Western Michigan University
ScholarWorks at WMU

Dissertations | Graduate College

12-2003

Load Sharing Strategies in Distributed Environments

Laurentiu Cucos
Western Michigan University

Follow this and additional works at: https://scholarworks.wmich.edu/dissertations

Part of the Philosophy Commons

Recommended Citation

Cucos, Laurentiu, "Load Sharing Strategies in Distributed Environments" (2003). *Dissertations*. 3156.
https://scholarworks.wmich.edu/dissertations/3156

This Dissertation-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 Dissertations by an authorized administrator of ScholarWorks at WMU. For more information, please contact wmu-scholarworks@wmich.edu.

LOAD SHARING STRATEGIES IN DISTRIBUTED ENVIRONMENTS

by

Laurentiu Cucos

A Disertation
Submitted to the
Faculty of The Graduate College
in partial fulfillment of the
requirements for the
Degree of Doctor of Philosophy
Department of Computer Science

Western Michigan University
Kalamazoo, Michigan
December 2003

LOAD SHARING STRATEGIES IN DISTRIBUTED ENVIRONMENTS

Laurentiu Cucos, Ph.D.

Western Michigan University, 2003

We investigate load sharing strategies in distributed systems based on work properties, environment and process organization. For various classes of problems we characterize the paradigms corresponding to applicable communication strategies. We develop suitable structure for the work, and work assignment techniques in order to maximize the efficiency. This research is based in part on work conducted on distributed numerical integration.

ACKNOWLEDGMENTS

I would like to offer my very special thanks to my dissertation advisor, Dr. Elise de Doncker. Your continuous support, guidance and patience made this entire work possible. It has been an honor working under your supervision.

I also thank the other members of my dissertation committee, Dr. Karlis Kaugars, and Dr. Alan Genz for reviewing my work and for providing invaluable feedback. Words are not enough to express how grateful I am to my parents for all the love and continuous support. Ultimately I would like to thank my wife and my son for their patience and understanding.

Laurentiu Cucos

Copyright by
Laurentiu Cucos
2003

TABLE OF CONTENTS

ACKNOWLEDGMENTS ii

LIST OF TABLES v

LIST OF FIGURES vi

1 Introduction 1

1.1 Properties of the Distributed Computing Environment 2

1.2 Properties of the Work 4

1.3 Inefficiency Factors in Distributed Computing 6

1.4 Metrics Overview 9

1.5 Survey of Literature 11

2 Problem Definition 14

3 Problems with a Known Number of Units and Known Unit Size (KnKs) 19

3.1 Parallel Files Distribution 19

3.2 Resource Allocation for Clusters 21

3.3 MaxFlow Algorithm 21

3.4 Efficient Parallel Files Distribution 22

3.5 Using MaxFlow to Find Cluster Allocation 24

4 Problems with an Unknown Number of Units - Known Unit Size (UnKs) 34

4.1 Abstract Problems 34

4.2 Numerical Integration 42

5 Problems with a Known Number of Units - Unknown Unit Size (KnUs) 50

5.1 Hierarchical Integration 50

5.2 Experimental Results 52

6 Problems with an Unknown Number of Units - Unknown Unit Size (UnUs) ... 58

6.1 Adaptive Integration 58

6.2 Further Remarks and Challenges 61

6.3 Adaptive Mesh Refinement 61

7 Speedup and Efficiency Analysis Based on Work Unit Boundaries for Master Slave paradigm 64

7.1 The Effect of Bounded/Unbounded Work Unit 64

7.2 Investigating the Bottleneck Effect 65

7.3 Overlapping Computation with Communication 67

7.4 Properties of the Communication System 72

8 Sharing Status Information 78

8.1 Computation Communication Interface 78

8.2 Efficiency Analysis when Using $fair_split$ Function 80

9 Conclusions 82

Appendix 84

A ”gRpas”, a Tool for Performance Testing and Analysis 84

Bibliography 95

LIST OF TABLES

4.1 Theoretical vs. Experimental Speedup for Problem 1 41

5.1 Genz Test Integrand Families 53

7.1 Round Trip Communication Time: Free Network 75

7.2 Round Trip Communication Time: Congested Network 76

7.3 Send Time: Master $\longrightarrow$ 29 Workers 76

LIST OF FIGURES

1.1 Classification of Inefficiency Factors 7

1.2 Inefficiency Factors Based on Ratio Work Unit Size/Workload 9

2.1 Work Structure Classification and Sample Applications 15

2.2 Probable Efficiency Function of Workload Information 16

2.3 Investigated Methods 17

3.1 Sample Problems 21

3.2 Ford and Fulkerson Method 22

3.3 Sample 1: MaxFlow Mapping 23

3.4 Sample 2: MaxFlow Mapping 24

3.5 Clusters for Resource Allocation Example 30

3.6 Communication Requirements for Resource Allocation Example 31

3.7 Flow Graph for Resource Allocation Example 33

4.1 Example of Work Structure 35

4.2 Single Cell Work Assignment 36

4.3 Example of Breaking Loss Effect for Problem 1 37

4.4 Row Distributed Work Split for $p = 4$ 37

4.5 Problem 2: Uniform Work Distribution in Heterogeneous Environment 38

4.6 Iteration Timing Algorithm 39

4.7 Monte Carlo: Speedup - Workload A 44
4.8 Monte Carlo: Speedup - Workload B 44
4.9 Monte Carlo: Total Function Evaluation Count - Workload B 45
4.10 Speedup for Finance Problem on Homogeneous System 46
4.11 Speedup for Student-T Distribution Problem on Homogeneous System 47
4.12 Speedup for Finance Problem on Heterogeneous System 49

5.1 Timing Results for $f_1(\vec{x}), \varepsilon_r = 10^{-9}$ 54
5.2 Timing Results for $f_2(\vec{x}), \varepsilon_r = 10^{-6}$ 54
5.3 Timing Results for $f_3(\vec{x}), \varepsilon_r = 10^{-4}$ 55
5.4 Timing Results for $f_4(\vec{x}), \varepsilon_r = 10^{-5}$ 55
5.5 Timing Results for $f_5(\vec{x}), \varepsilon_r = 10^{-3}$ 56
5.6 Timing Results for $f_6(\vec{x}), \varepsilon_r = 10^{-9}$ 56

6.1 Adaptive Integration Meta-Algorithm 59
6.2 Sample Integrand Function 62
6.3 Domain Subdivision Plot 63

7.1 Communication Time 64
7.2 Bottleneck at Controller 66
7.3 Communication Bottleneck Effect for 100 Work Units 67
7.4 Communication Bottleneck Effect for 1000 Work Units 68
7.5 Computation-Communication Overlap 69

7.6 Questions on: Work Estimation Function 70

7.7 Communication Overlapping Algorithm 72

7.8 Round Trip Communication Time 74

7.9 Timing Code for Round Trip Communication 75

7.10 Timing Code for Send Message 77

A.1 Tests Tree Structure 89

A.2 Individual Tests and Parameters 89

A.3 gRpas : Multiple Plots, Multiple Window 92

A.4 gRpas : Multiple Algorithm Plot 92

A.5 gRpas : Scalability (speedup) Plugin 93

CHAPTER 1

Introduction

An application is said to be executed in parallel when different portions of it are run concurrently on different functional units, processors or systems. The ratio of the sequential to the parallel runtime gives an indication of how efficient the parallel execution is. Ideally, the total execution time gets reduced proportionally with the number of processing units that are used. However, this does not often happen in practice. The main reasons for inefficiency are: idle processing units, and redundant and/or unnecessary computations.

To achieve efficiency in distributed computing, two major factors need to be investigated: the structure of the work to be performed and the characteristics of the environment that will carry the computation. In this section we present classifications of different types of work, different types of computing environments and various distributed computing factors that need to be taken into account. These will be used in later sections to guide our design of work assignment strategies for different problem characteristics.

Chapter 2 presents the problem to be addressed by this research. Considering the task to be computed in a distributed system as being characterized by two factors, the number of work units and the size of each work unit, we investigate how this information can affect the efficiency of the computation.

Chapters 3, 4, 5 and 6 contain representative test cases investigated in relation to the proposed problem. Chapter 3 presents two problems with known work structure [13], Chapter 4 presents a set of cases with known work units size but of an unknown num-

ber [12, 15, 21, 22, 65]. Chapter 5 introduces a problem with a known number of work units but of unknown size [17, 19], while Chapter 6 handles a problem where there is no information about the size of the workload [23].

Even though challenges exist regardless of the structure of the workload, in general, work units of an arbitrary size may constitute a more serious source of inefficiency than an unknown total number of units. In Chapter 7 we investigate how to approach the more difficult problems with an unknown work unit size, using a Master-Slave paradigm. Chapter 8 further presents an important aspect in finding an efficient computation for workloads of unknown work unit size, through the sharing of status information by the workers.

Chapter 9 summarizes this research and presents conclusions of our findings.

1.1 Properties of the Distributed Computing Environment

Michael Flynn [62] introduced a classification of various computer architectures based on notions of instructions and data stream, those are SISD, SIMD, MIMD and MISD. Intrinsic Parallel Computers are those that execute programs in MIMD mode (Multiple Instruction stream over Multiple Data stream).

1.1.1 Multiprocessors, Multicomputers

Regarding the memory location and accessibility, there are two major classes of parallel computers, namely, shared-memory multiprocessors and distributed-memory multicomputers. The major distinction between multiprocessors and multicomputers lies in memory accessibility and the mechanisms used for interprocess communication. Multicomputers are sometimes also called message-passing systems.

1.1.2 Homogeneous, Heterogeneous Systems

A cluster is said to be homogeneous when all CPUs are identical, and are connected by a symmetrical network. If these conditions are not satisfied, we have to consider different types of heterogeneity : architecture, data format, computational speed, machine and network load [26]. Heterogeneous parallel computing systems can range from a collection of MPP's (Massive Parallel Processors) or other supercomputers to clusters of various personal computers connected by a high speed network. The systems are usually programmed by message passing libraries such as PVM and MPI.

1.1.3 Dedicated vs. Shared Process Space

An important factor that affects performance is the number of applications that share the same resources. Sharing the process space increases the total runtime for each application; the delays introduced will have a random behavior. For a distributed application that has tightly coupled work units, this will significantly reduce the overall system efficiency. The delay of one unit will propagate to all other depending on it.

1.1.4 Grid Computing

A computational grid is a type of parallel and distributed system that enables sharing, selection, and aggregation of resources distributed across multiple administrative domains, based on their availability, performance and cost [6].

1.2 Properties of the Work

There are a number of issues that need to be addressed in determining the size, structure and origin of the work units assigned in a distributed application.

1.2.1 Total Amount of Work

For some applications we can estimate the total amount of work required to solve a certain problem. This information can be very useful in splitting the work among the participating processing units. Examples are matrix multiplication, direct methods for solving a large system of linear equations, etc. For other applications we cannot estimate the total amount of work required to solve a certain problem. In numerical integration, for example, it is very difficult to predict the total amount of work required for an arbitrary integrand function. Other problems with an unknown total amount of work include: branch and bound, and adaptive mesh refinement.

1.2.2 Structure of Task/Work Units

Work units can be very big or can be very small. Either one of these extremes can degrade performance. For irregular problems it is difficult to identify useful work units of at least a certain size. Spending additional computation time to identify good size work units may be beneficial.

With respect to the form of the overall work, we can identify fixed or variable size work units. The *fixed* form corresponds to non-adaptive methods such as performing a number of predetermined iteration steps. In the *variable* form, as in the case of adaptive integration methods, there is no a-priori outline of the work. This type of strategy adapts itself to the

problem at hand and concentrates the evaluation points in the areas of concern, consisting of subregions of high estimated error for the local cubature rule approximations.

For those problems where we can control the task size, such as for parallel Quasi-Monte Carlo, or Monte Carlo integration, we can make the following classification:

- **Worker-independent**. The task size is independent of the worker power.
  - **Time-constant**. The task size remains constant throughout the computation.
  - **Time-variable**. The task size (though predetermined), changes as the computation advances, e.g., in the assignment of QMC randomized sample evaluations as the work units [21]. Their increasing size may introduce significant breaking loss [12] or communication clutter. Uniform assignment can be applied effectively for homogeneous systems. In heterogeneous environments, slower workers will process less work than faster ones.
- **Worker-dependent**. The size of the work unit depends on the worker power.
  - **Static assignment**. Each worker receives an amount of work proportional with its processing power. However, it is difficult to estimate the worker's power accurately.

    Therefore we will devise a dynamic assignment based on processing time.
  - **Dynamic assignment**. The work unit does not have a specific size but a specific processing time. Each worker processes as much as it can in the specified time interval. This handles most of the problems introduced by heterogeneity in the

system. In this category, we should be able to partition the work into arbitrary chunk size.

1.2.3 Process Origin or Assignment

The work generation can be *centralized*, as in the case where one or possibly multiple controllers create parametrized work units and assign them to the workers. Or, the work can be generated *locally* by each worker, as in adaptive task partitioning, where the partitioning of a task yields subtasks that can be distributed among the other workers. For either class of methods, a measure of task importance is required as it dictates process load. If the tasks do not have any dependencies among each other, they can be loaded into a priority heap.

Very large problems may benefit from a combination of centralized and local work generation. On a large cluster, it is common practice to schedule problems on a number of subclusters simultaneously. Furthermore, with the current support for grid computing it has become feasible to distribute the computation of large sets of tasks over multiple sites.

1.3 Inefficiency Factors in Distributed Computing

This section presents an overview of inefficiency factors that occur in distributed computing. Figure 1.1 shows their hierarchical classification.

1.3.1 Communication Time

Traditionally, the most cited factor of inefficiency for distributed systems is communication time. This is the sum of Prepare Communication Time, Data Transfer Time and Synchronization Time.

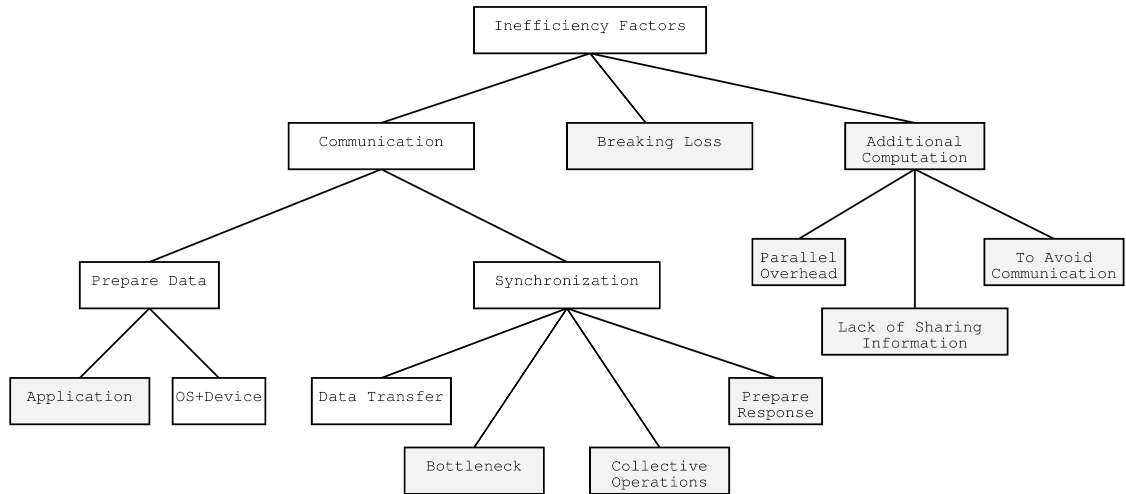


Figure 1.1 Classification of Inefficiency Factors

Prepare Communication Time. This is the sum, over all participating processors, of the total time spent by each worker to prepare the data for communication. In general this factor is not "very visible", as it involves a small fraction of all communication time.

Synchronization Time. Often a processor needs to gather data from other processors. The synchronization time is the total idle time between the moment a request was sent until an answer is received.

Collective Operations. This represents the additional time due to communication involving more than two processors. A few examples of global operations are Broadcast, (All)Reduce, (All)Gather. The main source of inefficiency in collective operations is the waiting time for all the participating processors to reach the communication entry point.

Data Transfer Time. This is the total required time to physically transport the data from one point to another. In general this factor can be eliminated (at least theoretically) by overlapping the communication with computation.

Communication Bottleneck. Especially for load balancing strategies, the majority of the communication time is used for sending updates with either results and/or load information. A communication bottleneck occurs when for one worker, the number of messages received is greater than the number of messages processed. To reduce the total number of messages received by a particular processor, we can either change the structure of the work, or redistribute the communication among more processors.

1.3.2 Breaking and Starvation Loss

Breaking loss is loss in parallelism caused by processors doing useless work towards the end of the computation. This is actually replaced by *starvation loss*, where processors are idle waiting for work, if the computation proceeds to the last possible work unit.

These factors are significant when the total work size is unknown, and the work unit size is relatively big. Let us denote: $w_i(t)$ for the work unit size of worker i at time t, n for the total number of workers, and W for the total work size. A sufficient condition for breaking loss to occur is that $\sum_{i=1}^{n} w_i(t_f)$ (where t_f represents termination time) is large (compared with W). When termination is signaled, some workers' results are not going to be considered, hence their work is useless.

1.3.3 Additional Computation

This represents the redundant, or unnecessary computation performed due to a lack of proper synchronization, to avoid more expensive communication, or due to the parallel logic overhead.

Using the same number of processors, Figure 1.2 shows that for larger work unit size

Figure 1.2 Inefficiency Factors Based on Ratio Work Unit Size/Workload

(relative to the total workload size) the main factor of inefficiency is Breaking Loss, while for very small work units it is Communication.

1.4 Metrics Overview

If we denote the execution time of the fastest sequential algorithm by T_0 and the parallel execution time on n processors by T_n, the *speedup* can be defined as $S_n = T_0/T_n$, and the *efficiency* is $E_n = S_n/n$.

One of the most common criteria to measure the efficiency of a parallel computation is *Scalability*, which refers to the ability of the parallel method to maintain efficiency as the number of processors increases.

Isoefficiency is the rate at which the problem size must increase with respect to the number of processors in order to keep the efficiency fixed. Considering the total work W

distributed over n processors, let us define $H_0(W, n)$ to be the total parallel overhead, i.e., $H_0(W, n) = nT(n) - W$. This is the difference between W units of time spent doing useful work and the cost of the parallel time $nT(n)$ for all the processors. To maintain constant efficiency E, the following must be satisfied: $W = KH_0(W, n)$, where $K = \frac{E}{1-E}$. The isoefficiency function $W(n, K)$ is obtained by extracting W as a function of n and K from the previous equation [42].

In the present work we used a definition of speedup where T_0 is considered the execution time of a parallel program when running with only one worker.

Most of our algorithms conform to the master-slave paradigm. In our scalability studies, we compute the speedup based on the number of workers involved and not the number of CPUs.

In a simple approach, the controller can be set up to perform useful work while idle between message processing. This strategy may introduce significant performance degradation if there are workers waiting for the controller to finish its computation before processing their messages. We will report our results with the controller not working, thus speedup plots are computed with T_0 generated by the parallel application executed with two CPUs: one controller and one worker. In a more complex solution, the controller may perform useful work in a separate thread or process.

Other issues include that the experimental results may be influenced by dynamic factors in the environment (network, operating system, system load). For this reason we repeat the same experiment a number of times and take the median of the run times.

1.5 Survey of Literature

1.5.1 Load Balancing Strategies

In [63], Xu at al. compare six work distribution strategies on two parallel systems for branch-and-bound computations. Three of the strategies are based on random allocation; the other three are based on local averages. They observed that the average strategies outperform the randomized allocation.

In general, load balancing strategies have been studied extensively for homogeneous systems. These strategies do not scale well in heterogeneous settings without modifications. The computational power of the nodes and the network bandwidth are not well known in advance; moreover they can change unpredictably at runtime. In [24], a general mechanism for converting scalable homogeneous algorithms to heterogeneous strategies is presented. The approach is to constantly monitor the system for the work unit processing time and the average communication turnaround time. Our work on distributed quasi-Monte Carlo algorithms takes possible heterogeneity of the environment into account [12].

For a-priori known work load, in the form of loops with variable running times, the iterations are executed in chunks of decreasing size, so that earlier larger chunks have relatively little overhead, and their non-uniformity in size can be smoothed over by later smaller chunks [58]. This method also known as *factoring* has been shown to be a robust and highly efficient technique on homogeneous shared address-space multiprocessors. In a *weighted factoring* method [38], the workers are dynamically assigned decreasing size chunks of iterations in proportion to their processing speed. In *adaptive weighted factoring* [2], the weight values are adapted after each iteration in the computation. Zheng [67],

presents the *adaptive weighted fractiling* method in a an N-Body simulation framework. Fractiling is a dynamic scheduling method that exploits the self-similarities properties of fractals to maintain data locality, and balances work load using factoring.

The architecture adaptive algorithms introduced in [41] provide a set of concepts and tools for writing portable parallel programs that run efficiently on a broad range of target machines. By taking into account information about processors, communication channels, memory hierarchies, etc., a portable algorithm can theoretically adapt its behavior to environment characteristics in order to increase its efficiency.

1.5.2 Tools on Distributed Systems

The **Portable Parallel Branch-and-Bound Library** [61] contains a set of tools to develop and compare load balancing algorithms. The library takes over the management of the subproblems which are created during execution of the Branch-and-Bound algorithm. The user has the option to use one of the library's application independent load balancers, or to implement a customized one.

LBsim is a software tool that simulates various existing load balancing strategies for data parallel applications in a distributed memory environment [40]. The authors provides various data sets of different distributions. Each data set contains N loop iterations, and each loop iteration has a variable running time. To model system-induced load imbalance, the workers are intentionally loaded with additional processes. The simulator is written in C and MPI.

DRAMA is a library for parallel dynamic load balancing of finite element applications [45]. The method used to restore the load balance is dynamic mesh re-partitioning. To accomplish this goal, there are two approaches: a) using a suitable sequence of local decisions, the current partition is updated by migrating parts of the mesh to another neighboring partition; b) by exploiting a global view of the partitioning problem, parallel static mesh partitioners are able to find a nearly optimal solution.

CHAPTER 2

Problem Definition

The present work addresses the following questions pertaining to the efficiency of distributed computations on a network of workstations.

How can information about the work structure improve the efficiency of a parallel computation? Having this information, how can we obtain a more efficient computation? How much do we need to know? We will consider a number of cases, and investigate how the information about the work structure can affect and/or predict the efficiency.

Assuming the work can be split into a number of units, the information about the total number or the size of the units can be estimated with certain accuracy. In some cases we can consider the size of the communicated data to be known as well.

Figure 2.1 shows the types of problems to be considered grouped based on the accuracy of predicting the total work. The X axis shows the probability of correctly predicting the number of work units while the Y axis shows the probability of correctly predicting the work units size.

Figure 2.2 depicts the probable efficiency, as a function of the two measures. The plot assumes solving problems of different characteristics on the same system with the same number of processors. It is highly probable to obtain a very efficient computation when the total amount of work, and the size of the work units are known. On the contrary, when we do not have any information about the size or structure of the work, the computation may be very inefficient.

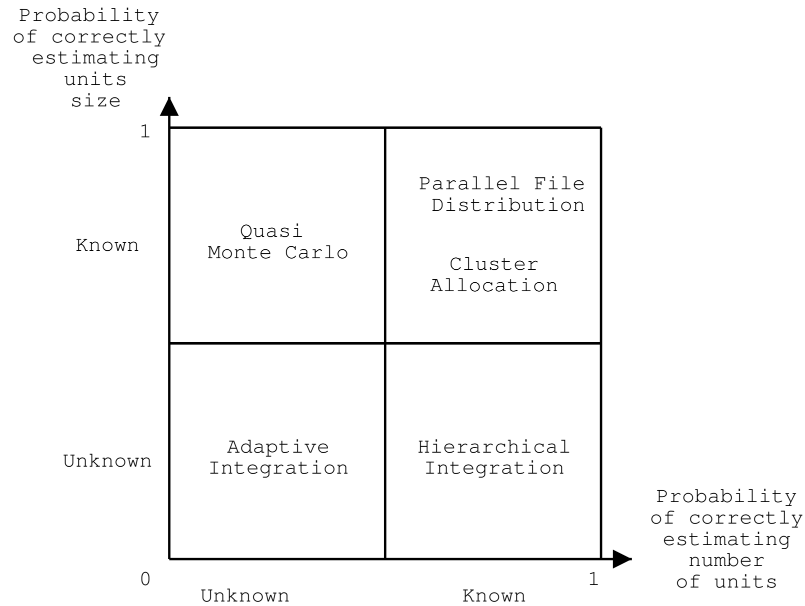


Figure 2.1 Work Structure Classification and Sample Applications

Figure 2.3 shows a set of methods that we will apply to investigate or generate efficient computations. In general, when the unit size is unknown, the task is more difficult. In this case it may be beneficial to introduce additional work to gather this information, in order to reduce the problem to the known category. Overall, having information about the work unit size has proved more useful than having information about the total number of work units.

Chapter 3 will investigate problems with a Known number of units and Known units size (KnKs). This includes parallel files distribution, which is important as parallel executions often produce and require re-distribution of large amounts of data on input and output. This chapter also handles a resource allocation problem [13]. The MaxFlow algorithm is used.

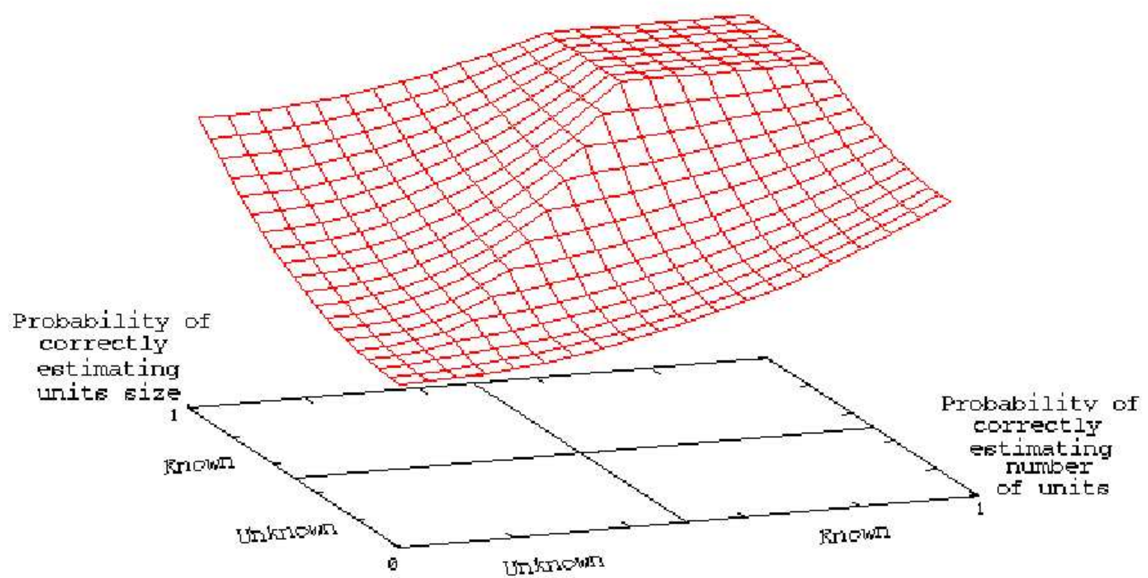


Figure 2.2 Probable Efficiency Function of Workload Information

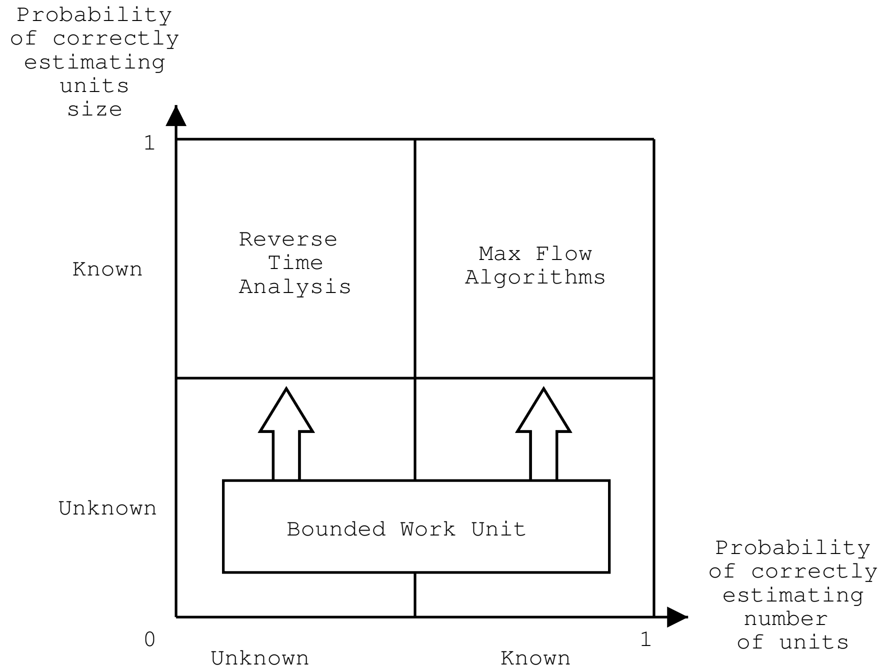


Figure 2.3 Investigated Methods

Problems with an Unknown number of work units and Known unit size (UnKs) include Monte Carlo and quasi-Monte Carlo integration, as outlined in Chapter 4 [12, 15, 22, 14, 65].

Hierarchical integration belongs to the Known number of units and Unknown unit size (KnUs) category, and is presented in Chapter 6 [17]. Distributed adaptive integration with load balancing is given as a problem with Unknown number of units and Unknown unit size (UnUs) in Chapter 6 [23, 65].

Chapter 7 gives a detailed analysis of the effects of bounded vs. unbounded work unit size. Chapter 8 deals with status information sharing and analyzes the effects of large tasks on parallel efficiency.

To help the investigation, we developed *gRpas*, a tool to gather analyze and store tests

results. Part of publication [11], Appendix A presents an overview of the application. Throughout the document most of the plots are either generated directly by gRpas (as in Figures 4.7, 4.8, 4.9), or contain data collected by it (e.g. Figures 5.1, 5.2, 5.3).

CHAPTER 3

Problems with a Known Number of Units and Known Unit Size (KnKs)

In this chapter we investigate two types of problems for which we know the total number of work units and the size of each work unit, *Parallel Files Distribution* and *Resource Allocation for Clusters*.

3.1 Parallel Files Distribution

We propose to devise an efficient and flexible mechanism to distribute data from multiple files to multiple processors/machines and vice-versa.

Often, a large data set is composed of a number of items distributed over multiple files. In parallel computations, each participating machine will write its own file, and for subsequent parallel processing the data must remain distributed over multiple machines.

There are a number of reasons to incorporate the file distribution in parallel programs, to name a few:

- using load balancing for uniform and efficient distribution;

- compactness (in the same program we can read, process and save files);

- a single machine cannot hold all the required data;

- standard filesystems usually hide the location of the files.

For reading and distributing data, we make the followings assumptions:

- the data is composed of a known, large number of items;
- the data files are located on machines that participate in the computation;
- there are no dependencies among items, or items and machines;
- the system is homogeneous;
- there is an arbitrary topology of shared filesystems.

Note: Even though a file is shared between a number of machines, it is physically located on one of them. We do not consider the differences in access time between machines sharing the same file.

3.1.1 Static Item Distribution

To avoid the synchronization overhead in homogeneous systems, and since the total amount of work is known, it is more efficient to compute in advance the item distribution in advance. In general, we can formulate the following problem.

Problem: Given a group of n machines, M = $\{m_i\}_{1\leq i\leq n}$, a collection of p sets, $S = \{S_j\}_{1\leq j\leq p}$, each having $|S_j|$ items, and a unidirectional bipartite graph $G = \{(M, S), E\}$ where $E = \{(v_i, u_j)\}$ with $v_i \in M$ and $u_j \in S$, which defines a mapping between machines and sets, compute the item distribution so that each machine gets an equal share of items. In case that an equal distribution cannot be achieved, find a solution with items shipped between machines at a minimum communication cost.

Figure 3.1 shows two examples with 4 machines, 3 sets, and different file sharing configurations. While for simple topologies (see Sample 1), it may be easy to identify an equal item distribution, the general case may be more challenging.

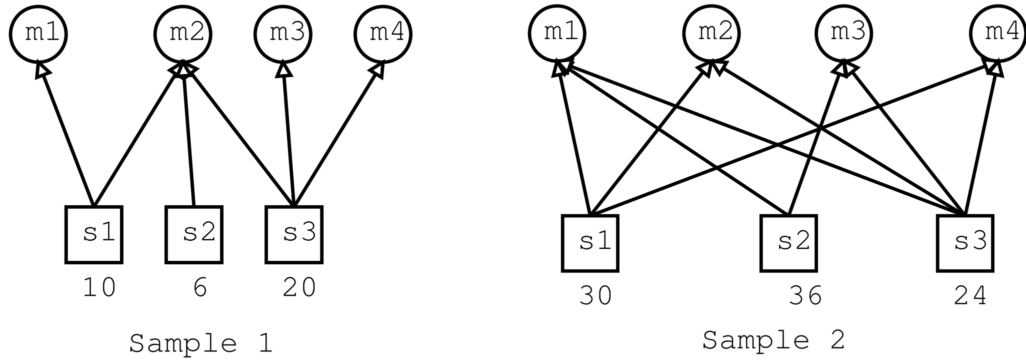


Figure 3.1 Sample Problems

3.2 Resource Allocation for Clusters

For this problem we consider the mapping of processes to a network with hierarchical structure, where the processors are grouped in clusters which may have different intracluster bandwidth. This work is part of publication [13]. We combine the communication requirements of the processes with the bandwidth characteristics of the target network by minimizing the total transfer time.

3.3 MaxFlow Algorithm

Given a directed graph with a non-negative capacity on each edge, having two special nodes: *source* and *sink*, the *flow* is a function that assigns to each edge a number between 0 and the capacity so that for all nodes but the sink and source, the total incident flow is equal to the total emerging flow. The MaxFlow algorithm computes the maximum value of the flow from the *source* to the *sink*. The total flow value for G is equal to the total flow emerging from the source.

An *augmenting path* is a path from the source to the sink along which we can push

```
Ford_Fulkerson_Method(G,s,t)
initialize flow F to 0
while there exists an augmenting path P
     do augment flow F along P
return F
```

Figure 3.2 Ford and Fulkerson Method

more flow.

The *residual capacity* $c_f(u, v)$ of an edge $e(u, v)$ is $c_f(u, v) = c(u, v) - f(u, v)$, where $c(u, v)$ is the edge capacity, and $f(u, v)$ is the current flow.

$G_f = (V, E_f)$ where $E_f = \{e(u, v) \text{ with } c_f(u, v) > 0\}$ is a *residual network*.

Theorem: MaxFlow MinCut: [8] If f is a flow in a flow network $G = (V, E)$ with source s and sink t, then the following conditions are equivalent:

1. f is a maximum flow in G.

2. The residual network G_f contains no augumenting paths.

3. $|f| = c(S, T)$ for some cut (S, T) of G.

The order of complexity is $\mathcal{O}(|f| \cdot |E|)$. The Edmonds-Karp algorithm differs from the original Ford-Fulkerson algorithm in that, at each iteration, it chooses an augumenting path with the smallest number of edges, leading to an order of complexity $\mathcal{O}(|V| \cdot |E|^2)$ for the algorithm [32].

3.4 Efficient Parallel Files Distribution

An efficient distribution can be achieved using a MaxFlow algorithm and the following procedure.

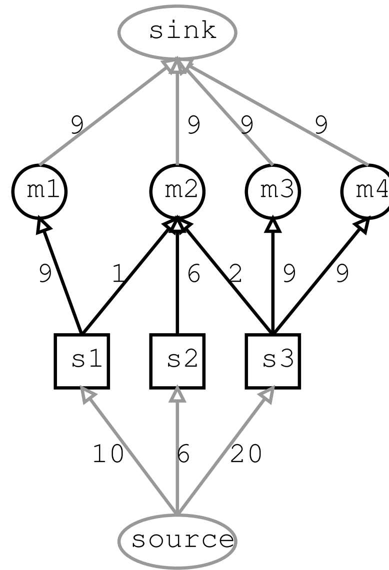


Figure 3.3 Sample 1: MaxFlow Mapping

1) Augment the graph with two new nodes: *source* and *sink*, and edges between all set-nodes and source, and all machine-nodes and sink, see Figure 3.3

2) Assign infinite capacity for edges between sets and machines, $|S_i|$ between set S_i and source, and $\lceil \frac{\sum_{i=1}^{p} |S_i|}{n} \rceil$ between each machine and the sink.

3) Apply the MaxFlow algorithm to the newly created graph.

4) If the resulting flow is $\sum_{i=1}^{p} |S_i|$, there exists an equal distribution of the sets. The distribution is given by the flow in the inner edges;

otherwise, the sets cannot be equally distributed without additional communication and go to step 5.

5) Equally distribute the items not allocated in step 3) to machines with a smaller number of items.

Figure 3.4 shows the item distribution for Sample 2 computed using the MaxFlow al-

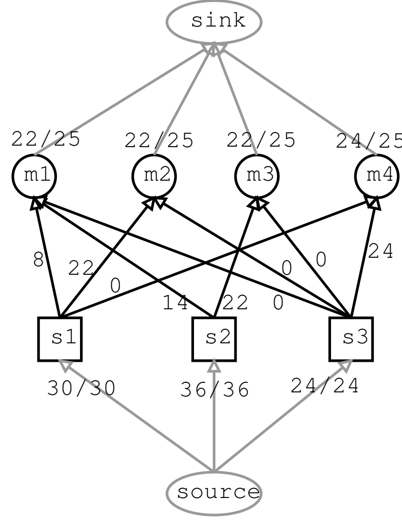


Figure 3.4 Sample 2: MaxFlow Mapping

gorithm. In this case, the solution generates an equal distribution without requiring synchronization.

3.5 Using MaxFlow to Find Cluster Allocation

Assume there are p processes to be assigned, and κ clusters. Let $\beta_{\ell_1 \ell_2}$ be the required communication traffic (# bytes) between processes ℓ_1 and ℓ_2. Let τ^{ij} be the transfer time per byte between clusters i and j. Then the communication time between processes ℓ_1 and ℓ_2 is $\beta_{\ell_1 \ell_2}\tau^{ij}$ if processes ℓ_1 and ℓ_2 reside in clusters i and j, respectively. Define indicator variable $x_\ell^j = 1$ if process ℓ belongs to cluster j and $x_\ell^j = 0$ otherwise, for $\ell = 1, 2, \ldots, p$ and $j = 1, 2, \ldots, \kappa$, and let $\bar{x}_\ell^j =$ (not)x_ℓ^j. The total transfer time to be minimized is $t(x) = \sum_i \sum_j \sum_{\ell_1} \sum_{\ell_2} \beta_{\ell_1 \ell_2} \tau^{ij} x_{\ell_1}^i x_{\ell_2}^j$.

We denote the intra-cluster transfer time per byte within cluster j by $\tau^j = \tau^{jj}$. For the inter-cluster transfer time per byte we use $\tau_g = \tau^{ij}, i \neq j$ (which may be considered as an

estimated average transfer time/ byte).

By requiring that each process is mapped to one cluster, the problem is to obtain an assignment x which solves

$$\begin{aligned}
&\min_x t(x) \\
&\textstyle\sum_j x_\ell^j = 1, \quad 1 \le \ell \le p \qquad (3.1)\\
&x_\ell^j \in \{0,1\}, \quad 1 \le j \le \kappa,\ 1 \le \ell \le p
\end{aligned}$$

The objective function which can be written as

$$\tau_g \sum_j \sum_{\ell_1} \sum_{\ell_2} \beta_{\ell_1 \ell_2} x_{\ell_1}^j \bar{x}_{\ell_2}^j + \sum_j \sum_{\ell_1} \sum_{\ell_2} \beta_{\ell_1 \ell_2} \tau^j x_{\ell_1}^j x_{\ell_2}^j, \qquad (3.2)$$

can furthermore be put in the form of the pseudo-boolean quadratic function

$$t(x) = x^\tau Q x + y^\tau x, \qquad (3.3)$$

where $x = (x_1^1, \ldots, x_p^1, \ldots, x_1^\kappa, \ldots, x_p^\kappa)^\tau$, $y^\tau = \tau_g \sum_{j=1}^{\kappa} \sum_{\ell_1=1}^{p} \sum_{\ell_2=1}^{p} \beta_{\ell_1 \ell_2}$, and the matrix Q is upper triangular, and block diagonal with κ upper triangular blocks each of dimensions $p \times p$, i.e.

$$Q = \begin{pmatrix} Q_{11} & & & \\ & Q_{22} & & \mathbf{0} \\ \mathbf{0} & & \ddots & \\ & & & Q_{\kappa\kappa} \end{pmatrix}.$$

The jth upper triangular block Q_{jj} has zeros on the diagonal and the element $-2\beta_{\ell_1 \ell_2}(\tau_g - \tau^j)$ at location (ℓ_1, ℓ_2) in the block.

As a small example, for $\kappa = 2$, $p = 3$,

$$Q_{jj} = \begin{pmatrix} 0 & -2\beta_{12}(\tau_g - \tau^j) & -2\beta_{13}(\tau_g - \tau^j) \\ & 0 & -2\beta_{23}(\tau_g - \tau^j) \\ \mathbf{0} & & \\ & & 0 \end{pmatrix},$$

for $j = 1, 2$.

Note that the objective function as the total (sum, or average) transfer time reflects the total usage of the network, i.e., the extent to which network links/channels are tied up by the application, loose from the actual point in time any transfer occurs, thereby allowing transfer overlaps (where transfers occur simultaneously on different network links). $t(x)$ does not depend on the underlying topology, and the processes and message passing may be synchronous or asynchronous. However, $t(x)$ does generally not represent the total communication time.

The problem of minimizing an unconstrained pseudo-boolean quadratic function of the form (3.3), although *NP* complete in general [3], has an efficient algorithmic solution when the elements of Q are all nonpositive, which is satisfied by $t(x)$ assuming $\tau_g \geq \tau^j$; cf. the following property from [52]. The problem

$$\min_x \{\sum_{j=1}^{n} y_j x_j + \sum_{i=1}^{n}\sum_{j=1}^{n} q_{ij} x_i x_j\} \tag{3.4}$$

subject to

$$x_j \in \{0, 1\},\ 1 \leq j \leq n$$

where y_j and q_{ij} are real valued constants and $q_{ij} \leq 0$ for $1 \leq i \leq n$ and $1 \leq j \leq n$ can be solved efficiently by solving for a minimum cut on a related graph.

The constructions in [51, 52] deliver a network with a set of non-negative capacities, so that the capacity through the cut equals the objective function, thereby reducing the 0-1 minimization problem to determining a cut with minimum capacity. The algorithm of Ford and Fulkerson or its improvements [48] can then be used to obtain a minimum cut.

Taking the constraint of (3.1) into account leads to a problem of the form considered in [5]. In the latter, a branch-and-bound method is used where, at each branch-and-bound node, bounds (for the primal problem) are obtained via the Lagrangian dual, which is mapped to a max-flow problem with an efficient solution.

The task allocation models given so far in this section will not, in general, guarantee a particular restriction on the number of processes assigned to any cluster, which would be necessary if, for example, the number of processes assigned to cluster j cannot exceed its number of processors (allowing for assigning at most one process per processor). With this type of constraint the quadratic program becomes

$$
\begin{aligned}
&\min_x t(x) \\
&\textstyle\sum_j x_\ell^j = 1, \quad 1 \leq \ell \leq p \\
&\textstyle\sum_\ell x_\ell^j \leq \nu_j, \quad 1 \leq j \leq \kappa \\
&x_\ell^j \in \{0,1\}, \quad 1 \leq j \leq \kappa,\ 1 \leq \ell \leq p
\end{aligned}
$$

assuming $p \leq \sum_j \nu_j$.

3.5.1 Methods

We did some test runs with a "general" optimization code (for mixed integer programs) available from the NAG library [46]. The algorithm uses a branch-and-bound technique which branches by splitting a variable range, and obtains bounds for the objective function by solving the corresponding continuous problem at each node. The problem specification is of the form $\min_x\{y^\tau x + \frac{1}{2}x^\tau Hx\}$, where each variable $x_i,\ 1 \leq i \leq n$, is specified to be in a given range and may be restricted to be integer, subject to a linear constraint set. The only restriction on H is to be symmetric. In general the method is only expected to deliver a global optimum when H is positive definite; yet for some applications a result corresponding to a local optimum may yield useful information. This method is generally too time consuming for practical cases.

Alternatively, special purpose methods can be investigated, in order to exploit the structure of the objective function. *Generalized Lagrange multiplier* type methods solve a sequence of Lagrangian dual (relaxation) problems, searching the multiplier space to maximize the minimum Lagrange function. Indeed, it is well known that a solution minimizing the Lagrange function provides a lower bound on the original objective function. We thus seek to maximize the lower bound.

Using the definition of [33], the Lagrangian dual corresponding to $\min_{x\in X} f(x)$ under constraints $h(x) = 0$ and $g(x) \leq 0$ (primal) is

$$\max\{L^*(\lambda, \pi) : \lambda \geq 0\}$$

where

$$L^*(\lambda, \pi) = \inf_{x \in X} \{L(x, \lambda, \pi)\}$$

with

$$L(x, \lambda, \pi) = f(x) + \lambda g(x) + \pi h(x).$$

If L^* is reached at x^*, then

$$L^*(\lambda, \pi) - \lambda b - \pi c \leq \inf_{x \in X} \{f(x) \mid g(x) \leq b, h(x) = c\}.$$

Furthermore, $x^* \in \text{argmin}\{L(x, \lambda, \pi) \mid x \in X\}$ implies $x^* \in \text{argmin}\{f(x) \mid g(x) \leq b \text{ and } h(x) = h(x^*)\}$ for all b satisfying $(g(x^*) \leq b)$ and $(\lambda g(x^*) = \lambda b)$.

A generalized Lagrange multiplier method then incorporates a strategy of the form [33]

1. For (λ, π) with $\lambda \geq 0$, find $x \in \text{argmin}\{L(x, \lambda, \pi) \mid x \in X\}$.

2. Terminate if 0 is in the convex hull of $S(x, \lambda, \pi) = \{(b, c) \mid g(x) \leq b, h(x) = c, \lambda g(x) = \lambda b\}$ (then x solves the original program). If no min generates 0 as a member of S, the right hand side (0) is in a duality gap.

3. Given (x, λ, π), determine (λ', π') to increase $L^*(\lambda, \pi)$.

A strategy of this form is proposed in [25]. In [34] the Lagrangian model is put in the framework of a generalized penalty function/ surrogate model. Considering the Lagrange multiplier method as a mapping from the space of the multipliers to the space of the constraint vectors, the duality gap arises from the fact that this mapping is not necessarily onto. This may leave inaccessible regions or *gaps* corresponding to (right hand side) constraint

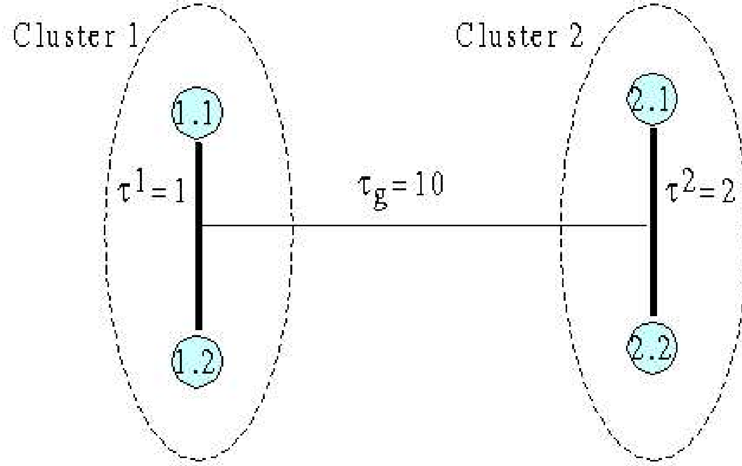


Figure 3.5 Clusters for Resource Allocation Example

vectors which can never be generated [25]. In [5], an approximate Lagrangian dual problem is solved to provide a lower bound; the duality gap is tested by solving a posiform equation at each branch-and-bound node.

3.5.2 Example

As an example, let us consider two clusters C_1 and C_2, each having two processors, and four processes that need to be efficiently distributed. Figure 3.5, shows the communication speed (seconds/MegaBytes) between and within clusters, while Figure 3.6 shows the communication requirements (MegaBytes) between the processes.

For this example the optimum solution is to assign processes p_1 and p_2 to Cluster 1 and $p3$ and $p4$ to Cluster 2. The minimum communication time is

$2 \cdot (10 \cdot 1 + 1 \cdot 10 + 5 \cdot 2) = 60$. The unknowns are $x = (x_1^1, x_2^1, x_3^1, x_4^1, x_1^2, x_2^2, x_3^2, x_4^2)^\tau$,

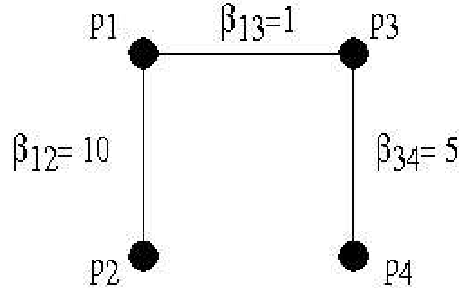


Figure 3.6 Communication Requirements for Resource Allocation Example

$y = (110, 100, 60, 50, 110, 100, 60, 50)$, and Equation (3.3) becomes:

$$t(x) = 110x_1^1 + 100x_2^1 + 60x_3^1 + 50x_4^1 + 110x_1^2 + 100x_2^2 + 60x_3^2 + 50x_4^2 - \\ - 180x_1^1x_2^1 - 18x_1^1x_3^1 - 90x_3^1x_4^1 - 160x_1^2x_2^2 - 16x_1^2x_3^2 - 80x_3^2x_4^2, \quad (3.5)$$

with constraints $h(x)$ and $g(x)$:

$$\begin{cases} h(x) = \begin{cases} x_1^1 + x_1^2 = 1 \\ x_2^1 + x_2^2 = 1 \\ x_3^1 + x_3^2 = 1 \\ x_4^1 + x_4^2 = 1 \end{cases} \\ g(x) = \begin{cases} x_1^1 + x_2^1 + x_3^1 + x_4^1 \leq 2 \\ x_1^2 + x_2^2 + x_3^2 + x_4^2 \leq 2 \end{cases} \end{cases} \quad (3.6)$$

In order to eliminate these constraints, we set out to solve the Lagrangian dual problem:

$$L^*(\lambda, \pi) = inf_x\{t(x) + \lambda \cdot g(x) + \pi \cdot h(x)\}. \quad (3.7)$$

For $\lambda = (-10, -10, -20, 4)$ and $\pi = (1, 2)$, the new equation to minimize is:

$$t(x) = 101x_1^1 + 91x_2^1 + 41x_3^1 + 55x_4^1 + 102x_1^2 + 92x_2^2 + 42x_3^2 + 56x_4^2 - \\ - 180x_1^1x_2^1 - 18x_1^1x_3^1 - 90x_3^1x_4^1 - 160x_1^2x_2^2 - 16x_1^2x_3^2 - 80x_3^2x_4^2. \quad (3.8)$$

From Equation (3.8) we generate the flow graph presented in Figure 3.7. The mapping steps are: a) add one node for every equation term plus a source and a sink; b) add a directed edge between the source and each linear term in the equation, and between every quadratic term and the sink (the edge capacities are the absolute values of the coefficients); c) for each quadratic term add an edge (of inifinite capacity) between the constituent linear term nodes and the corresponding quadratic node.

Applying the Ford and Fulkerson algorithm to the above graph, we find a solution to Equation (3.8). If this solution satisfies the constraints (3.6), then we have found a solution for the principal problem. If not, we have to solve another dual problem with a different λ, π vector.

In conclusion, even if the workload structure is known, it may be difficult to obtain an efficient computation.

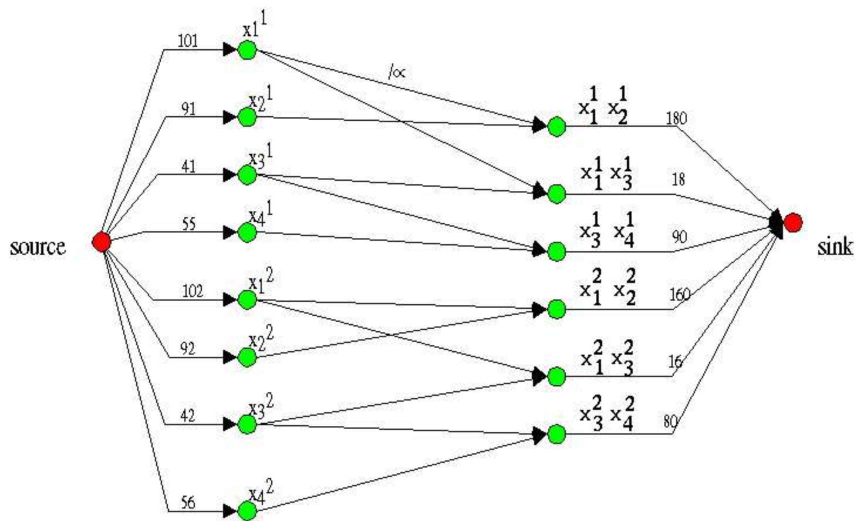


Figure 3.7 Flow Graph for Resource Allocation Example

CHAPTER 4

Problems with an Unknown Number of Units - Known Unit Size (UnKs)

In this chapter we study problems for which we know the size of the work units but we do not know the total number of units. In Section 4.2 we present distributed quasi-Monte Carlo and Monte Carlo integration methods. This chapter is part of publications [12] and [14].

4.1 Abstract Problems

Problem 1: First we consider the case where the total amount of work is unknown but the required work increases in known size units. Let us assume that the computation can be characterized as in Figure 4.1. The algorithm requires a termination check only when a row is completed. Each row has the same number of cells; all cells in the same row are of equal size, and there are no dependencies among them. However, the size of the work unit increases considerably from one row to the next. Consider using p processors for the computation, one of which is assigned a controlling role, and the remaining ones are workers. One distribution strategy is to assign one cell of work k_{ij} to one worker at a time. We will refer to this strategy as *single cell work assignment*.

The controller updates the result and sends a new cell if necessary. Figure 4.2 illustrates a case with less workers than cells in a row. Since the computations are asynchronous, different processors may be working on different rows. Results from this algorithm tend to

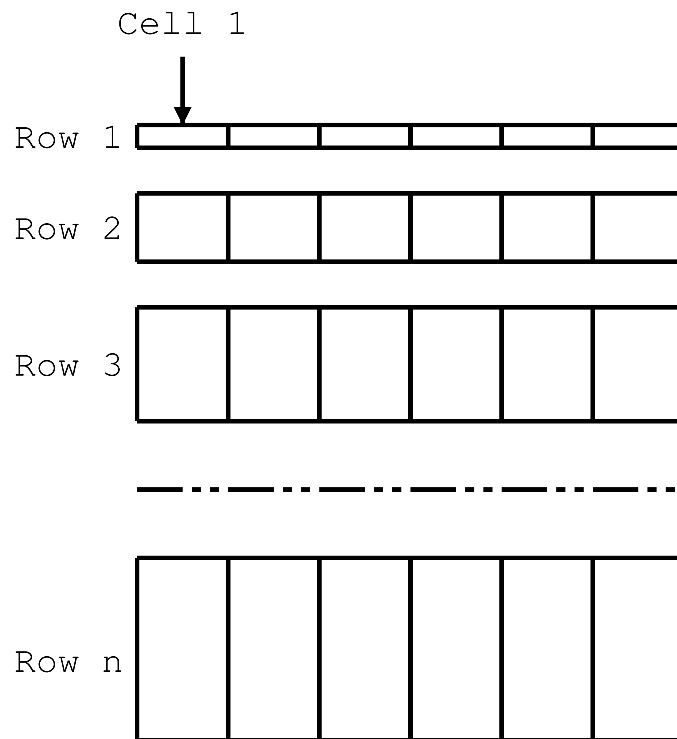


Figure 4.1 Example of Work Structure

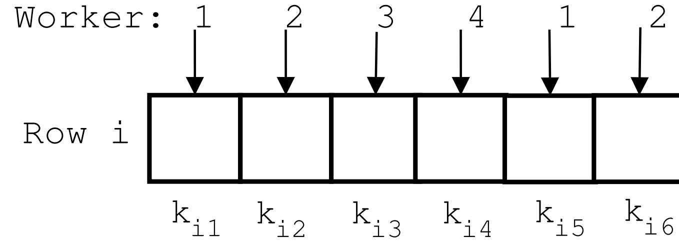


Figure 4.2 Single Cell Work Assignment

show a degradation in performance when the number of workers is relatively large (compared with the number of cells in a row q). We found that this behavior can usually be attributed to *breaking loss*. This is replaced by *starvation loss*, where processors are idle waiting for work, when the computation proceeds to the last possible row (of index N_{max}).

For example, assume $p = 2 \cdot q$ workers (two groups of q), and a computation that requires five rows to finish. In this case, the first row is assigned to the first group of workers, the next row to the next group and so on. In Figure 4.3 the amount of time wasted on unwanted work (breaking loss) is marked with light gray.

Another strategy is to split every row among all the workers, so that (in a synchronous setting) all workers at a time would process the same row. This strategy splits every cell k_{ij} into p pieces $k_{ij}^m, m = 1, \ldots, p$ of size N_i/p where N_i is the size of a cell in row i (see Figure 4.4).

Every worker w is assigned a work unit $u_i^w = \{k_{ij}^w \mid j = 1, \ldots, q\}$. After the controller has received the row results from all workers, it assembles the cell results. We will refer to this strategy as *row distribution work assignment*. We found that this new strategy significantly decreases the breaking loss.

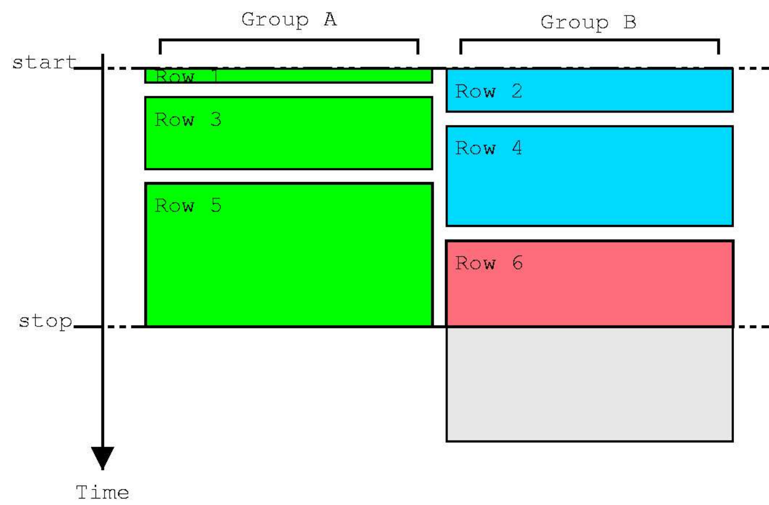


Figure 4.3 Example of Breaking Loss Effect for Problem 1

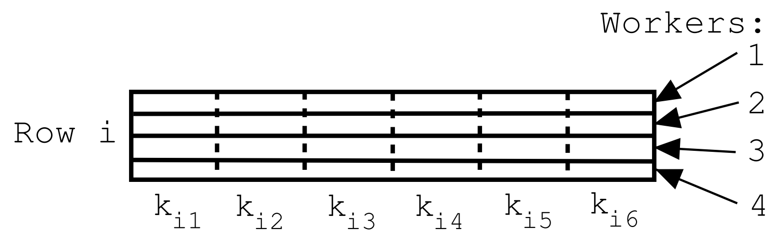


Figure 4.4 Row Distributed Work Split for $p = 4$

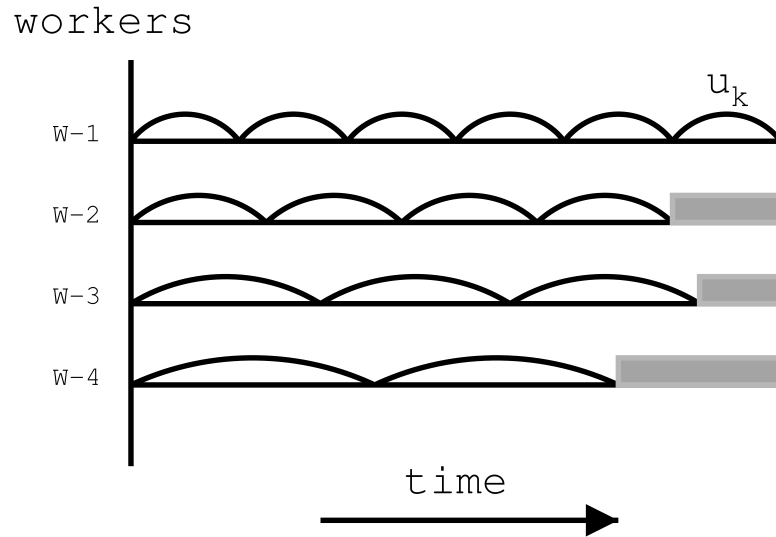


Figure 4.5 Problem 2: Uniform Work Distribution in Heterogeneous Environment

An issue that remains is that the performance easily degrades if some processors are slower than others, making this strategy, as is, unsuitable for heterogeneous systems. Fortunately we can easely adapt it by splitting each row into a number of units that is a multiple of the number of processors, so that the faster workers process more units from the same row.

Problem 2: Consider an unknown amount of work in a worker-dependent/dynamic task size assignment. Assume further that there are no dependencies between work units, and each unit can be made as small or as big as desired. One option for the work split is to ask each worker to report results after a fixed number of iterations. However this strategy introduces breaking loss in heterogeneous systems. Figure 4.5 represents the load assignment for four workers of different power. After worker *W-1* reports the result for unit u_k, the computation terminates. The gray boxes represent unwanted work.

```
Time one iteration
Estimate the number of iterations (NLI) that can be performed
in the requested time interval
do {
    Read the time after NLI/10 iterations
    Use the new time read to adjust NLI and next time-read interval
}
until (Time expired)
```

Figure 4.6 Iteration Timing Algorithm

A better strategy is to ask all the workers to report results after a fixed time interval. Each worker basically adjusts the computation performed in a work unit according to its power. By fixing the work unit time and not the work unit amount, we can control the number of messages received by the controller. In this way we can avoid both bottleneck and breaking loss effects.

This strategy can be accomplished by asking each worker to send an update every t seconds. The time interval must be determined to obtain a fair trade-off between communication overhead and possible breaking loss.

Let us assume the work is composed of a number of iterations, and we want to report results after a certain time interval. In order that the application would not be flooded with too many clock reads we can use the iteration execution time as a measurement unit. Figure 4.6 outlines a simple algorithm.

4.1.1 Speedup Analysis

Based on the structure of the work, we propose to create a model describing the speedup that can be obtained on a particular system. A case study is presented in [12]. For Problem 1 and single cell work assignment, let us consider the case where the execution behaves in a synchronous manner and the number of processors p is a multiple of the number of cells in a row, that is, $p = tq$ with $t > 0$.

Assume the problem requires n rows to terminate. Initially, the first q processors are working on row R_1, the next q processors work on R_2 and so on. Assuming that the processors finish their cells and receive new tasks at the same time, then once R_1 is done, the first set of q processors moves on to R_{t+1}; once R_2 is done the second set starts row R_{t+2} and so on. When the last row required for this problem is assigned, all other rows of lower index are either completed or in the process of being completed. Since the work required for lower index rows is much smaller than the work required for R_n, and there are only q processors assigned to R_n, all other workers will be reassigned to rows R_ℓ, $\ell > n$ before R_n is completed. However this work is useless, since the controller stops the computation when R_n is completed; thus breaking loss is incurred.

In order to evaluate the parallel performance, let τ be the time required to complete a function evaluation; then the sequential computing time can be estimated by $T_1 = q \sum_{i=1}^{n} N_i \tau$. In the case under consideration, the total parallel execution time equals the time of the processors that eventually work on the last row R_n. That is,

$$T_p = \sum_{s=1}^{r} N_{k_s} \tau, \tag{4.1}$$

where $r = \lceil \frac{nq}{p} \rceil = \lceil \frac{n}{t} \rceil$ is the number of rows computed by these processors, $k_s = k_1 + t(s-1)$, is the row index, $k_1 = n - (r-1)t$, and $k_r = n$. In this case the theoretical speedup is

$$S = \frac{T_1}{T_p} = q \frac{\sum_{i=1}^{n} N_i}{\sum_{s=1}^{\lceil n/t \rceil} N_{k_s}}. \tag{4.2}$$

For example, with $q = 6, p = 30, n = 25$, (4.2) gives $S = \frac{6 \sum_{i=1}^{25} N_i}{N_5 + N_{10} + N_{15} + N_{20} + N_{25}}$ = 15.628. Measuring the runtime experimentally (for a problem in quasi-Monte Carlo integration, on a homogeneous cluster), we obtain a speedup of 15.495. Table 4.1.1 lists the experimental vs. theoretical speedup, obtained for numbers of workers between 1 and 30.

No.Workers	Theoretical	Experimental
1	1.000	1.000
2	2.000	1.998
4	3.749	3.746
6	6.000	5.990
8	6.841	6.828
10	7.961	7.950
12	9.999	9.974
14	10.320	10.296
16	10.974	10.933
20	12.778	12.722
24	14.443	14.389
28	14.659	14.594
30	15.628	15.495

Table 4.1 Theoretical vs. Experimental Speedup for Problem 1

The total communication time is small as only one pair of messages per cell is required, amounting to $\mathcal{O}(nq)$. However, the parallel efficiency of the single cell work assignment is affected considerably by breaking loss.

Now let us consider $p = q$ $(t = 1)$. All the processors work in one row at a time. Once a row is finished all workers move to the next one. When the last row is completed there are no processors working on cells beyond the last row, resulting in a near ideal speedup. The same can be shown if $p < q$ and q is a multiple of p (however, breaking or starvation loss may occur when q is not a multiple of p).

Since, in our application, q is small (e.g $q = 6$), we have hereby shown that the single cell work assignment technique is not scalable, and justified our introduction of the row distribution strategy. The goal is to force all the workers to process units from the same row at every moment in time.

4.2 Numerical Integration

The computational objective is to obtain an approximation Q to the multivariate integral $I = \int_{\mathcal{D}} \mathrm{f}(\mathbf{x})d\mathbf{x}$ and an absolute error bound E_a such that $\hat{E} = |I - Q| \leq E_a \leq \hat{\varepsilon} = \max\{\varepsilon_a, \varepsilon_r|I|\}$, for given absolute and relative error tolerances ε_a and ε_r, respectively. The integration domain $\mathcal{D}$ is a hyper-rectangular region or simplex. The integrand is generally defined as a vector function $f : \Re^n \rightarrow \Re^m$, and the error (estimate) value is interpreted in maximum norm.

4.2.1 Monte Carlo

The work structure for the Monte Carlo integration method fits the description presented in Problem 2 of Section 4.1. To avoid the breaking loss problem in a heterogeneous environment, we have implemented the *fixed time unit* distributed strategy.

Each worker continuously evaluates the integrand function at random points. At fixed

time intervals it sends the result to a controller that updates the global result and standard error, see Figure 4.6. The communication is sender initiated. The workers do not receive any other messages beside the stop signal.

In the parallel Monte Carlo implementation, special attention must be payed to the random number generator used. The sequence of random values generated by one processor must not overlap with the sequence generated by another processor.

Initial scalability tests have shown near ideal speedup. In Figure 4.7, workload A requires 43 seconds to complete by one processor. Note that as the work unit size increases, the parallel efficiency decreases. For a Work Unit of 2 seconds the speedup is half of that with a work unit of 0.01 seconds at 30 processors. We attributed this behavior to breaking loss. At the end of the computation, the last work unit is processed for 2 seconds, most of which is not necessary.

This observation is sustained by the Figure 4.8. The work unit is set to 2 seconds, the sequential computation time is 6.2 seconds. The total function evaluation count for each run is shown in Figure 4.9.

The superlinear speedup spikes are explained by the fact that the sequential time is obtained by running the parallel program on one CPU. The speedup peaks indeed coincide with runs of a lower number of function evaluations (compare with Figure 4.9).

4.2.2 Quasi-Monte Carlo

Quasi-Monte Carlo (QMC) methods are effective for higher dimensions and fairly smooth functions. At the basis, we use a sequence of Korobov lattice rules [9, 29, 21, 15].

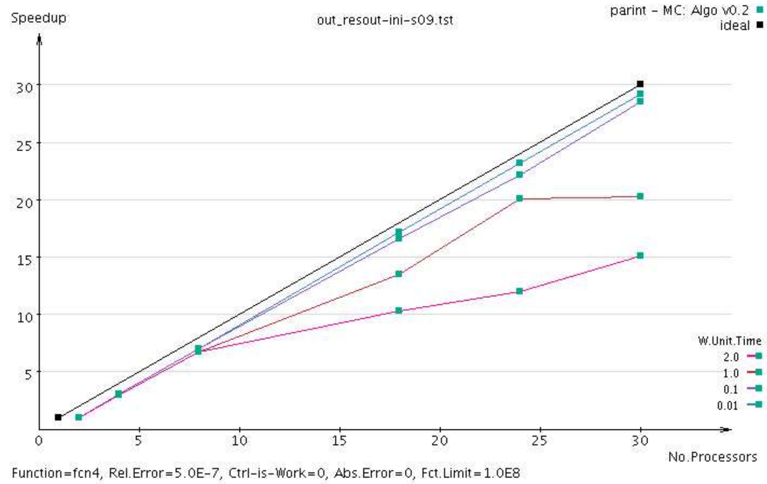


Figure 4.7 Monte Carlo: Speedup - Workload A

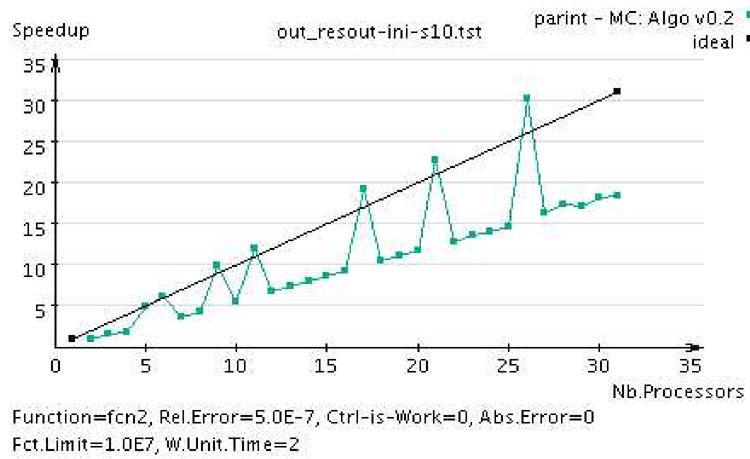


Figure 4.8 Monte Carlo: Speedup - Workload B

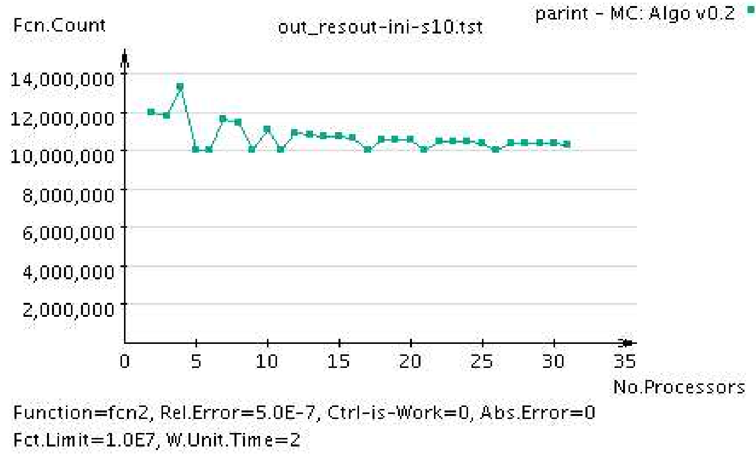


Figure 4.9 Monte Carlo: Total Function Evaluation Count - Workload B

For each rule, a fixed number of randomized samples are computed by adding a random vector to the integrand evaluation points. The randomization serves the purpose of obtaining an estimated error based on the sample variance. Let

$$K_N(\boldsymbol{\beta}) = \frac{1}{N}\sum_{i=1}^{N} f(\{\frac{i}{N}\mathbf{v} + \boldsymbol{\beta}\}), \tag{4.3}$$

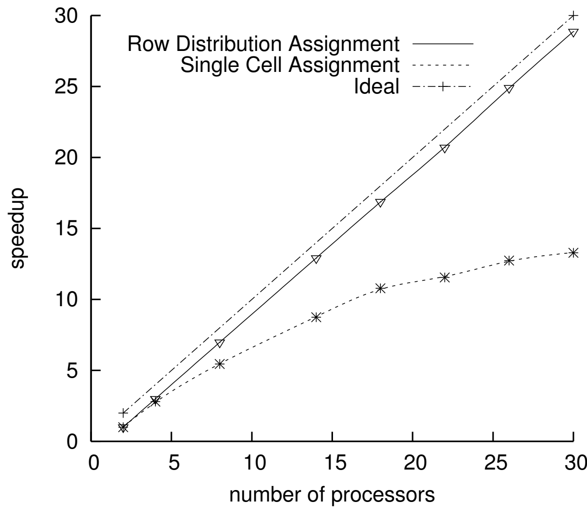
where $\mathbf{v}$ is the (Korobov) lattice generator vector, randomized by a uniformly distributed random vector $\boldsymbol{\beta}$ [9]. Then with a sample set (of size q) of random $\boldsymbol{\beta}_j$, the mean

$$\bar{K}_N = \frac{1}{q}\sum_{j=1}^{q} K_N(\boldsymbol{\beta}_j) \tag{4.4}$$

allows for a standard error estimation by

$$E_N^2 = \frac{1}{q(q-1)}\sum_{j=1}^{q}(K_N(\boldsymbol{\beta}_j) - \bar{K}_N)^2. \tag{4.5}$$

The algorithm calculates the $\bar{K}_N$ values of (4.4) for a sequence of rules with successively larger numbers N of integration points, until either an answer is found to the user-specified



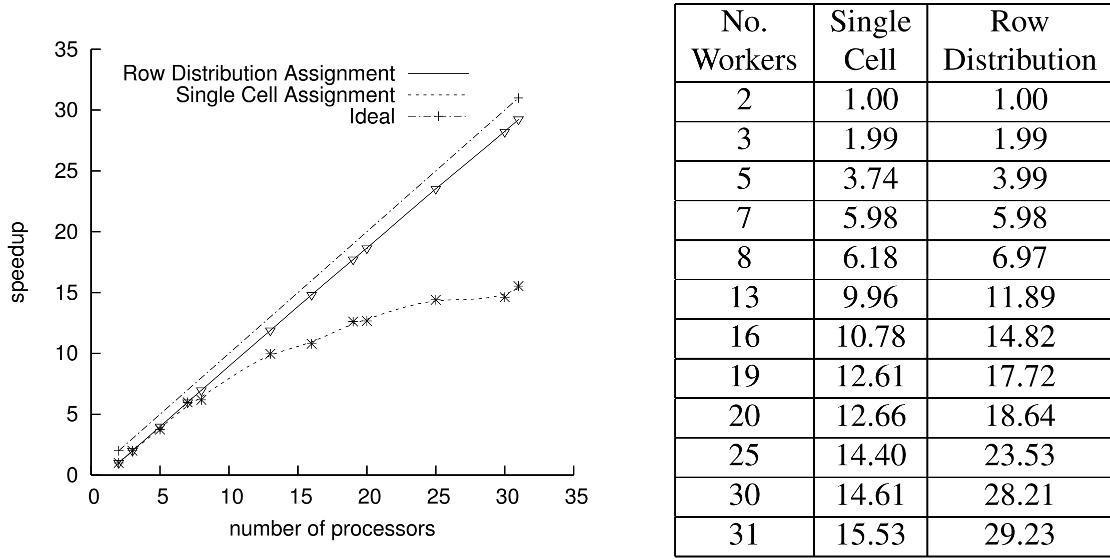
No. Workers	Single Cell	Row Distribution
2.0	1.00	1.00
4.0	2.80	2.99
8.0	5.45	6.97
14.0	8.74	12.90
18.0	10.78	16.87
22.0	11.54	20.70
26.0	12.73	24.89
30.0	13.28	28.86

Figure 4.10 Speedup for Finance Problem on Homogeneous System

accuracy, or the function count limit is reached (or the end of the sequence has been reached).

Using the mortgage-backed security problem from [47, 49], we tested both strategies presented in Problem 1 of Section 4.1. Figure 4.10 shows the speedup obtained when calculating the (360-dimensional) integral in a homogeneous network, showing the excellent behavior of the row distribution assignment (versus the relatively poor behavior of the single cell assignment).

Figure 4.11 shows the speedup obtained for a Student-T distribution problem using the two algorithms.



No. Workers	Single Cell	Row Distribution
2	1.00	1.00
3	1.99	1.99
5	3.74	3.99
7	5.98	5.98
8	6.18	6.97
13	9.96	11.89
16	10.78	14.82
19	12.61	17.72
20	12.66	18.64
25	14.40	23.53
30	14.61	28.21
31	15.53	29.23

Figure 4.11 Speedup for Student-T Distribution Problem on Homogeneous System

Adaptations for Heterogeneous Systems

Since for each row, every processor performs the same amount of work in the strategy of Problem 1, the total run time for the completion of N rows is determined by the slower machines. To overcome this limitation we consider three alternative strategies.

Proportional row splitting: For each row, a worker gets a slice of work proportional with its processor power. This information can be acquired either statically, dynamically or using a combination of both. Statically, the controller can retrieve the physical properties of each machine (CPU, memory, bus speed, etc). For a cluster, these numbers almost never change, and can be stored in and retrieved from a file. Dynamically, each worker reports its average time per function evaluation. Or, as a combination, the controller first splits the work according to the static information, and later adjusts the distribution based on reported function evaluation times and worker response times as the computation proceeds. This

strategy has a number of disadvantages. The static approach gives a very rough estimation. The processing time is affected by a number of factors besides the ones mentioned above. These include: memory latency, caching, etc., which can affect the computation time significantly from problem to problem. The dynamic approach works well if the workers are dedicated to the computation. Otherwise, processes sharing the same machine, affect the estimation of the worker power. A similar approach is presented in [41].

Fixed work splitting: In this approach, the work units are of the same size, so that the earlier rows are split in fewer units than the later rows. This gives faster processors a chance to request work more often than the slower ones. This strategy has a few drawbacks. On the one hand the total number of messages increases from row to row. Furthermore, choosing the optimum work unit size is a delicate task, mainly because it depends on the integrand function. A big work unit may introduce a significant starvation or breaking loss effect (in case the last units from the last row are assigned to the slower workers); while a small work unit may introduce significant communication time.

Constant row splitting: This strategy splits each row in a constant number of work units, $C = Kp$, where K is a constant. The size of each work unit is proportional with the row size. If some processors are slower than others, they will process fewer units from each row. This strategy has the advantage that the total number of messages is constant across the rows. As in the previous case, starvation or breaking loss can occur in situations where the last work units are assigned to the slower processors. Figure 4.12 shows the speedup obtained for the same problem in a heterogeneous system using different K values. The first 30 machines have 1.2Ghz CPUs and the last 30 machines have 0.8Ghz CPUs.

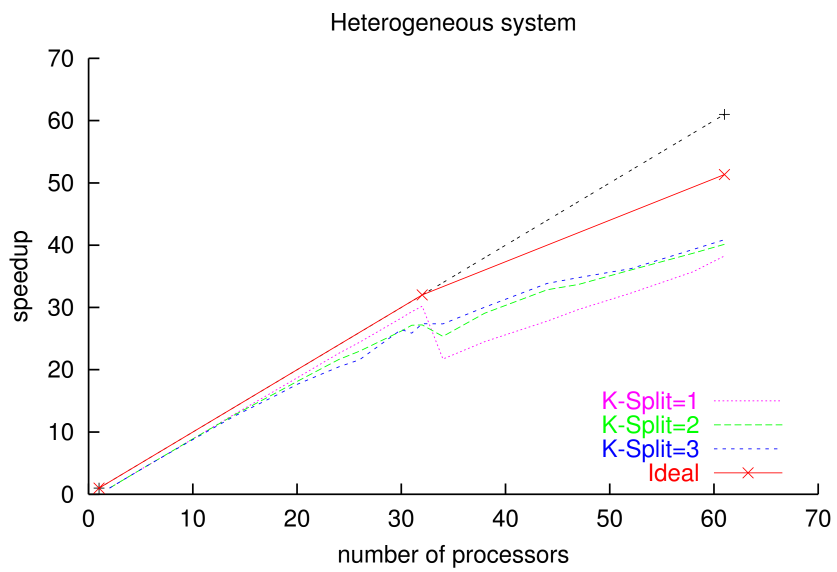


Figure 4.12 Speedup for Finance Problem on Heterogeneous System

CHAPTER 5

Problems with a Known Number of Units - Unknown Unit Size (KnUs)

In this chapter we investigate a problem with known number of work units, and unknown work unit size. This section is part of the publication [17].

5.1 Hierarchical Integration

We consider efficient strategies for the parallel and distributed computation of large sets of multivariate integrals. These arise in many applications such as computational chemistry, high energy physics and finite element problems.

Let $\{I_1, I_2, \cdots, I_P\}$ be a set of P integrals. Our problem is to efficiently calculate a numerical approximation Q_k to the integral I_k, together with an error estimate E_k, for $1 \leq k \leq P$. Integral I_k is of the form $I_k = \int_{\mathcal{D}} F_k(\vec{x}) d\vec{x}$ where $\mathcal{D}$ is a hyper-rectangular region in $\mathcal{R}^N$ and $F_k(\vec{x})$ is an N-dimensional integrand function over $\mathcal{D}$. The approximation Q_k and its error estimate E_k need to satisfy the condition $|I_k - Q_k| \leq E_k \leq Max\{\varepsilon_{ak}, \varepsilon_{rk}|I_k|\}$ where ε_{ak} and ε_{rk} are given absolute and relative tolerances, respectively.

Let p be the number of processors in a given parallel and distributed computing system. One approach would be to insert the integrals I_k into a queue and compute each integral using all p processors until the queue is empty. In cases where some of the integrals are easy to compute this may lead to a wastage of resources because of system overheads. Load balancing not only within an individual integral computation but also among the integrals I_k thus becomes an important issue. Furthermore, in situations where communication over-

heads in the given p-processor system are considerable, it may be more efficient to use, say, $s < p$ processors to compute an individual integral I_k. This would allow the computation of $g = [p/s]$ integrals in parallel and would also lead to a scalable solution. This suggests a three level hierarchical approach with a global controller, a level of group controllers and a level of workers. The number of workers does not need to be equal in the different groups.

The global controller manages g group controllers and is responsible for assigning integrals (tasks) to groups initially and whenever a group becomes idle. For example, the global controller may give one task to each group initially; whenever a group becomes idle, it is supplied with a new integration problem. If the problems are all sufficiently similar and P is a multiple of g, this is equivalent to assigning $\frac{P}{g}$ tasks to each group initially and letting each group evaluate its sequence of integrals. If the problems are not similar, then this results in assigning the integrals depending on their computational time requirements in a load balanced manner.

Parallelization with very different characteristics can be obtained by varying the number of groups g. When $g = 1$, all p processors participate in the integration for each integral, i.e., the parallelization is on the subregion level, which may be appropriate when p is relatively small and the integrals are difficult. When p is large and/or communication costs are high, this approach may lead to too much communication overhead. When $g = p$, the integrations are performed sequentially by the individual processors, i.e., the parallelization is on the integral level. In situations where integrals of varying difficulty are involved, this may lead to load imbalances and breaking loss. For example, one processor may be working on a difficult problem while all the other ones have finished and are idle. Other

values of g, $1 < g < p$, result in a true hierarchical structure. Load balancing within a group may be needed depending on the integration problem.

The hierarchical algorithm can be analyzed given the details of the communication and computation costs of the p-processor system, specific implementation of load balancing and priority queues within a group, and details of the integral approximation locally in a worker. For a detailed theoretical description we refer the reader to [19].

5.2 Experimental Results

In order to test the viability of the hierarchical approach, we selected six function families proposed by Genz [27], which characterize different peculiarities in the integrand behavior. Schürer also considers these integral families in [56] and compares quasi-Monte Carlo techniques with adaptive methods based on cubature rules. The function families are given in Table 5.1. In this table, N is the dimension of the integral, the u_i's and a_i's, $1 \leq i \leq N$, are the unaffective and affective parameters, respectively, and the $||a||_1$ attribute determines the difficulty level.

The test platform was our 64-node linux cluster (*Athena* [1]), of which we used its available AMD Thunderbird 1.2GHz processors each with 512 MB RAM, and its Myrinet interconnect. The head node of the cluster is a dual-processor 1.5GHz AMD Athlon processor with 1GB RAM.

We report typical test results for $N = 10$ and various levels of difficulty. The u_i's and a_i's are randomly chosen from $[\frac{1}{20}, 1 - \frac{1}{20}]$ and the a_i's scaled to obtain the desired difficulty level. Figures 5.1 to 5.6 graph the run times as a function of the group size for 100 instances

Integrand Family	$\|\|a\|\|_1$ Attribute
$f_1(\vec{x}) = \cos(2\pi u_1 + \sum_{i=1}^N a_i x_i)$	$110\beta/\sqrt{N^3}$ Oscillatory
$f_2(\vec{x}) = \prod_{i=1}^N 1/(a_i^{-2} + (x_i - u_i)^2)$	$600\beta/N^2$ Product Peak
$f_3(\vec{x}) = 1/(1 + \sum_{i=1}^N a_i x_i)^{(s+1)}$	$600\beta/N^2$ Corner Peak
$f_4(\vec{x}) = \exp(-\sum_{i=1}^N a_i^2 (x_i - u_i)^2)$	$100\beta/N$ Gaussian
$f_5(\vec{x}) = \exp(-\sum_{i=1}^N a_i \|x_i - u_i\|)$	$150\beta/N^2$ C_0 Function
$f_6(\vec{x}) = \{ \begin{array}{ll} 0 & \text{if } x_1 > u_1 \vee x_2 > u_2 \\ \exp(\sum_{i=1}^N a_i x_i) & \text{otherwise} \end{array}$	$100\beta/N^2$ Discontinuous

Table 5.1 Genz Test Integrand Families

of each parametrized family and using 28 processors. The function evaluation limit for each problem was set to 10^8. The difficulty values are given in the legends.

For the runs of Figure 5.1 most problems were hard, indicating a uniform work spread over the processors. For the higher difficulty levels the problems parallelize optimally using a small number of fairly large groups. The other runs typically involved a mix of easy to hard problems, leading to breaking loss when many small groups were used. We further observed, especially for the higher difficulty level of function 6 (Figure 5.6) and to some extent for function 3 (Figure 5.3) that the number of function evaluations performed was significantly higher using large groups of processors, leading to a decreased efficiency. This work anomaly effect may result due to a very local nature of the difficulties in the integrand behavior, such as the corner peak of function 3 and the discontinuity in function 6. As a result, some workers perform less important tasks than those available at the processors

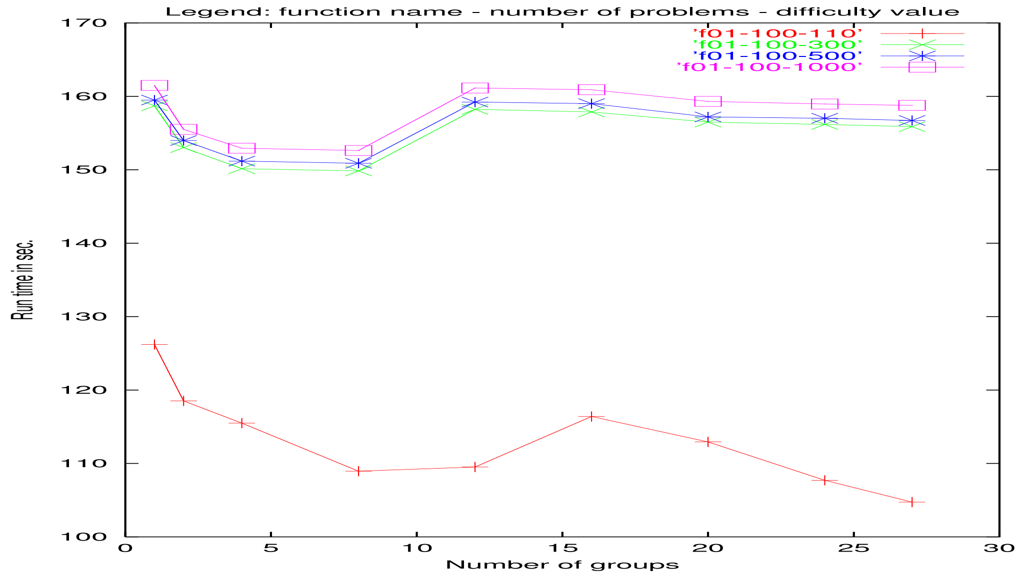


Figure 5.1 Timing Results for $f_1(\vec{x}), \varepsilon_r = 10^{-9}$

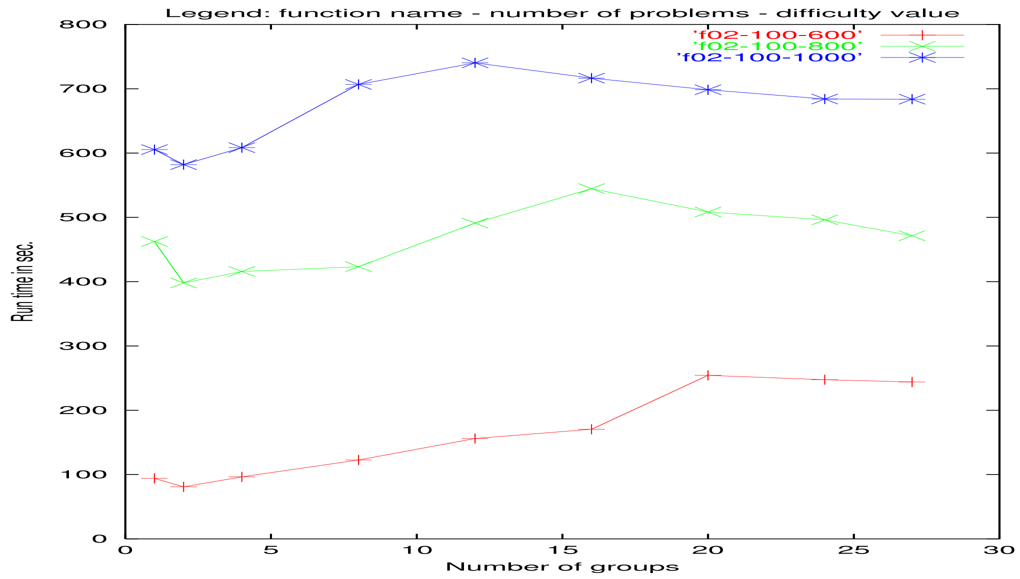


Figure 5.2 Timing Results for $f_2(\vec{x}), \varepsilon_r = 10^{-6}$

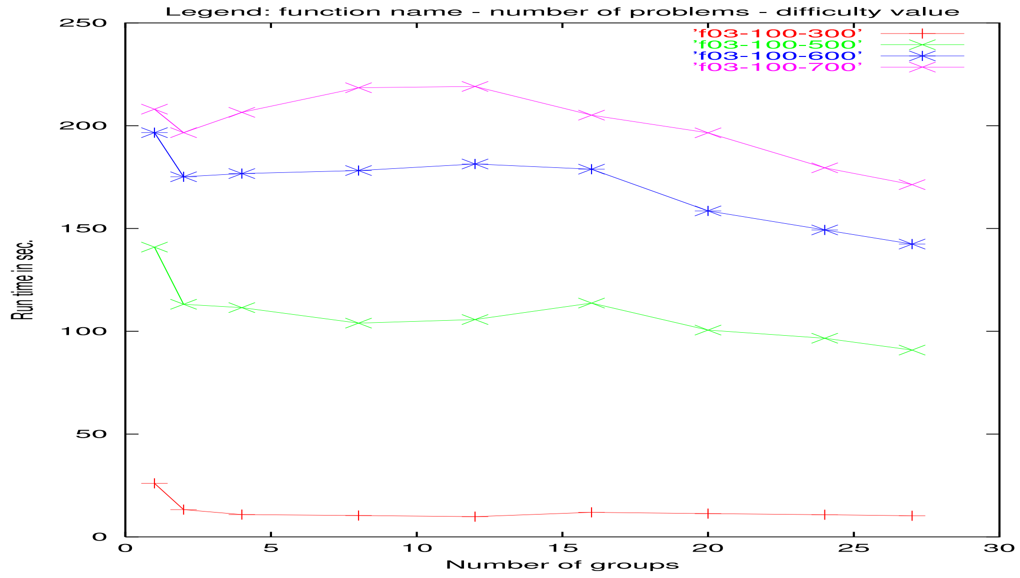


Figure 5.3 Timing Results for $f_3(\vec{x}), \varepsilon_r = 10^{-4}$

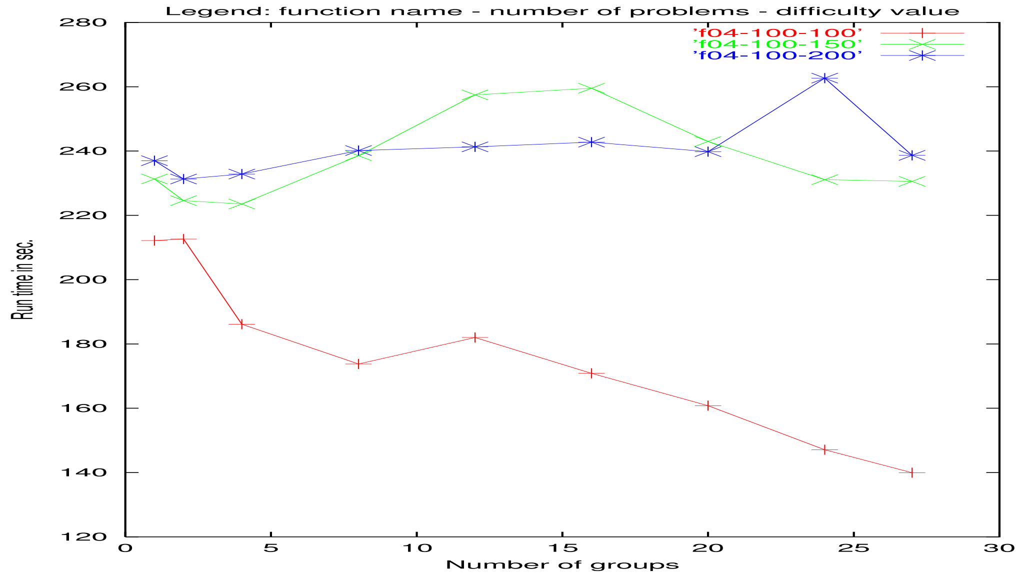


Figure 5.4 Timing Results for $f_4(\vec{x}), \varepsilon_r = 10^{-5}$

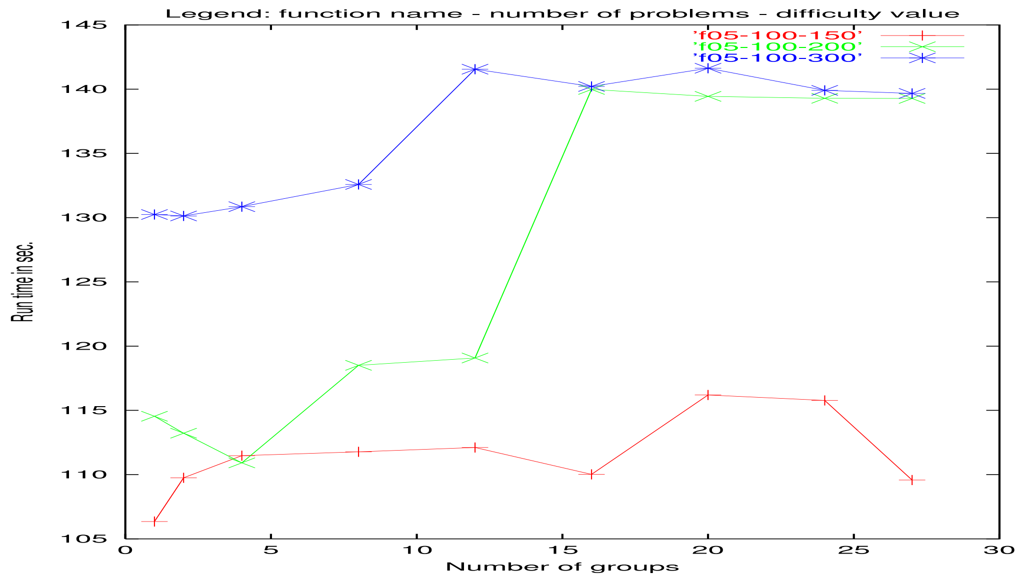


Figure 5.5 Timing Results for $f_5(\vec{x}), \varepsilon_r = 10^{-3}$

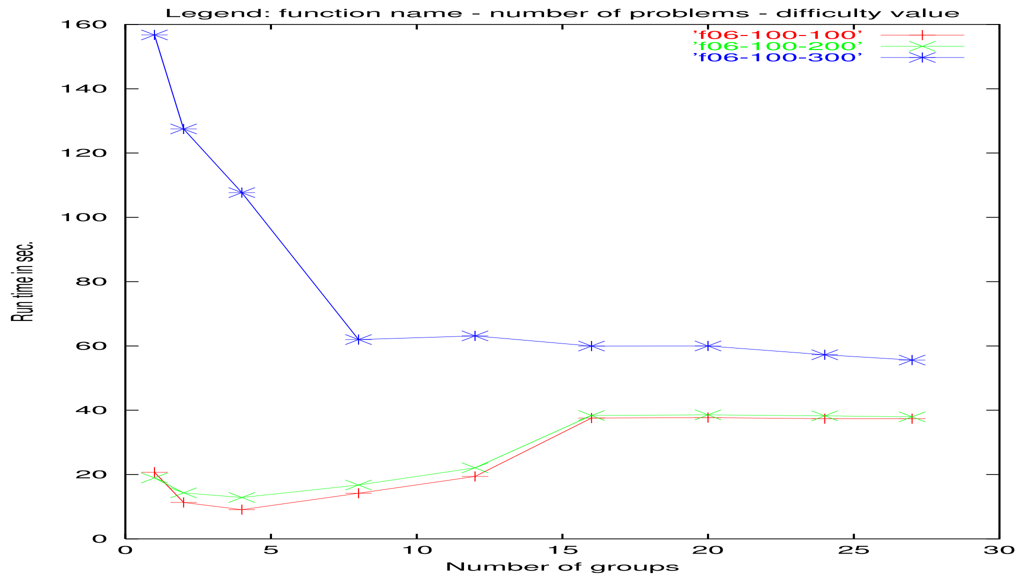


Figure 5.6 Timing Results for $f_6(\vec{x}), \varepsilon_r = 10^{-9}$

which have difficult subregions [64].

In conclusion, even though the number of work units is known for this type of problems, it is very hard to develop an efficient solution without information about the work unit size.

CHAPTER 6

Problems with an Unknown Number of Units - Unknown Unit Size (UnUs)

In this chapter we present a problem with an unknown number of work units, and unknown work unit size. This section is part of [22, 23, 20].

6.1 Adaptive Integration

The computational objective is to obtain an approximation Q to the multivariate integral $I = \int_{\mathcal{D}} \mathrm{f}(\mathbf{x}) d\mathbf{x}$ and an absolute error bound E_a such that $\hat{E} = |I - Q| \leq E_a \leq \hat{\varepsilon} = \max\{\varepsilon_a, \varepsilon_r|I|\}$, for given absolute and relative error tolerances ε_a and ε_r, respectively. The integration domain $\mathcal{D}$ is a hyper-rectangular region or simplex. The integrand is defined as a vector function $f : \Re^n \rightarrow \Re^m$.

We use a *global adaptive partitioning algorithm* (adhering to the *meta-* algorithm of Figure 6.1), in order to concentrate the integration points in areas of the domain $\mathcal{D}$ where the integrand is the least well-behaved. The bulk of the computational work is in the calculation of the function f at the integration points. The granularity of the problem primarily depends on the time needed to evaluate f and on the dimension; an expensive f generally results in a problem of large granularity, such as the Bayesian statistics problems in [16, 18] and the high energy physics problem in [23].

Multivariate integration rules for hyper-rectangular regions from Genz and Malik [30, 31] (with error estimation techniques from [4]), simplex rules from [28], and univariate

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

Figure 6.1 Adaptive Integration Meta-Algorithm

(Gauss-Kronrod) rules (as in Quadpack [53]) are used to provide a result $Q(\mathcal{R})$ and an error estimate $E(\mathcal{R})$ for any subregion $\mathcal{R}$ obtained from the initial domain $\mathcal{D}$. Thus $Q(\mathcal{R})$ is of the form $Q(\mathcal{R}) = \sum_{k=1}^{r} w_k \mathrm{f}(\mathbf{x_k})$. We will refer to the calculation of the pair $Q(\mathcal{R})$ and $E(\mathcal{R})$ for a region $\mathcal{R}$ as a *region evaluation*.

The meta-algorithm of Figure 6.1 subsumes many different designs and implementations [55, 57]. In the PARINT distributed implementation of the adaptive partitioning algorithm, all processors act as *integration worker*; one processor additionally assumes the role of an *integration controller*. The initial region is divided up among the workers. Each executes the adaptive integration algorithm on their own portion of the initial region, largely independent of the other workers, while maintaining a local priority queue of regions, stored as a heap and keyed with the estimated error of the regions.

In the initial steps the workers evaluate their part of the domain and initialize their local heap with it. The regular adaptive loop iteration consists of: removing the region with the highest estimated error from the priority queue; this region is split in half and the two new subregions are evaluated; the overall result and error estimate now needs to be updated and

the new subregions inserted into the queue.

All workers periodically send updates of their results to the controller; in turn, the controller provides the workers with updated values of the estimated required accuracy ε, calculated as $\varepsilon = \max\{\varepsilon_a, \varepsilon_r|Q|\}$. The workers use this value to determine if they have achieved a satisfactory approximation of the integral over their initial subregion. If they have, they become *idle*. To maintain efficiency, a dynamic load balancing technique is employed to move work to the idle workers (see Load Balancing section bellow).

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

6.1.1 Load Balancing

A worker i informs the controller via a regular update message that it is not idle; the controller selects (in a round-robin fashion) an idle worker j and sends i a message containing the id of j. Worker i will then send j a region off the top of its queue or a message indicating that no work is available. Worker j receives this message and either resumes working or informs the controller that it is still idle.

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

6.2 Further Remarks and Challenges

Adaptive task partitioning is an important paradigm in numerical integration. However, it is also used in other problem domains, such as ray tracing, function approximation, optimization, and mesh refinement [39].

The adaptive strategy implements a priority queue using a heap. There are problems that require more regions stored in the queue than the amount of memory in any one CPU. In such cases, a sequential execution may not be possible, and/or the parallel implementation may give superlinear speedup.

Although this method helps to direct the work in difficult areas, any adaptive partitioning strategy may suffer from singularity loss, where less important tasks are done before more important ones are available.

If the error estimate does not improve or improves very slowly as the computations proceed, it is possible that the subdivision focus went wrong and that there are important regions which did not get processed.

A related type of work anomaly, which is incurred by generating and processing unnecessary work is studied in [66] for adaptive integration and branch and bound applications.

6.3 Adaptive Mesh Refinement

We developed a techique for investigating the integration domain, based on a sequence of nested grids with finer and finer mesh spacing. The problem domain is initially divided in a coarse grid that will create a first approximation of the solution. Based on error estimation, cells of the grid are subdivided further for the improvement of the partition.

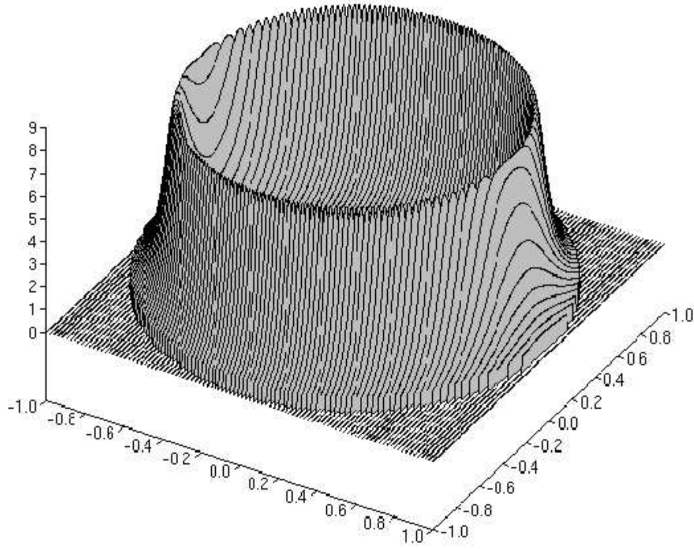


Figure 6.2 Sample Integrand Function

This technique is frequently used in fluid dynamics, radiative transport methods, progressive radiosity, etc. We note that, in one extreme, may have work units with difficulties uniformly distributed in the problem domain, or it may be very imbalanced.

If, after a set number of function evaluations, the error does not show any tendency improvement, a region is split into a number of subregions. The subdivision stops when the error estimate is smaller than the requested error relative to the region size. In that case the region becomes inactive based on the local adaptive criteria.

We studied a problem with applications for the calculation of cross sections in high energy physics [59]. The integrand of the sample integral $\int_{-1}^{1}\int_{-1}^{1} dx_1 dx_2 \frac{\varepsilon\sqrt{x_1^2+x_2^2}\,\theta(1-x_1^2-x_2^2)}{(x_1^2+x_2^2-a^2)^2+\varepsilon^2}$ is shown in Figure 6.2.

Results given in [20] indicated that regular adaptive methods failed to integrate this function to a relative accuracy of 10^{-5} for small values of ε, e.g. $\varepsilon = 10^{-5}$. This was

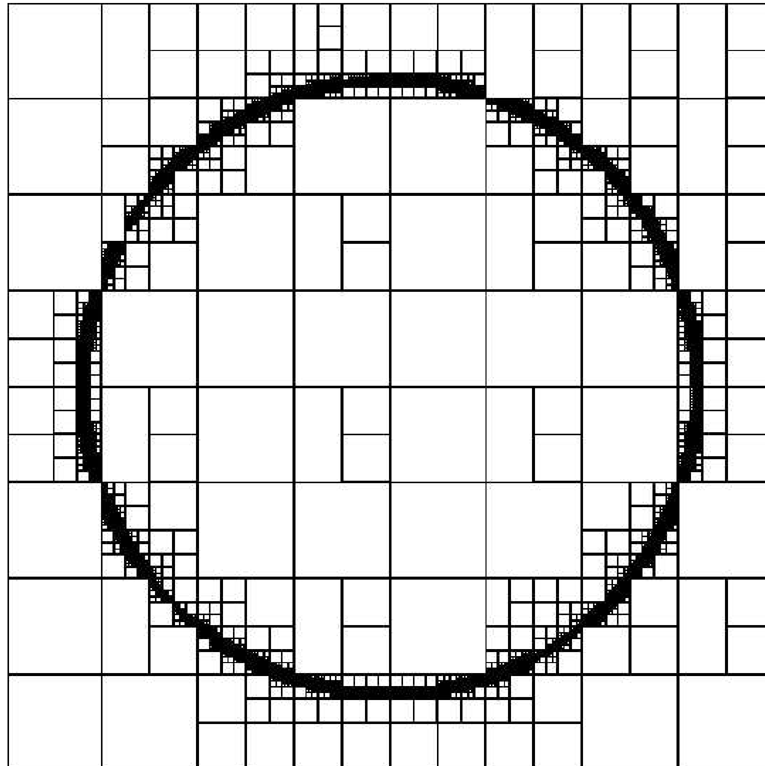


Figure 6.3 Domain Subdivision Plot

ascribed to a problem with the error estimates over the subregions generated, which caused the adaptive algorithm to focus on certain sections of the singularity, but leave large portions untouched, as shown in Figure 6.3.

More research is needed in this area, possible leading to specialized methods for handling these extremely difficult problems.

CHAPTER 7

Speedup and Efficiency Analysis Based on Work Unit Boundaries for Master Slave paradigm

In this chapter we investigate how to approach the more difficult problems with unknown work unit size, using a Master-Slave paradigm.

7.1 The Effect of Bounded/Unbounded Work Unit

For workload W, we denote by T_s the sequential runtime, by T_p the parallel runtime (using p processors), by T_c the penalty for the transfer of a work unit w, and by T_w the processing time of the work unit. Figure 7.1 shows a communication process between two processors (typical for sending a request, processing the request and sending a response). Thus T_c is a type of round trip communication time.

Assuming there is no significant breaking loss or network congestion, the total CPU time used by the parallel run is $T_{totalCPU} = p \cdot T_p = T_s + k \cdot T_c$, where k is the number of

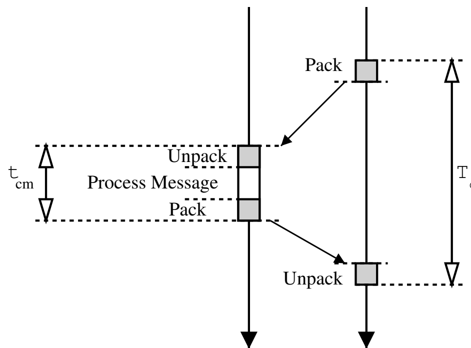


Figure 7.1 Communication Time

work units processed [43]. The efficiency is:

$$E = \frac{T_s}{p \cdot T_p} = \frac{T_s}{T_s + k \cdot T_c} = \frac{1}{1 + k \cdot \frac{T_c}{T_s}}. \quad (7.1)$$

To investigate the effect of communication, we compute the number of work units k_h that reduces the efficiency to $E_h = 50\%$ (this corresponds to an isoefficiency function for a parallel efficiency of 50%).

$$E_h = \frac{1}{2} = \frac{1}{1 + k_h \cdot \frac{T_c}{T_s}} \quad \Rightarrow \quad k_h = \frac{T_s}{T_c} \quad (7.2)$$

For a given system and problem, it is possible to find T_s and T_c. We can estimate the number of messages per second that need to be processed by the controller when the efficiency is reduced to this level:

$$K_c = \frac{k_h}{T_p} = \frac{T_s}{T_c \cdot T_p} = \frac{p}{2 \cdot T_c} \quad (7.3)$$

For example, for a problem of T_s=10 sec, in a system where T_c=0.2 milliseconds, k_h is 50,000, while for $p = 2$, K_c is 5,000. At this rate, we expect a significant network congestion. Figure 7.2 shows T_c in a worst case scenario.

For problems which have no bounds on the size of the work unit, the number of work transfers can be unlimited, the efficiency depreciating accordingly.

7.2 Investigating the Bottleneck Effect

The time used by the controller to process one incoming message is denoted by t_{cm}. There will be congestion at the controller if the incoming message rate exceeds $\frac{1}{t_{cm}}$ mes-

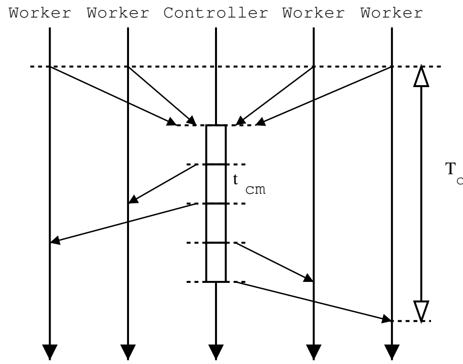


Figure 7.2 Bottleneck at Controller

sages per second, or for p workers, the average time between two consecutive messages sent by one worker is less than $p \cdot t_{cm}$.

We can reduce the bottleneck effect at the controller if we can force the work units size to satisfy:

$$T_w + T_c \geq p \cdot t_{cm}. \tag{7.4}$$

If $T_w + T_c \leq p \cdot t_{cm}$, the total parallel runtime is $T_p = k \cdot t_{cm}$, regardless of the number of processors.

To investigate this effect, we developed a simple simulator for which we can control: the number of work units, the size of a unit, and the amount of work performed by the controller for each work unit. The simulator runs over a network of workstations and uses MPI. Figure 7.3 shows the speedup obtained for a problem with 100 work units each requiring $T_w = 150$ milliseconds and different values of t_{cm}. Considering $T_c \simeq t_{cm}$ we observed a sharp decrease in efficiency when $T_w + T_c$ reaches $p \cdot t_{cm}$. After this point, the speedup remains fairly constant near $\frac{T_s}{T_p} = \frac{T_w}{t_{cm}}$

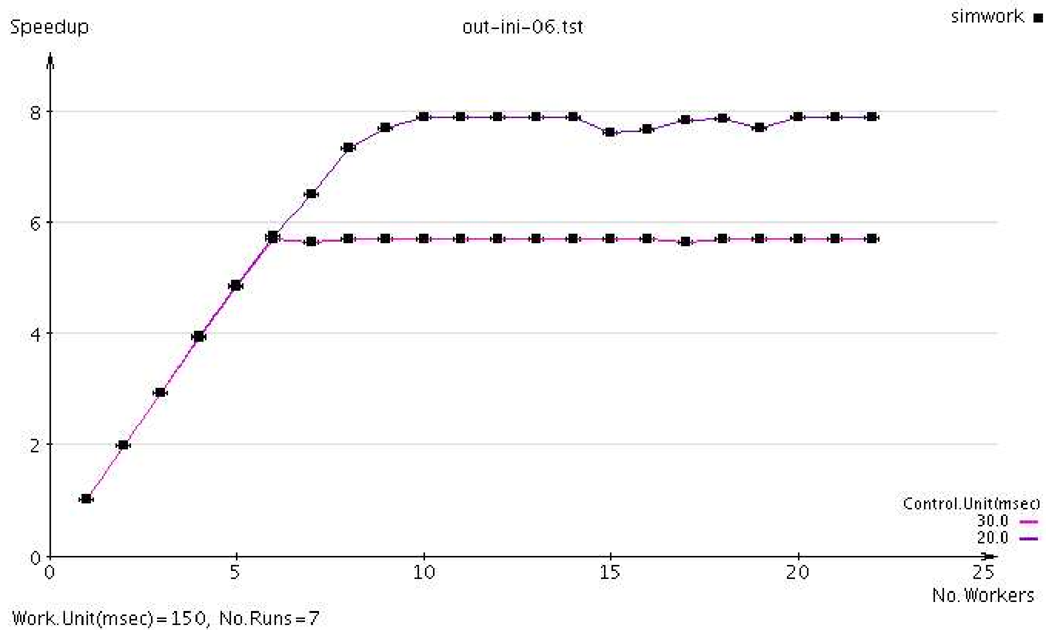


Figure 7.3 Communication Bottleneck Effect for 100 Work Units

Indeed, increasing the workload size to 1000 work units does not improve the efficiency, see Figure 7.4.

If we increase the size of the work unit proportionally with the number of processors, we obtain an increase in the efficiency due to alleviating the bottleneck.

Another way of eliminating the bottleneck is by using intermediate collectors of information. Some of the workers can collect data from other workers and send a unique package to the controller. The effect of additional controllers on the overall efficiency is a topic for further investigation.

7.3 Overlapping Computation with Communication

If we have knowledge of the work unit size, we can overlap the computation $T_w^{(i)}$ with the communication $T_c^{(i+1)}$ if $T_w^{(i)} \geq T_c^{(i+1)}$ (Figure 7.5). A new work unit will be available

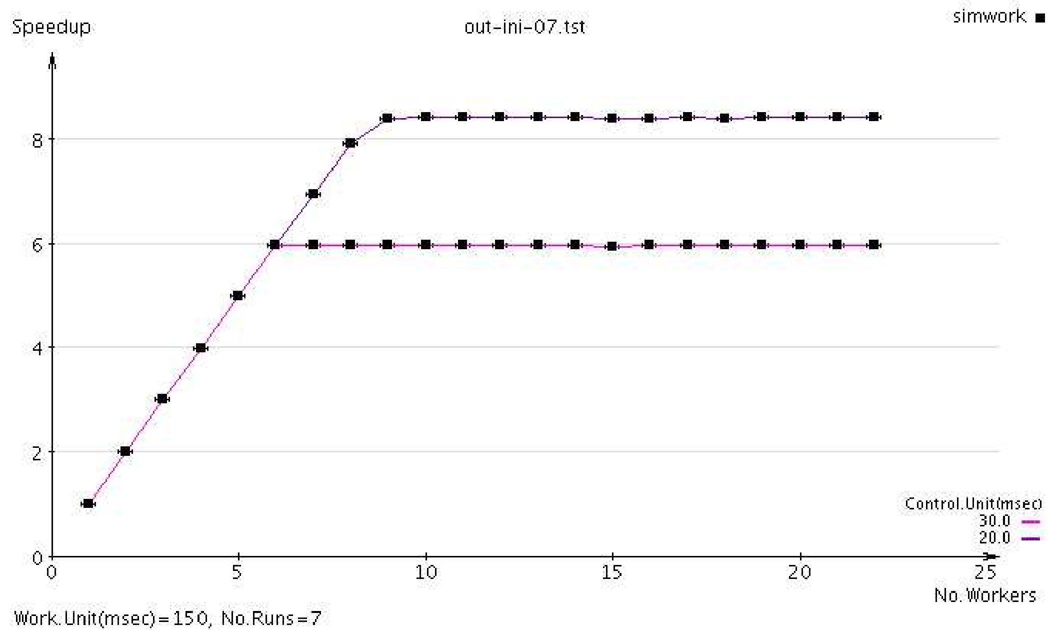


Figure 7.4 Communication Bottleneck Effect for 1000 Work Units

just before completing the current one. It is generally desired that no more than one work unit be transferred in advance to any one worker.

Depending on the location where the new work is generated, we consider the following cases:

I) the controller distributes work units from a work pool readily available;

II) each worker generates new work on the fly (for example, by splitting the current work).

For case I) there is an easy solution to overlap the computation with communication which, for work units with known boundaries, falls under the "embarrassingly parallel" category. As soon as the controller receives a result, it sends a new work unit. The new work unit can wait in the worker message queue until the current one is finished. Knowledge about the work unit boundaries serves to avoid the bottleneck at the controller and breaking loss at

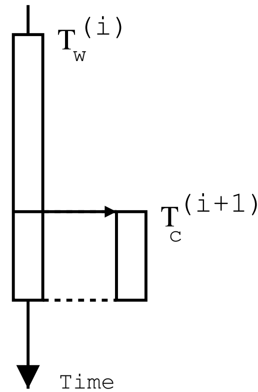


Figure 7.5 Computation-Communication Overlap

the end of the computation (otherwise the computation needs to be investigated under case II)).

Case II) is more interesting and is studied in the following section.

7.3.1 Overlapping Computation with Communication in Work Redistribution

When the work load requires dynamic redistribution among the participating workers, overlapping computation with communication can be an expensive process.

We investigate a strategy based on two special functions: *work_estimate* and *fair_split*. The *work_estimate* function must give an approximate finishing time of the current work unit while *fair_split* must generate a new work unit of a certain minimum size.

Every t_r seconds each worker will send a status message to the controller with an estimate of its work. The id's of the workers to finish in a certain interval are sent to those with more work. Note the particular cases where: a worker receives more than one work unit request at a time; or no worker receives a work request message.

We can estimate the current work unit termination time by quantifying the execution

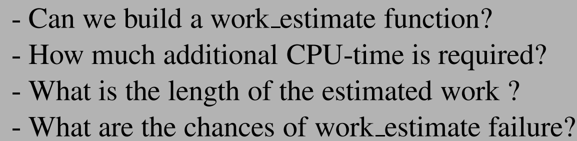


Figure 7.6 Questions on: Work Estimation Function

progress by a real value, for example, the number of iterations, accuracy, etc. Every t_r seconds, each worker calls a function to record the work unit progress. Using an extrapolation function, we may be able to estimate the finishing time for reaching the goal.

Creating a *fair_split* function presents challenges. This function must generate a new work unit of a minimum processing time. Note that, in order to properly use *fair_split*, we need a *work_estimate* function (unless the size of each work unit generated by *fair_split* is known precisely). The *work_estimate* function is particularly important when we do not know the upper boundaries of the unit, or we lack information on the CPU processing time. For a given work load and a given system, answering the questions in Figure 7.6 may provide valuable efficiency information.

To assure immediate response to a work request, we consider computation and communication performed in separate threads (or using a similar approach).

We use a clock synchronization mechanism (either provided by the system or implemented in the program), such that the wall clocks of different machines are within dt seconds. Every t_r seconds, each worker sends its status to the controller; the message includes the estimated finishing time t_f of its work unit. The controller translates t_f to a global time

T_f (by looking at worker's clock shift). A work request is sent on behalf of this worker if the following condition is satisfied: $T_f - T_{crt} \leq b \cdot T_c$ where b is a constant, and T_{crt} is the current time. Figure 7.7 depicts in gray the time spent by each worker performing the estimate and sending the update to the controller.

Consider the scenario where a worker makes a request for a new work unit only when it becomes idle. The request goes to the controller, from which it gets forwarded to another worker, which (if still busy) sends a new work unit to the requester. Since we have defined T_c as being round trip communication time, the total idle time is $1.5 \cdot T_c + T_{awt}$, where T_{awt} is the Average Waiting Time at the controller before the message gets processed.

Considering T_2 to be the time a worker sends an update message used by the controller to identify its finishing time, then computation can be overlapped with communication during a time span of length $T_f - T_2 \approx 1.5 \cdot T_c + T_{awt}$, given that the work unit size satisfies:

$$T_w \geq t_r + 1.5 \cdot T_c + T_{awt} \tag{7.5}$$

What is the efficiency degradation due to the estimation function and update message?

If we assume that the estimation is performed in δu seconds, then every worker, every t_r seconds, will spend an extra δu time performing the estimation function. The new parallel time is $T'_p = T_p + \delta u \cdot T_p / t_r$ and the new efficiency is:

$$E' = \frac{T_s}{p \cdot T'_p} = \frac{T_s}{p \cdot T_p \cdot (1 + \frac{\delta u}{t_r})} = \frac{E}{1 + \frac{\delta u}{t_r}} \tag{7.6}$$

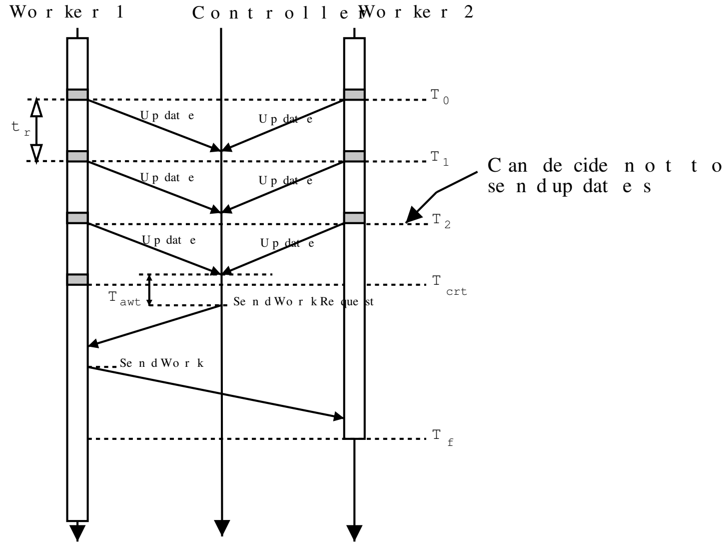


Figure 7.7 Communication Overlapping Algorithm

Given δu we can compute t_r so that in order to have δE efficiency degradation:

$$t_r \geq \delta u(\frac{E}{\delta E} - 1) \tag{7.7}$$

Putting together (7.4), (7.5), and (7.7), and assuming $T_{awt} \approx 0$, the work unit size must be:

$$T_w \geq 1.5 \cdot T_c + max(\delta u(\frac{E}{\delta E} - 1), p \cdot t_{cm}) \tag{7.8}$$

7.4 Properties of the Communication System

In this section we investigate a number of characteristics of the communication system that can help in deciding the distribution strategy:

1) The actual values for round trip communication time T_c (ethernet, myrinet);

2) The real value for the t_{cm} processing time of an incoming message, which gives an estimate of the number of messages per second that leads to the bottleneck effect (at any

time, the controller must not have more than a constant number of message in its message queue);

3) How much of T_c is used for packing/unpacking a message and how much is used for data transfer;

4) What is the effect of multiple messages in the system. This can be assessed by checking the network switch capabilities, and by measuring the communication time between two processors when there is no other communication in the network, and when the network is loaded (i.e., when there are $p/2$ distinct pairs or processors, out of p processors, that communicate simultaneously);

5) What is the effect of processing a work unit and communicating at the same time. In other words if communication and processing is done in parallel (by having separate threads or processes), how much will that slow down the work unit processing.

A grand goal is: given a network of workstations, by collecting some information about the system, generate requirements for the work unit size to obtain a given efficiency for a certain number of processors; or, given certain work unit size boundaries, to estimate the efficiency for a number of processors.

In our preliminary tests we found the measurement data to be highly skewed. Figure 7.8 plots the round trip communication time for three runs, each having 40 measurements. The message size is 1 byte. 85% of the measurements are in a small interval around $1.6 \cdot 10^{-4}$ seconds, but a number of times we consistently found values up to 70% higher than the median. We attribute these high values to some external factors that need to be considered separately. In the following sections we use the median of 40 observations on three different

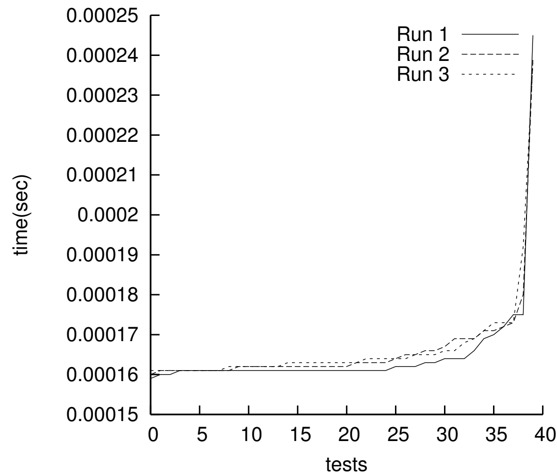


Figure 7.8 Round Trip Communication Time

architectures:

a) Fast Ethernet and 1.2MHz Athlon CPU;

b) Fast Ethernet and 0.8Mhz Athlon CPU;

c) Myrinet and 1.2Mhz Athlon CPU.

7.4.1 Round Trip Communication

This section presents results of measuring the round trip communication time T_c for varying message length. Figure 7.9 lists the algorithm used for this measurement. Table 7.4.1 shows T_c when the network and the CPUs are dedicated, while Table 7.4.1 shows the corresponding communication times when the network is congested: 30 processors execute the same communication pattern. We note a small increase in communication time (of $20\mu sec$) when the ethernet network is flooded with messages, however the myrinet network does not seem to be affected.

```
/* the even processors start the communication: i <--> i+1*/
if(id%2 == 0)
{
        run_time = MPI_Wtime() ;
        MPI_Send(buf, MsgSize, MPI_CHAR, id+1, tag_1, MPI_COMM_WORLD);
        MPI_Recv(buf, MsgSize, MPI_CHAR, id+1, tag_2, MPI_COMM_WORLD, &status);
        run_time = MPI_Wtime() - run_time;
}else
{
        MPI_Recv(buf, MsgSize, MPI_CHAR, id-1, tag_1, MPI_COMM_WORLD, &status);
        MPI_Send(buf, MsgSize, MPI_CHAR, id-1, tag_2, MPI_COMM_WORLD);
}
```

Figure 7.9 Timing Code for Round Trip Communication

MsgSize(bytes)	ethernet+0.8Ghz	ethernet+1.2Ghz	myrinet+1.2Ghz
1	0.000193	0.000163	0.000019
10	0.000197	0.000166	0.000019
20	0.000198	0.000168	0.000020
30	0.000203	0.000172	0.000020
40	0.000209	0.000177	0.000021
50	0.000213	0.000182	0.000021
100	0.000234	0.000201	0.000027
200	0.000271	0.000241	0.000043
500	0.000389	0.000362	0.000070
1000	0.000581	0.000560	0.000106
5000	0.001371	0.001343	0.000351

Table 7.1 Round Trip Communication Time: Free Network

MsgSize(bytes)	ethernet+0.8Ghz	ethernet+1.2Ghz	myrinet+1.2Ghz
1	0.000228	0.000180	0.000019
10	0.000229	0.000183	0.000019
20	0.000233	0.000186	0.000020
30	0.000237	0.000189	0.000020
40	0.000241	0.000194	0.000021
50	0.000247	0.000200	0.000021
100	0.000268	0.000219	0.000027
200	0.000307	0.000259	0.000043
500	0.000424	0.000380	0.000070
1000	0.000617	0.000578	0.000106
5000	0.001373	0.001343	0.000349

Table 7.2 Round Trip Communication Time: Congested Network

7.4.2 MPI Send Message

This measurement captures the time used by the non-blocking MPI_Send. A message is sent from the controller to 29 workers. The algorithm is presented in Figure 7.10. Table 7.4.2 shows a very small overhead for the non-blocking send operation (as the message size increases).

MsgSize(bytes)	ethernet+0.8Ghz	ethernet+1.2Ghz	myrinet+1.2Ghz
1	0.000031	0.000019	0.000001
10	0.000004	0.000009	0.000000
20	0.000004	0.000012	0.000000
30	0.000004	0.000002	0.000000
40	0.000004	0.000003	0.000000
50	0.000005	0.000003	0.000000
100	0.000010	0.000004	0.000001
200	0.000014	0.000005	0.000001
500	0.000010	0.000006	0.000002
1000	0.000014	0.000008	0.000003
5000	0.000044	0.000033	0.000021

Table 7.3 Send Time: Master $\longrightarrow$ 29 Workers

```
if(id==0)
{
        time_start = MPI_Wtime() ;
        for(k=1;k<nb_proc;k++)
                MPI_Send(buf, MsgSize, MPI_CHAR, k, tag_1, MPI_COMM_WORLD);
        time = (MPI_Wtime() - time_start)/nb_proc;
}
else
        MPI_Recv(buf, MsgSize, MPI_CHAR, 0, tag_1, MPI_COMM_WORLD, &status);
```

Figure 7.10 Timing Code for Send Message

7.4.3 Collective Communication

For the collective communications measurements we start the clock when all the participating workers are about to enter the communication function. Otherwise the communication time will reflect arbitrary delays representing the time to reach the communication function by some workers. For this reason we use a barrier just before starting the timer. Often the MPI device for a network of workstations implements the barrier with a broadcast [35]. The broadcast is often implemented with a tree expansion communication algorithm. The difference between the exit point from the barrier for different CPUs can be up to T_c. This difference may be reflected in the collective communication time.

CHAPTER 8

Sharing Status Information

One problem that needs to be addressed is how the workers will share status information. The controller has to receive status information periodically in order to properly redirect work requests to those CPUs which have more work left.

8.1 Computation Communication Interface

We define a work unit as the amount of work executed by a worker that, once started, does not require any additional information to be completed.

Let us first look at the structure of the work unit from two perspectives:

I) how easy it is to split a work unit;

II) how easy it is to extract information about the amount of work performed (progress) on a work unit at any given point in time.

We assume that a work unit can be composed of two main building blocks:

1) for loops,

2) indivisible tasks.

The “for loops” may have a known finishing time or an arbitrary finishing time. The “indivisible tasks” are purely sequential; but without prior knowledge it may be difficult to estimate their finishing time.

If we cannot estimate the termination time, it is very difficult to predict the efficiency of the parallel computation.

Let us assume the work unit can be split into arbitrary chunks of known size. This can be done if the work unit has a main loop of known size. Under this condition, the question arises how often data has to be exchanged between processors in order to maintain an efficient computation.

We propose a computation-communication interface based on two procedures:

1. Update Function Call. An update function is called within the main work unit loop every X iterations, which can decide to send an update with the computation status and/or handle requests for help.

2. Parallel Communication. The communication and the computation are performed in separate threads or processes [36], [60]. Assuming that while processing a work unit, the computation process (PR) does not require any communication, the communication process (CM) will: process all the messages, ask PR for its status information, or ask PR to split its work.

Assuming a work unit composed of k_u indivisible subtasks of size t_u, without having the communication and the computation in separate processes, a work request has to wait t_i seconds until a subtask is finished in order to send out a response. For a uniform distribution of work request arrival time, t_i is approximately $t_u/2$, and the corresponding efficiency degradation is:

$$E = \frac{1}{1 + k \cdot \frac{t_i}{T_s}} \tag{8.1}$$

where k is the number of work units, and T_s is the total sequential time, $T_s = k \cdot k_u \cdot t_u$.

Thus the efficiency degradation is:

$$E = \frac{1}{1 + \frac{k \cdot t_u}{2 \cdot k \cdot k_u \cdot t_u}} = \frac{2 \cdot k_u}{2 \cdot k_u + 1} \tag{8.2}$$

For $k_u = 10$, the efficiency degradation is 5%. This shows that for relatively big work units, waiting for a response can be a significant source of inefficiency.

The proposed mechanism presents a number of challenges:

1. The interval between two updates can be very small or very large, both will affect the efficiency.

2. The system requires additional work to synchronize the processes. CM needs to identify how much work PR has left by either consulting a shared memory variable (updated by PR) or by investigating the data processed by PR (in which case the data access must be protected by monitors).

8.2 Efficiency Analysis when Using $fair_split$ Function

For a work load of $T_s = k \cdot T_w$, of k work units, the parallel execution time (without considering the breaking loss effect) is $T_p = T_s + k \cdot T_c$. Let us consider a $fair_split$ function that takes t_s seconds to execute, and reduce the total number of work units from k to $\rho_x \cdot k$. In estimating the influence of the $fair_split$ function on the overall efficiency, the following questions arise: (i) Given ρ_x, what should the execution time t_s be in order to maintain an efficient computation? (ii) For a given t_s, what should the value of ρ_x be?

The parallel execution time when using the $fair_split$ function is

$T'_p = T_s + k \cdot \rho_x \cdot T_c + k \cdot \rho_x \cdot t_s$. For a better efficiency, $T'_p < T_p$, which implies $k \cdot T_c - k \cdot \rho_x \cdot T_c - k \cdot \rho_x \cdot t_s > 0$, i.e., $T_c \cdot (1 - \rho_x) - \rho_x \cdot t_s > 0$. For a given ρ_x, this leads

to

$$t_s < T_c \cdot \frac{1 - \rho_x}{\rho_x} \tag{8.3}$$

and for a given t_s,

$$\rho_x < \frac{T_c}{T_c + t_s}. \tag{8.4}$$

Example, for $T_c = .2$ milliseconds, and $\rho_x = 0.01$, t_s must be less than $99 * .2 \approx 20$ milliseconds.

It may be noted that a $fair_split$ function it can also be used to alleviate the bottleneck effect. Furthermore the $fair_split$ function will generally depend on the type of problem at hand.

CHAPTER 9

Conclusions

The present work has addressed various efficiency issues in distributed computing. Considering the task to be computed in a distributed system as being characterized by two factors, the number of work units and the size of each work unit, we investigated how these may affect the efficiency of the computation.

We studied a number of representative test cases, where either the number of work units, or/and the size of the work units is known, or no information about the structure of the work is available.

For problems with an a-priori known workload structure, we demonstrated how the MaxFlow algorithm can improve the efficiency. In this category we studied file distribution and cluster optimization problems. As representative cases where the number of work units is unknown but the unit size is known, we presented distributed Monte Carlo and quasi-Monte Carlo integration methods. We built a theoretical model to explain various inefficiency issues. The Known number of units and Unknown unit size (KnUs) case is represented by the Hierarchical integration method. The most difficult case, when the workload structure is unknown, is represented by adaptive numerical integration.

Even though challenges exist regardless of the structure of the workload, in general, work units of arbitrary size may constitute a more serious source of inefficiency than an unknown total number of units.

We developed a theoretical foundation for problems with bounded work units. Investi-

gating workloads with bounded work units is important not only for problem with unknown work units, but also for problems with known units that run on a network of heterogeneous workstations under variable additional loads.

Appendix

A ”gRpas”, a Tool for Performance Testing and Analysis

This section introduces a testing tool developed by the author to plot, analyze and store experimental results. The tool has been used extensively for the majority of the experiments presented in this work. This section is part of publication [11].

A.1 Introduction

Software applications in general and scientific computing programs in particular can be seen as multidimensional functions: $f : R^n \rightarrow R^m$. A program can take an n-variate input, and generate an m-variate output. To analyze the program behavior, the investigator selects a set of input values and compares the output with results obtained using a different method. The complexity of evaluating the program arises through a number of factors including: the problem to be solved and its parameters, the algorithm used and its parameters, random factors, total number of processors involved, etc.

For a given problem, some algorithms perform better than others, while for a given algorithm some problems will be solved more efficiently. Efficiency related results are generally affected by the problem parameters and the algorithm parameters.

The random factor is related to the fact that some algorithms require a set of random values. Ideally, the outcome must not depend on the random values; however this is not always the case. By repeating the same test a number of times, the developer is able to see if the output of the algorithm tends to the same value.

The number of processors involved can complicate the analysis. On the one hand, for

each processor, the developer must monitor a number of efficiency related parameters; on the other hand, a disturbance of the distributed environment (such as in sharing bandwidth with another program) can possibly affect the outcome, introducing a random factor.

For those problems for which there are no proved solutions (either because they are too hard and require approximations - as for NP complete problems, or because they involve too many extra conditions - as in parallel processing, adaptive numerical integration, etc.), the development cycle often iterates as follows:

a) find a new algorithm (or start from a current version);

b) perform a set of tests;

c) based on results analysis improve the algorithm.

For the later stages, some common questions are: How much testing must be done to obtain significant results? Is the new algorithm better and how much better? Despite the fact that there are no practical limits on the amount of data that can be collected from the tests, it is desired to perform as few tests as possible (to save CPU cycles, or decrease development time).

Tests are usually automated by varying one input parameter while keeping the others constant. If, for the ith parameter, k_i different values are tested, then a problem with n input parameters and m output values generates $m \prod_{1 \leq i \leq n} k_i$ numbers to analyze.

Often, to evaluate improvements, the same set of tests is performed for different versions. In [37], Hooker mentions two types of algorithm comparisons: (i) competitive testing, and (ii) controlled experimentation. Competitive testing is more suitable for development, and tells which algorithm is faster but not why, whereas controlled experimentation

is suitable for research and gives insight on how the code behaves under certain conditions. We can relate this classification with the following: (i) compare results from two or more versions; (ii) examine results from the same algorithm version.

Algorithm comparison in (i) relies on examining the difference in result vectors from different algorithms for corresponding input parameters. There is often a trade-off between the amount of work done and the quality of the result. Algorithm A may be pessimistic and B more optimistic, so that requesting less accuracy from A may lead to results equivalent to B's. The performance profiles technique of [44] accounts for this characteristic in the comparison of A and B.

In (ii), the algorithm may be tested with respect to the influence of random factors, either due to the environment or a random component in the algorithm (such as through the use of random numbers). The difference between the (sample) result vectors in m-space is significant to measure performance dependence on the random factor(s). Their distance can be accounted for by using the mean and variance with respect to some or all of the output components.

Performance measures to test algorithm behavior as a function of varying input parameters are problem related. For example, "result" and "estimated error" output parameters can be analyzed as a function of input "tolerated error" by comparing the actual error (if known) and estimated error to the level tolerated. The situation is complicated by the fact that the dependence on different input parameters may be correlated, such as that of "actual error" on "tolerated error" and "allowed number of iterations". As another example, various scalability measures (as a function of the number of processors used) are discussed

in [66].

Another issue to take into account is the stability of the algorithm. An unstable algorithm may cause a large change in results for a small change in the input parameters, or, magnify the small errors from earlier computation stages until the result deviates completely from the true answer. Examples can be found in recurrence relations, like in computing the nth order Bessel function denoted by $J_n(x)$ using the (forward) recurrence, $J_{n+1} = \frac{2n}{x} J_n(x) - J_{n-1}(x)$, which leads to a numerical catastrophe for $n \geq x$ when finite precision arithmetic is used [50]. Solving Fredholm equations of the first kind also belongs to this category [54].

A.2 Testing Numerical Integration Algorithms

The two main ingredients of a numerical integration problem are: $I(\alpha)$ the integral, and $M(\beta)$ the integration method, where α and β are various parameters. We denote by I_j a problem $I(\alpha_j)$ and by M_k a method $M(\beta_k)$. We consider the integrand function, integration region, requested accuracy and maximum number of function evaluations as part of the integration parameters α. A few examples of integration methods are: adaptive, Monte Carlo and quasi-Monte Carlo methods. Various cubature rules that can be used at the basis of adaptive multivariate integration methods are listed in [7].

A major issue in numerical integration is that often we don't know the characteristics of the integrand function. In general, we don't have answers to questions like: are there any singularities, how steep are they and where are they located?

We denote by M_j^o the best algorithm that solves I_j, that is, the algorithm with the best

convergence. Let $C(I, M)$ be the cost of solving I using algorithm M, expressed in an appropriate measurement unit (e.g., number of function evaluations performed).

Given M, it is our goal to develop a set of tests to:

a) Find β_k such that on the average over all $I_j, C(I, M)$ is minimum. - Since it is impossible to handle all I_j, we need to determine a suitable test set for which we require that on the average $C(I, M)$ is minimized. Alternatively we may require that $M(\beta_k) = M_j^o$ for I_j in the test set. Or, we may just want to extract as much information as possible about the algorithm behavior.

b) Compare M_k with other algorithms for the entire test space.

A.3 gRpas tool

gRpas [10] is a small application designed to help analyze results from scientific applications. Its main features are listed bellow.

Flexible Application Interface

For inter-operability and flexibility, all data is stored in an XML like format. For each input/output parameter, the user specifies a set of attributes: name, type, and tag. These are stored in a type repository and are used by the GUI module and re-used for similar tests. All results, along with input parameters, are stored in a tagged format.

Option for filtering or combining results

Tests are performed in groups. Usually a set of tests is performed, analyzed and if there are satisfactory results, another group of tests is performed. If the results are not as

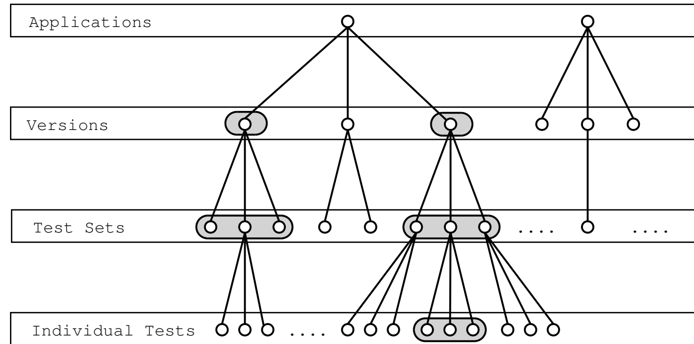


Figure A.1 Tests Tree Structure

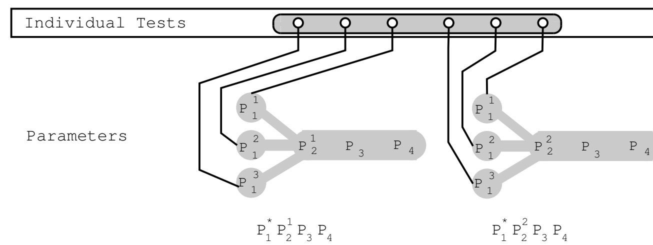


Figure A.2 Individual Tests and Parameters

expected, the code is changed and the same set of tests is repeated. *gRpas* can combine the results from different groups of tests or can present them individually. Figure A.1 shows the tree structure: applications - versions - tests. In Figure A.2 a Test Set is expanded to show the relation between Individual Tests and input parameters for an application with 4 parameters: p_1, p_2, p_3, p_4, where p_1 gets assigned three values and p_2 two. The Test Set is composed of 6 Individual Tests.

Note that although the version can be seen as an algorithm parameter, from the practical point of view it is better to consider it as a separate entity.

The user can filter certain results across multiple sets of tests for individual analysis, or

can combine multiple tests from multiple sources.

Flexible User Interface

The application is mainly designed for algorithm comparisons, and is mainly intended for scientific software developers. The user interface is simple and is point-and-click based. This allows for a short learning curve and fast response to the user. After data is collected, a wide range of plots can be generated. Parameters defined in the input files can be selected by their names from drop-down menus. Multiple plots can be drawn in the same window and multiple windows can be displayed at once.

Figure A.3 shows the main user interface. The table in the lower part lists all the Test Sets performed for a particular version of the code (this corresponds to the shaded areas in the *Test Sets* level in Figure A.1). In the rightmost column, the title shows the name of the code version, and the cells show the time when the Sets were generated. The middle column shows the parameters used in the Set, and the left column shows the associated name for the Set.

Once the user has selected a Test Set, its parameters are shown in the control group situated on the right (corresponding to the type of area that is shaded in the *Individual Tests* level in Figure A.1). This section has a double role: to show and to select the parameters to be plotted. In the top part, the user selects the Y coordinate from a drop list containing all the output variables, while the X coordinate is selected from the input parameters with multiple values. Once the user has selected a parameter set, the plots are displayed in the center-left area.

The plots in Figure A.3 result from solving multivariate integration problems using a quasi-Monte Carlo method with up to 60 processors. The variable input parameters are: Integrand Function, Number of Processors, and Function Limit. This particular Set consists of 325 Individual Tests, each repeated a number of times. On the left, the Run Time is plotted versus the Number of Processors for all the integrand functions using a different Function evaluation Limit, whereas on the right the obtained error is plotted. Each plotted point represents the median of multiple values. The data interval is shown as a closed segment.

Figure A.4 represents two algorithm versions (note an additional column in the table Test Sets area). The two versions have a number of identical Test Sets: *Test s01*, *Test s02*, *Test s05*, and *Test s06* (which can be identified by the date in the algorithm version column). The plot shows the differences in the result values.

User defined plugins

Extern modules can be linked to gRpas for additional functionality. The user can pre-process the output before displaying, or can link in a different application. The plugins must be written in java. The extern modules are called using menus. For example, Figure A.5 shows the effect of applying the scalability (speedup) plugin: for each test, the plot shows the inverse of the runtime multiplied with the runtime of the first test (which corresponds to the sequential run).

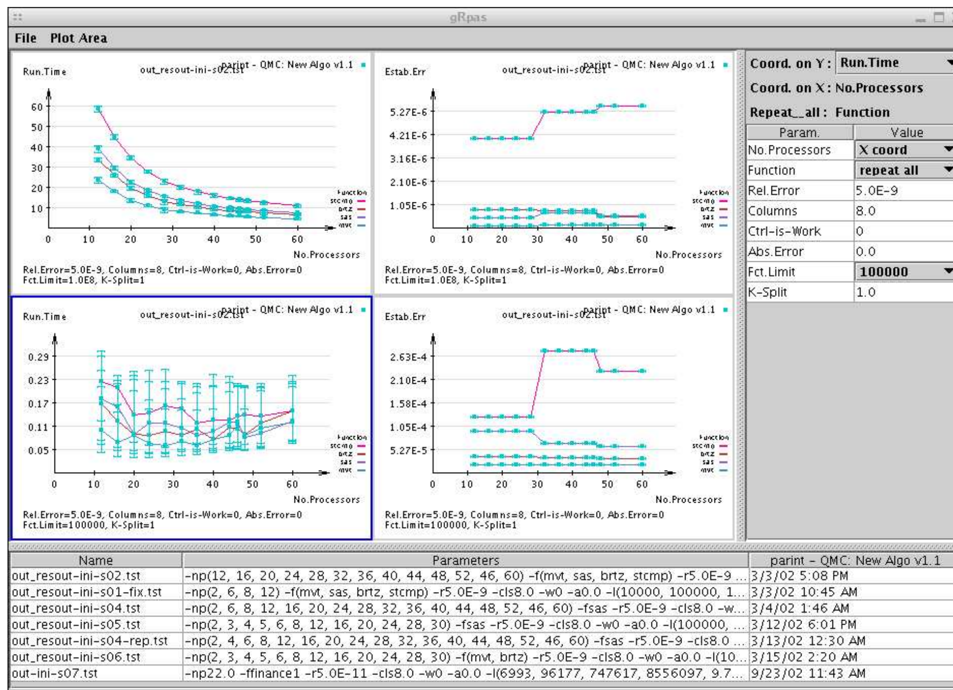


Figure A.3 gRpas : Multiple Plots, Multiple Window

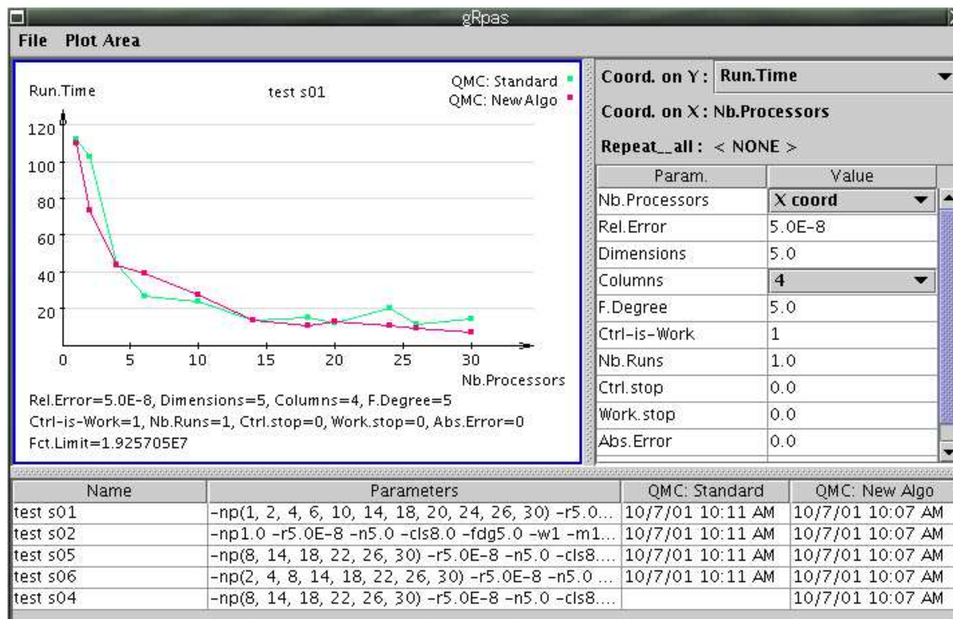


Figure A.4 gRpas : Multiple Algorithm Plot

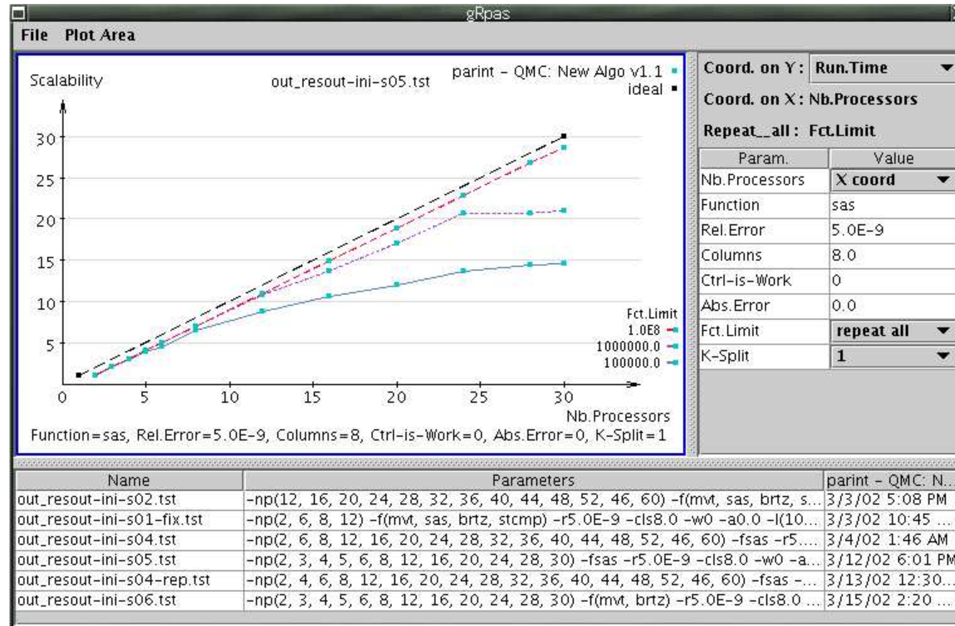


Figure A.5 gRpas : Scalability (speedup) Plugin

Statistical reports

As mentioned earlier, the developer is interested in two types of comparisons. First, for a given algorithm M, what is the value of β such that $M(\beta)$ gives the best performance for a set of problems I_j. Second, given two or more algorithms, which one performs better for the same set of problems.

Most of the time, a simple inspection of the plots gives enough information to continue analyzing the algorithm or to change it.

To generate more elaborate statistics we have two options, either implement a small set of functions in java as plugins, or export the data to applications like Splus or SAS. It is our goal to implement a set of functions that can perform comparisons at the click of a button.

For further consideration, we envision a new functionality for generating plots from

non-common-parameters tests. Instead of performing Test Sets where some parameters are fixed while others vary, is possible to generate indicative plots from Tests that do not share common parameters. In this way it is possible to reduce the number of tests performed, while increasing the problem investigation space.

A.4 Conclusions

In this section we presented “gRpas”, a tool to help analyze test results for scientific computing applications. Intended to be used for both competitive testing as well as controlled experimentation, the application has been used extensively in our parallel numerical integration research. We included samples of its usage from solving multivariate t-distribution (MVT) problems using a quasi-Monte Carlo method with up to 60 processors. The “gRpas” application and code is available for download at [10]. It can also be executed online as an applet at [10].

BIBLIOGRAPHY

1. ATHENA web site, https://aegis.cs.wmich.edu.
2. BANICESCU, I., AND VELUSAMY, V. Performance of scheduling scientific applications with adaptive weighted factoring. In *Proceeding of the IEEE International Parallel and Distributed Processing Symposium Heterogeneous Computing Workshop (on CD-ROM)* (2001).
3. BARAHONA, F., JÜNGER, M., AND REINELT, G. Experiments in quadratic 0-1 programming. *Mathematical Programming 44* (1989), 127–137.
4. BERNTSEN, J., ESPELID, T. O., AND GENZ, A. An adaptive algorithm for the approximate calculation of multiple integrals. *ACM Trans. Math. Softw. 17* (1991), 437–451.
5. BILLIONNET, A., COSTA, M. C., AND SUTTER, A. An efficient algorithm for a task allocation problem. *J. ACM 39*, 2 (1992), 502–518.
6. BUYYA, R. Grid computing info center. http://www.gridcomputing.com.
7. COOLS, R. Monomial cubature rules since "Stroud": a compilation – part 2. *Journal of Computational and Applied Mathematics 112(1-2)* (1999), 21–27.
8. CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to Algorithms*. The MIT press, 1994.
9. CRANLEY, R., AND PATTERSON, T. N. L. Randomization of number theoretic methods for multiple integration. *SIAM J. Numer. Anal. 13* (1976), 904–914.
10. CUCOS, L. grpas - home page. http://aegis.cs.wmich.edu/~lcucos/prjs/grpas/doc/gRpas.html.
11. CUCOS, L., AND DE DONCKER, E. gRpas, a tool for performance testing and analysis. Submitted.
12. CUCOS, L., AND DE DONCKER, E. Distributed QMC algorithms: New strategies and performance evaluation. In *Proceedings of the High Performance Computing Symposium (HPC'02)* (2002), pp. 155–159.
13. DE DONCKER, E., CUCOS, L., AND GUAN, Y. Resouce allocation for clusters. In *Proceedings of the High Performance Computing Symposium (HPC'01)* (2001).
14. DE DONCKER, E., CUCOS, L., AND ZANNY, R. Parallel multivariate integration: Paradigms and applications. In *Joint Statistical Conferences (JSM'02)* (2002). CD-ROM Proceedings.

15. DE DONCKER, E., CUCOS, L., ZANNY, R., AND GENZ, A. Parallel computation of the multivariate t-distribution. In *Proceedings of the High Performance Computing Symposium (HPC'01)* (2001), pp. 129–134.

16. DE DONCKER, E., GUPTA, A., BALL, J., EALY, P., AND GENZ, A. ParInt: A software package for parallel integration. In 10th *ACM International Conference on Supercomputing* (1996), Kluwer Academic Publishers, pp. 149–156.

17. DE DONCKER, E., GUPTA, A., AND CUCOS, L. On the scalable computation of large sets of integrals. In *16th International Conference on Parallel and Distributed Computing Systems* (2003). CD ROM proceedings.

18. DE DONCKER, E., GUPTA, A., GENZ, A., EALY, P., LIU, J., AND SUREKA, A. Use of ParInt for the parallel computation of statistics integrals. *Computing Science and Statistics 27* (1996).

19. DE DONCKER, E., GUPTA, A., AND ZANNY, R. Large scale parallel numerical integration. *Journal of Computational and Applied Mathematics 112* (1999), 29–44.

20. DE DONCKER, E., KAUGARS, K., CUCOS, L., AND ZANNY, R. Current status of the ParInt package for parallel multivariate integration. In *Proc. of Computational Particle Physics Symposium (CPP 2001)* (2001), pp. 110–119.

21. DE DONCKER, E., ZANNY, R., CIOBANU, M., AND GUAN, Y. Distributed Quasi Monte-Carlo methods in a heterogeneous environment. In *Proceedings of the IPDPS Heterogeneous Computing Workshop* (2000), pp. 200–206.

22. DE DONCKER, E., ZANNY, R., KAUGARS, K., AND CUCOS, L. Multivariate integration and visualization with ParInt/ParVis. Submitted.

23. DE DONCKER, E., ZANNY, R., KAUGARS, K., AND CUCOS, L. Performance and irregular behavior of adaptive task partitioning. In *Lecture Notes in Computer Science* (2001), vol. 2074, Springer-Verlag, pp. 118–127.

24. DECKER, T., FISHER, M., LÜLING, R., AND TSCHÖKE, S. A distributed load balancing algorithm for heterogeneous parallel computing systems. In *PDPTA* (1998).

25. EVERETT, H. I. Generalized Lagrange multiplier method for solving problems of optimal allocation of resources. *Operations Research 11* (1963), 399–417.

26. GEIST, A., BEGUELIN, A., DONGARRA, J., JIANG, W., MANCHEK, R., AND SUNDERAM, V. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

27. GENZ, A. Testing multidimensional integration routines. In *Tools, Methods and Languages for Scientific and Engineering Computation* (1984), pp. 81–94.

28. GENZ, A. An adaptive numerical integration algorithm for simplices. In *Lecture Notes in Computer Science* (1990), N. A. Sherwani, E. de Doncker, and J. A. Kapenga, Eds., vol. 507, pp. 279–285.

29. GENZ, A. MVNDST: Software for the numerical computation of multivariate normal probabilities. http://www.sci.wsu.edu/math/faculty/genz/homepage.

30. GENZ, A., AND MALIK, A. An adaptive algorithm for numerical integration over an n-dimensional rectangular region. *Journal of Computational and Applied Mathematics 6* (1980), 295–302.

31. GENZ, A., AND MALIK, A. An imbedded family of multidimensional integration rules. *SIAM J. Numer. Anal. 20* (1983), 580–588.

32. GOODRICH, M. T., AND TAMASSIA, R. *Algorithm design, Fundations, Analysis, and Internet Examples*. John Wiley and Sons, Inc., 2002.

33. GREENBERG, H. J. Mathematical programming glossary (2001) . http://carbon.cudenver.edu/greenbe/∼glossary.

34. GREENBERG, H. J. The generalized penalty function/ surrogate model. *Operations Research 21* (1973), 162–178.

35. GROPP, W., AND LUSK, E. Installation and User's Guide to mpich, a Portable Implementation of MPI Version 1.2.5 The chp4 device for Workstation Networks. http://www-unix.mcs.anl.gov/mpi/mpich/.

36. GROPP, W., LUSK, E., AND SKJELLUM, A. *Using MPI: portable parallel programming with the message-passing interface*. MIT Press, 1994.

37. HOOKER, J. N. Testing heuristics: We have it all wrong. *Journal of Heuristics 1*, 1 (1995), 33–42.

38. HUMMEL, S. F., SCHMIDT, J. P., UMA, R. N., AND WEIN, J. Load-sharing in heterogeneous systems via weighted factoring. In *ACM Symposium on Parallel Algorithms and Architectures* (1996), pp. 318–328.

39. KAPENGA, J., AND DE DONCKER, E. A parallelization of adaptive task partitioning algorithms. *Parallel Computing 7* (1988), 211–225.

40. KRISHNAN, M., GHAFOOR, S. K., AND BANICESCU, I. Lbsim: A load balancing simulator for data parallel applications. In *Proceedings of the High Performance Computing Symposium (on CD-ROM)* (2002).

41. KROMMER, A. R., AND UEBERHUBER, C. W. Architecture adaptive algorithms. *Parallel Computing 19* (1993), 409–435.

42. KUMAR, V., GRAMA, A., GUPTA, A., AND KARYPIS, G. *Introduction to Parallel Computing*. Benjamin Cummings Publishing Co., 1994.

43. KUMAR, V., GRAMA, A. Y., AND VEMPATY, N. R. Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing 22*, 1 (1994), 60–79.

44. LYNESS, J. N., AND KAGANOVE, J. J. A technique for comparing automatic quadrature routines. *The Computer Journal 20*, 2 (1977), 170–177.

45. MAERTEN, B., ROOSE, D., BASERMANN, A., FINGBERG, J., AND LONSDALE, G. DRAMA: A library for parallel dynamic load balancing of finite element applications. In *European Conference on Parallel Processing* (1999), pp. 313–316.

46. NAG, 1999. Numerical Algorithms Group, C Library Mark 5.

47. NINOMYA, S., AND TEZUKA, S. Toward real-time pricing of complex financial derivatives. *Applied Mathematical Finance 3* (1996), 1–20.

48. PAPADIMITRIOU, C., AND STEIGLITZ, K. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982.

49. PASKOV, S. H. Computing high dimensional integrals with applications to finance. Tech. rep., Columbia University, Department of Computer Science, 1994. CUCS-023-94.

50. PATTERSON, T. Sorry - wrong number!, 1988. Inaugural lecture.

51. PICARD, J., AND RATLIFF, H. A graph theoretic equivalence for integer programs. *Operations Research 21* (1973), 261–269.

52. PICARD, J., AND RATLIFF, H. Minimum cuts and related problems. *Networks 5* (1975), 357–370.

53. PIESSENS, R., DE DONCKER, E., ÜBERHUBER, C. W., AND KAHANER, D. K. *QUADPACK, A Subroutine Package for Automatic Integration*. Springer Series in Computational Mathematics. Springer-Verlag, 1983.

54. PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., AND FLANNERY, B. P. *Numerical Recipes in C - The Art of Scientific Computing*. Cambridge University Press, 1997. Second Edition.

55. RICE, J. R. A metalgorithm for adaptive quadrature. *Journal of the Association for Computing Machinery 22* (1975), 61–82.

56. SCHUERER, R. Parallel high-dimensional integration: Quasi-Monte Carlo versus adaptive cubature rules. In *Lecture Notes in Computer Science* (2001), vol. 2073, Springer-Verlag, pp. 1262–1271.

57. SHAPIRO, H. D. Increasing robustness in global adaptive quadrature through interval selection heuristics. *ACM Transactions on Mathematical Software 10*, 2 (1984), 117–139.

58. SUSAN, F. H., SCHONBERG, E., AND FLYNN, L. E. Factoring: A method for scheduling parallel loops. *Communications of the ACM 35(8)* (1992), 90–101. get.

59. TOBIMATSU, K., AND KAWABATA, S. Multidimesional integration routine dice. Tech. rep., Kogakuin University, 1998. Tech.Rep.85.

60. TREUMANN, R. Experiences in the implementation of a Thread Safe, Threads Based MPI for the IBM RS/6000 SP. http://www.research.ibm.com/actc/Tools/MPI_Threads.htm.

61. TSCHÖKE, S. Portable parallel branch-and-bound library. http:// www.uni-paderborn.de/∼ppbb-lib.

62. WHANG, K. *Advanced Computer Architecture*. McGraw-Hill, Inc., 1993.

63. XU, C., TSCHÖKE, S., AND MONIEN, B. Performance evaluation of load distribution strategies in parallel branch and bound computations. In *Proc. of the 7th IEEE Symposium on Parallel and Distributed Processing, SPDP'95* (1995), pp. 402–405.

64. ZANNY, R., AND DE DONCKER, E. Work anomaly in distributed adaptive partitioning algorithms. In *Proceedings of the High Performance Computing Symposium (HPC'00)* (2000), pp. 178–183.

65. ZANNY, R., DE DONCKER, E., KAUGARS, K., AND CUCOS, L. PARINT1.2 *User Manual*, 2002. Available at http://www.cs.wmich.edu/∼parint.

66. ZANNY, R., KAUGARS, K., AND DE DONCKER, E. Scalability of branch-and-bound and adaptive integration. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'01)* (2001), pp. 674–680.

67. ZENG, H., DEVAPRASAD, J., KRISHNAN, M., BANICESCU, I., AND ZHU, J. Improving load balancing and data locality in n-body simulations via adaptive weighted fractiling. In *Proceedings of the High Performance Computing Symposium (on CD-ROM)* (2002).