap design of Gupta and Photiou, a parent node retrieves all necessary data items (regions) from its children in two messages (one from each child) by receiving the worst case number of regions. (Note, though, that technically more than two messages were used, due to the relatively small maximum message size of the nCUBE-2, and the large messages (arrays of regions) that needed to be sent. Rather, the regions were grouped and sent in a sequence of messages until all regions were sent. The ideas of Mans indicate that if processing were done between these messages, the number of regions sent might have been reduced. The implementation in this thesis contains similar child-to-parent messages, and they were implemented as a single, large, MPI message). Mans did get good results from his implementation. It should be noted though, that the architectures used feature communication latencies smaller than that of the nCUBE-2, and much smaller than the NOW model used in this thesis.

In [31], some other concerns are noted, which should be discussed as they
Two models of application behavior, as applied to the use of the DPQ, are noted. The first is the self-scheduling model, in which the priority queue has a single initial item, processors request DELETEMIN operations and perform subsequent INSERT operations, and execution terminates when the DPQ is empty. The second is the event-set model, in which the queue has initial items in it, each processor repeatedly requests items and inserts new items, and terminates after a given number of iterations, or when an appropriate result is obtained. The good results of Mans were obtained over both of these models.

The adaptive partitioning algorithm fits neither of these models. It begins with an empty heap, the processes go through iterations in which they request regions and then insert even more regions, and terminate after some initially unknown number of iterations, during which time the number of regions in the global heap grows monotonically.
CHAPTER VII

CURRENT DESIGN FOR THE GLOBAL HEAP

This chapter explains the design of the global heap developed in this thesis. This algorithm, distinguished from the synchronized version just presented, is termed the GHEAP algorithm, and is based on the horizontally sliced global heap of [23]. (The GHEAP and LLB algorithms are the primary algorithms compared in this thesis.) The implementation, detailed in Chapter VIII, used MPI as an experimental extension, based on PARINT1.0, of the PARINT system of numerical integration. (Note that a partial design and implementation of the GHEAP algorithm was previously completed in [42]).

Design Overview

The design, as previously mentioned, is asynchronous. The integral workers are viewed as independent clients of the global heap. Each is free to execute the adaptive partitioning algorithm at its own pace, receiving regions from and sending regions to the global heap as needed. Additionally, each periodically sends updates up to the IC. No $\varepsilon_{global}$ values need to be sent from the IC to the IW's, as the workers never become locally idle over some initial piece of the total region.
Rather, they continue working on non-contiguous subregions received from the global heap until the IC sends out the STOP message. (Though, as in the LLB algorithm, the initial region is divided a priori among the workers; each evaluates its initial piece and sends an initial update to the IC. This ensures that the controller has some result estimate for the entire initial region. After evaluating its initial region, each worker begins normal iterations of the GHEAP algorithm.)

An asynchronous design was chosen because the targeted hardware model was the NOW; in this model the communication is assumed to be slow, and, synchronization in a parallel algorithm generally requires additional communication. Also, the workstations are assumed to be of a wide-range of capabilities; in a synchronized implementation the faster workstations may end up idling at each iteration while the slower workstations catch up.

The root of the global heap is termed the Global Heap Controller (GHC), the other nodes in the global heap are the Global Heap Workers (GHW’s). The global heap must again be a complete binary tree of $p$ nodes, with the number of nodes less than or equal to the total number of processors. The global heap will store $n$ regions, and as in the synchronized version, it is guaranteed that each node in the global heap stores $\left\lfloor \frac{n}{p} \right\rfloor$ or $\left\lceil \frac{n}{p} \right\rceil$ regions. The global max-heap condition is the same as in the synchronous algorithm: each processor holds only regions that have errors less than or equal to those at the processor’s parent node and that have errors greater than or equal to those at the processor’s children nodes.
The total group of PE's in the algorithm consist of the Integral Controller (IC), the Global Heap Controller (GHC), the Integral Workers (IW), the Global Heap Workers (GHW), and, the Controller-as-Worker (CAW). The key to the implementation is designing each MPI process to variably fulfill multiple functions (i.e., to act as multiple PE's). The IC can optionally also be a CAW, or not, by simply switching the value of a boolean flag (via a command line parameter). The GHC functionality can be assigned to the same process as the IC, or, be assigned to the lowest-ranked IW process. Each IW might also be a GHW, based on the rank of the IW and the number of nodes in the global heap.

This flexibility is important because it is not known beforehand which combinations of functionalities within each process will produce the best results. Experiments must be performed to determine this. Also, it may be the case that the best combination depends upon the particular network and workstations used, requiring the switching of combinations based on run-time information.

Chapter VIII discusses how this variability is achieved. For now, we will consider each functionality of each process to be a distinct process, without regard to the overall sum of what each process is actually doing.

**Design of the GHC**

The GHC design supports asynchronous operation and tries to reduce the number of global heap operations needed.
In the synchronous global heap, regions were always inserted and deleted in a large group, as all workers sent regions to or received regions from the GHC at the same time.

In the new design, the GHC receives regions from a single worker at a time and there needs to be a quick response to each request. To reduce the number of messages, each request for an insertion of regions is considered an implicit request to send new regions back to the worker. (Note that throughout the entire algorithm, no worker ever sends regions to the global heap without, at the same time, wanting additional regions from the global heap.)

We do not want to have to perform a global heap adjustment (insertion or deletion) as each request is serviced. Yet requests cannot be held pending the next global insertion or deletion.

The solution is as follows: The GHC stores regions in two data structures, a local deap $D$ (filling the same basic role as the local deap in the synchronous algorithm) and a sorted linked list $L$. Each message sent to the GHC contains $r$ regions, though generally $r$ will be 2, as workers will receive a single region, divide it and evaluate the new subregions, and then send them both back to the global heap. The incoming regions will be inserted into $L$ sorted by descending estimated error. The regions on $L$ are not considered to yet be part of the global heap.

To fill a request for regions, the GHC will compare the regions on $L$ and $D$. 
If \( L \) contains one or more regions of higher error than the max region on \( D \), then those regions have a higher error than any regions in the global heap and they can be removed from \( L \) and sent back to the requesting worker. (Since the GHC sits at the top of the global heap, the regions in its local heap have a larger error than those in any other node in the global heap.)

If \( D \) contains regions with the highest error, then they can be removed from \( D \) and sent back to the requesting worker. (This may reduce the value of \( n \), possibly resulting in fewer than \( \lceil \frac{n}{p} \rceil \) regions at the root node. This imbalance is only temporary, persisting until the next global adjustment of regions.) One multi-region request from a worker may be filled using regions from \( L, D \), or both.

\textbf{Global Heap Adjustments}

This pattern of retrieving regions from \( D \) and \( L \) without bothering other nodes in the global heap can continue until one of two situations occurs. First, \( D \) may become empty. At this point, it is no longer known whether the regions in \( L \) have a higher error than the regions on the global heap, as the max regions of the global heap now reside at the children of the GHC. Therefore, a \textit{global heap adjustment} must occur. This adjustment will result in the \( m \) regions on \( L \) being inserted into the global heap, the list \( L \) becoming empty, and the global heap being readjusted such that all nodes (including the root node) hold either \( \lceil \frac{n+m}{p} \rceil \) or \( \lfloor \frac{n+m}{p} \rfloor \) regions.
Secondly, the list $L$ may become very large. If it grows too large, then there will generally be too many regions at the root node, unbalancing the global heap, and it will be desirable to initiate a global adjustment. The threshold value is set (somewhat arbitrarily) at $\lfloor \frac{n}{p} \rfloor$. A value lower than this will work, but may result in too many costly global adjustments. At this value, $D$ can contain as many as $\lfloor \frac{n}{p} \rfloor$ regions; when combined with the maximum $L$ size, this results in a maximum of $\leq \lfloor \frac{n+n}{p} \rfloor$ regions at the root node, or, basically one "node's" extra regions. This global adjustment achieves the same results as when $D$ becomes empty: the $m$ regions in $L$ are inserted into the global heap and all nodes are re-balanced. Note that at the point when $L$ is too large, $D$ will generally contain fewer than $\lfloor \frac{n}{p} \rfloor$ regions, as some regions in $D$ will have been sent to an integral worker since the last global adjustment.

Note that neither type of adjustment is explicitly a global insertion or deletion. Rather, each adjustment takes the regions in $L$ and inserts them into the global heap and "re-fills" $D$ with new regions. If the sum of the regions in $L$ and $D$ is greater than $n/p$, then it can be considered an insertion, if that sum is less than $n/p$ then it can be considered a deletion.

The overall pattern of an adjustment is the same as with the synchronized global heap: the root node performs some calculations, requests regions from its children nodes, receives those regions, shuffles regions between its local deep and the received regions, and then sends the locally lowest error regions back down
to its children. The children receive those regions and initiate their own heap adjustment at the subtrees for which they are the root nodes.

The details of the number and pattern of regions sent up and down the tree to complete the global adjustment are complex and are basically the same as in the synchronized algorithm; the reader is referred to [23] for details.

If a worker is requesting multiple regions, and only part of the worker's request can be filled before a global adjustment, then the regions that can be immediately sent will be, so that the worker does not have to wait for the global heap adjustment to complete before receiving regions. After filling a partial request, the heap controller must complete its portion of the heap adjustment before being able to service any additional requests.

**Pattern of Errors Over Time**

Consider now the pattern of errors in the regions arriving at the GHC. Over time, the errors in the incoming regions should decrease as the total error estimate is gradually reduced. This pattern should ensure that over time, regions are taken from $D$ more often than they are taken from $L$. The global adjustments will have the net effect of shifting regions from $L$ into the global heap, gradually increasing the value of $n$. There may be times, though, when a sequence of incoming regions allows the GHC to serve many requests from $L$ without requiring a global adjustment. Increasing the global re-heapify threshold will result in a
slightly unbalanced global heap, but may decrease the total number of global adjustments.

**Initial Handling of Heap**

There were some problems in the initialization of the synchronized global heap due to the synchronous nature of the algorithm. These problems are resolved in the GHEAP algorithm. Initially, all of the nodes' local deaps will be empty, including the root node. The first few region requests will be handled solely by the root node; it will store each incoming region in its local deap and then retrieve maximum regions from the deap to send back to the worker. This technique is efficient, as there is no point in initiating a global heap adjustment when each node will receive only zero or one nodes from it. Global adjustments are delayed until \( n = p^2 \) regions are in the root's local deap. At that point, the first global adjustment is completed, after which each node holds exactly \( p \) regions. Until this first adjustment, all GHW's simply wait idle to receive their first global heap message. (Though recall that each of the corresponding MPI processes will not be waiting idle, as the process that performs the GHW work will also be acting as an IW.)
Single-Node Version

The GHEAP algorithm includes a single-node global heap version in which the GHC stores all the regions in a local heap and services all requests from integral workers by itself. A local heap is used (as opposed to a deap) as only the max region ever needs to be retrieved; the resulting speedup is of a constant factor, but can be substantial when the heap becomes very large. This single-node version is used whenever the value of \( p \) (a run-time parameter) selected by the user happens to be 1.

Details of the Global Heap Algorithm

This section presents the details of the global heap algorithm.

Because the algorithm is asynchronous, and because each GHW must also handle IW functionality, all of the message receive function calls must be non-blocking.

Due to this behavior, the algorithm is best presented not as pseudo-code, but as a flow through a state diagram. Each of the global heap controller and worker flows through a different pattern of states. It is the receipt of one of a particular state-dependent set of messages (considered the “preferred” messages for a state) that can transition the process into a different state. If a non-preferred message is received, it is generally ignored by the process. (Note that messages
are discovered to have arrived at a process via the non-blocking MPI probe function (MPI_Iprobe()) before they are actually received via the receive function MPI_Recv(); if a non-preferred message is discovered via the probe, it is actually not received, but left on the incoming message queue for receipt at a later time.)

Message Types for the Global Heap

Before the states can be presented, the possible message types must be explained (for any message that can be received from any global heap process). They are:

1. **TAG_IW2GHC_RGNS** This is the message that an IW uses to send regions to the GHC, and, implicitly, to request new regions.

2. **TAG_GH_P2C_START** This message is the first message that a worker receives; it tells it to soon expect the first set of regions for the initial global heap adjustment.

3. **TAG_GH_P2C_REQ** This is the message from a global heap parent to one of its child nodes requesting regions.

4. **TAG_GH_C2P_SEND** This is the reply message from a child to its parent, returning the regions that the parent requested.

5. **TAG_GH_P2C_SEND** This message contains regions and is sent from a parent node to a child node, telling the child that it should begin its own global heap adjustment on its own subtree.
6. **TAG_GH_INTG_STOP** This message is sent from the IC to all processes; it is the global stop message.

**State Flow for the GHW**

As the functionality of the GHW is simpler, it is presented first. The states are as follows:

1. **STATE-BEGIN** The worker begins in this state. From this state the worker expects to receive a **TAG_GH_P2C_START** message.

2. **STATE_BETWEEN** This is the “steady-state” state; the worker is in this state between global heap adjustments.

3. **STATE_WAIT_FOR_FOLKS** The worker is in this state whenever it expects to next receive regions from its parent.

4. **STATE_WAIT_FOR_KID1** In this state, the worker has requested regions from its children, and has not yet received any regions back.

5. **STATE_WAIT_FOR_KID2** In this state, the worker has requested regions from its children, and has received regions from one child, but not the other.

Figure 6 presents the state diagram for the global heap worker process. The diagram is simplified in that the **STOP** message is not shown; this message can be received at any point and will cause the process to begin cleaning up resources and then halt. This diagram also does not hold for the leaf nodes in the global heap; their state diagram is much simpler, as they only send regions to their parents and
receive regions in return. They never have to initiate their own global adjustment, for they are not the root of any non-empty subtree.
Note that the worker is done with a global heap adjustment after it has sent regions to its children. At any time after this, a new global heap adjustment can commence, regardless of whether or not the previous adjustment has been completed at the lower levels of the tree. I.e., the adjustments are pipelined, so that the tree can be in the process of completing multiple adjustments at any instant in time.

State Flow for the GHC

The states for the heap controller are similar to those of the heap workers:

1. STATE_LOCAL_HEAP This is the initial state for the controller; in this state, the controller has not yet received enough regions to initiate the first global adjustment, so the controller is still solely using its local heap to serve regions back to the integral workers.

2. STATE_BETWEEN As with the heap workers, this is the steady-state state for the controller; in this state the controller is not currently in the middle of any global adjustment (though previous global adjustments may still be progressing down the tree).

3. STATE_WAIT_4_KID1 In this state the controller has requested regions from its children, but has not yet received any reply.

4. STATE_WAIT_4_KID2 In this state the controller has requested regions from its children, and has received a reply from exactly one child.
Figure 7 shows the state diagram for the controller. Its additional complexity over the worker diagram is the result of additional logic needed to determine when a global adjustment should be started, and, how the current integral worker request should be filled. As with the worker diagram, the receipt and handling of a STOP message is not shown.

The complexity of the GHC state diagram requires some additional explanation. The legend for this diagram is as the same as in Figure 6. Flow begins at the `STATE_LOCAL_DEAP` node in the upper-left hand corner (abbreviated as `LOCAL_DEAP`). Execution loops around the small upper loop until the controller has received too many regions. At that point it exits the small loop, starts and finishes its portion of the initial adjustment, and then settles into the `BETWEEN` state.

Receipt of a message from an integral worker will send it down the diagram's left vertical path, where it tries to fill the request from its list $L$ and deap $D$. If $L$ becomes too full, or $D$ becomes empty, then it must begin a global adjustment, else it can just fill the request and return to the `BETWEEN` state. If the list $L$ is full, a flag variable `pending_svc_flag` is set to indicate that the integral worker is still waiting for service. The final vertical path on the right side of the diagram is the sequence used to complete a global adjustment, after which the controller returns to the `BETWEEN` state.
Figure 7. State Diagram for the GHC.
The global heap modification to the base code in PARINT1.0 required substantial modification to the IW, IC, and CAW code. Additional code needed to be brought in to handle the linked list and deap data structures.

A lot of the base code was able to be re-used. Little had to be changed to the upper-level code that begins the integration process. As a result, the user of the global heap version need not worry about the change in the algorithm they are using; they can simply run PARINT as before.

To modify the behavior of the global heap, two additional run-time parameters were added to PARINT. (The term run-time parameter is used to refer to any parameter that the user can change immediately before running PARINT; no re-compiling of the code is required to change these values). The user is able to change the global heap size and specify whether or not the process that is the IC should also be the GHC, or if the lowest ranked integral worker should be the GHC. These parameters can be changed on the PARINT executable command line or through functions in the PARINT API. No support for these parameters was added to the PARINT GUI; if the global heap design becomes part of a future PARINT release, then they will be added at that time.

The remaining aspect of the design / implementation that has not yet been explained is the method by which multiple functionalities were combined
into single MPI processes. This is explained in the next chapter.
CHAPTER VIII

GLOBAL HEAP IMPLEMENTATION DETAILS

Chapter VII presented the design of the global heap. Key to the design was the flexibility in assigning different tasks to different MPI processes. This chapter explains the technique used. To fully understand this chapter, the reader should be familiar with the writing of parallel programs in a message passing environment, though familiarity with MPI itself is not necessary.

Overview of the Implementation

The key idea of this chapter is the technique termed the Single Pass Message Loop (SPML). This is an implementation technique developed independently by the author, though it may be in use elsewhere (certainly, parts of the idea have been used, if only implicitly, see for example [14]). It is a technique for designing and coding any asynchronous parallel program in which a variety of message types must be received and handled, and, local work (work that is performed independent of the receipt of any message) must also be completed.

As is needed in the global heap implementation, the SPML allows for a single process to easily perform multiple roles, when normally each role might be
assigned to a separate process.

Terminology Used in This Chapter

Some of the terms used in this chapter must be carefully defined.

The term *message passing process* will be used to refer to the entire process (as the operating system views it) that is sending and receiving messages. This is a more general term than referring to a process as, e.g., an "MPI process".

The term *functionality* will be used to refer to a specific role played by a message passing process. For example, a process in PARINT may be an IC. If that process is also acting as the CAW, then that message passing process has two functionalities. A process will perform one or more functionalities.

Any time the functionalities are viewed separately from the processes, then one can consider the idea of a *mapping* from the processes to the functionalities, i.e., assigning one or more functionalities to each process.

The term *local work* is used in this chapter to refer to some specific kinds of work performed by a process. It applies only when the work is not done in response to any received message, but rather is work that the process can do regardless of which messages are received. For example, the IW process performs local work consisting of evaluating regions. The local work can include sending messages, but generally does not include receiving any messages.
Rationale for the Single Pass Message Loop

An alternative to the SPML is to actually create a separate process for each desired functionality. There are drawbacks to this approach.

Process Spawning Drawbacks

It may not be as flexible to have each functionality be a separate process. Completing this assignment requires some sophistication at the process spawning time. To spread multiple functionalities over multiple workstations may require multiple processes to be spawned on each workstation.

To do this in MPI, for example, each process must have its workstation and program to run specified in a text file. If a user wants to modify the mapping from functionalities to workstations, the process/workstation list has to be modified (edited), possibly automatically using shell scripts for novice users. In addition, if each functionality is implemented as a separate program, the processes have to be terminated and respawned again any time the mapping needs to be changed.

Process spawning time is generally very expensive. In addition, the MPI standard is not designed to handle the spawning of new processes from within existing processes.

The competing product PVM [18] can better control the spawning of processes (a process is easily able to create additional processes on other worksta-
tions), but it can still be time consuming.

In contrast, the SPML design allows a process to switch its functionality (within limits) while the program is running. For example, each time an integral is solved (within a single run of the GHEAP version of the PARINT executable) the functionalities performed by each process can be changed.

Control Over Interleaving of Execution

If the SPML technique is used then the programmer has control over the receiving of different kinds of messages and the amount of time spent in each functionality. Alternatively, if each functionality is assigned to a separate process, then the operating system controls the amount of time each functionality is given via the CPU scheduling algorithm. Granted, there may be times when the programmer does not care and prefers to have scheduling handled automatically.

Speed of Message Passing

If the SPML technique is used, then it is likely that one functionality will end up sending a message to another functionality even though a single process controls both those functionalities. (For example, the GHC may send a message containing regions to the lowest-ranked integral worker, yet the process that is that worker may also be acting as the GHC.) It is likely that a well designed message-passing implementation will optimize this kind of message passing.
For example, on a message send, the sending message buffer may be copied into a separate buffer within the same process (in the process's incoming message queue) to await receipt. The message contents will never leave the process's memory space.

If two message passing processes want to communicate on the same workstation, then at best the operating system's shared memory functionality will be used if not some slightly slower technique. Regardless, the operating system must manage the communication, and this will require context switching, hence slowing execution.

The fastest alternative is to custom design each possible combination of functionalities to work together as a separate message passing process. In such an implementation the different functionalities can be invoked via simple function calls, and the "message passing" between functionalities can be achieved through the programming language's parameter passing mechanism. The CAW/IC functionality combination uses this technique, as the CAW modification was completed before the SPML was developed. The CAW is invoked by the IC via a C function call; after performing a round of work, the CAW builds an update structure with the results (as do all workers) but the structure is passed directly back to the IC via a function parameter.

The drawback of this approach is the combinatorial explosion of combinations of functionalities. Consider the functionalities presented in this thesis; there
is a need for the following combinations: IC, IC/CAW, IC/GHC, IC/CAW/GHC, IW, IW/GHC, IW/GHW, etc.

However, it is clear that passing messages via function parameters is the fastest technique.

Terminology Used Within the Pseudocode

The next section begins to develop the SPML technique. It assumes the existence of a message passing programming environment. The messages are assumed to have message types, identified by message tags (usually represented by distinct integral values). Each process has a unique id or rank used to specify a message sender or destination. The pseudocode will assume the existence of several message passing primitives. They are:

1. msg_probe(tag, sender, &msg-type, &msg-sender) This function waits until a message of a particular type tag from a particular sender has been received into the process’es incoming message queue (specifying ANY_TAG or ANY_SENDER allows for the checking of any kind of tag or sender.) The parameters &msg-type and &msg-sender are filled in with the message’s type and sender if a message is found. The message is not actually retrieved.

2. msg_iprobe(tag, sender, &msg-type, &msg-sender) This function will test (without waiting) to see if a message of a particular type tag from sender has been received into the process’es incoming message queue. The param-
eters are the same as with msg_probe(). It will not actually receive the message. It returns TRUE if the message is there, FALSE otherwise.

3. msg_recv(tag, sender, &msg-type, &msg-sender) This function block-receives a message. If the message has been determined to already be on the incoming message queue (via a msg_iprobe() call) then it returns immediately. Otherwise, it will wait until the message arrives. The parameters are the same as with msg_iprobe().

4. msg_send(tag, receiver) This function sends a message of type tag to the process with id receiver. It locally-blocks, i.e., it blocks until the outgoing message queue has room for the message, but it does not wait until the destination process is ready to receive this particular message.

The functions above also need parameters containing the message itself. These parameters do not need to be considered in this chapter, so they are skipped.

Both PVM and MPI support these functions.

Single Pass Message Loop

The key behind the SPML is to develop a single, simple, organized program loop that controls all of the execution of a process (other than initialization and cleanup activities). And as the name suggests, this loop will receive at most a single message per loop iteration.

This approach is designed for programs that might receive several different
types of messages. The loop is designed to show no arbitrary preferences for the receiving of any particular type of message.

**Simple Example**

Consider the message loop (presented in a C-style pseudocode) in Figure 8. In this example it is assumed that the “handling” of one of the given message types (TAG1 and TAG2) will eventually set the boolean done variable and end the loop. A similar variable will be used in most of the examples in this chapter. This loop is shown with two distinct messages tags; there could be any number of tags.

```c
Message Loop:
while (!done)
{
    while (msg_iprobe(TAG1, ANY_SENDER, msg-type, msg-sender))
        handle-message-of-type-TAG1;
    while (msg_iprobe(TAG2, ANY_SENDER, msg-type, msg-sender))
        handle-message-of-type-TAG2;
}
```

Figure 8. Simple Message Loop.

There are several problems with this code. First of all, if too many messages arrive of a certain type, then the messages of the other type will not be handled. This may adversely affect the behavior of the program. In addition, the incoming message queue may become clogged with the type of message not being received, as long as a single message of the type being checked for is on the queue. Secondly, if
the process is supposed to terminate as the result of a STOP message its termination may be delayed because the STOP message may be waiting on the incoming message queue while the process handles other kinds of messages.

Figure 9 shows a SPML implementation of the code in Figure 8. This loop shows no preference for either type of message. It block-probes and receives whichever message is detected first. It receives a single message per loop iteration. Code can also be added to detect the receipt of an unexpected message (a message with an unknown tag).

```c
Single Pass Message Loop:
while (!done)
{
    msg_probe(ANY_TAG, ANY_SENDER, msg-type, msg-sender)
    if (msg-type == TAG1)
        handle-message-of-type-TAG1;
    else if (msg-type == TAG2)
        handle-message-of-type-TAG2;
}
```

Figure 9. Initial Single Pass Message Loop.

Mixing Message Receiving With Local Work

The previous examples were simple in that the process was only responsible for receiving and responding to messages. While the PARINT processes must do this, they also have to perform local work. As previously defined on page 69, this specifically means work that must be performed regardless of the receipt of
For example, the integral worker processes receive $\varepsilon_{global}$ messages, and may occasionally receive a message from the IC indicating the id of an idle worker. Regardless of the receipt of these messages, though, the worker must continue to perform rounds of work on the integration problem: retrieving regions from its heap, dividing, evaluating, and storing them back on the heap, and sending updates to the controller.

This is not work done in response to any message, and can be contrasted with the instance of when the worker receives a message indicating the id of an idle worker (as part of an instance of load balancing). The work it performs in response is: determine if it is idle or close to idle, prepare a region to send, and send the region to the worker with the given id. This is certainly work, but is not considered local work.

The design of the SPML is to prefer to receive a message rather than perform local work. There are several reasons behind this. First of all, the message to be received may be the global STOP message, and it is preferable to receive this message as soon as possible (indeed, receiving this message soon can directly lead to a reduction in braking loss).

Secondly, there is not assumed to be any predefined "end" to the local work. The process can perform local work indefinitely until some time when it receives a STOP message, or, it possibly determines independently that it is finished. But,
if it is continually busy performing local work, it will never probe for nor receive a
STOP message! The local work must be broken into pieces, between which it must
check for the receipt of a message.

Third, it is assumed that the work done in response to a message does
not take long to complete. Since the receipt of a message often indicates that
some other process is requesting a service of the receiving process, it is desirable
to complete that service as quickly as possible. Consider again an instance of
PARINT load balancing: If a worker receives the id of an idle worker, then that
explicitly means that another worker is idle. The receiving worker should not
make the idle worker wait while it performs one or more rounds of work; it should
respond as soon as possible so that the idle worker can get back to work solving
the integral. In fact, if we assume for a moment identical speeds of processing, we
can assume that every round of work done by one worker while another is idle is a
round of work that the idle worker could have done concurrently. And as already
discussed, it is important to receive STOP messages as soon as possible.

Figure 10 shows the SPML pseudo-code when there is local work to perform.
It is possible for a process to receive so many messages that it never gets around to
performing local work. This is considered a flaw in the algorithm's design. Even
if the local work were given preference, with that many messages arriving, it is
likely that the incoming message queue would fill up.

There are some additional considerations when identifying and coding the
local work. The local work should be divided into small pieces, as small as is reasonably possible. A small unit of local work will allow the process to quickly get back to receiving and processing messages. If the unit is too small, though, it may cause additional overhead in keeping track of partial results. In the GHEAP algorithm, the unit of work is a single round of the adaptive partitioning algorithm: dividing a region, evaluating the two new regions, forming the update, and sending a message to the IC and the GHC.

Also, the local work should not do any actions that block it from continuing. For example, it should not attempt to block-receive any messages. If it did, it might cause long delays before the process was able to receive any other messages. It is fine for the process to send a message as part of the local work, as sends are
assumed to be only locally blocking.

**State Information**

If a unit of local work requires a lot of computation, then it may be best to break that unit of work into even smaller pieces. Doing this may require storing state information between iterations of the SPML. The state information, stored in some variable, specifies which kind of work should be performed in the next iteration (or, alternately, what kind of work was performed in the previous iteration).

A convenient unit of work for the LLB algorithm is a single round of the adaptive partitioning algorithm. If we want to make the unit of local work smaller, we would need state information to record what step to perform next. The state information would record whether we should next do: the retrieving of a region, the dividing of a region, the first evaluation, the second evaluation, or the storing of the regions on the heap.

This control is performed within the block of code that implements the local work; generally the rest of the loop will not care about the state of the local work.
Multiple Functionalities in a Single Process

As previously presented, one of the key benefits of the SPML is the ease at which it can allow a single process to variably perform multiple functionalities. This design can now be shown.

In this section, the multiple functionalities will be named \( \text{func1} \), \( \text{func2} \), etc. The existence of several boolean variables is assumed, one per process, per functionality. Each boolean variable specifies whether or not a given process is to perform a given functionality. This example is from the point of view of the single executing process, so these variables will be named \( \text{i...am...func1} \), \( \text{i...am...func2} \), etc. For example, in the case of the GHEAP algorithm, a processor will have the variables \( \text{i...am_ghc} \), \( \text{i...am_ghw} \), \( \text{i...am_ic} \), etc., defined.

If there are multiple functionalities, then it is assumed that every incoming message "belongs" to, or can be handled by, a particular functionality. The design specifies that a single message handling routine (a message handler) be written for each functionality. This routine will be called when a message is found on the incoming message queue (i.e., when the \text{msg_iprobe()} call returns a TRUE value). The routine examines the type of the message and if it is one of the messages that is to be handled by that functionality, then the message is received, it is handled, and a TRUE value is returned to indicate that the message was handled. If the incoming message type does not belong to that functionality, then it is left on the
incoming message queue, and FALSE is returned. The sender of a message may also be used to indicate whether a particular functionality should receive or ignore a message.

The message types of an application need to be designed, across all functionalities, to handle this method of receiving messages.

The pseudocode for one of these receiving functions is shown in Figure 11 for func1, for two message types for this functionality. Figure 12 shows the use of several of these routines in a message loop (with no local work):

```
bool handler-routine-func1(msg-type, msg-sender):
{
  if (msg-type == FUNC1_TAG1)
  {
    handle-msg-tag1;
    return TRUE;
  }
  else if (msg-type == FUNC1_TAG2)
  {
    handle-msg-tag2;
    return TRUE;
  }
  return FALSE;
}
```

Figure 11. Message Handler Routine for a Single Pass Message Loop.

(The \texttt{?} operator is assumed to have the semantics of a C conditional logical conjunction operator.)

Note that if one handler function declines to handle a message, then the next handler function (based on the \texttt{i.am.xxx} variables) is called to handle the
message. If the processor does not have any of the flag variables set, then the error statement will be executed (this condition may be preferred to be checked using pre-processor controlled assertion statements, as if it occurs, it indicates a serious flaw in the program logic). If a message is found that no handler message will receive, then again the error statement will be executed. The large if statement can also be re-coded in a more succinct statement through the heavier use of the conditional logical operators. The displayed form of the function is clearer, though.

If local work needs to be performed, then the msg_probe() is turned into a msg_iprobe(), and the message handlers are checked only if a message is available, else, a round of local work is performed. If there are multiple functionalities that need to perform local work, then logic needs to be added to allow them to take turns.
In practice, it may be easier for a process to have a default functionality. For example, in the GHEAP algorithm, a process is either an IW or an IC; if it is an IW then it may also be a GHC or a GHW; if it is an IC then it may also be a CAW or a GHC. It is easier for the default functionality not to have a handler routine, rather, the message tag is checked and possibly handled in the body of the SPML itself before being optionally passed to a secondary handler routine. Local work performed by the default functionality can also be executed directly from the body of the loop.

Functionality States and Handling of Preferred Messages

A functionality may have to receive and send a sequence of several messages in order to complete a portion of its execution. After receiving a message, it cannot simply wait for the second message to arrive. It must exit the handler function and allow for possibly another handler function or a round of local work to execute next. Therefore, the functionality may have to store state information to know where it is in a sequence of execution.

For example, the GHC has state information which can tell it whether it is in the middle of a global heap adjustment or not, and if it is, where it is within the adjustment.

The existence of state information may complicate the SPML and the handler function. For example, if the GHC process is in the middle of an adjustment,
it may be "waiting" to receive regions from one or both of its children. The GHC is the functionality that receives and handles message from the IW's when they send and request regions (a TAG_IW2GHC_RGNS message). But if it is in the middle of an adjustment, it cannot receive this message, because it cannot reply to requests for regions until the adjustment has been completed.

The solution is to identify the functionalities and states where these conflicts exist, and identify the message types that must be received before other messages types. Then, preferred message routines can be written as needed for each functionality. These routines perform msg_iprobe's on the messages for which they are looking (by message type or sender), and if a looked-for message is found, it can be received and handled.

The pseudocode for this is not shown, as it can vary a lot depending upon the requirements. Generally, though, a single function can be called before the SPML calls its msg_probe() or msg_iprobe() function. This will call the preferred message routine for each functionality that has one, until one of them indicates that it received a message. If any message is received, then that information will be returned to the SPML so it knows whether to loop around and receive another message, or, possibly perform some local work.

If multiple functionalities have preferred message routines, then care must be taken to ensure that no functionality is given preference for receiving messages, or messages may become stuck on the incoming message queue. This can simply
be handled by changing the order in which the preferred message routines are called. In the GHEAP algorithm this is not a problem, as only the GHC or GHW functionalities have preferred messages, and these two functionalities cannot both be assigned to the same process.

Finally, if a process is going to be told to terminate via a STOP message then that message should probably always be specifically checked for (using a msg_iprobe() at the top of the SPML before any other message checking calls are performed).

Use of a SPML in the LLB Algorithm

The implementation of the LLB algorithm used (PARINT1.0) does not use a SPML. The primary loop in which it receives messages does have several smaller msg_iprobe() loops and the local work performed is always completed in units of the ns algorithm parameter.

This algorithm only has three functionalities: the IW, the IC, and the CAW. There are restrictions on how they are assigned to processes: the CAW can only be assigned to the process that is acting as the IC, and the IC and the IW cannot be assigned to the same process.

The experimental version of PARINT in [12] allowed for very large numbers of processes to solve multiple integrals in parallel by dividing the processes into groups. Each group consisted of an IC and one or more IW's. An overall Global
Controller (GC) controlled the assigning of integral problems to the groups (acting, in a sense, as an “integral server”). This multi-group design clearly shows the possibilities of using a SPML (though one was not used, as this work was completed before the ideas of the SPML were developed). If there are only a few groups, then the GC can probably simply be assigned to a process that also acts as an IC; if a message comes in to that process for the GC it will be handled quickly given the SPML design.

If there are a lot of groups, then contention for the GC’s attention may lead to experimenting with having the GC be the sole functionality for a process separate from any IC or IW responsibilities. This separation can easily be achieved and changed, even at run-time (e.g., between sets of integrals), by using the SPML.

A rough design has been completed for modifying the PARINT1.0 code to use a SPML; this change may be part of the next PARINT release. This design assumes that the process code has a default functionality of either an IW or an IC. Furthermore, since the CAW functionality is already implemented in a “local work” style (passing information back to the IC using function parameters rather than messages), it will probably be kept in that style (since the function parameters are faster than message passing, even within a single process).

The largest change will be at the IW, where a unit of local work will be a single round of the adaptive partitioning algorithm. State information will be kept to determine after which round of work an update should be sent to the IC.
It is hoped that by modifying the code in this fashion it will be easier to experiment with different local load-balancing techniques and other modifications.
CHAPTER IX

EXPERIMENTAL RESULTS

This chapter presents the experimental results, predominately comparing the GHEAP algorithm with the existing LLB algorithm.

Hypothesis

The primary hypothesis is that the global heap algorithm will be able to solve integrals in fewer region evaluations than the LLB algorithm, with the results getting better as the number of workers increases. It is also hypothesised that the global heap will perform iterations of work slower than the LLB algorithm. What is not clear is whether or not the reduction in the number of regions evaluated with the GHEAP algorithm will be great enough to offset the slower iteration rate and result in an overall faster algorithm.

Initial Results

A subset of the integrals and tolerances presented in Table 1 (on page 33) were used. Each function used was run on values of \( p \) (recall that \( p \) is the number of processes) going from 1 to 24; each run was performed 5 times and averaged.
For these initial runs, the global heap size was one and the GHC functionality was assigned to the same processor as the IC. Sample initial results are presented in Figure 13 though Figure 22.

Figure 13. GHEAP vs. LLB Data for *fcn1*.

**Poor Results**

Figure 16 shows the results for *fcn7*. They are not very good. What was hoped to be a large (possibly 50%) reduction in regions evaluated is only a small improvement, and, the improvement does not grow much at all as the number of processors increases to 24. Figure 14 and Figure 15 (showing results for *fcn2* and *fcn4* respectively), are other examples where the GHEAP algorithm shows only a
Figure 14. GHEAP vs. LLB Data for fcnn2.

slight improvement.

Figure 22 (fcnn22) has even worse results, as there appears to be no improvement at all when using the global heap. Similar results appear in Figures 17, 20, and 21.

In Figure 19, the GHEAP algorithm actually is worse (requires more regions) than the LLB algorithm; similar results are in Figure 13 and Figure 18.
Figure 15. GHEAP vs. LLB Data for $fcn_4$.

Explanation of Initial Results

The initially poor results are cause for concern, but present the challenge of explaining why they are poor.

More information was needed to analyze the results, so the next set of experiments were designed to get more data about what occurs during a single run of each algorithm. Each graph (Figure 23 through Figure 28, to be introduced formally after the parameters shown in these graphs are derived) shows the changing of what is termed the error ratio versus the number of regions evaluated to some intermediate point in the algorithm. Recall from Chapter II that the algorithm
will terminate (assuming that the function count limit is not reached) when the condition:

\[ E_a \leq \max\{\varepsilon_a, \varepsilon_r |Q|\} \]

is reached. In these runs, the \( \varepsilon_a \) value is zero, so that term can be removed. Rewriting and rearranging leads to termination when the condition

\[ E_r = \frac{E_a}{\varepsilon_r |Q|} \leq 1.0 \]
Figure 17. GHEAP vs. LLB Data for *fcn10*.

holds. The value $E_p$ is the error ratio; it provides a convenient, single value that is an estimate of how close we are to having an answer. The next figures specifically show the $\log_{10} E_p$ versus the iteration number (the log factor makes it easier to visualize the results, as the error ratio generally goes from a number in the billions very quickly down to a number that hovers near 1.0 for a while before dropping to be $\leq 1.0$). An error ratio of 1.0 corresponds to a $\log_{10} E_p$ value of 0.0. Note that while the integral is being solved the value of $Q$ (the current value of the answer to the integral) changes, so both the numerator and the denominator of the error ratio can change.

Figure 23 through Figure 28 show results for functions *fcn22* (at 4, 16, and
Figure 18. GHEAP vs. LLB Data for fcn14.

24 workers) and fcn18 (also at 4, 16, and 24 workers). These results were run with the same $\varepsilon_r$ values as the previous set of graphs, so a $y$-axis value of 0.0 is reached at an $x$-axis value that corresponds to the plotted $y$-axis value in the previous graphs (for the corresponding number of workers, also, the earlier graphs were averaged over multiple runs and are therefore more representative than the single runs in the error curve graphs). For example, in Figure 23 the error curve crosses the $x$-axis at $\approx 9500$ iterations; this corresponds to the data point in Figure 22 at $p = 4$ and number of regions equal to $\approx 9500$.

These figures make clear the behavior of the algorithms. In these runs, the global heap algorithm is able to initially reduce the error ratio quicker (in fewer
iterations) than the LLB algorithm. Consider Figure 23, where the global heap curve swoops down very smoothly, consistently, and swiftly, while the LLB curve does more meandering.

The differences are even more pronounced in the *fcn22* results at 16 and 24 workers. Consider that at 24 workers there are 24 local heaps; at any given iteration there are a lot of workers probably working on regions that have relatively low error estimates (regions that under the assumptions of Chapter V cannot reduce the error ratio much), resulting in the horizontal pieces of the error ratio curve. Then, some worker evaluates some region that is important, resulting in a large $\Delta E_p$ (change in $E_p$) decrease. Presumably, the GHEAP algorithm encountered the
same region earlier on, when that $\Delta E_p$ resulted in only a small downward tick in the error ratio curve. When the LLB algorithm evaluated that region, the $E_p$ was much lower, the $\Delta E_p$ value appeared relatively large, and resulted in the large vertical pieces of the LLB curve. In general, the failure of the LLB algorithm to select the highest error region at each iteration created the meandering of the LLB error ratio curve.

Note that these results are for *fcn22*, the function for which (in Figure 22) there was no improvement for the global heap. Figures 26, 27, and 28 show the error ratio curve for *fcn18*, the function for which the LLB algorithm outperformed the GHEAP algorithm.
Figure 21. GHEAP vs. LLB Data for fcn21.

These results are similar to the error curves for fcn22, but show more wandering by both algorithms. There are horizontal and vertical pieces to the error ratio curves for both algorithms.

It is not clear why the GHEAP algorithm would have such irregular behavior. Possibly this is a function where even the sequential run's error curve would wander more. It is also not clear why the GHEAP would be the clear victor at 4 and 24 workers, but would appear to be an even match for the LLB algorithm at 16 workers. The variation in the initial regions and in the load balancing behavior probably have some effect on the varying results. An avenue of future work is to explore in more details the behavior of the the GHEAP and LLB algorithms on
functions like \textit{fcn18} and \textit{fcn22}.

\textbf{Trail Ends of the Error Curve}

The explanation of the initial poor results lies in the far ends of the error curve. In all six error curve graphs the two lines converge as the algorithm is about to complete its work. The previous set of graphs showed a single data point corresponding to the number of iterations it took to complete the integration; this data point corresponds to the position on the error curve graphs where the two error ratio curves converge.
Figure 23. Single Run Error Ratio Curve for fcn22 and $p = 4$.

The trend over all of the GHEAP error ratio curves is that the curves level off over time. They do not continue their initial fast descent. The proposed rationale for this is that the initial fast descent is due to the regions in the worst portions of the initial region being repeatedly subdivided and evaluated. After a while, the error ratio has settled down to a lower value. The regions on the global heap generally then all have a lower error, and, each individual evaluation of a region does not greatly contribute to a decreasing error ratio.

The speculated, overall answer to the conundrum of these results is as follows: The GHEAP reduces the error ratio quicker, but ultimately the LLB algorithm
Figure 24. Single Run Error Ratio Curve for *fcn22* and *p* = 16.

thoroughly evaluates the same “crucial” regions as the GHEAP algorithm. It takes longer for the LLB to evaluate these crucial regions because they may be buried within a single worker’s local heap along with lots of other high-error regions.

After these crucial regions have been evaluated, both algorithms are faced with a vast but calm sea of regions, all generally having a low error estimate (but some of which are still important in reducing the error to below the tolerance). At this point the GHEAP algorithm’s benefit of being able to select the region with the overall lowest error for division and evaluation is of reduced benefit. It is speculated that at this point, the LLB’s method of choosing locally important
regions is no worse than the GHEAP's global region selection.

At this point in the execution, both algorithms have evaluated the crucial regions and both have evaluated enough non-critical regions to get their estimated error within the desired tolerance. Hence, both complete their run at approximately the same number of region evaluations.

More support for the idea of crucial regions comes from some previous experiments performed with $fcn7$ when the kill-heap feature was still used in PARINT [39]. It was noticed that with kill-heap on, it was common for the algorithm to get stuck with the estimated error at a particular value. No matter how
Figure 26. Single Run Error Ratio Curve for *fcn18* and *p* = 4.

many additional evaluations were performed, the error estimate barely moved. With keep-heap turned on, this never happened. The proposed explanation was that a crucial region was getting deleted when a worker's local heap was removed during an instance of load balancing, and without that region, the error was not able to be reduced below the error tolerance.

**Lower Error Tolerances**

It might be thought that by lowering the desired error tolerance, the algorithm would terminate while the crucial regions were still being evaluated, and
Figure 27. Single Run Error Ratio Curve for fcn18 and p = 16.

since the GHEAP algorithm does that quicker, it would get an answer quicker. Consider, however, the graph in Figure 29. This graph shows a single run for fcn22 with p = 16, similar to Figure 24, but in this case, the $\varepsilon_r$ is $10^{-6}$ (versus $\varepsilon_r = 10^{-11}$ in Figure 24).

Even though the value of $\varepsilon_r$ is larger, the curves still converge, and the GHEAP algorithm still gets an answer in about the same number of region evaluations as the LLB algorithm. The key lies in the makeup of the error ratio. In the latter run, the denominator of the error ratio was much larger (recall that $\varepsilon_r$ is a factor in the denominator of $E_\rho$). This implies that the change in the error ratio
from evaluating a given region will be much less than in the former run. Consider
the effect of evaluating a region in both runs, resulting in a change in the current
estimated error from $E_a$ to $E'_a$, and a change in the current result from $Q$ to $Q'$.
The change in error ratio for runs #1 and #2, $\Delta E_{p_1}$ and $\Delta E_{p_2}$, respectively, are
related as follows:

$$\Delta E_{p_2} = \left| \frac{E'_a}{10^{-6}|Q'|} - \frac{E_a}{10^{-6}|Q|} \right|$$
$$= 10^6 \left| \frac{E'_a}{|Q'|} - \frac{E_a}{|Q|} \right|$$

$$\Delta E_{p_1} = \left| \frac{E'_a}{10^{-11}|Q'|} - \frac{E_a}{10^{-11}|Q|} \right|$$
Figure 29. Single Run Error Ratio Curve for \textit{fcn22}, \( p = 16 \), and \( \varepsilon_r = 10^{-6} \).

\[
\begin{align*}
\Delta E' &= 10^{11} \left| \frac{E'_a}{|Q'|} - \frac{E_a}{|Q|} \right| \\
&> 10^6 \left| \frac{E'_a}{|Q'|} - \frac{E_a}{|Q|} \right| \\
&= \Delta E_{p2}
\end{align*}
\]

I.e., the effect of evaluating the region in Run \#2 is less than evaluating that same region in Run \#1. In fact, the net effect of evaluating any region in Run \#2 is less than when evaluating that same region in Run \#1. The importance of all regions is shifted, meaning that there will still be, in a relative sense, the same number of crucial regions, and the same large number of regions that are
not so crucial. So, regardless of the tolerances required, we can expect the GHEAP algorithm's error curve to initially have a fast descent, but always to level off.

Note that the same effect would occur if just an $\varepsilon_a$ value was used. In this case, the identical value of $\varepsilon_a$ to use for a given $\varepsilon_r$ would be $\varepsilon_a = \varepsilon_r|I|$, where $I$ is the actual answer to the integral. Consider the evaluation of a region again; the effect on the denominators of the error ratios would be the same, so the same $\Delta E_p$ effect would be seen. The only difference is that the $Q$ term is eliminated, so the $\varepsilon_{global}$ value does not change over time.

Figure 30 through Figure 32 verify these results for functions $fcn7$, $fcn18$, and $fcn22$, respectively. They show the results of running each function, at $p = 16$, across $\varepsilon_r$ values from $10^{-1}$ through $10^{-10}$. As is now expected, there is no benefit (in terms of number of regions evaluated) with using the global heap algorithm when requesting less accuracy (i.e., when $\varepsilon_r$ is larger).

Revisiting the Hypothesis

The original hypothesis was that the global heap would reduce the number of regions required to get an answer. The rationale was that by concentrating on the globally most important regions, the global error ratio would be reduced quicker, resulting in a final answer in fewer region evaluations.

It has now been revealed that the effect does result in the error being reduced quicker, but that the rate of reduction levels off before the final answer
is reached, and that the levelling off allows the LLB algorithm to catch up and generally match the GHEAP algorithm in the number of regions required to get the answer.

Note that the GHEAP algorithm suffers from the same problem as the LLB algorithm: the number of regions required to get an answer increases as $p$ increases. This aspect has not yet been explained. Regardless of the number of workers, all are accessing the global heap, and all should be always retrieving the most important region at every iteration. There are several possible explanations for this.
Non-Ideal Regions From the Global Heap

One problem is that workers are not actually retrieving from the global heap the globally most important region. At any given retrieval from the global heap, the extracted region is the most important over all the regions on the global heap. But, there are regions that are not currently stored on the global heap, specifically, all the regions that are currently being sent to the GHC for insertion on the heap, and all regions in the GHC’s incoming message queue. One or more of these regions may have a higher estimated error than the regions currently on the global heap.
Since the GHC has not yet received these regions, they cannot be considered as possible regions to be sent back to other workers. Therefore, the region sent back to a worker is not necessarily the most important region over all known regions. Furthermore, it is increasingly likely that the region a worker gets is not the most important as the total number of workers increases. (As the value of $p$ increases, the number of regions on their way to the GHC at any given instant increases.)

This appears to be a partial explanation for why the number of regions to get an answer increases in the GHEAP algorithm.
Effects of Singularities on Region Retrieval

An idea proposed in [15] is that a singularity in the region may have an adverse effect upon the retrieval of regions from the global heap.

As an example, suppose that the integrand function has a single point singularity in the region and that over the rest of the region the function is well-behaved.

At some point in the algorithm there will only be a single worker working on a region containing the point of the singularity. However, all of the workers will still be requesting regions from the global heap. Only a single worker on each iteration will get the region containing the singularity; the other workers get regions of lesser importance, possibly even regions that would not be evaluated by the sequential algorithm.

The problem is that the important regions can only be produced one at a time as the worker with the region containing the singularity performs a round of work. This does not affect the sequential algorithm, as it only ever does one round of work at a time.

As the number of potential important regions that can exist at one time increases (as in the case with several point singularities, or a line singularity, etc.), this effect should be reduced. This phenomenon has not yet been experimentally verified.
Domino Effect of Evaluating Incorrect Regions

An additional aspect may cause the increase in regions to get an answer, though this theory has not yet been tested. Consider the idea from Chapter V of considering the dividing of regions by the adaptive partitioning algorithm to be filling out a binary tree. The sequential algorithm is assumed to build a binary tree of the minimum number of nodes. As the number of processors used increases, the size of the binary tree increases. Consider the effects of dividing a region (in some parallel run) that would not have been divided by the sequential run. The immediate result is that an additional region is evaluated, resulting in a small additional amount of work, and (by previous assumptions), the reduction in the global error estimate is not as large as if a correct region had been evaluated.

But there are now two additional active regions (regions in the global heap or in a local heap) that never became active regions in the sequential run. It is now possible for them to become selected as well, after which their child regions could become selected, and so on. The effect of evaluating a single incorrect region can have a domino effect resulting in many other incorrect regions being evaluated. This is especially so if we are on the trail end of the error ratio curve, where the crucial regions have already been thoroughly evaluated, and, most regions are generally of a low error (and where slightly incorrect error estimations for a region may greatly mis-rank that region's importance against the other regions in the
priority queue(s)).

It seems that the domino effect will be worse for the LLB algorithm and will be affected by the locally-idle criteria and the load balancing strategy used.

Analysis of the Domino Effect

It would be nice to be able to analyze and verify the domino effect. This is rather difficult.

One possible analysis would take two binary tree representations of two executions of the adaptive partitioning algorithm, and compare them. It would find where a single incorrect region evaluation led to other incorrect region evaluations.

Consider, though, that the binary tree of regions may consist of upwards of \(10^5\) regions.

It is presumed that the sequential algorithm will have the minimum number of nodes on its tree. However, there may be other distinctly shaped trees of the same number of nodes, slightly more nodes, or even slightly fewer nodes (if some of the given assumptions do not always hold), that result in similar accuracy of answer. To simply say that a portion of a tree does not match the sequential tree does not provide a measure of the incorrectness of evaluating that portion of the tree.
Benefit of the Global Heap

The sole benefit of the global heap, then (within the context of considering
the number of regions to get an answer), is that it can initially reduce the error
ratio quicker than the LLB algorithm. As the next section discusses, there are
circumstances where this can be of great benefit, namely, when the function count
limit is reduced.

Lowering the Function Count Limit

All of the runs performed so far in this thesis had the function count
limit \( L \) set high enough that it was not reached, i.e., an answer to satisfactory
accuracy was found in fewer function evaluations than the function count limit.
If \( L \) had not been set this high, then it would not have been possible to measure
the total number of regions required to get an answer to a desired tolerance, as
that accurate an answer would not have been found.

There are practical situations where the total amount of work (or, similarly,
the total amount of time) spent on a problem needs to be limited. After that limit
has been reached, the answer found, to whatever estimated accuracy, is used.

To understand the effect on limiting the value of \( L \), reconsider Figure 24.
Limiting \( L \) to 325000 (correspond to limiting the number of regions to 5000, as
\textit{fcn22} is integrated using an integration rule that uses 65 function evaluations
per rule evaluation) has the effect of drawing a vertical line through the graph at \( x = 5000 \) region evaluations. This vertical line represents the halting point for this value of \( \mathcal{L} \). On this line, the GHEAP algorithm has reduced the log of the error ratio to 2.6 while the LLB algorithm has reduced it to only 8.2.

In a more concrete example, consider the actual output of such a run. Averaging over 5 runs, using \( fcn22 \) with \( \varepsilon_a = 0.0, \varepsilon_r = 10^{-11}, \mathcal{L} = 325000 \), the GHEAP algorithm gets an answer \( Q = 4.0000000000377 \), and an \( E_a = 1.8134419632370E-08 \); the LLB algorithm gets an answer of \( Q = 4.0000000940704 \), and an \( E_a = 3.1331609302382E-05 \). With an actual answer of \( I = 4.0 \), GHEAP has an actual error of \( 3.78E-11 \) and LLB has an error of \( 9.40E-08 \), a significant improvement when using the GHEAP algorithm.

Unfortunately the reason for limiting \( \mathcal{L} \) such that it will usually be reached is to get an answer in a limited amount of time. So far, we have not considered the execution time of the two algorithms. This is the subject of the next section.

Initial Timing Results

There are several methods and measures for getting experimental data concerning the execution timing of an implementation of the adaptive integration algorithm.

The easiest to consider is just to measure the total time that it takes for an answer to be found. As the previous sections indicate, different algorithms can
have a tremendous effect on the amount of work required (i.e., number of regions evaluated) to get an answer. If one algorithm requires twice as many regions to get an answer, comparing total execution times does not yield an accurate picture of the raw speed of an algorithm.

Therefore, an alternate measure can be used, that of measuring the time per basic unit of work. Regardless of how many units of work are performed, this measure gives an indication of how fast the algorithm executes. Within the PARINT project, a common unit of work used for timing is the time to complete 100,000 function evaluations (abbreviated \( t_{\text{per}} \)). This measure was chosen because the time to complete a single function evaluation is generally a very small fraction of a second for a modern workstation; by multiplying by 100,000 a value closer to 1 second is often obtained. Note that while this value explicitly refers to the time to evaluate functions, all of the overhead of the algorithm is also included, i.e., the \( t_{\text{per}} \) value for a run is the total time of the run, divided by the number of function evaluations, times 100,000.

**Expected Timing Results**

It is expected that the \( t_{\text{per}} \) for the global heap algorithm will be several times larger than that of the LLB algorithm due to all the communication overhead involved. In the GHEAP algorithm each round of work is preceded by the time to send a message to the GHC, and, have the reply sent back. In the LLB algorithm,
there is only the delay to send or receive messages between rounds of work; there is no waiting for a round-trip message delay.

In addition, it is expected the speedup as the number of workers increases to be less for the GHEAP algorithm, due to the increased contention at the GHC.

**Timing and Efficiency Results**

Figure 33 shows the result of running *fcn22* at 1 through 24 workers. The $t_{per}$ value (in seconds) is graphed versus $p$. The runs were done with $\varepsilon_d = 11$, $\varepsilon_a = 0.0$, and $L = 520000$ (corresponding to 8000 regions, ensuring that the limit was always reached). Each data point is the averaged value over 20 runs. Figure 34 contains the same data manipulated to show the efficiency versus $p$. In this graph the traditional definition of efficiency is used (efficiency defined as $t_1/(t_p p)$; recall that $t_1$ is the sequential time and $t_p$ is the parallel time of the same algorithm).

The LLB does a fairly good job at reducing the $t_{per}$ value as $p$ increases. Its lack of efficiency comes from imperfect load balancing. However, the GHEAP algorithm performs very poorly. In fact, the algorithm actually gets progressively slower (as shown in the *rising* time versus $p$ in Figure 33). This is due to the contention at the GHC as the number of integral workers increases, and, due to the large delay between each round of work. The next chapter discusses some modifications that would speed up the execution of the GHEAP algorithm. From these results, it is clear that these modifications are needed.
Multi-Node Global Heap

All of the results were run on a global heap of a single node. While the multi-node global heap was implemented and tested, no final experiments were performed on it.

From some initial results it was found that the number of regions to get an answer did not vary with global heap size.

It is anticipated that a very large number of regions will be stored more efficiently on a multi-node global heap.
Figure 34. GHEAP vs. LLB Efficiency Data for fcn22.
CHAPTER X

FUTURE WORK

This chapter discusses future directions of research, as well as proposed improvements to the global heap design.

Improvements to the Global Heap

The global heap implemented in this thesis can really be considered a "base" asynchronous implementation of the synchronized global heap of [23]. There are some obvious modifications that could be added which should improve the heap's performance. The following sections detail some of these improvements. Along with each improvement is a discussion of the estimated change in the performance characteristics that the improvement would bring.

Modify the Number of Regions Per Request

Currently workers request only a single region per message to the GHC. This could be modified to allow each worker to request \( n_{rpm} \) regions per message, i.e., allowing a variable "access granularity". The value of \( n_{rpm} \) could even vary across the workers, allowing faster workers to perform more rounds of work be-
between accessing the global heap. The number of regions sent back to the GHC would also change; if $n_{rpm}$ regions are requested, after evaluating all the regions, there will be $n_{rpm} + 1$ regions to send back.

The expected effect of this change would be to increase the speed (i.e., decrease the $t_{per}$ for the implementation), because multiple rounds of work would be able to be completed for each long delay caused by a request for more regions. Also, the contention at the GHC could be controlled by modifying $n_{rpm}$. The cost, though, would be a further increase in the number of region evaluations required to get an answer. If workers request multiple regions at a time, and then perform multiple rounds of work on those regions, then there is an increased chance that they are working on regions that are not of the highest priority over all known regions.

**Increase Work Between Requests**

Regardless of how many regions are requested in a message, it is possible to have each worker perform some arbitrary number of rounds of work, $n_{wbr}$ ("wbr" stands for "Work Between Requests"), between sending requests for more regions to the GHC. If $n_{rpm}$ regions were requested, then it is expected that $n_{wbr} \geq n_{rpm}$, and $n_{wbr} + 1$ regions would be sent back to the GHC.

The expected effect is the same as with adding $n_{rpm}$: the $t_{per}$ value would decrease, but the total number of regions evaluated to get an answer would in-
crease, as again the region being evaluated at any given round of work is less likely to be the globally most important region.

**Increase Work Between Updates**

This improvement is related to the previous two. Currently each worker sends an update to the IC after each round of work (so that each pair of region evaluations is followed by the sending of a message requesting more regions and the sending of a message to the IC containing an update. This could result in too many messages being sent to the IC. The solution is to add a parameter, \( n_{wbu} \) ("wbu" stands for "Work Between Updates"), specifying the number of rounds of work to be done between sending updates to the controller. The value of \( n_{wbu} \) can be independent of the values of \( n_{rpm} \) and \( n_{wbr} \), but, if a worker is about to wait for regions to come back from the GHC, it might as well send off an update during that time.

A further, but slight, decrease in \( t_{per} \) is again expected for this improvement, due to the ability to better manage contention at the IC.

Consider the cost of this improvement. At some point in the algorithm, a worker completes a round of work that decreases the global error estimate below the requested tolerances. However, the IC may have to wait longer than usual to get the update reporting this because it will not be sent until \( n_{wbu} \) rounds of work have been completed since the last update. The result is a small increase in the
number of regions to get the answer. This can be considered a form of braking loss, as introduced on page 36.

Continual Working

Currently, when a worker sends regions to be inserted into the global heap, it sends all of the regions on its heap to the GHC. It could hold back $n_{kih}$ regions ("kis" stands for "keep in heap") and continue working on those regions while it waits for fresh regions from the global heap. When those regions arrive, they would be added to the current heap of regions. After $n_{wbu}$ rounds of work were completed, the entire heap minus $n_{kis}$ regions would again be sent to the GHC for insertion into the global heap.

As with the previous proposed improvements, this would reduce the $t_{per}$ value for the algorithm, but would increase the number of regions required to get an answer.

Dynamic Allocation of Arrays

This improvement deals with a detail of the implementation. Currently the PARINT base code (the code on which PARINT1.0, the GHEAP implementation, and all experimental versions are based) considers the maximum values for the number of dimensions and the number of functions in a vector function to be compile-time constants, and, these values are to used declare arrays that store
one value per dimension or per function in a vector. The data structure that stores a region has two vectors based on the maximum number of functions in a vector and two vectors based on the maximum dimension value.

The default value for each of these parameters is 10, meaning that a non-vector function of 2 dimensions (the current minimum function dimensionality in PARINT 1.0) wastes an incredible amount of space per region structure.

This is a problem for the LLB version, as regions are stored in the local heap, and local memory should not be wasted. It is a greater problem for the GHEAP version, as not only are regions stored on local deaps, but large numbers of regions are constantly sent back and forth constantly as the algorithm is executed.

Making all these arrays be allocated dynamically would result in a small additional computational cost (for the memory allocation time), but would greatly reduce the size of the region-containing messages. This improvement is on a list of considered improvements for future PARINT releases.

The negative effects of this wasted memory space were reduced for this thesis by reducing the compile time constants for the number of functions in a vector to 1, and the maximum dimensionality to 6. These were the maximum values needed for the experiments used in this thesis. Having smaller maximum values reduced the maximum amount of wasted space.
Attempt to Keep Regions Local in the Global Heap

If an IW is also a GHW, then there is a chance that a region sent to be inserted by that IW will end up within that process'es local deap after the next global adjustment. If that is the case, then the region should never have to leave the memory space of the process. It can be "sent" to the GHW functionality and stored in its local deap. The GHW can then send a message to its parent node in the global heap and inform it that it now holds an extra region. This information can be used to rebalance the global heap at the next global adjustment.

Further, it seems that the probability that this occurs is high. The chance is $\frac{p}{p_{gh} \log_2 p_{gh}}$ (where $p$ is the total number of processes, $p_{gh}$ is the number of processes managing the global heap; the formula reflects the chance that an integral worker is also a heap worker, times the chance that the region fits into that workers level of the global heap). For example, if there are 3 nodes in the global heap, and there are 4 processes total, then the chance of any given node belonging to a process’es local deap is $3/8$.

This improvement would reduce the number of messages sent back and forth to the GHC, and would reduce the total number of global adjustments needed. Therefore, the $t_{per}$ value would go down. The costs include a possibly unbalanced global heap (i.e., the chance that the regions in the global heap are not evenly distributed across the nodes of the global heap). It seems likely that a method
could be devised for moving the global heap back towards perfect adjustment at each global adjustment, reducing the imbalance. The resulting increase in the complexities of the global adjustment sequence must also be considered a cost.

**Increasing the Global Heap Size**

It is hypothesized (but not yet rigorously tested) that a large number of regions is better managed (i.e., faster global heap access is provided) by a multi-node global heap. Correspondingly, as the number of regions in the global heap grows yet larger, it makes sense that an even larger number of heap nodes may better manage the regions.

The proposed improvement, then, is to allow the number of nodes in the global heap grow as the algorithm progresses and the number of regions to manage grows. At some point, when a threshold number of regions was reached, the leaf nodes of the heap would inform their as-yet-idle child nodes that they were now going to operate as the global heap leaf nodes, and then the previous leaf nodes would begin operating as internal heap nodes.

If the hypothesis concerning the relationship between the number of regions on the heap, the heap size, and the speed of accessing the heap is correct, then this should result in quicker heap access.
Optimizing Inter-Heap-Node Message Sizes

In the middle of a global heap adjustment, an internal node in the global heap forms a worst case estimate of the number of regions that it must request from its child nodes and the children each send up that many regions.

It is likely that a method could be devised to reduce the number of nodes sent from the current worst-case value. For example, the message containing regions could be broken into several messages. The parent node could make a partial request, and after receiving, examining, and shuffling regions around, make a request for further regions based on the range of region errors in the first request. In this manner a single large request could be broken into several requests whose sum total number of regions would be less.

It is not clear whether the increasing in the number of messages would be offset by the decrease in the total number of regions passed back and forth during a global adjustment.

Overall, the heap improvements suggested do nothing to reduce the efficiency, in terms of region evaluations, of the algorithm. Rather, they increase the speed of the algorithm. Since the primary goal of the GHEAP algorithm was to reduce the number of region evaluations, these improvements do not overall seem to be a good idea.
The results in Chapter IX were not what was expected when this thesis was begun. While the results were able to be explained, not all of these final explanations were able to be verified by experimental evidence.

A starting point for future research directions is therefore to devise methods for verifying these explanations. The explanations that need verification include the theory of the existence of "crucial" regions that need to be subdivided satisfactorily for the algorithm to get an adequate answer, and the corresponding idea that there exist many other regions that are somewhat interchangeable. Also, the idea of the "domino effect" of evaluating an improper region needs further examination.

All these ideas would benefit from a close examination of the binary tree view of the evaluated regions. Other areas of computer science could be surveyed for theories and results involving large $d$-ary trees in general, and possibly those techniques could be applied in this area.

Another possible avenue is to develop finer models of the behavior of the adaptive partitioning algorithm, and possibly even test them using simulations.

Further experimentation with additional types of integrands may also yield more clues to the GHEAP algorithm's behavior.

If the final conclusion of this thesis is that the global heap algorithm is not
effective for solving integration problems on a NOW, then that can serve to focus future improvements to the adaptive partitioning algorithm in other areas.

Initial discussions [41] have centered around the idea of improving the load balancing techniques of the LLB algorithm in an attempt to combine the initial fast decrease in the error ratio of the GHEAP algorithm with the lower $t_{\text{per}}$ values of the LLB algorithm. It is known that the LLB algorithm currently uses a very small amount of message-passing to implement the load balancing. Paying a slightly larger communication cost (i.e., passing more messages) in load balancing may allow for workers to have a greater assurance that the regions they are evaluating are globally important, though maybe not as globally important as with using a global heap.

And lastly, the initial problem of low efficiency in the adaptive partitioning algorithm has not been solved. If the goal is to improve the speedup of this algorithm to near-optimal, then this inefficiency must be reduced.
CHAPTER XI

CONCLUSIONS

The results in Chapter IX were not what was expected when this thesis was begun. It was assumed, without question, that the global heap would complete its work more efficiently than the LLB algorithm.

As a result, the primary benefit of this thesis was not to provide an improved algorithm for parallel numerical integration, but rather to extend the knowledge of the characteristics and behavior of the adaptive partitioning integration algorithm. Some of the experimental techniques used in this thesis had not been used before to analyze the PARINT algorithms; they can now easily be applied to future modifications to get better experimental data.

Finally, the knowledge gained through the completion of this research will aid greatly in finding future improvements to the PARINT system of parallel numerical integration.
BIBLIOGRAPHY


