To the Graduate Council:

I am submitting herewith a thesis written by Chris M. Jepeway entitled "The Design, Implementation, and Evaluation of RCalc." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science with a major in Computer Science.

Dr. Jeffrey D. Case, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

Associate Vice Chancellor
and Dean of The Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Chris M. Ippenway

Date 27 APR 94

THE DESIGN, IMPLEMENTATION, AND EVALUATION OF RCALC

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Chris Markland Jepeway

May 1994

Acknowledgments

This work would never have been completed without the support of a host of friends, colleagues and mentors. I thank them all as a group; there are some, however, that I wish to thank individually.

I thank Dr. Case for dragging me through this endeavor and for sponsoring my research. I thank Drs. Dunigan and Mutchler for serving on my Thesis Committee. I thank Dr. Poore for supporting me in my work. I thank everyone who has worked in the Computer Science Department Labs over the past 7 years for their help. I thank Sam Nicholson for his glee when reading an early draft of this thesis. I thank Therese Pepin for her support and encouragement when I began writing RCalc. I thank Bob Manchek for sharing in the development of my views on the academic process. I thank Arthur C. Guinness, as others have, for inspiration. And I thank Aaron and Audrey Bruner for still liking me, even though I have given them many, many reasons why they shouldn't. Mostly, though, I thank Sharon Lewis for her patience and help during the four years it took me to track down this vampire and pound a stake through its heart.

Abstract

The software framework presented herein, RCalc, provides Artificial Intelligence researchers programming in the Common Lisp language with a multiple-instruction, multiple-data (MIMD) super-computer built from a collection of Unix[1] workstations interconnected via a local area network (LAN) utilizing the Transmission Control Protocol/Internet Protocol (TCP/IP) protocol suite.

RCalc is most suitable for parallel algorithms adhering to the bag-of-work paradigm. A model of bag-of-work algorithms as implemented using RCalc-like systems is presented. It is used to analytically derive a predictor for the efficiency, E, of parallel algorithms fitting the model.

To apply the predictor, one must know certain characteristics of the hardware and software realizing the algorithm. A testing methodology that can determine these characteristics is developed, and it is applied to RCalc. The results of this application are used to improve the predictor of E, and the improved predictor is validated by applying it and the testing methodology to a Parallelized Simulated Annealing Solver of the Travelling Salesman Problem (PSA-TSP) implemented in RCalc.

[1] Unix is a trademark of Unix System Laboratories, Inc.

Contents

List of Figures

List of Tables

Chapter 1

Introduction to The Problem

It is a truism that parallel computing offers the only means of sidestepping limits placed on processor speeds by the absolute nature of the speed of light. Ready access to parallel architectures attempts to keep pace with the compute power required by the growing number of research problems that push these limits. One cost-effective means of providing such access can be found in software that combines an institution's general purpose workstations using the local area network (LAN) that interconnects them: in effect, the LAN then functions as the backplane of a large multiple-instruction, multiple-data computer [CM87]. Software systems that combine general-purpose, networked workstations into a multiple instruction, multiple data (MIMD) virtual computer will be called MIMD-LAN systems or MIMD-LAN frameworks. At the time this research began investigating the problems implementing MIMD-LAN frameworks[1], few software frameworks facilitating this combina-

[1] Working prototypes of the software described here existed as of late June 1989.

tion existed. Those that did exist were often limited in their implementation. Some, *e.g.*, [GGM+, Kil88], had a low ceiling on the number of workstations that could participate in a computation; some, [WL88], *e.g.*, were only implemented on mainframe class hosts; some, such as [Gai85], were of historical interest only; and others, such as [LMN88, Sun88b], were predominately intended for distributed computing in heterogeneous environments and did not support parallelism.

Since the commencement of the research reported herein, a vast number of frameworks combining workstations into MIMD computers have been developed; [Sun90, RSL92, BKT92, CS92, AOC+88, Che92, PD91] are but a few. Though the shortcomings, if any, of these newer frameworks are less severe than earlier frameworks, of necessity all are limited in the languages they support. Some, *e.g.*, [RSL92, Sun90, CS92], are intended for use by scientific researchers, and therefore support the C or Fortran programming Languages. Some, such as [BKT92, Che92, JLHB88, AOC+88], implement new languages with which to write parallel programs. Few target artificial intelligence researchers programming in Common Lisp (CL); [PD91] does.

The problem, then, is to provide AI programmers using Common Lisp with a framework for applying the compute power of a LAN of workstations to their parallel applications. The focus of the problem is narrowed to Unix[2] workstations that communicate using the Transmission Control Protocol/Internet Protocol (TCP/IP) protocol suite, since these workstations are widely available in the research community. It is further narrowed to certain types of of Artificial Intelligence (AI) applications, to those heuristic searches where

[2]Unix is a trademark of Unix Systems Laboratories, Inc.

"bag-of-work" style parallelism[3] applies. Two examples of such search techniques occur in genetic algorithms [Hol75], where individuals in a population are elements of the bag, and in simulated annealing [KGV83], where annealing schedules are bag elements.

1.1 Statement of The Problem

The purpose of this research is to

- design, implement, and evaluate a software framework, known as RCalc, that enables Unix workstations to participate in bag-of-work style parallel computations written in the Common Lisp language, and
- to provide potential users of the framework with a prediction of the performance they might expect from the framework.

RCalc has been implemented on a network of Sun Sparc workstations using a combination of Austin Kyoto Common Lisp and the American National Standards Institute standardization of the C language [KR88].

[3]Briefly, computations patterned on the bag-of-work paradigm can be decomposed into multiple tasks that can proceed in parallel. One or more decomposing processes store task descriptors in an abstract data structure, the "bag." In parallel, worker processes repeatedly remove a task descriptor from the "bag" and perform their chosen task. Further details can be found in [CG88].

1.2 Significance of the Study

1.2.1 National and International Impacts

Arguably, there is no need for another software system tying a collection of loosely coupled workstations into a MIMD computer: as noted in the introduction, there are already many such. However, while comparative studies of these alternatives exist ([CS92, DS92, Bal91]), only [Bal91] analyzes the suitability of a particular framework to general classes of applications. One national and international impact of this study will be to facilitate pragmatic comparisons of MIMD-LAN systems: an evaluation strategy is presented that aids a programmer in determining whether a given system is suitable to a particular bag-of-work application.

While further development of RCalc would only duplicate on-going research efforts across the world, the application of the author's experience with low-level communication routines and with Common Lisp interfaces to other MIMD-LAN systems is, potentially, a second national and international impact of this study.

1.2.2 Local Impacts

At present, Lisp programmers within the Computer Science Department, University of Tennessee, Knoxville use the Department's distributed file system as a message-passing mechanism between workstations participating in parallel computations. One impact of the study will be to provide these researchers with a better means of parallelizing their applications, while giving them some idea of the performance they can expect from the framework.

Instrumentation tools developed during the implementation of RCalc have a wide applicability. Another local impact will occur when these tools are made available to other software development efforts within the department.

1.3 Research Questions

The goal in answering this study's research questions is twofold:

- to provide a predictor for the performance that can be obtained using RCalc, where the standard notion of efficiency[4] is taken as an appropriate performance metric, and
- to provide a notion of the accuracy of this predictor.

A first-order model of bag-of-work computations as realizable using RCalc is presented. The efficiency of the modelled computations is then derived analytically, based on the model's parameters. To apply this analytic predictor to a given bag-of-work algorithm in a given environment, performance characteristics of RCalc must be known. A testing methodology is devised to obtain them, and it is applied to RCalc. The analytic predictor is applied to those test results which it can forecast, and a comparison is made between the empirically obtained results and the analytically predicted results. The analytic predictor is then modified to account for discrepancies discovered by the comparison. Finally, the analytic predictor of efficiency and the testing methodology necessary to apply it are validated by comparing their predictions to an implementation of an example AI bag-of-work algorithm.

[4]The efficiency of a parallel computation is the ratio of the speedup obtained by the parallel computation to the number of processors used in that computation.

1.3.1 QUESTION ONE: Derive a Predictor of Efficiency

Formally stated, Q1 is:

> To what extent can a predictor for the efficiency of bag-of-work algorithms as realized using RCalc be analytically derived?

This question is answered by first modelling a subset of bag-of-work algorithms which can be realized using RCalc. The efficiency of these algorithms is then analytically derived as a function of the model's parameters and performance characteristics of the RCalc implementation. The model is described in Section 3.3, and the analytic derivation of its efficiency is given in Section 3.4. An attempt will be made to provide an upper bound on the efficiency and to predict its average value.

Important issues in answering this question include determining the nature of the algorithms that can be modelled, determining the parameters of the model, understanding the assumptions made about the modelled algorithms and about RCalc, and determining which characteristics of RCalc–as implemented in the experimental environment–must be known in order to apply the predictor to an example bag-of-work algorithm.

1.3.2 QUESTION TWO: Test and Modify the Predictor

Q2 is broken into the following two sub-questions:

> To what extent can a testing methodology be devised so that the analytic predictor of efficiency may be applied to bag-of-work algorithms as implemented in RCalc?

and

To what extent can the analytic predictor of efficiency be modified to account for the results of applying the testing methodology to RCalc?

The result of answering the first of these questions is a testing methodology; the results of the second, the fruits of applying the methodology in an experimental environment, including additional insight into the RCalc implementation and bag-of-work algorithms in general.

The approach to be taken in answering Q2-A begins by designing a simple bag-of-work algorithm which fits the model and which has model parameters that can be varied in a known manner. Given an implementation in RCalc of this tunable algorithm, one can vary its input, observe its behavior, and derive performance characteristics of both the algorithm and RCalc that must be known in order to apply the analytic predictor. As an example, one portion of the test methodology will likely involve discovering the relationship between the algorithm's input and the complexity of its bag tasks. Determining which of the algorithm's model parameters can be varied with its input, how the model parameters are to be varied to obtain clear results, and what performance characteristics can be obtained by varying them are all important aspects of answering Q2-A.

Answering Q2-B requires that the testing methodology be applied to an RCalc implementation of the test algorithm in an experimental environment, that the analytic predictor of efficiency be applied to the algorithm's implementation, that the prediction so obtained be compared with empirical results, and that the predictor be modified to the extent possible to account for these results. The goal in answering Q2-B is to improve the accuracy of the predictor by utilizing it in a controlled test environment.

1.3.3 QUESTION THREE: Validate the Predictor

A formal statement of Q3 is:

> To what extent is the combination of the analytic predictor and the testing methodology validated when applying it to an example bag-of-work algorithm?

An example bag-of-work application, a solver for the Travelling Salesman Problem (TSP) based on Simulated Annealing (SA), is implemented in Common Lisp using RCalc. Those parts of the testing methodology that are algorithm-specific are applied to the SA-based TSP-solver, and the analytic predictor is used to forecast its efficiency when using different numbers of Unix workstations. The efficiency of the TSP-solver is then empirically obtained, and the forecasted and observed efficiencies are compared. The goal in answering Q3 is to provide the potential user of RCalc with an idea of how well the predicted efficiency will match actual performance obtained using RCalc.

1.4 Assumptions of the Study

This study is based on several assumptions. Those that are not discussed later are presented here.

1.4.1 No Kernel Modifications

Some MIMD-LAN systems rely on changes to the Unix kernel, typically requiring special-purpose network drivers for their improved performance. For reasons of portability, RCalc does not rely on kernel modifications.

1.4.2 Security is Unimportant

It is assumed that the framework will be used in a research environment, one in which security is undesirable. There is no provision in the framework for access control, user authentication, data encryption, malicious packet rejection, and the like. Providing security measures under the current implementation is not considered.

1.5 Limitations of the Study

This study addresses the problems presented in Section 1.1. However, it is limited in the scope to which it does so.

1.5.1 Only First-Order Effects are Modelled

The model used to predict and observe RCalc's behavior does not consider second-order effects. In particular, no attempt is made to model demand-varying network throughput, as is found when using Ethernet.

1.5.2 No Similar Framework is Examined

The results of this study only apply to RCalc. No experimental comparison is made between RCalc and other frameworks providing similar capabilities.

Chapter 2

Review of the Literature

There are two major parts of the framework:

- its model of parallel, distributed programming
- its communication method

The first of the following two sections surveys other parallel, distributed programming frameworks; the second, reliable datagram protocols, the communications method used by the framework.

2.1 Parallel, Distributed Programming Frameworks

This section presents an overview of methods both available and historic that address the problem of how to accomplish parallel, distributed computation using general-purpose processors sharing some communications medium. The sections are ordered chronologically,

either by the date of the method's publication or the date of the method's most recent implementation, whichever is more recent.

2.1.1 Remote Procedure Calls

A remote procedure call facility (an RPC) provides communication between network nodes via a programmatic interface that passes parameters to, and returns results from, a procedure running on another node. The RPC paradigm has been theoretically presented in [Nel79], and its seminal implementation has been described in [BN84]. The reader desiring further details of the remote procedure call facility is directed to those two papers. A widely available implementation of RPC, the drawbacks of which inspired parts of RCalc's implementation, is presented as illustrative of the form available in the late 1980's: an examination of RPC as developed and distributed by Sun Microsystems, Inc. is given.

Sun's remote procedure call facility, confusingly called Remote Procedure Call (RPC), allows heterogeneous computing platforms to make procedure calls to one another using the network interconnecting them. It uses the eXternal Data Representation (XDR) [Sun87] to encode all data transmitted in a canonical form. Because Sun's popular Network File System (NFS) is RPC based, RPC is widely available across various architectures and operating systems.

In theory, RPC could be directly used to implement parallel computation: one would simply write the procedures one wished to call, make them available to the RPC mechanisms on the hosts one wished to use, call them asynchronously or via a broadcast mechanism, and reap their results. In practice, however, this approach is not feasible for three reasons:

1. *Using RPC is difficult.* As the RPC manual distributed by Sun Microsystems, Inc.

itself states, the upper layers of RPC are sufficiently inflexible that one generally makes use of RPC's bottommost layer for "serious" applications [Sun88a]. The details of that layer are overwhelming [Sun88b], and their complexity hampers an understanding of how one might use it well.

2. *Broadcasting over RPC is resource intensive.* The RPC broadcast mechanism utilizes the broadcast mechanism of the underlying transport. Every host sharing that transport, then, will be interrupted to either service or discard the RPC broadcast. While this overhead is acceptable for applications broadcasting only infrequently and aperiodically (for instance, when a person requests the names of users currently logged into all hosts on a network), it is unacceptable for the repeated, insistent usage generated by the message-passing necessary to parallel, distributed processing[1].

3. *Asynchronous RPC is not available.* Though the current RPC specification [Sun88b] clearly states they need not be, implementations of RPC are singly threaded from the calling side; no mechanism for doing asynchronous procedure calls exists in known RPC implementations[2]. Without asynchronous calls allowing the caller to start procedures while other remote procedures execute, parallelism cannot be achieved.

4. *Encapsulating procedure calls in light-weight processes is not portable.* As [NHSS87]

[1] In some vendors' platforms, RPC is so tightly coupled to system services that continuous, heavy use of RPC's broadcast mechanism causes catastrophic failures when critical network servers "thrash" while coping with the interrupts generated by broadcast packets. At the University of Tennessee, Knoxville, this is a well known and much loathed occurrence that results in simultaneous loss of service to over 400 such systems.

[2] At the time of RCalc's design, the author was familiar with RPC implementations from Sun Microsystems, Inc.; International Business Machines, Inc.; the University Of California, Berkeley; and Digital Equipment Corporation.

reports, a light-weight process (LWP)[3] can encapsulate a synchronous RPC while other LWPs sharing the same address space can continue to do useful work. However, most implementations of LWPs are either not freely available, or are architecture dependent.

Because of the ease with which RPC is incorporated into both standard procedure-based or functional programming languages, the RCalc design is based on the general RPC paradigm, extended to support parallelism through the use of asynchronous calls. For reasons presented later in Section 3.2.1, RCalc leverages off Sun's development of XDR.

2.1.2 MP

Developed in the mid 1980's at Tektronix, Inc, MP's purpose was to put the processing power of a collection of engineering workstations in the hands of parallel programmers, quite similar to the goals of RCalc [Gai85]. Processes in MP communicate via message-passing over a network capable of multicasting. MP's message-passing protocol is flow-controlled, but not reliable. MP processes are assumed to be short lived and to retain no state between receiving messages. Messages are assumed to be either requests that computations be performed or results of computations.

In MP,[4] a process is a set of shadow processes, each of which is built from the same executable image, each of which executes on a different workstation, and each of which

[3] A related group of light-weight processes share the same address space, but have different threads of control. LWPs thus have access to the same data for co-ordination purposes, the user is free to let some LWPs block while others execute, and the operating system implementing them can rapidly context switch between them.

[4] [Gai85] does not give an expansion for the acronym "MP."

receives the same input messages. This latter feature is implemented using the multicasting capabilities of the underlying network: each message destined for a particular process is multicast to all shadows in its set. One shadow process is elected to output messages on behalf of the process set; it is said to be "primary" for that set. Shadow members of the process dynamically re-elect a primary whenever the process has a new result to output by designating that shadow which responds first as the new primary. Since MP processes are stateless, there is no need to communicate any information from the old primary to the new primary. This "auto-election" mechanism provides fault tolerance, load balancing, and speedup, in favorable circumstances, of the square-root of the number of shadow processes [BD80].

RCalc provides mechanisms whereby MP's concept of stateless can be incorporated by the RCalc user to provide some degree of fault tolerance and load balancing (*cf* Section 3.2.2).

2.1.3 Linda

The designers of Linda[5] at Princeton University maintain that, when constructing programming models, there is a natural division between computation and co-ordination: "computation models allow construction of a single computational activity; the co-ordination model will bind separate activities into an ensemble" [GC92]. Linda, then, is their implementation of a co-ordination language based on a global tuple space, essentially "a shared associative memory." A tuple is a list of fields, each of which has a type and an optional value.

[5]Linda is a trademark of Scientific Computing Associates, Inc.

Co-ordination occurs through four operators that manipulate the tuple space:

out The *out* operator places tuples in the tuple space. It takes a tuple as an argument. Typically, fields in *out*'ed tuples have a value in addition to their mandatory type.

in The *in* operator removes a matching tuple from tuple space. Tuples match when they have the same number of fields and their fields match. Two fields match when

- their types match and
- their values, if both are specified, match.

in blocks if there are no matches in the tuple space.

rd *rd* is like *in*, except the matching tuple is not removed from the tuple space.

eval The *eval* operator *out*s what is known as a "live" tuple. Live tuples have a process associated with them that computes values of the fields. When the process finishes its computation(s), the fields of the live tuple take on the computed values, and the tuple is indistinguishable from any other *out*ed field.

At first glance, the utility of these primitives may not be apparent. [Lel90], however, attests to their power: an operating system with a Unix-like user interface has been implemented that uses *in*, *out*, *rd*, and *eval* to co-ordinate all of its kernel activities.

Linda has been implemented on a variety of parallel architectures in a variety of languages, including implementations embedded in C for networks of hosts [WL88, DS92]. [DS92] reports these implementations manage the global tuple-space efficiently and compare favorably with message-based parallel-processing systems such as the Parallel Virtual Machine (*cf* Section 2.1.6).

RCalc borrows little from Linda save the concept of bag-of-work parallelism, which, while not an explicit part of Linda, did help inspire Linda's tuple space.

2.1.4 The Distributed Processing Utilities Package

Developed at the Computer Science Department at the University of Colorado, Boulder, the Distributed Processing Utilities Package (DPUP) was designed to achieve coarse-grained parallelism out of a collection of workstations connected via a LAN [GGM+]. It provided a programmer with two communication paradigms: master/slave and broadcast group. Using the master/slave paradigm, a single master process sends free-format messages to slave processes initiated by the master using DPUP primitives based on the Transmission Control Protocol (TCP). Using the broadcast group model, a group of processes register their interest in a set of replicated broadcast variables. Processes access broadcast variables by exchanging messages with a local variable server. These servers loosely maintain the variables' consistency using protocol based on the User Datagram Protocol (UDP). DPUP's broadcast group model foreshadows the "data-objects" of Orca (*cf* Section 2.1.8) and borrows somewhat from Linda's "tuple-space" (*cf* Section 2.1.3).

While historically useful, DPUP's lasting success was in demonstrating the utility of a MIMD-LAN system that relied solely on standard Unix networking primitives. Many systems–PVM, the PrOgramming SYstem for distriButed appLications (POSYBL), RCalc itself– owe their development to DPUP's trailblazing.

After observing the tribulations of a study conducted within the Computer Science Department, University of Tennessee, Knoxville of a distributed algorithm implemented using DPUP [Sur89], the author concluded that a number of DPUP's drawbacks should be

eliminated in any similar framework. In particular, an explicit decision was made to allow RCalc to run on a host without intervention from a privileged user.

2.1.5 The Distributed Hypercube

Developed at the University of Tennessee, Knoxville, the Distributed Hypercube is a set of of library routines that provide a programming interface to a collection of homogeneous workstations that looks very much like that of an Intel Hypercube. The interface is based on TCP. [Kil88] reports that the Distributed Hypercube can yield performance equal to an Intel iPSC/2 when running compute-intensive parallel applications.

Both the limitations and the successes of the Distributed Hypercube directly spurred RCalc's development. As noted in [Kil88], the Distributed Hypercube could gain considerable performance improvement by utilizing a connectionless protocol instead of TCP, a connection-based protocol: the overhead of establishing and destroying connections would be eliminated. Because of this recommendation, RCalc utilizes a reliable datagram protocol as its network transport. [Kil88] also notes that the Distributed Hypercube suffers from a requirement that all hosts participating in a computation be of the same architecture and use the same data representation. RCalc uses XDR to achieve architecture independence, allowing hosts of different byte-ordering and languages that utilize different data formats to interoperate. The Distributed Hypercube's performance directly inspired the author to produce an ancestor of RCalc when confronted with a long-running genetic algorithm: while it was clear that run times could be drastically reduced by implementing this algorithm using the Distributed Hypercube, the onus of casting the algorithm in a form suitable for a hypercube implementation pushed the author into developing a new framework more suitable

for heuristic search.

Some frameworks have been developed in parallel with the one described in this research. These are described in the following sections.

2.1.6 The Parallel Virtual Machine

The Parallel Virtual Machine (PVM), first presented in [Sun90] addresses the same problems as does the framework. One achieves parallelism by decomposing one's computation into functional units that can operate in separate address spaces, then combining the units via message-based parallel programming primitives. Different functional units may reside on different hosts attached via a network, and one obtains parallelism from

- the concurrent execution of functional units
- any parallelism inherent in the hosts themselves.

One can consider the collection of hosts as a single virtual machine, hence the name PVM. The semantics of the primitives are managed and enforced on behalf of the functional units by per-host PVM daemons[6] which reliably transfer messages, synchronize interdependent functional units, choose hosts on which to instantiate new units, give notification of failed units and hosts, etc. Advantages of PVM include

- the ability to implement a functional unit on the architecture most appropriate to its function,

[6] In Unix parlance, a *daemon* is a long-lived server process invoked to do work on behalf of the invoking process.

- ready support for functional units written in differing languages, and
- a console process that can orchestrate PVM calculations under interactive control.

PVM has seen widespread use and is currently considered one of the better MIMD-LAN systems available. A great deal of supporting software for PVM exists, including a graphical programming language [BDG+93] and a load-balancing system [Sep93]. Experienced PVM researchers have concluded that bag-of-work parallelism is the paradigm of choice when communicating between functional units.

2.1.7 Jade

The Jade parallel-programming language [RSL92] extends the C language with syntactic additions that allow programmers to specify data-parallelism. Given these hints, the Jade system detects an application's coarse-grained concurrency, exploiting it dynamically as appropriate for the application's target environment. Jade programs can run unchanged at the source level on various architectures[7] and, regardless of environment, are guaranteed the execution semantics of a sequential, imperative, single-address space programming model. This guarantee allows debugging of the program to occur in an environment where subtle timing interactions need not be considered. The Jade implementation on a network of workstation-class machines uses PVM (*cf* Section 2.1.6) as a typed, reliable transport protocol.

By relying on a MIMD-LAN system as just another parallel computer, Jade demon-

[7] [RSL92] reports that Jade has been implemented on shared-memory multiprocessors, message-passing machines, systems with special-purpose functional units, as well as networks of heterogeneous workstations.

strates the maturity of the MIMD-LAN paradigm. With its transparent interface to a MIMD-LAN system hiding details of how many and what types of hosts are available to the parallel programmer, Jade is arguably the culmination of MIMD-LAN development.

2.1.8 Amoeba and Orca

Amoeba is an efficient distributed operating system designed on an RPC communication model which supports light-weight processes. An Amoeba system is built of

- workstations, used for tasks such as editing text which require fast interactive response;
- a pool of central processing units (CPUs), dynamically allocated to execute processes;
- specialized servers, residing at well-known locations, providing file service, directory service, etc; and
- a gateway, enabling communication with remote Amoeba systems.

As reported in [TvRvS+90], Amoeba is one of the fastest distributed operating systems implemented on slower CPUs: its RPC mechanism, when used to communicate between two workstations based on the Motorola Corporation 68030 CPU, is roughly six times as fast as that developed by Sun Microsystems, Inc. for very short messages, and twice as fast for medium-length messages. While originally conceived of as a system for distributed computing [TvRvS+90], Amoeba's robust design and efficient implementation led to its use as a parallel system. The parallel language Orca, providing a shared-memory paradigm, was designed and implemented to work under Amoeba.

In Orca, one shares data through "data-objects," essentially instances of a type defining private data and public interfaces to manipulate the data. Data-objects are analogous to a tuple space in Linda where the tuple operators are arbitrarily defined by the user. Data-objects are shared at process invocation: starting child processes can be passed data-objects, which are then shared between parent and child. This sharing is transitive: a data-object shared between parent and child and child and grandchild will be shared between parent and grandchild. Sharing is implemented in the distributed memory environment of Amoeba through replication and reliable broadcast: every processor has a copy of each shared data-object, and updates to the data-objects are reliably broadcast to all processors. This scheme is efficient: programs solving the Travelling Salesman Problem with 12 cities have achieved near-linear speedup using 16 processors [BKT92].

Both Amoeba and Orca have been implemented as new systems, unconstrained by issues of backwards compatibility or heterogeneity. A large part of Amoeba's and Orca's power is derived from such an uncluttered design environment. In contrast, RCalc seeks to provide a MIMD-LAN system to an already established audience utilizing heterogeneous architectures. As such, RCalc is neither as efficient as Amoeba nor as elegant as Orca.

2.1.9 The Parform

The Parform [CS92], developed at the University of Zurich Information Institute, is, in essence, a load-balanced PVM-like software system. Subtasks in The Parform are statically or dynamically load-balanced across a collection of workstations based on the following metrics:

- application-specific heuristics that predict load generated by subtasks

- CPU utilization as retained by the workstation's operating system
- available compute power on a workstation, as given by the number of times a counter is incremented during a sampling interval

Host-to-host communication in the Parform occurs directly at the data-link layer, without the overhead of intervening software at the transport and network level. While the Parform gains performance in bypassing these protocol layers[8], it does so at the cost of scalability: without utilizing network layer software, the Parform can only communicate with hosts directly attached to the LAN on which it operates.

2.1.10 Condor

Condor [LS92] is a software system that transparently provides access to compute power on idle Unix workstations. By restricting all I/O to read-only or write-only file access, a program may operate in the Condor environment. Condor programs are compiled, linked to a special run-time library, started on an initiator machine, and then, using check-point files maintained by the Condor system, migrated to idle workstations. The runtime library replaces Unix system calls with RPC routines that call back to a shadow process running on the initiator host. The shadow process handles I/O on behalf of the migratory process; thus, the migratory process appears to run in the file system of the initiator host. While not provided with explicit parallelism by the Condor system, Condor programs can achieve parallelism by running multiple copies of themselves. That is, once migrated, different copies may concurrently execute on different CPUs. Note that any communication

[8] [CS92] shows that the Parform performs better than PVM.

between these copies must take place in the file system of the initiator host. Condor's restrictions on I/O make file-based communication cumbersome and limit Condor programs to large-grained parallelism. Developing versions of Condor incorporate Linda's communication primitives [LS92] and will schedule PVM tasks [LP93]. Once a Linda/PVM-enhanced version of Condor is produced, Condor will become one of the most functional MIMD-LAN systems available. Until such time, Condor's primary utility lies in enabling programs to run for months at a time.

2.1.11 The CLOUDS Lisp Distributed Environment

The core of the CLOUDS Lisp Distributed Environment (CLiDE) is comprised of large-grained, persistent, distributed objects capable of evaluating Lisp expressions in the context of their own address space [PD91]. Access to CLiDE objects occurs through public interfaces. Both protection and consistency of these objects are enforced by the underlying CLOUDS operating system. Parallelism is achieved through support of the *future* abstraction reported in [RH85]: remote evaluations are first tagged as belonging to a group, then started asynchronously on other hosts, and, sometime in the *future*, are reaped or terminated based on which evaluations have completed.

The CLiDE developers have identified the same need for a parallel, distributed Lisp system that spurred the development of RCalc.

2.2 Reliable Datagram Protocols

RCalc uses OCARD[9], an unsequenced reliable datagram protocol built as part of this research, as a data transport. Such a protocol is inherently record-oriented, and no guarantees are made about the order in which messages are received, hence the appellation "unsequenced." The following sections survey both sequenced and unsequenced reliable datagram protocols.

2.2.1 Delta-t

Delta-t, developed at Lawrence Livermore National Labs, is an implicitly-connected transport protocol that supports both client/server transactions and byte stream style communications. A basic premise of its design is that the connection abstraction is an artifact of the services provided by a protocol (*e.g.*, error control, sequenced delivery) and has no bearing on the data the user of the protocol wishes to transfer from one site to another. Based on this premise, one of Delta-t's stated design goals is "to remove connection management as a user interface issue" [Wat81]: there are neither open nor close primitives in Delta-t. Rather, connections in Delta-t exist implicitly between every pair of nodes in a default state. Connections move to the active state when data are sent from one node to another, and they return to the default state when no data are sent during some interval. Delta-t uses timer-based mechanisms to place bounds on the lifetime of both packets and state information associated with connections, ensuring that old packets do not interfere with on-going data transfers. To enforce these bounds, Delta-t requires estimates of

[9]There is no acronymic expansion for the sequence of characters O-C-A-R-D.

- the maximum packet lifetime (MPL), the time it takes a packet to traverse the network,
- the maximum time a receiver delays before acknowledging a packet, and
- the maximum time a sender will attempt to deliver a packet in the absence of acknowledgements.

[Wat81] presents inexpensive methods to obtain the MPL from an underlying link-level protocol; other estimates are available based on the Delta-t implementation.

OCARD borrows Delta-t's notion of implicit connections, though it restricts the nodes to which implicit connections can exist to a set of hosts specified at the time OCARD is initialized.

2.2.2 The Reliable Datagram Protocol

The Reliable Datagram Protocol (RDP) was developed to support a network transparent filesystem, the Extended File System, at the Massachusetts Computer Corporation for their proprietary operating system [CFA85]. Layered on the Internet Protocol (IP), it extends the functionality of the standard IP datagram protocol, UDP, by providing reliable, ordered delivery of datagrams and by further demultiplexing datagrams based on transaction[10] identifiers. This latter extension amortizes the overhead of establishing a single connection across multiple processes sharing that connection, while providing the sharing processes with a means to quickly negotiate and manage sub-connections with peers on the remote host.

[10] A *transaction* is an exchange of related messages between two network entities. Transactions are often thought of as request/response messages between a client and a server.

It is particularly useful in an environment supporting LWPs (*cf* page 13). RDP's interface to its transaction mechanism accommodates degenerate transactions, where no reply is expected, and it allows for single message or multiple message transactions. By specifying the appropriate transaction model, RDP can thus act as an unsequenced or a sequenced reliable datagram protocol, with or without support for transactions. RDP's capability for tailoring its behavior to the needs of the application using it foreshadows the complex set of options provided by more recently designed protocols (*cf* Section 2.2.7, Section 2.2.8).

OCARD provides fixed functionality to the user, resembling RDP when used with degenerate transactions of a single message.

2.2.3 The Reliable Data Protocol

The Reliable Data Protocol (RDP), designed by Bolt, Beranek and Neuman Communications Corporation, Inc, supports full-duplex, reliable delivery of segments, units of data exchanged between peers that must be delivered atomically [VHS84]. The RDP user can, at connection-initiation, specify if segments must be delivered in the order sent by the remote peer or if it is acceptable to deliver them as they are received by the local RDP protocol module. Sequenced delivery may be necessary, for example, when presenting data to a human user; unsequenced delivery would be acceptable when loading a memory image across the network. To implement sequenced delivery, as well as for other bookkeeping purposes, RDP tags segments with monotonically increasing sequence numbers. RDP utilizes two types of piggy-backed acknowledgment[11]:

[11] Acknowledgements sent in packets containing user data are said to be *piggy-backed* on the user data.

- a *cumulative* acknowledgment lets a sender know that all segments up to and including a certain sequence number have been received, and
- a *non-cumulative* or *extended* acknowledgment indicates to a sender that a set of segments with designated sequence numbers has been received.

The type of acknowledgment used is a function of a particular implementation of RDP and is not under control of the RDP user: [VHS84] states that *extended* acknowledgments are intended to be used whenever segments are received out of order.

RDP provides flow control, using parameters negotiated at connection establishment. One parameter indicates the maximum size segment RDP is willing to accept. Another gives the maximum number of outstanding, unacknowledged segments allowed during a connection. The sender maintains a window of segments the width of this maximum and may not transmit segments with sequence numbers outside it. The sending window is anchored at the last sequence number cumulatively acknowledged by the receiver; it advances whenever another cumulative acknowledgment is received. The receiver maintains a window twice the width of this maximum, anchored similarly. Any segment received with a sequence number outside this window is dropped.

OCARD borrows RDP's extended acknowledgment mechanism as its sole form of acknowledgment (*cf* Section 3.2.4) under the assumption that, in an unsequenced reliable datagram protocol, cumulative acknowledgments might be needlessly delayed while awaiting mis-sequenced packets.

2.2.4 ISIS

Initially developed at Cornell University, ISIS is a distributed programming software system using the abstraction of a process group as its basic building block. A single process is a collection of LWPs (*cf* page 13), each known as a task. A process group collects one or more processes under a single name. ISIS provides primitives for a process to dynamically join, leave, and communicate with process groups using the group name. ISIS primitives also enable the spawning of new tasks based on "interesting" events, for example, when a message arrives or when a new process joins a group. ISIS further guarantees that all members of a process group see the same events in the same order. This guarantee, coupled with hooks provided to the user by ISIS to regenerate a process group's state, greatly simplifies the task of implementing fault-tolerant, distributed programs. ISIS runs on Unix workstations, and the ISIS libraries are usable from many different languages, including C, C++, LISP, and Prolog [BCJ+90]. Further development of ISIS has transferred from Cornell to Isis Distributed Systems, Inc., as has the copyright on a number of technical reports concerning ISIS. ISIS research has therefore had little impact on OCARD's development.

2.2.5 Prospero's Asynchronous Reliable Delivery Protocol

The Asynchronous Reliable Delivery Protocol (ARDP) was designed to provide reliability on top of UDP for Prospero, a distributed file system intended for use on a global scale. As such, ARDP directly supports client/server transactions: it allows transaction requests and responses to span multiple packets (*cf* segments in Section 2.2.3), a client can obtain the status of its transaction request, a server can redirect requests away from itself or forward them to other servers, a server can report the estimated time it will take for a transaction

to complete, etc. Acknowledgments in an ARDP packet can contain a bit vector specifying to the sender a list of received packets. [NA93] does not specify a flow control mechanism for ARDP; however, the implementation distributed with version 5 of Prospero limits the number of outstanding packets for a single request to seventy-five.

Had ARDP been available early in the design of RCalc, it is likely that it would have been used in place of OCARD.

2.2.6 The RHODOS Reliable Datagram Protocol

The RHODOS Reliable Datagram Protocol (RRDP) is designed to be an efficient protocol which the distributed operating system RHODOS can use for both internal and external communication. When used externally to communicate with remote RHODOS sites, RRDP is, like ARDP, layered on UDP. Using RRDP, messages bound from one process to another are split into datagrams which are then transmitted by the sender in fixed-sized groups. If the receiver successfully receives all datagrams in a group, it acknowledges the entire group. If, instead, the receiver detects the loss of one or more datagrams in the group, it negatively acknowledgesthe missing datagrams, indicating to the sender that the lost datagrams are to be retransmitted. Protocols that batch datagrams and their acknowledgements in this manner are termed multidatagram protocols[12]. RRDP transmits messages reliably between processes running on different hosts, where a message is analogous to an RDP segment (*cf* Section 2.2.3). [GZ90] has shown that grouping increases throughput and reduces message transfer delays, and that the benefits of grouping diminish as message

[12]Note that for a group size of one, multidatagram protocols degenerate into using a stop-and-wait (re-)transmission policy.

size increases from 8 kilobytes to 32 kilobytes. None of these effects would be apparent in OCARD's intended usage environment, where short, un-ordered messages are exchanged between hosts.

2.2.7 The Versatile Message Transaction Protocol

The Versatile Message Transaction Protocol (VMTP) was developed at Stanford University to support the V Kernel [Che88], a distributed operating system that makes heavy use of transaction-oriented remote procedure calls. VMTP thus presupposes all communication occurs between a single client making a request to one or more servers which, under normal circumstances, respond to that request. Transaction requests and responses can be of arbitrary length, delivered to applications in pieces or as a whole.

VMTP packets are sent to and from "entities," which provide or consume some service, such as a distributed database or a location service. Entities are tagged with 64-bit identifiers which are valid within a given administrative domain. Entity identifiers are independent of any network, any host, or any interface on a host; entities can thus migrate arbitrarily and can be composed of arbitrary groups of machines. Entity identifiers are assumed to be strictly stable, meaning a given identifier refers to the same entity forever and is thus valid forever, or t-stable, meaning that a given identifier is no longer valid, it is invalid for a period of time t.

VMTP is a multidatagram protocol layered atop IP. It bundles packets into groups of no more than 32 packets, to which rate-control, ordering and reliability are, at the user's discretion, applied. As with RRDP, each group can be acknowledged in part or in whole and can be delivered to an entity in order or unsequenced. Unlike RRDP, which acknowledges

messages, the units VMTP acknowledges are 512 byte blocks of user data.

VMTP has been designed with a number of real-time applications in mind. Most of VMTP's functionality is therefore optional: checksumming can be turned off, acknowledgements are not mandatory, transaction responses are not required, and data delivery need not be delayed while an entire message is accumulated. Messages can be prioritized. A message can be sent such that it is only delivered if an application is awaiting that message. Further, much of VMTP's design reflects the goal that a VMTP implementation be efficiently realizable using hardware. For example, packet checksums occur at the end of packets, where they may be calculated on the fly as packets are copied from buffer to buffer. As another example, VMTP headers are of fixed length and format so that header fields may be accessed using efficient hardware instructions.

[Mas90] shows that VMTP meets the apparently conflicting goals of providing for low latency when sending short messages and high throughput when sending long messages.

While a Unix implementation of VMTP existed at the time of RCalc's design, it relied upon modifications to the Unix kernel. Such modifications were termed unsuitable in the usage environment of RCalc, and so VMTP was not considered as a candidate transport for RCalc. OCARD, however, uses the notion of fixed length headers espoused by the VMTP designers.

2.2.8 The Xpress Transfer Protocol

The Xpress Transfer Protocol (XTP) was developed by a consortium of for-profit corporate members and affiliated not-for-profit institutions. It is a lightweight, connection-oriented,

byte-stream transfer protocol[13] providing a rich set of user-tunable options, reflecting its design goal of separating policy (what to do) from mechanism (what can be done) [SDW92]. Functionally a superset of VMTP (*cf* Section 2.2.7), it was designed in a similar manner as VMTP, from a clean slate with the intent that it provide efficient access to high-speed giga-bit or tera-bit networks. Unlike VMTP, it does not presume anything about the communication paradigms it supports. XTP can support a wide variety of communication styles, including connection-oriented exchanges, graceful or abrupt connection teardown, half-closed connections, two-packet request/response transactions that establish and tear down a connection, and both reliable and un-acknowledged datagrams.

Since XTP is a transfer protocol, using it requires that intermediate systems be XTP-aware. XTP end systems must use tunneling techniques[14] to communicate through intermediate systems that do not understand XTP. XTP uses cut-through routing, where the packet establishing a connection between two endpoints cuts a route through the subnetwork between the endpoints; this route is subsequently used for all packets between endpoints until it is released, either by the endpoints or by detecting a failure along the route. Another major design goal of XTP is to enable reduction of protocol processing to silicon. To accommodate this goal

- checksums, as in VMTP, are in a packet's trailer
- headers and trailers are fixed in length, and fields are offset at fixed positions in them

[13] A transfer protocol is one merging levels 3 and 4 of the Open Systems Interconnection reference model, the network and transport levels.

[14] A protocol that encapsulates its packets for delivery by a higher level protocol is said to *tunnel* through the higher level protocol.

- fields in XTP are aligned on 4 byte boundaries
- the protocol engine is entirely realizable as a finite state machine.

In a break with traditional network protocols, which regard network bandwidth as a limited resource and end-system compute cycles as inexpensive, certain fields in XTP packets exist solely to facilitate efficient implementation of protocol software. The example of XTP's *key* field will demonstrate the use of such fields. The *key* field is initialized by the initiator of a connection. All subsequent packets sent by the responder are expected to be tagged with its value. The initiator can then use the *key* field in these return packets to demultiplex the incoming stream of XTP packets to a particular connection. The XTP protocol definition does not specify the interpretation of the *key* field's value; instead, when a connection is initiated, an XTP implementation could place the index into a table of corresponding to that connection in the *key* field. The context for a particular incoming packet could then be found quickly by using the *key* field as an index into the table of connection contexts.

In addition to one-to-one, duplex communication, the XTP definition specifies one-to-many simplex communication semantics. The group communications are simplex, according to XTP designers, because the concentration of multiple data streams into one is both unclear and best left to the application desiring duplex communication. Realizing the dogmatic nature of this position regarding group communication, the XTP definition goes on to suggest a heuristic called "cloning" that combines a one-to-many simplex association with one or more one-to-one simplex associations, effecting a full-duplex one-to-many association.

Early implementations of OCARD, developed independently of XTP, attempted to use

header fields to facilitate lookups in much the same way as does XTP. However, the OCARD implementation was flawed in its handling of re-used keys, and re-using keys was subsequently abandoned. Given XTP's resolution of the problem of using a key again, lookup fields may return to later revisions of OCARD.

Chapter 3

Methods and Procedures

This Chapter presents the methods and procedures used to address the problems of Section 1.1 and to answer the Research Questions of Section 1.3. First, the software that implements the framework is discussed. Next, by deriving an analytic predictor of efficiency, Research Question One is answered. Finally, a testing methodology is developed and experiments are proposed to both test and improve the accuracy of the analytic predictor.

3.1 Framework Design Goals

The general goals to be met by this framework, that of providing Common Lisp programmers with a MIMD-LAN system based on Unix workstations, have been discussed in the Problem Statement (*cf* Section 1.1). In this section, those goals which have fundamental impact on the character of the framework are expounded upon. Further, additional goals felt to be

desirable in such a framework are presented.

3.1.1 Efficiency

The framework considers the resources it uses precious: they should be allocated only when necessary and should be freed as soon as they are no longer required. Efficient use of resources takes precedence over all other design goals.

3.1.2 Simplicity

Many of the available methods for harnessing the idle CPU cycles in a workstation environment require at least an intermediate level understanding of the network connecting the workstations. Using RPC, for example, presupposes a familiarity with networks at the protocol level (*cf* Section 2.1.1). Installing these methods often requires elevated privileges; the DPUP (*cf* Section 2.1.4) installation does. Some require advanced knowledge of accepted coding practices in the C programming language in order to use them. Others require familiarity with esoteric parallel programming paradigms; the Distributed Hypercube Simulator (*cf* Section 2.1.5) is one such. In short, either by design or by default, most of the methods reported in Chapter 2 are intended for use by systems programmers. However, there are large classes of users who could benefit from parallel computation who do not have the requisite systems programming experience to use them. Simplicity, therefore, takes precedence in the design over other concerns that might introduce the user to a detailed understanding of the underlying system.

3.1.3 Heterogeneity

So that the framework may have as many CPU types as possible available to it on any particular LAN, it should support many architectures, regardless of their CPU-type or operating system. While this goal is in some ways impractical, it influences many of design decisions made while implementing the framework.

3.1.4 Support for Medium-Grained Computations

An explicit attempt has been made to develop a framework with low enough overhead that finer grained parallel computation may be achieved. Some problems are very coarse-grained in nature and can be effectively parallelized by such crude methods as distributing input data to processors via human-loaded magnetic tapes as reported in [Pet87]; this framework is intended to support finer grained computations.

3.1.5 Extensibility

So that future work can build on the results obtained from this research, the framework's initial implementation should be both modular and extensible. The initial implementation should be general enough to incorporate future advances in operating system design.

3.2 Implementation Particulars

The framework, known as RCalc, implements a simple bag-of-work model of parallel computation as described in [CG88]. Calculations are performed by *servers*, hosts that do work, and co-ordinated by *drivers*, hosts that are responsible for maintaining the "bag." *Servers*

perform pieces of the calculation; *drivers* distribute the calculation amongst *servers* and assemble results produced by *servers* into a whole.

The following parts of RCalc are surveyed next:

- the syntax and semantics of messages passed between drivers and servers,
- a dispatcher that facilitates writing server programs,
- the Common Lisp interface to RCalc, and
- the protocol used by RCalc as a data transport.

3.2.1 RCalc Messages

RCalc messages are sent by a reliable datagram protocol OCARD, described in Section 3.2.4. It is clear why reliability is necessary for RCalc, while choosing a datagram protocol, which transfers data without the overhead of exchanging connection establishment packets, is in keeping with the design goal of using a resource (in this case the bandwidth of the network) efficiently (*cf* Section 3.1.1). While implementations of other reliable connectionless protocols exist for some versions of Unix (*cf* Chapter 2), OCARD was implemented in conjunction with RCalc to be portable across many versions of Unix.

In keeping with the design goal of heterogeneity (*cf* Section 3.1.3), RCalc messages are canonically encoded so that any host may interpret them. XDR [Sun87] was chosen as RCalc's encoding scheme. Two features of XDR marked it as a suitable choice. First, proven implementations of XDR exist on nearly all Unix platforms. Second, XDR encodes data implicitly, meaning it does not encode any information regarding the type or format of the data presented to it. In a remote procedure call environment, XDR's implicit representation

Table 3.1: RCalc message format.

Field Name	Size (in bytes)	Description
id	4	an integer identifying this message so **START** and **RESULT** or **FAILED** messages can be paired
type	4	one of **START**, **RESULT**, **RESET**, **FAILED**, or **DIE**
msg	[0, 500]	empty for **RESET** & **DIE** messages, arbitrary data for **START** or **RESULT** messages, data sent in **START** message for **FAILED** messages

is more appropriate than explicit encoding schemes as in the Abstract Syntax Notation One [ISO87] or the Network Data Representation [DLM+88]: when using an RPC, the types, sizes, and so forth of the data are already known by the procedure interfaces, and they need not be transmitted across the LAN.

Upon initialization, RCalc is directed at a file containing the names of hosts with which to communicate. The host file is read, parsed, and treated as a zero-based array. Hosts in RCalc are thereafter referred to in the **host** field of RCalc messages by their index in this array. Typically, the host file on driver hosts lists all server hosts, while the host file on each server host contains just the name of the driver host(s).

The format of RCalc messages is given in Table 3.1.

In normal operation, driver hosts send **START** messages containing calculation parameters to server hosts; after performing them, server hosts return the results of these requested calculations in **RESULT** messages. The **START**/**RESULT** transaction is asynchronous: driver hosts are not blocked while awaiting the results of server hosts' calculations. **RESULT** messages thus arrive at driver hosts in any order, and the user of RCalc must be prepared to

pair up START messages with RESULT messages based on their **id** fields. Should an error occur while computing a result, a server host can return a FAILED message that contains the START message causing the failure.

Servers can be either persistent or transient. A persistent server maintains its state between START messages; a transient server does not. The two types of servers have different CPU utilization characteristics; these are discussed in Section 3.2.2.

A driver host can send a server a RESET message, indicating it should halt all outstanding calculations and reinitialize its internal state. It can also send a DIE message to a server, indicating that the server should remove the framework from that host.

3.2.2 Calculation Dispatcher

An RCalc server can be functionally decomposed into two parts: the part which interacts with driver hosts, and the part that performs calculations. The latter part varies from application to application, while the former is similar for all applications. The Calculation Dispatcher provides the common functionality needed by any server host to interact with driver hosts.

Servers need not be implemented using the Calculation Dispatcher. To implement one that does, however, one need only write a filter[1] that performs the bag calculation. The Dispatcher starts the calculation filter, delivers the decoded data of a START message it has received to the calculator's standard input, collects the result from the calculator's

[1] In Unix parlance, a filter is a program that reads data from a well-known source, called "standard input," transforms that data, and writes the transformed data to a well-known destination, termed "standard output."

standard output, marshals the result as either a **RESULT** or **FAILED** message, and returns it to the appropriate driver host. It also processes **RESET** and **DIE** messages by killing all currently executing calculators, and, when handling **DIE** messages, by performing an orderly shutdown.

The Dispatcher provides a convenient point where administrative measures may be applied. For example, the current implementation can run the calculations at an altered priority and can enforce a limit on the number of calculations in progress, returning **FAILED** messages for **START** messages that exceed the limit. Future implementations could limit the resources available to a calculator, perhaps placing a ceiling on the CPU time any one calculator could use.

RCalc provides for two types of calculation servers: transient servers and persistent servers. Persistent servers are allowed to maintain state between calculations; transient servers cannot maintain any state between calculations. The distinction between the two becomes clear when considering how the Calculation Dispatcher handles calculator filters of each type. When a calculator filter is declared to be transient, the Dispatcher spawns a filter for each **START** message it receives; when declared to be persistent, the Dispatcher spawns a single filter once and continually delivers **START** messages to it. The two types of servers can thus achieve different CPU utilizations on different architectures when performing different kinds of calculations. On uniprocessors when calculations make heavy use of I/O, transient calculators will better utilize the server host's CPU, since the I/O of one calculation can overlap with the computation of another. Conversely, persistent calculators obtain higher utilization when performing CPU bound calculations on uniprocessors, since calculations won't incur the overhead of process creation. On multiprocessor architectures where a

process is tied to a single CPU, greater CPU utilization can be obtained with transient calculators: multiple calculators, each running on a different CPU, can proceed in parallel. On multiprocessor architectures where a single process can schedule threads on multiple CPUs, a persistent calculator that can encapsulate a single calculation in one thread will perform better. The RCalc user must decide which type of calculator is best suited to a particular application and architecture.

When properly constructed, an RCalc application, much like an MP process (*cf* Section 2.1.2), can achieve fault tolerance with very little effort on the part of the programmer. By enforcing inter-message statelessness in a calculator (either by declaring a calculator transient or by not utilizing the state available to a persistent calculator), an RCalc user need only write a driver that sends a `FAILed START` message to a different server. By using the Dispatcher's ability to limit the number of outstanding calculations executing on a single host, load balancing can likewise be achieved.

3.2.3 RCalc Common Lisp Interface

RCalc-Lisp, the Common Lisp Interface to RCalc, is written for Austin Kyoto Common Lisp (AKCL), a freely available version of Common Lisp maintained at the University of Texas, Austin. The interface consists of a loadable CL package containing two parts. The first part provides functions to send and receive Lisp objects as RCalc Messages. The second part provides functions to turn a running AKCL image into an eval server suitable for use as a filter by RCalc's Calculation Dispatcher. The work in the bag of parallel algorithms implemented using RCalc-Lisp thus consists of Lisp forms to be evaluated. The results of the evaluation are returned by the server hosts. Both the Lisp forms sent to the servers

and results returned to the driver are limited to S-expressions that can be parsed by the Lisp reader. This precludes passing such Lisp objects as streams, compiled functions, etc. Further details are provided in the Appendix, which contains the Unix manual pages for RCalc-Lisp.

3.2.4 OCARD Protocol

OCARD is the reliable datagram protocol used by RCalc. OCARD provides end-to-end communication. An underlying assumption of OCARD is that it will be used to transmit short messages in a transaction-based communication where ordered delivery of messages is not required.

OCARD uses the User Datagram Protocol [Pos81], a member of the TCP/IP protocol suite, as an underlying transport protocol. TCP/IP was chosen as the proven, ubiquitous networking protocol for Unix hosts, and UDP is its lightest-weight member available to non-privileged user-level processes on most Unix implementations.

An OCARD packet has 3 parts: *AckMask*, a bitmap used to determine whether this packet acknowledges previously received packets; *SeqNum*, a sequence number used to uniquely identify each packet for purposes of duplicate rejection and receipt acknowledgment; and *data*, from zero to 512 bytes of data to or from the OCARD user. See Table 3.2 for formats and names of these parts.

So that OCARD can migrate to operating systems supporting multiple threads of control inside a single execution context as in [Che88, ABB+86], so that it can take advantage of multiprocessor architectures, and so that it need not be driven by a heartbeat routine, OCARD is implemented using three user-level Unix processes: a client, a reader,

Table 3.2: OCARD message format.

Field Name	Size (in bytes)	Description
AckMask	4	a bitmap of acknowledged packets
SeqNum	4	an integer identifying this message
data	[0, 512]	upper level data

and a writer. Figure 3.1 shows these processes and their interactions. A user-level

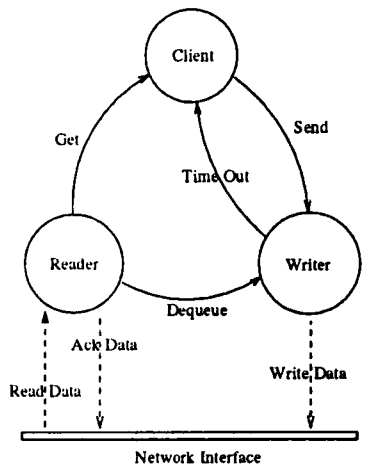

Figure 3.1: Overview of Data Flow in OCARD

implementation is not as efficient as one built into the Unix kernel; however, a kernel-level implementation is inherently not portable across different versions of Unix and would violate the design goal of heterogeneity (*cf* Section 3.1.3). A multiple process implementation introduces the overhead of using Unix interprocess communication primitives; however, it is in keeping with the design goal of allowing for future extensions (*cf* Section 3.1.5).

The *client* process uses OCARD libraries to communicate with peer processes on other

hosts that are themselves *client* processes of OCARD. The client never transmits data directly: rather, messages known as *OGrams* are passed to and from *writer* and *reader* processes, which interact with the network on the client's behalf. OGrams contain a host identifier as in Section 3.2.1 and data. If an OGram is sent to the writer, then the data are transmitted to the specified host. If an OGram is delivered by the reader, then the host identifier tells what host transmitted the data to the client. OGrams are returned to the client if the writer cannot deliver them.

The *reader* process is responsible for reading all OCARD packets from the network. Acknowledgement packets are summarized and passed to the *writer*; data packets are passed to the *client*. The *AckMask* field in an *OGram* distinguishes between the two packet types.

When nonzero, *AckMask* is a bitmap of packets that the sender of the packet in question has successfully received. If the bit numbered i (where the low-order bit is numbered 0) in *AckMask* is set, then packet $SeqNum + i$ is acknowledged. Acknowledgements are summarized and sent via Unix interprocess communication primitives to the *writer*, which removes the acknowledged packets from its retransmission queue.

When *AckMask* is zero, then the packet in question is a data packet, and must be acknowledged. If packet *SeqNum* has never been seen by the reader, it is enqueued as an OGram for delivery to the client. Acknowledgements are batched while incoming packets are processed. When there are no more packets to read, the reader composes acknowledgements and sends them, combining as many as possible into a single *AckMask*.

The *writer* process is responsible for reliably delivering data from its associated client process to a remote reader process acting on behalf of a client process on another host. After being passed an OGram by its client process, the writer will

- encapsulate the data portion of the OGram in a packet, marking it with a unique *SeqNum* and setting *AckMask* to zero;
- transmit the resultant packet to the specified destination host via UDP;
- and, finally, enqueue the packet for later retransmission.

When notified by the reader process that a particular packet has been acknowledged, it removes that packet from its retransmission queue. Any packet that is transmitted too many times is returned to the client process as an undeliverable OGram.

Like any reliable service built atop unreliable ones, OCARD achieves its reliability through acknowledgment, discussed above, and retransmission. OCARD's retransmission scheme uses a simple binary exponential backoff strategy. A maximum retransmission delay is stored with each unacknowledged packet. Immediately before retransmitting a packet, the maximum retransmission delay is doubled, and its next retransmission time is set to the current time plus a delay randomly chosen between 1 second and the maximum delay. The initial maximum delay is randomly chosen from between 1 and 5 seconds. This scheme is readily changed should the evaluation prove it to be unacceptable.

3.3 Computational Model

The framework may be used in a variety of bag-of-work configurations. However, the behavior modelled in this work is restricted to usage that could arise from non-deterministic searches such as those in Simulated Annealing or Genetic Algorithms. Such searches can be outlined as follows:

1. compute starting points for some number of similar calculations,

2. perform the calculations,

3. based on calculation results, compute starting points for new calculations, and

4. repeat steps 2 through 3 until done.

The efficiency of these searches is derived. Before elaborating on the derivation, however, the steps outlined above must be re-cast in a bag-of-work paradigm suitable for realization using RCalc.

The calculations of step 2 correspond to work in the bag, performing them corresponds to doing the work, and step 3 corresponds to refilling the bag with new work. Conceptually, the simplest RCalc implementation of this behavior utilizes a single driver host that maintains the bag and a varying number of server hosts that perform the work in the bag. The driver then alternates between the two states of emptying and refilling the bag until the final result is computed. In the emptying state, the driver distributes the work in the bag across the servers, starting calculations by sending `START` messages to the server hosts and collecting their results received in `RESULT` messages. In the refilling state, the driver refills the bag with new calculations computationally derived from results collected in the emptying state.

For performance reasons, the number of calculations in the bag is restricted to the number of server hosts used. This restriction minimizes communication overhead, while suffering from no great loss of generality: the parameters to more than one step 2 calculation can be batched together in a single `START` message, so that each each piece of work in the bag is one or more step 2 calculations.

Figure 3.2 pictorially represents a single iteration of the driver as it starts in the emptying state and finishes the next refilling state. There are B server hosts, where B is also the

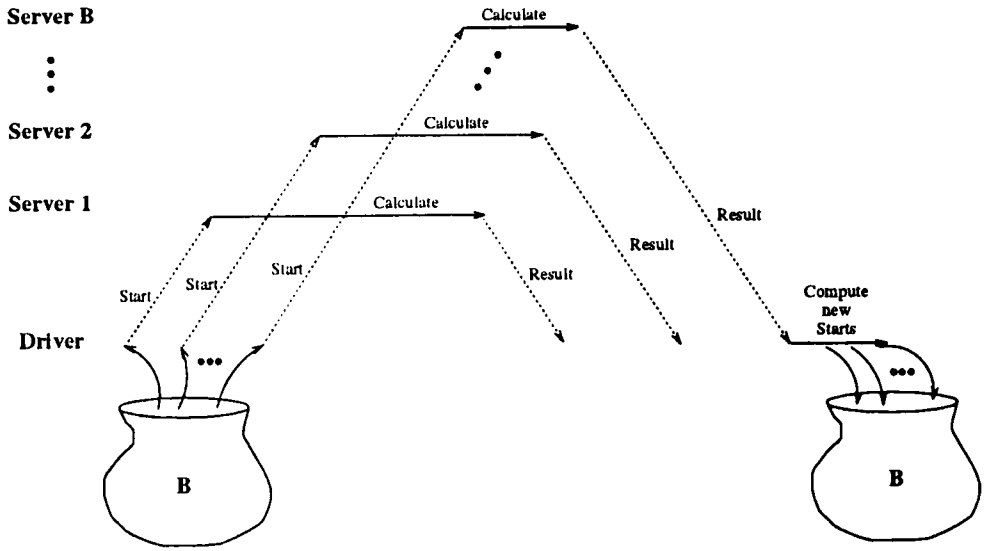

Figure 3.2: Single Iteration of Bag Emptying and Refilling

bag size. On the left of the figure, the bag is emptied by the driver as B START messages are sent; in the middle, the B server hosts perform the work by calculating results; at the right, the servers return RESULT messages; and at the far right, the driver refills the bag by computing B new START messages.

Some aspects of Research Question One concerned with the model of computation to which the predictor applies have been addressed. The model applies to a subset of bag-of-work algorithms characterized as follows:

- The algorithm iterates through the steps of first distributing all work in the bag across server processes, and then refilling the bag with new work based on the results. All

the outstanding work completes before the bag is refilled with new work.

- The algorithm's bag tasks are all similar in nature.
- The size of the bag is not changed by the algorithm.

The model is parameterized by the following quantities:

- the size of the bag;
- the complexity, or strength, of the bag calculations;
- the strength of the calculation to refill the bag; and
- the size of the `START/RESULT` messages.

3.4 Analytic Derivation of Efficiency

A large part of the derivation of E, RCalc's predicted efficiency when used as presented in the previous section, involves accounting for execution time on either the driver or server hosts. It can be argued that the execution time to perform any particular action is fixed, since the set of instructions which implement that action is invariant. While this argument will hold for dedicated, non-interruptible processors, it clearly does not apply to multi-programmed Unix hosts. It does, however, provide a basis for modelling CPU times in a multiprogramming environment: the time to perform a given task is the sum of execution times of the instructions needed to perform it, plus a noise factor introduced when the CPU services interrupts or switches to another task. In the derivation that follows, the

execution time of a particular task is represented by an upper-case Roman letter, while the multiprogramming delay associated with it is represented by a lower-case Greek letter.

E, the efficiency of a parallel algorithm, is defined to be the speedup it achieves, S, divided by the number of processors used by the algorithm. In the modelled computation, one driver host and B server hosts are used. So, $E = \frac{S}{B+1}$. The speedup, S, of a parallel algorithm is the ratio of the time it takes to perform the algorithm serially, s, divided by the time it takes to perform in parallel p. A formula for s is first derived, followed by a derivation of a formula for p.

The time to perform a bag-of-work algorithm serially is equal to the time it would take to perform all the work that ever comes out of the bag, plus all the time it would take to recompute new pieces of work to go into the bag. When the bag-of-work algorithm in question is one that oscillates between the two states of emptying and refilling the bag, then the serial time to perform it is equal to the sum over the number of oscillations of the times it takes to compute each B pieces of work in the bag plus the time it takes to refill the bag. The number of oscillations is equal to the total number of bag calculations to be performed, N, divided by the bag size, B. Assume that $N \mathrm{mod} B = 0$. If C_r denotes the execution time of the instructions needed to refill the bag, ϕ_i denotes the multiprogramming delay incurred while refilling the bag for the ith time, C_e denotes the execution time of the bag calculations and $\rho_{i,j}$ denotes the multiprogramming delay incurred by the jth server while it performs its calculation in the ith bag, then

$$s = \sum_{i=1}^{\frac{N}{B}} (C_r + \phi_i + \sum_{j=1}^{B} (C_e + \rho_{i,j}))$$

$$
\begin{aligned}
&= \frac{N}{B}C_r + \sum_{i=1}^{\frac{N}{B}}(\phi_i + BC_e + \sum_{j=1}^{B}\rho_{i,j}) \\
&= \frac{N}{B}C_r + NC_e + \sum_{i=1}^{\frac{N}{B}}(\phi_i + \sum_{j=1}^{B}\rho_{i,j}) \\
&= \frac{N}{B}C_r + NC_e + \frac{N}{B}\overline{\phi} + \sum_{i=1}^{\frac{N}{B}}\sum_{j=1}^{B}\rho_{i,j} \\
&= \frac{N}{B}(C_r + \overline{\phi}) + NC_e + N\overline{\rho} \\
&= N(\frac{(C_r + \overline{\phi})}{B} + C_e + \overline{\rho}) \qquad (3.1)
\end{aligned}
$$

Note that the above derivation makes use of the restriction on the modelled bag-of-work algorithms that bag calculations are all similar by assuming that the execution times of the calculations performed on the driver and the servers are the same for each iteration of emptying and refilling the bag. The remainder of this derivation makes this assumption, as well.

As in the derivation of s, the time it takes to perform the bag-of-work algorithm in parallel, p, is equal to the sum over all oscillations of the times it takes to perform each oscillation. If T_{B_i} is equal to the time it takes to empty and refill the bag for the ith time, then

$$
\begin{aligned}
p &= \sum_{i=1}^{\frac{N}{B}} T_{B_i} \\
&= \frac{N}{B}\overline{T_B}
\end{aligned}
$$

If communication delays are ignored, a lower bound on T_{B_i} and p, and thus an upper bound on E, may be obtained. These bounds reflect the use of of a hypothetical instantaneous communication mechanism. The time for the driver to prepare and send each **START**

message will be zero, as will the time for messages to reach servers, as will all other times involving parsing or transmitting messages. The time to empty the ith bag will therefore be the longest time each server takes to calculate a result, plus the time it takes to refill the bag. So,

$$\begin{aligned} T_{B_i} &= C_e + \max\{\rho_{i,1}, \cdots, \rho_{i,B}\} + C_r + \phi_i \\ \overline{T_B} &= C_e + \overline{\max\{\rho_1, \cdots, \rho_B\}} + C_r + \overline{\phi} \end{aligned}$$

If the statistical distributions of multiprogramming delays on each server are assumed to be similar to and independent of each other, as is reasonable to suppose of like machines running the same operating system and process mix, and if they are assumed to be of low variance, then $\overline{\max\{\rho_1, \cdots, \rho_B\}} \approx \overline{\rho}$ and

$$\begin{aligned} \overline{T_B} &\approx C_e + \overline{\rho} + C_r + \overline{\phi} \\ p &\approx \frac{N}{B}(C_e + \overline{\rho} + C_r + \overline{\rho}) \end{aligned}$$

Computing the speedup $S = \frac{s}{p}$ yields

$$\begin{aligned} S &\approx \frac{N(\frac{(C_r+\overline{\phi})}{B} + C_e + \overline{\rho})}{\frac{N}{B}(C_e + \overline{\rho} + C_r + \overline{\rho})} \\ &\approx \frac{B(C_e + \overline{\rho}) + C_r + \overline{\phi}}{C_e + \overline{\rho} + C_r + \overline{\phi}} \end{aligned}$$

As an aside, this equation is precisely that for "scaled speedup" given in [Gus88], where, in apparent contraction of Amdahl's law[2], S is shown to increase without bound as more

[2]Stated using the variables introduced in this paper, Amdahl's law says $\lim_{B\to\infty} S = \frac{1}{C_r}$

and more processors are added. The contradiction is resolved by noting that in [Amd67] the problem size $C_r + C_e$ remains fixed, while both here and in [Gus88] the problem size $C_r + BC_e$ increases linearly in B.

To continue, dividing S through by $B + 1$ to obtain a upper bound on the efficiency E gives

$$E_{\text{upper}} \approx \frac{B(C_e + \overline{\rho}) + C_r + \overline{\phi}}{(B+1)(C_e + \overline{\rho} + C_r + \overline{\phi})} \tag{3.2}$$

Note that, as more and more servers are added to a particular computation, that is, as $B \to \infty$, $E \to \frac{C_e+\overline{\rho}}{C_e+\overline{\rho}+C_r+\overline{\phi}}$. Too, note that for a given number of server hosts B, E reaches its maximum of $\frac{B}{B+1}$ when $C_e \to \infty$ and its minimum of $\frac{1}{B+1}$ when $C_r \to \infty$. Ignoring message-passing delays, then, one may unsurprisingly conclude that the efficiency of a bag-of-work algorithm using RCalc may be improved by giving each server as much work to do as possible, while having the driver do as little work as needed.

To predict an average value for E, T_{B_i} must account both for time to perform bag calculations and for delays introduced by RCalc. To do so, the **START/RESULT** transaction between the driver and the jth server during the ith bag oscillation is examined. The events that occur during this transaction are shown in Figure 3.3, and the time when they complete, relative to the start of the bag oscillation, is denoted $t_{i,j}$.

Before sending the jth server its **START** message, the driver must first compose and send the previous $j - 1$ servers their **START** messages. Let T_s be the execution time to compose and send a single **START** message, and let $\eta_{i,k}$ be the multiprogramming delay associated with composing and sending the kth **START** message before sending the jth. Let the time to send the jth be be $T_s + \eta_{i,j}$. Let the time it takes for the jth **START** message to travel from the driver to server j be $\alpha_{i,j}$. Next, server j must receive and parse the **START** message.

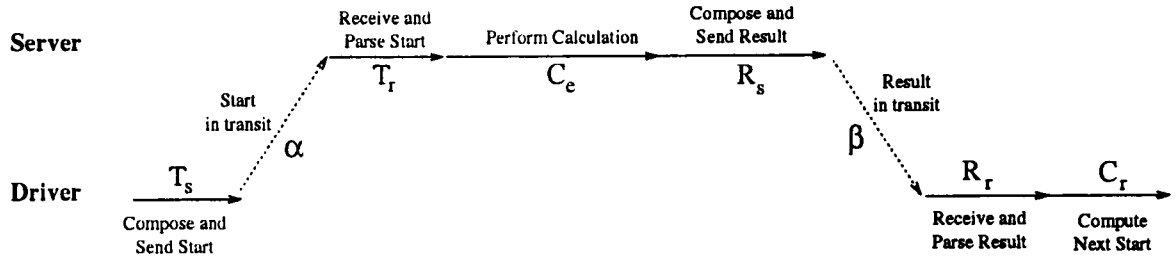

Figure 3.3: Events of a Single START/RESULT Transaction

Let this time be $T_r + \theta_{i,j}$. Server j then performs the calculation; as in the serial case, this time is $C_e + \rho_{i,j}$. The server then packages the result and sends it. Let this time be $R_{i,j} + \mu_{i,j}$. The RESULT message takes time $\beta_{i,j}$ to reach the driver, and the driver takes time $R_{r_{i,j}} + \nu_{i,j}$ to receive and parse it once it arrives. The total time for all these events is thus

$$\begin{aligned} t_{i,j} &= \sum_{k=1}^{j-1}(T_{s_{i,k}} + \eta_{i,k}) + T_{s_{i,j}} + \eta_{i,j} + \alpha_{i,j} + T_{r_{i,j}} + \theta_{i,j} + C_e + \rho_{i,j} \\ &\quad + R_{s_{i,j}} + \mu_{i,j} + \beta_{i,j} + R_{r_{i,j}} + \nu_{i,j} \\ &= \sum_{k=1}^{j}(T_{s_{i,k}} + \eta_{i,k}) + \alpha_{i,j} + T_{r_{i,j}} + \theta_{i,j} + C_e + \rho_{i,j} + R_{s_{i,j}} + \mu_{i,j} + \beta_{i,j} + R_{r_{i,j}} + \nu_{i,j} \end{aligned}$$

Letting $\tau_{i,j}$ stand for that part of $t_{i,j}$ due entirely to the overhead of RCalc yields

$$t_{i,j} = \tau_{i,j} + C_e + \rho_{i,j},$$

where

$$\tau_{i,j} = \sum_{k=1}^{j}(T_{s_{i,k}} + \eta_{i,k}) + \alpha_{i,j} + T_{r_{i,j}} + \theta_{i,j} + R_{s_{i,j}} + \mu_{i,j} + \beta_{i,j} + R_{r_{i,j}} + \nu_{i,j}. \qquad (3.3)$$

To continue, note that while parts of the B **START/RESULT** transactions overlap (see Figure 3.2), they must all complete before the driver can compute new **START** messages. Therefore

$$\begin{aligned} T_{B_i} &= \max\{t_{i,1}, \cdots, t_{i,B}\} + C_r + \phi_i \\ &= \max\{\tau_{i,1} + C_e + \rho_{i,1}, \cdots, \tau_{i,B} + C_e + \rho_{i,B}\} + C_r + \phi_i \\ &= C_r + \phi_i + C_e + \rho_{i,m} + \tau_{i,m} \\ \overline{T_B} &= C_r + \overline{\phi} + C_e + \overline{\rho_m} + \overline{\tau_m} \end{aligned}$$

Note that $\rho_{i,m}$ and $\tau_{i,m}$ represent the values of $\{\tau_{i,1}, \cdots, \tau_{i,B}\}$ and $\{\rho_{i,1}, \cdots, \rho_{i,B}\}$, respectively, where the sum $\tau_{i,j} + \rho_{i,j}$ is maximized and not, as one might erroneously suppose, $\max\{\tau_{i,1}, \cdots, \tau_{i,B}\}$ and $\max\{\rho_{i,1}, \cdots, \rho_{i,B}\}$.

The speedup S is now computable.

$$\begin{aligned} S &= \frac{s}{p} \\ &= \frac{N(\frac{(C_r+\overline{\phi})}{B} + C_e + \overline{\rho})}{\frac{N}{B}(C_r + \overline{\phi} + C_e + \overline{\rho_m} + \overline{\tau_m})} \\ &= \frac{B(\frac{(C_r+\overline{\phi})}{B} + C_e + \overline{\rho})}{C_r + \overline{\phi} + C_e + \overline{\rho_m} + \overline{\tau_m}} \\ &= \frac{C_r + \overline{\phi} + B(C_e + \overline{\rho})}{C_r + \overline{\phi} + C_e + \overline{\rho_m} + \overline{\tau_m}} \end{aligned}$$

Dividing S by $B + 1$ to obtain the efficiency yields

$$E = \frac{C_r + \overline{\phi} + B(C_e + \overline{\rho})}{(B+1)(C_r + \overline{\phi} + C_e + \overline{\rho_m} + \overline{\tau_m})} \tag{3.4}$$

Note that Equation 3.4 does not yet include terms containing the lengths of either **START** or **RESULT** messages; to include such terms, $\overline{\tau_m}$ must be expanded by making a number of arguably reasonable assumptions about the characteristics of RCalc or by further restricting the computational model. For the first of these, assume that RCalc, the application using RCalc, and the underlying network used by RCalc are all well-behaved in that the time it takes each to process a message is some per-message overhead plus time proportional to that message's length. If l_T is the length of **START** messages, then, using T_s as an example, $T_{s_{i,m}} \approx c_{T_s} l_T + k_{T_s}$. Substituting similar approximations for α, T_r, R_s, β, and R_r into Equation 3.3 for $\tau_{i,j}$ when $j = m$ yields

$$\begin{aligned}
\tau_m &\approx \sum_{k=1}^{m}(c_{T_s} l_T + k_{T_s} + \eta_{i,k}) + c_\alpha l_T + k_\alpha + c_{T_r} l_T + k_{T_r} + \theta_{i,m} \\
&\quad + c_{R_s} l_R + k_{R_s} + \mu_{i,m} + c_\beta l_R + k_\beta + c_{R_r} l_R + k_{R_r} + \nu_{i,m} \\
&\approx m(c_{T_s} l_T + k_{T_s}) + c_\alpha l_T + c_{T_r} l_T + c_{R_s} l_R + c_\beta l_R + c_{R_r} l_r \\
&\quad + k_\alpha + k_{T_r} + k_{R_s} + k_\beta + k_{R_r} + \theta_{i,m} + \mu_{i,m} + \nu_{i,m} + \sum_{k=1}^{m} + \eta_{i,k} \\
&\approx (mc_{T_s} + c_\alpha + c_{T_r}) l_T + (c_{R_s} + c_\beta + c_{R_r}) l_R \\
&\quad + mk_{T_s} + k_\alpha + k_{T_r} + k_{R_s} + k_\beta + k_{R_r} \\
&\quad + \theta_{i,m} + \mu_{i,m} + \nu_{i,m} + \sum_{k=1}^{m} \eta_{i,k}
\end{aligned}$$

The second simplifying assumption made is that the process of sending and receiving a **START** message incurs nearly the same delays as sending and receiving a **RESULT** message.

That is, $c_{T_s} \approx c_{R_s}$, $k_{T_s} \approx k_{R_s}$, $c_\alpha \approx c_\beta$, $k_\alpha \approx k_\beta$, $c_{T_r} \approx c_{R_r}$, and $k_{T_r} \approx k_{R_r}$. This gives[3]

$$\begin{aligned}
\tau_{i,m} &\approx (mc_s + c_{\alpha\beta} + c_r)l_T + (c_s + c_{\alpha\beta} + c_r)l_R \\
&\quad + mk_s + k_{\alpha\beta} + k_r + k_s + k_{\alpha\beta} + k_r \\
&\quad + \theta_{i,m} + \mu_{i,m} + \nu_{i,m} + \sum_{k=1}^{m} \eta_{i,k} \\
&\approx (m-1)c_s l_T + (c_s + c_{\alpha\beta} + c_r)(l_T + l_R) \\
&\quad + (m+1)k_{T_s} + 2k_{\alpha\beta} + 2k_r \\
&\quad + \theta_{i,m} + \mu_{i,m} + \nu_{i,m} + \sum_{k=1}^{m} \eta_{i,k}
\end{aligned}$$

If all constants are collapsed into single values, then

$$\tau_{i,m} = (m-1)c_s l_T + c_l(l_T + l_R) + (m+1)k_m + k_0 + \theta_{i,m} + \mu_{i,m} + \nu_{i,m} + \sum_{k=1}^{m} \eta_{i,k}$$

The third simplification made concerns the multiprogramming delays η, θ, μ and ν. While in the general case, no assumptions can be made about these delays, if the modelled bag-of-work calculation is the only application running on the driver and server hosts[4], one can attribute multiprogramming delays almost entirely to the requests made by the bag-of-work application on RCalc to send and receive messages. The delays should then be either negligible, or proportional to the number or length of messages sent and received. They can therefore be eliminated or absorbed into the various c's and k's. This yields

$$\tau_{i,m} \approx (m-1)c_s l_T + c_l(l_T + l_R) + (m+1)k_m + k_0$$

Recall that $\tau_{i,m}$ is the value of $\{\tau_{i,1}, \cdots, \tau_{i,B}\}$ where the sum $\tau_{i,j} + \rho_{i,j}$ is maximized, and that $\rho_{i,j}$ is the multiprogramming delay incurred when performing the bag calculation

[3]The second subscripts, T and R, have been eliminated.

[4]This is certainly the case those desiring the results of the bag-of-work calculation would prefer.

on the jth server during the ith bag iteration. To proceed any further, some assumption about where this maximum occurs must be made. Assume, then, that m is usually B. This gives

$$\overline{\tau_m} = (B-1)c_s l_T + c_l(l_T + l_R) + (B+1)k_B + k_0 \tag{3.5}$$

After so many assumptions and approximations, a careful examination of the implications of Equation 3.5 is in order. First, terms involving each model parameter are grouped together: $\overline{\tau_m} = c_s B l_T + (c_l - c_s)l_T + c_l l_R + k_B B + k_0 + k_B$. Rewritten in this manner, Equation 3.5 states that the average delay incurred in using RCalc to request that each of B servers perform and return the results of a piece of work is equal to a term, $c_s B l_T$, proportional to the number of bytes sent as requests ($B l_T$), plus another term, $(c_l - c_s)l_T$, proportional to the length of a single request (l_T), plus another term, $c_l l_R$, proportional to the length of a single answer (l_R), plus a term, $k_B B$, proportional to the number of servers (B), plus a term, $k_0 + k_B$, representing the fixed overhead of using RCalc. Note that there is no term proportional to the number of bytes returned, that is, no term involving $B l_R$ appears. This is intuitively satisfying, since it suggests that receiving answers overlaps useful work being done on the servers. As will be seen in Chapter 4, however, some of the assumptions leading to Equation 3.5 do not always apply.

Substituting Equation 3.5 and $m = B$ into Equation 3.4 yields

$$\begin{aligned} E &= \frac{C_r + \overline{\phi} + B(C_e + \overline{\rho})}{(B+1)(C_r + \overline{\phi} + C_e + \overline{\rho_B} + (B-1)c_s l_T + c_l(l_T + l_R) + (B+1)k_B + k_0)} \\ &= \frac{C_r + \overline{\phi} + B(C_e + \overline{\rho})}{(B+1)(C_r + \overline{\phi} + C_e + \overline{\rho} + c_l(l_T + l_R) + k_0) + (B^2-1)c_s l_T + (B+1)^2 k_B} \end{aligned} \tag{3.6}$$

Before proceeding to answer Research Question One any further, a qualitative analysis of Equation 3.6 is enlightening.

First, note that RCalc places bounds on l_T and l_R. Therefore, there is no reason to examine in any great detail how E varies with them.

Second, note that as more and more hosts are added to a particular computation, that is, as $B \to \infty$, the terms in the denominator involving B^2 will dominate and $E \to 0$. This suggests that improvements to RCalc concentrate on reducing k_B, RCalc's per-server/message overhead and on reducing c_l, RCalc's per-byte overhead. This is not a startling conclusion, but its agreement with intuition is comforting.

Third, note that for a given number of server hosts, B, the maximum value E can obtain is $\frac{B}{B+1}$ when $C_e \to \infty$. Its minimum is reached $\frac{1}{B+1}$ when $C_r \to \infty$. These limits are the same as those of Equation 3.2, and one may again conclude that efficiency is improved by giving the servers as much work to do as possible and the driver as little work as needed.

3.5 Testing Methodology

To apply Equation 3.6 to a given bag-of-work algorithm in a given environment, values in that equation must be either be chosen, such as B, or be determined empirically, such as C_e. Some values, C_e and C_r, *e.g.*, are properties of the algorithm, while others, *e.g.*, c_l and c_s, are properties of RCalc-Lisp. The steps of the testing methodology are divided into those that are applied to the algorithm in its implementation environment, and those that are applied to RCalc-Lisp in the environment in question.

3.5.1 Steps Applied To the Algorithm

The steps applied to the bag-of-work algorithm seek to determine values for the model parameters that appear in Equation 3.6. One must know how C_e, $\overline{p}$, C_r, $\overline{\phi}$, l_R, and l_T all vary with the algorithm's input and the number of servers participating in the computation.

Step 1: Determine How C_e and C_r vary with Input

By taking repeated timings of executing the algorithm's bag calculation on a server host, one can approximate $C_e + \overline{\phi}$ as the average of these timings. This is only an approximate value since multiprogramming delay will vary based on the number and nature of the processes running on a host. By varying the algorithm's input while such timings are made, one can approximate $C_e + \overline{\phi}$ as a function of input. $C_r + \overline{p}$ can be similarly approximated.

Step 2: Determine How l_T and l_R vary with Input

Typically, the manner in which l_T and l_R vary with input can be determined by inspecting the source code implementing the algorithm. For some algorithms, however, it may be necessary to instrument the code so that the lengths of the **START** and **RESULT** messages are recorded.

3.5.2 Steps Applied to RCalc

The steps of the testing methodology that are applied to RCalc seek to empirically determine the various c's and k's of Equation 3.6. Given a bag-of-work algorithm that fits the model and has controllable model parameters, one can vary its input and observe its execution time. By plotting the execution time versus the various parameters, one can discover the

relationships between the model parameters and RCalc's performance characteristics. The derivation of Section 3.4 makes assumptions about these relationships, generally that they are linear and are independent of the algorithm using RCalc-Lisp. These assumptions can be checked while answering Research Question Two-B.

The Trivial, Tunable Algorithm

The bag-of-work algorithm used to explore the performance of RCalc-Lisp in the experimental environment is named the Trivial, Tunable Algorithm (henceforth, the TTA).

The bag calculation of the TTA takes a single value, g, as its starting parameter. If g is an integer, it initializes a counter to g, and proceeds to decrement the counter until it is negative, returning the Lisp atom NIL as its value. This calculation utilizes a CPU for time proportional to g, so that $C_e \propto g$. By varying g, C_e can be varied in a known manner. The precise nature of this variance will be determined empirically.

If, on the other hand, g is a string, the bag calculation of the TTA returns g as its value. By varying the length of g, the length of the TTA's **START** and **RESULT** messages can be varied directly.

The TTA begins with a bag of a given size full of task descriptors that invoke this bag calculation with either a given integer, or a given string. The TTA iterates through the steps of emptying and refilling the bag. It empties the bag by passing task descriptors to workers awaiting their results. Once all work completes, it refills the bag with the same work it had initially. As an optimization, the results of the worker processes are discarded and the data structure representing the bag is never re-initialized. The time to calculate new work is therefore zero, *i.e.*, $C_r = 0$.

Note that not all model parameters of the TTA can be varied by varying its starting conditions and that, of those which can be varied, not all can be varied independently of one another. Timings of other bag-of-work algorithms that did not suffer from these shortcomings showed that the projected experiments to fully determine the manner in which RCalc's characteristics varied with model parameters would require almost three weeks of exclusive access to the experimental environment. Since this access was not feasible, the TTA evolved to explore only certain relationships between parameters and performance. The limitations in the TTA are discussed next.

C_r for the TTA is always 0. Since in the normal case when using RCalc, there are no OCARD packets being sent when the driver is refilling the bag, one presumes that RCalc's performance does not vary with C_r, *i.e.*, the c's and k's of Equation 3.6 will not vary with C_r.

For the TTA, l_R and l_T are always the same when g is a string. Many bag-of-work algorithms used in AI have the property that their bag calculations return as a result new task descriptors, and the process of refilling the bag is one of choosing one or more of the results to replace previous bag elements. Since task descriptors are both sent to and returned by the worker processes, the lengths of the messages exchanged are often the same, *i.e.*, $l_T = l_R$. SA implemented as a bag-of-work algorithm, for example, has this property.

C_e and $l_{R,T}$ cannot be varied independently of one another. When C_e is varied by passing different integers as the value of g for the TTA's bag calculation, message lengths are near zero. When $l_{R,T}$ are varied by passing strings of different length to the TTA's bag calculation, C_e is zero. One might therefore expect the results obtained using the TTA as a test of RCalc to be most accurate where little interference exists between communication (as

reflected in varying message lengths) and computation (as reflected in varying calculation strengths).

Discovering Constants

The various constants in Equation 3.6 must be determined. To do so, some observations on $\overline{T_B}$, the duration of one bag iteration, are required.

For the TTA, $C_r + \overline{\phi}$ is always zero. When the TTA uses strings as its bag elements, $C_e + \overline{\rho} = 0$. Substituting both these and Equation 3.5 into the equation for $\overline{T_B}$, yields:

$$\begin{aligned} \overline{T_B} &= C_r + \overline{\phi} + C_e + \overline{\rho} + \overline{\tau} \\ &= \overline{\tau} \\ &= (B-1)c_s l + c_l l + (B+1)k_B + k_0 \end{aligned} \tag{3.7}$$

On the other hand, when the TTA uses integers as bag elements, l, the length of the messages passed by the TTA, is fixed at a value near 0. Then,

$$\begin{aligned} \overline{T_B} &= C_r + \overline{\phi} + C_e + \overline{\rho} + \overline{\tau} \\ &= C_e + \overline{\rho} + \overline{\tau} \\ &= C_e + \overline{\rho} + (B-1)c_s l + c_l l + (B+1)k_B + k_0 \\ &= C_e + \overline{\rho} + (B+1)k_B + k_0 \end{aligned} \tag{3.8}$$

By repeatedly measuring the duration of a single bag iteration of the TTA, T_B, while first varying B and integer values of g, and then varying B and string values of g, the

various c's and k's can be determined by fitting Equations 3.7 and 3.8 to the observed data. The remaining steps in the testing methodology perform this process.

Step 3: Determine How $\overline{T_B}$ varies with B and l

The TTA is implemented in RCalc. Repeated timings are made of a single bag iteration for different bag sizes and string lengths for g. $\overline{T_B}$ is taken to the average of these times.

Step 4: Fit Curves and Determine Constants

The values of $\overline{T_B}$ obtained in Step 1 are grouped into different curves by the value of B that produced them. These curves are plotted versus l, and curve-fitting techniques are applied to determine $\overline{T_B}$ as a function of l for each curve. Next, the Step 1 values are grouped by l and plotted versus B. Curve-fitting techniques are again applied, this time to determine $\overline{T_B}$ as a function of B. These curves are compared with Equation 3.7 to obtain a system of equations in c_s, c_l, k_B and k_0. The system is then solved, to the extent that they can be solved, to obtain values for c_s, c_l, k_B and k_0.

Step 5: Determine How $\overline{T_B}$ Varies with B and C_e

Using the RCalc-Lisp implementation of the TTA, repeated timings are made of a single bag iteration for different bag sizes and integer values of g. $\overline{T_B}$ is taken to the average of these times.

These times are plotted as in Step 3. Equation 3.8 suggests that the slopes of the curves so obtained are one. This step seeks to confirm the validity of Equation 3.8.

3.6 Improving The Predictor

The testing methodology will be applied to the TTA. An attempt will be made to account for any results which contradict some of the assumptions leading to Equations 3.6, 3.7, or 3.8. Further, modifications will be made either to the model of computation leading to these equations, to their derivation, or to the testing methodology so that the accuracy of the predictor may be improved.

3.7 Model Validation

To test the accuracy of the combination of the analytic predictor of efficiency and the testing methodology when it is applied to non-trivial bag-of-work algorithms, Steps 1 through 3 will be applied to a Travelling Salesman Problem (TSP) solver based on Simulated Annealing (SA). The values obtained will be used to predict E. Timings will be made of this algorithm, and the predicted efficiency will be compared to the observed efficiency.

The TSP and SA are introduced, then how SA applies to the TSP is presented, and, lastly, the RCalc-Lisp implementation of SA as applied to the TSP is given.

3.7.1 The Travelling Salesman Problem

The Travelling Salesman Problem consists of finding the shortest route for a salesman that must visit all cities in a given set. Because of its complexity, and because it has been widely studied, the TSP is often used to compare different programming environments (*cf* [Bal91]). Many different algorithms to solve the TSP exist; however, those that attempt to solve TSPs with many cities are usually non-deterministic.

3.7.2 Simulated Annealing

Simulated Annealing was first presented in [MRRT53]. SA simulates the metallurgical process of annealing, whereby a metal is made harder by first melting it and then slowly cooling it. As the metal is cooled, its internal energy is minimized. [KGV83] shows that one can use such a simulation to solve optimization problems in which a cost function, c, of a perturbable state, S, is minimized as follows:

1. let the cost function $c(S)$ be used to represent the energy of state S;
2. let $p(S)$ be a perturbation function that produces new states from S;
3. let T be the initial temperature of the system;
4. let $b(t)$ be a monotonically decreasing function returning a value less than t;
5. let $r(x)$ return a random real number between 0 and x;
6. set s equal to a randomly chosen member of $p(S)$;
7. set δ equal to $c(s) - c(S)$;
8. if $c(s) < c(S)$, set S equal to s;
9. if $\exp(-\frac{\delta}{T}) > r(1)$, set S equal to s;
10. repeat steps 6 through 9 until S reaches equilibrium;
11. set T equal to $b(T)$;
12. repeat steps 6 through 11 until S becomes frozen

The initial temperature, the definitions of equilibrium and frozen, and the characteristics of the cooling function $b(T)$ all influence the quality of the solution found by SA.

SA can be applied to the TSP by using the length of a tour to represent its energy, and by perturbing a given tour by choosing a random leg of that tour and reversing it. The resulting algorithm is called SA-TSP.

3.7.3 Parallel SA and the TSP

By defining equilibrium to mean that each city in the TSP has been moved at least m times, the SA-TSP algorithm can be parallelized using RCalc-Lisp if each piece of work in the bag requires that a server host anneal a given tour until each city has moved $\frac{m}{B}$ times. New work for the bag is produced by choosing one result from the servers according to the Boltzmann distribution given in step 9 to be the next state. This parallel algorithm, henceforth known as PSA-TSP, has been shown to produce solutions of the same quality as that of the serial SA [DLS93].

PSA-TSP will be implemented in RCalc-Lisp. It will be run on a randomly generated 30 city TSP with m set to 160. The testing methodology described later will be applied to it, the analytic predictor will be used to forecast its efficiency as the number of servers varies, and its empirically observed efficiency will be compared with the forecasted efficiency.

3.8 Experiments

The goal of the experiments conducted in this study is to derive enough information about RCalc-Lisp, the TTA, and the PSA-TSP so that the research questions of Section 1.3 may

be answered. The first three experiments apply the testing methodology of Section 3.5 to the TTA. The results of these experiments will be used to both improve the analytic predictor and to discover the performance characteristics of RCalc-Lisp needed to apply it to a bag-of-work algorithm. The next two experiments apply the algorithm-specific parts of the test method to the PSA-TSP. The results of these experiments, coupled with the earlier results, will be used to predict the efficiency of the PSA-TSP as implemented in RCalc-Lisp. The final experiment empirically measures the efficiency of the PSA-TSP so that the accuracy of the predictions of its efficiency may be judged.

3.8.1 Experimental Environment

The experimental environment used in this study consists of two or more Sun SparcStation IPXs with 16 megabytes of memory running a version of SunOS-4.1.3 modified so that idle processes are not gratuitously swapped to backing store.[5] These machines are interconnected by thin-wire Ethernet. They are isolated from other hosts on the network by a bridge. All the software built as part of the experiment will reside on disks directly connected to each host. All experiments are performed when the network is relatively idle.

Austin Kyoto Common Lisp, version 1.615, is the Common Lisp engine used to implement RCalc-Lisp. The ANSI C compiler used to compile RCalc source code and to compile AKCL's intermediate C code is the Gnu C Compiler, version 2.3.3.

[5] Far from an unusual modification, disabling idle swapping so that processes remain in memory whenever possible is a recommended performance enhancement on memory rich machines.

3.8.2 Experiment 1: Determine how C_e varies with g for TTA

Experiment 1 corresponds to applying Step 1 of the testing methodology to the TTA. This experiment therefore seeks to determine the relationship between integer values of g, the input parameter of the TTA's bag calculation, and C_e, the strength of the TTA's bag calculation.

Multiple timings of the TTA's bag calculation using this implementation will be made to determine $C_e + \overline{p}$ as a function of g. The minimum and maximum values will be recorded, and the standard deviation will be computed. It is expected that C_e will vary linearly with g.

The results of this experiment will be used in two ways. First, they will be used as given in Step 1 of the testing methodology. Second, since s, the serial time to complete the TTA, is the sum of the execution times of each bag calculation times the size of the bag for each bag iteration, these results will be used to compute s for the TTA using the equation $s = BC_e$.

3.8.3 Experiment 2: Determine How $\overline{T_B}$ varies with B and l for TTA

Experiment 2 corresponds to performing Step 3 in the testing methodology of Section 3.5.

Using strings to be presented to the TTA as work in the bag, measurements of the time it takes to empty and refill the work bag will be repeatedly taken for different numbers of servers. Strings of length 2, 5, 10, 20, 50, 100, 200, 300, 400, and 499 bytes will be used. As for the previous experiment, the extrema are saved, and the mean and standard deviation are calculated. The mean values will be taken as $\overline{T_B}$ and will be used in the curve-fitting performed in Step 4 of the testing methodology.

3.8.4 Experiment 3: Determine How $\overline{T_B}$ Varies with B and C_e for TTA

Experiment 3 measures times to be used in calculating E for the TTA. Its results will be used to judge the accuracy the analytic predictor as improved by applying the testing methodology

Timings will be made of the TTA as implemented in RCalc-Lisp while B and C_e are varied. Using the results of the first experiment, the calculation strength of the TTA is varied from a minimum value up to one second. This minimum is picked using the results of the previous experiment. Equation 3.1 (and common sense) shows that E will be far from unity if $C_e \ll \tau$. Recall from Section 3.5.2 that $T_B = \tau$ in the previous experiment. The minimum will therefore be chosen as the value of T_B in Experiment 2 for short messages. The minimum and maximum times will be recorded, the mean and standard deviation will be calculated, and the mean values will be taken as $\overline{T_B}$ to be used in the strength-based curve-fitting of Step 5 of the testing methodology. Since the overhead of using RCalc should be fixed for a given number of servers and a given message length and will therefore not change as C_e changes, it is expected that the constant of proportionality between C_e and T_B will be near one.

3.8.5 Experiment 4: Determine How C_e varies with Input for PSA-TSP

Experiment 4 corresponds to applying part of Step 1 of the testing methodology to the PSA-TSP. It seeks to discover how C_e varies with the PSA-TSP's input, the number of times to perturb each city before equilibrium is reached. A method similar to that used in the first experiment will be used in Experiment 4.

As in Experiment 1, these results will be used for two purposes. The first use is in

applying Step 1 of the test method. The second use will be to determine s for the PSA-TSP. The bag calculation of the PSA-TSP for $B = 1$ is precisely SA-TSP applied with $m = 160$.

3.8.6 Experiment 5: Determine How C_r varies with Input for PSA-TSP

Unlike the TTA, the PSA-TSP requires that some computation be performed on the RESULTs returned by the servers before the work bag can be refilled. That half of Step 1 of the testing methodology which seeks to determine how C_r varies with input must be performed, and Experiment 5 does so. A method similar to that used in the previous experiment will be used to determine C_r

3.8.7 Experiment 6: Determine Runtime of PSA-TSP

To calculate E for different numbers of server hosts, the time required to run the PSA-TSP must be known. The PSA-TSP algorithm will be run using varying number of server hosts, and timings will be made. The mean, standard deviation, and extrema values will be recorded.

The results of this experiment will be used to determine the accuracy of the analytic model after any modifications made when interpreting the results of the previous experiments.

Chapter 4

Results and Findings

The results reported in this study are of two types: analytical and experimental.

Some of the experimental results are used to confirm or deny assumptions made while initially deriving the analytical results. These experiments correspond to applying the testing methodology devised in Section 3.5, where their goal is determining performance characteristics of RCalc-Lisp. Their results are then used to improve the model and the assumptions from which the analytical results are derived, and improved analytical results are produced. The remainder of the experiments seek to determine the accuracy of the improved analytical results. They correspond to first applying the testing methodology to an example bag-of-work algorithm, and then checking the analytic predictor against empirical results.

The results of all experiments outlined in the previous chapter are based on timings of various activities. Before proceeding with any experiments, a preliminary investigation of how a user application running in the experimental environment can obtain such timings

was conducted. The results of this investigation are reported next.

4.1 Timing Issues

The Unix *gettimeofday()* system call is used by a user application to consistently sample a host's clock. On Sun SparcStation 1+'s, repeated back-to-back calls to *gettimeofday()* show that it takes nearly 50 microseconds to sample the clock. The initial direction of this study required that timings be made of events initiated by RCalc-Lisp and the algorithm using it. These events, numbering in the tens of thousands, had durations of around 30 microseconds. Clock resolution to 50 microseconds was felt to be too coarse and too intrusive. Another method of sampling the clock, termed the *map-clock*, was devised which does not incur the overhead of a system call. Instead, the system call *mmap()* is used to create a page table entry that maps an address in a process's virtual address space to the physical address of a SparcStation's countdown register, which has a resolution of one microsecond. Another mapping is made to the Unix kernel's time value. Once these mappings are in place, a process can add the two values to determine the current time of day. The *map-clock* works by first sampling the system's time value, and then sampling the countdown register. Sampling continues until the sampled value of the system time is equal to the mapped value of the system time. This loop is necessary because the sampling code may be interrupted when the kernel changes the value of the system time. Since the loop terminates only when the sampled value equals the actual value, the sampled value of the countdown register is consistent with the sampled value of the system clock.

While this method is architecture-dependent, it is much faster than using *gettimeofday()*.

Table 4.1: Clock Sampling Granularity

Sampling Method	# of Filtered Deltas	Total Time (μsecs)	Average Time (μsecs)
[.05in] *gettimeofday()*	49410	2382937	48.2278
map-clock	49761	537218	10.796

Table 4.1 compares deltas between fifty thousand back-to-back uses of the two sampling methods, where deltas that have been skewed by hardware interrupts have been filtered out. While the deviations for both methods, which are not shown, were small and comparable, the *map-clock* method is roughly four and one-half times faster than the ***gettimeofday()*** method. Since the *map-clock* software was immediately available, it was used to obtain timestamps for all experiments.

It should be noted, however, that when timing independent events, the ***gettimeofday()*** method is as precise as the *map-clock* method: given that the deviations in both methods are small, one can remove the overhead of sampling the clock from the timing of an event by subtracting the time required to obtain a clock sample from the computed duration of the event. The length of time it takes to sample the clock, then, is irrelevant to the accuracy of the timing made of any particular event.

4.2 Experimental Results Seeking to Improve Predictor

Understanding gained while interpreting the results of the first three experiments are used to improve the the analytical results of §§ 3.3, 3.4, and 3.5. Where feasible, these improvements are made during the reporting and interpretation of experimental results.

Table 4.2: Timings of Initial TTA Implementation

g	Minimum (μsecs)	Maximum (μsecs)	Average (μsecs)	Standard Deviation
[.05in] 1349	8907	5154374	23979	173519
1624	11602	655692	30331	86065
2129	16455	1109594	50141	116249
2398	19050	1091451	61550	127515

4.2.1 Results of Experiment 1

Experiment 1 seeks to discover the relationship between C_e, the strength of the TTA's bag calculation, and g, the input parameter to the TTA's bag calculation. This experiment corresponds to applying Step 1 of the testing methodology devised in Section 3.5 to the TTA. By discovering how the strength of the TTA's bag calculation varies with its input parameter, C_e can later be varied in a known manner during other experiments.

Timings of the TTA's bag calculation as initially implemented in AKCL varied wildly with integer g, as can be seen by observing the Standard Deviation column of Table 4.2. The standard deviations are up to seven times larger than the mean values. After some thought, it became clear that these data showed an unforeseen consequence of using Lisp: the Lisp Garbage Collector drastically skews an algorithm's expected time.

Using features of AKCL, it was observed that the Garbage Collector (GC) requires from 1 to 20 seconds to complete its job. In one sense, a Lisp algorithm should be charged the time accrued by the GC during its run. However, when timing repeated runs of an algorithm, there is no way to distinguish garbage generated by the algorithm itself and garbage generated by the timing harness. What, then, should be done about GC? There are several possibilities:

1. Don't write Lisp code that generates garbage.

2. Increase allocated memory so that garbage collection occurs less frequently.

3. Throw out timing samples during which garbage collection occurs.

The first approach was tried by modifying the TTA to only use hardware registers. An inspection of the intermediate code produced by the Lisp compiler showed that no function calls were made in the body of the TTA. Presumably such code cannot generate garbage. Timings of the new TTA, however, still varied.

Next, using AKCL primitives, more memory–nearly five megabytes–was allocated for use by the runtime memory allocator. When timings still varied, an inspection was made of the code implementing the memory allocator. It was determined that the memory allocator acquires unused memory in blocks, and that it first calls the GC before acquiring any new memory. The GC will therefore be called whenever the memory allocator needs another block of memory.

For algorithms that clearly do not generate garbage, then, the final approach of ignoring samples in which garbage collection occurs is a reasonable step. Table 4.3 shows statistics for the filtered timings of the modified TTA, where the value initially loaded in the TTA's loop register was scaled so that the TTA runs for approximately one millisecond when presented with a value of one. Note the deviations of this modified algorithm are roughly an order of magnitude smaller than the means.

Table 4.3: Timings of Final TTA Implementation

g	Minimum (μsecs)	Maximum (μsecs)	Average (μsecs)	Standard Deviation
[.05in] 10	9584	69464	9772	1206
20	19238	51902	19383	645
53	50998	305262	51263	3479
104	100050	131752	100503	2358
205	197095	253008	198077	4133
510	490412	558689	492957	7527
1000	965113	1099447	973582	16933

4.2.2 Results of Experiment 2

Experiment 2 seeks to determine how $\overline{T_B}$, the average time to empty and refill the work bag of the TTA, varies with both B, the number of servers, and the length of the messages sent by the TTA (recall that the lengths of the TTA's **START** and **RESULT** messages are the same). It corresponds to performing Step 3 of the testing methodology.

The points plotted in Figure 4.1 show values of $\overline{T_B}$ as message length varies from 2 to 499 bytes for different numbers of servers, and the lines in the figure are approximations of the data using a least-squares fit. Clearly, there is a linear relation between T_B and message length.

If, however, the T_B samples are plotted against the number of servers for different message lengths as in Figure 4.2, a non-linearity in the data becomes apparent. At all message lengths involving calculations with more than four servers, the slope of the line increases sharply, nearly doubling in each case. This anomaly can be seen more clearly when plotting the slopes of the lines in Figure 4.1 against the number of servers, as in Figure 4.3.

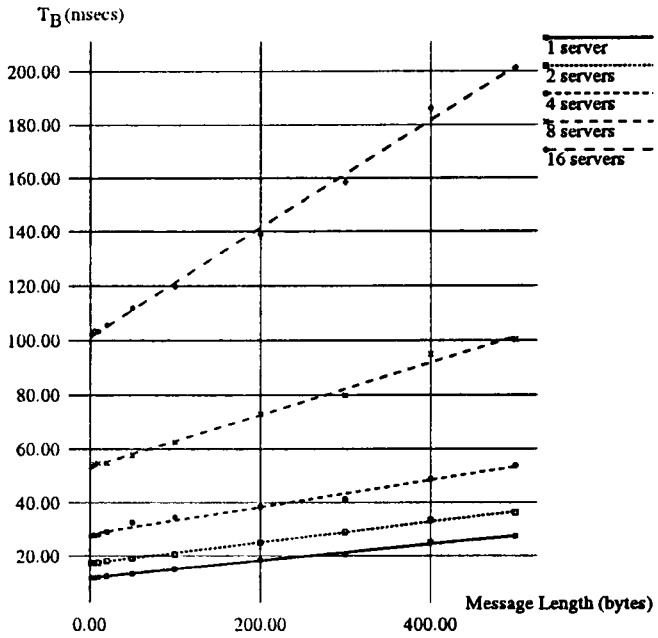

Figure 4.1: T_B versus l for Different B

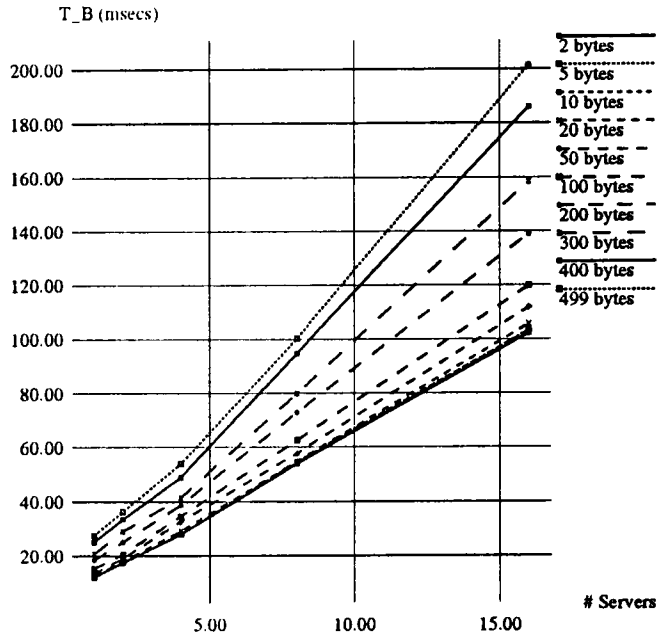

Figure 4.2: $\overline{T_B}$ versus B for Different l

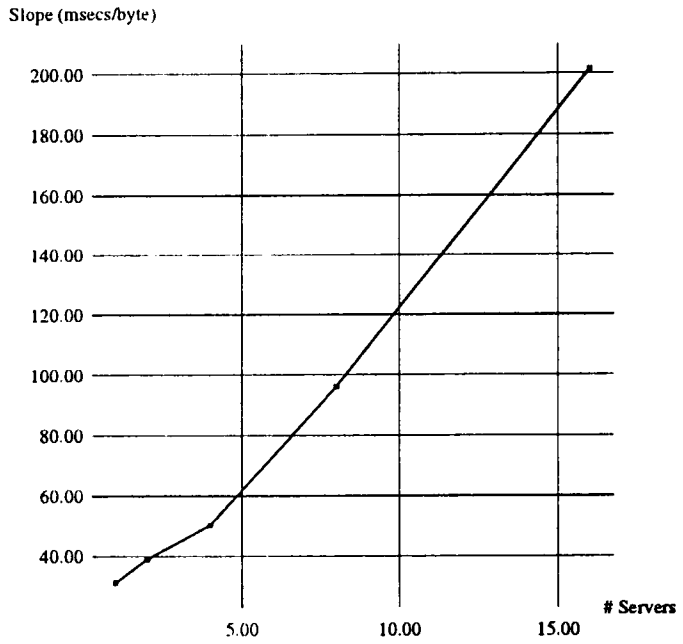

Figure 4.3: Slope of $\overline{T_B}$ versus B

Accounting for the knee at four servers is difficult, but there are a few observations about the nature of the curves in Figure 4.2 and Figure 4.3 that are helpful in eliminating a number of possibilities as to its origin. Firstly, and most obviously, the phenomenon causing the knee is clearly not dependent on message length (this conclusion can be drawn from a casual observation of Figure 4.1, as well). Secondly, it does not arise because of flow control problems in RCalc: were the knee the result of flow control issues, it would appear after a set amount of data had been presented to RCalc in a given time, and, *e.g.*, if the increase in slope appeared at 4 servers for 200 byte messages, it would also appear at 8 servers for 100 byte messages. Lastly, one may conclude that the knee phenomenon does not arise because processes are suddenly exceeding their allotted time slice: since the lines on either side of the knee include points that plot events occurring in less than 100 milliseconds, the timesharing quanta for the systems used in the experiment, timesharing effects do not drive the location of the knee.

More possibilities may be eliminated when examining Figure 4.4, the graph of $T_{B_{\min}}$, the minimum times observed during Experiment 2, versus the number of servers. The knee still appears at four servers. This suggests that the knee is not a result of retransmissions, since one might presume minimum timings to be those of events during which retransmissions do not occur. Similarly, the knee's appearance at four server in Figure 4.4 reinforces the conclusion that the knee is not an effect of context switching.

The knee effect could occur when the hardware buffers which hold incoming, unserviced packets are all in use. An inspection of the relevant source code reveals that SunOS

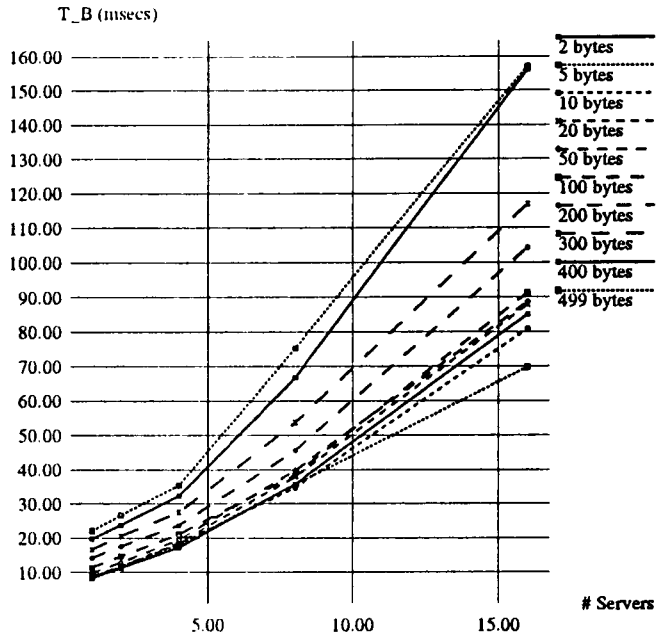

Figure 4.4: $T_{B_{min}}$ Versus B Different l

configures the Ethernet chip used on the experimental systems[1] with 40 buffers capable of holding 1600 bytes each. Filling these buffers, then, is an unlikely cause of an effect seen when moving from 4 to 8 hosts.

Likewise, the knee effect could be the result of exhausting operating system buffers that hold packets until a user level program accepts them. Since SunOS allocates 50 buffers to hold incoming IP packets, each of which can grow up to a kilobyte, it is unlikely that the knee effect occurs when these buffers are all in use.

The knee effect could be an artifact of the code implementing RCalc and OCARD. While no conclusive evidence can be presented that rules out this possibility, it is useful to note that the author has spent many days inspecting RCalc and OCARD code, vainly attempting to uncover just such an artifact.

Having eliminated a number of possible causes for the non-linear nature of how $\overline{T_B}$ varies with B, one is forced to re-examine the assumptions that suggested a linear relation between $\overline{T_B}$ and B. These assumptions are repeated here from Section 3.4:

1. RCalc is well-behaved in that the time to process a message of length l is $cl + k$.

2. The overhead of sending a `START` message is the same as that of sending a `RESULT` message.

3. Multiprogramming delays incurred by RCalc are entirely attributable to RCalc's operation.

4. During a single bag iteration, the last `RESULT` message to be received is, on average,

[1]The IPX uses the AMD 7990 LANCE Ethernet chip.

the one returned by the Bth server.

Assumption 1 is borne out by Figure 4.1, showing the linear relation between $\overline{T_B}$ and message length. Since the experiments do not explore the case where the length of the START and RESULT messages differ, their results can neither confirm nor deny Assumption 2. Note that even if Assumption 2 is not made, one would still derive a linear equation in l for $\overline{\tau_m}$ and thus for $\overline{T_B}$. Assumption 3 is nearly a given in the experimental environment: if no other processes were running on the hosts performing the experiment, from where else would multiprogramming delays arise other than the experiment? Assumption 4 is suspect, if only in the sense that one questions the validity of an equation based on which RESULT message the driver receives last. Such doubt leads to a new derivation of T_B based on the total amount of work the driver must perform, rather than the order in which it is performed. To make this derivation, one defines T_{B_e} as the time it takes to empty the bag.

By thinking of the sending of a START message as scheduling work for the driver to complete a single START/RESULT transaction, T_{B_e} can be considered the time it takes for the work scheduled by all START messages to complete. The case of one server is shown in Figure 4.5, where sending a single START message schedules both the work needed to send that START message and the work needed to receive its corresponding RESULT message. Let δ be the RCalc overhead that does not take place on the driver, so that $\delta = \alpha + R_s + T_r + \beta$. The work scheduled on the driver by sending a single START message at time t to one server completes at time $t + T_s + \delta + C_e + R_r$, and $T_{B_e} = T_s + \delta + C_e + R_r + k_0$ (recall that k_0 is the fixed overhead of using RCalc). The work scheduled on the driver when distributing the bag over two servers, shown in Figure 4.6, completes at time $t + T_s + \delta + C_e + \max(T_s, R_r) + R_r$. The max operator reflects the scheduling of some of the driver's work during the latency

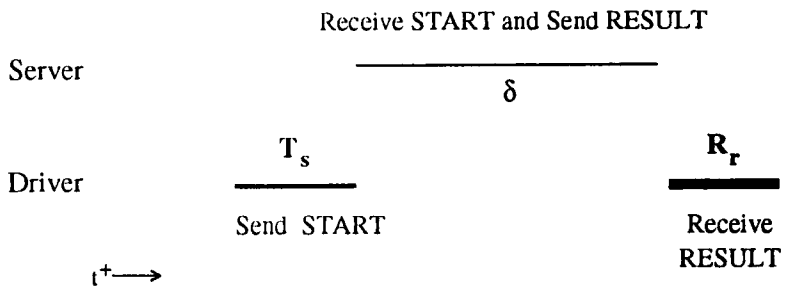

Figure 4.5: Work Schedule for Driver when $B = 1$

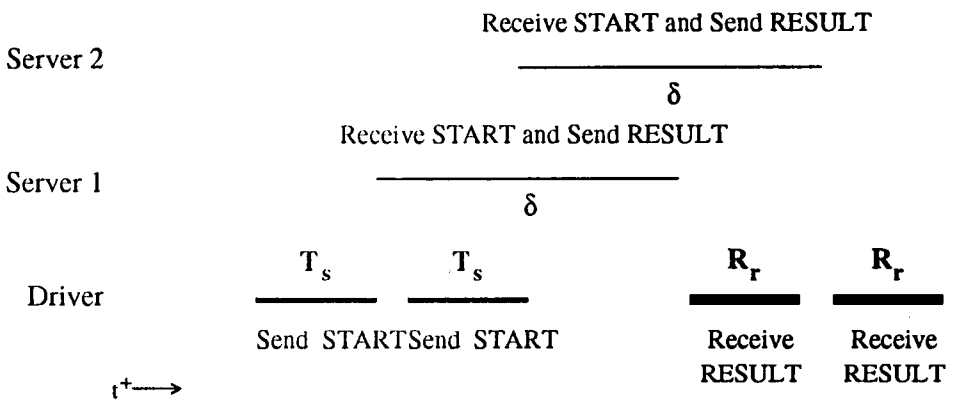

Figure 4.6: Work Schedule for Driver when $B = 2$

introduced by δ. The behavior as more servers are added is captured in the following equation

$$T_{B_e} = T_s + \delta + C_e + (B-1)\max(T_s, R_r) + R_r + k_0 \tag{4.1}$$

This equation holds until so many START messages are to be sent that the work of receiving RESULT messages interferes with the work of sending START messages. At that point, the driver is saturated with work, the latency introduced by δ and C_e disappears, and the equation for T_{B_e} changes to:

$$T_{B_e} = B(T_s + R_r) + k_0 \tag{4.2}$$

To pictorially represent this saturation effect, the work of receiving a RESULT message, R_r, must be broken down further into two components:

- that work done by the kernel while servicing interrupts generated by the driver's Ethernet controller as it accepts RESULT messages, and
- that work done by user level code variously implementing OCARD, RCalc and the driver program, as it parses and passes the RESULT message across process boundaries.

The interrupt level work occurs at fixed times which are not pre-emptable by other work. In contrast, the user level work to receive RESULT messages is pre-emptable and is merely constrained to occur after the corresponding interrupt-level work has completed and, in the case of the work done by the driver program, after all START messages have been sent.

Figure 4.7 shows the case where $B = 4$ and the saturation effect comes into play. In it, work on the driver that is pre-emptable is shown as dotted lines, and solid lines represent driver work that cannot be pre-empted. R_r is decomposed into two parts as above, where the interrupt level work is labelled in the legend as "Field RESULT" and the user level work

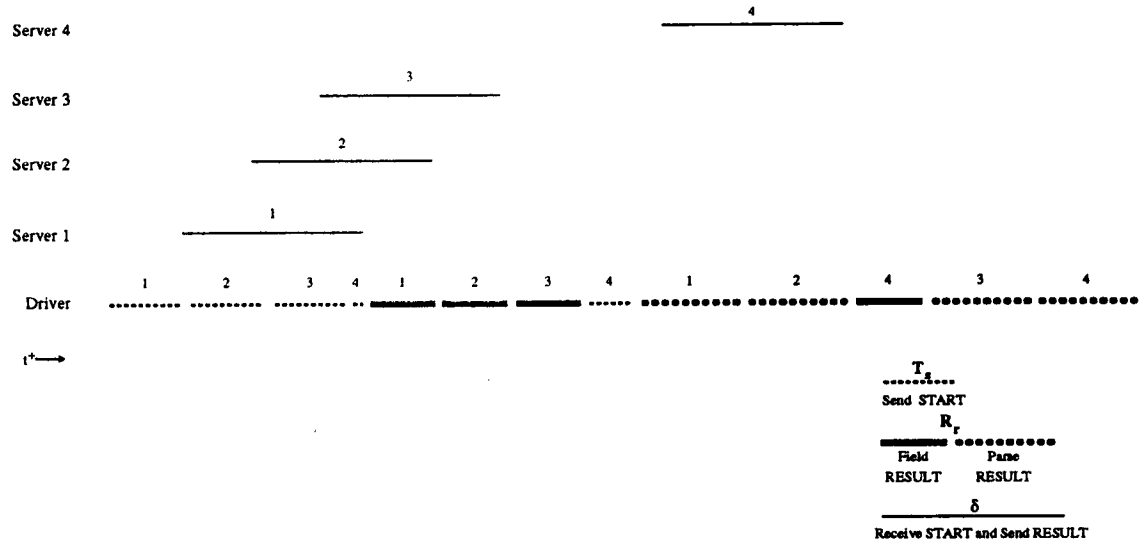

Figure 4.7: Work Schedule for Driver Showing Saturation Effect ($B = 4$)

is labelled there as "Parse RESULT." Each piece of work is tagged with the number of the START/RESULT transaction to which it belongs.

The saturation effect can be seen in Figure 4.7 while the fourth START message is being sent. Shortly after the work to send START 4 begins, the driver's CPU is interrupted to receive three RESULT messages. Next, sending START 4 finishes, scheduling the interrupt-level work of RESULT 4 to occur $\delta + C_e$ seconds from the time at which START 4 completes. Unlike the unsaturated case, however, previously scheduled user-level work fills the time between now and then, and the driver proceeds to parse RESULTS 1 and 2. This work completes, after which the final interrupt generated by this bag iteration is handled. The driver finishes by parsing RESULTS 3 and 4. Note that there is no time during which work is not scheduled on the driver.

While somewhat contrived, the previous example demonstrates a likely cause of the knee effect seen in Figure 4.2. The ratio $\frac{(\delta+C_e)}{T_s}$ determines where the knee occurs; that is, the number of of **START** messages that can be sent before the first **RESULT** message arrives will determine the number of servers that saturate the driver host. In the example, this ratio is close to three; the results of Experiment 2, where $C_e = 0$, show that $\frac{\delta}{T_s}$ for the experimental environment is closer to four.

If the T_s, δ, and R_r are assumed to be linear in l, the length of **START** and **RESULT** messages, and if it is further assumed that $R_r > T_s$ (as will be argued later), then Equations 4.1 and 4.2 change to

$$T_{B_e} = (c_s + c_\delta)l + B(c_r l + k_r) + k_s + k_\delta + k_0 + C_e, \quad B \leq 4 \tag{4.3}$$

$$= B((c_s + c_r)l + k_r + k_s) + k_0, \quad B \geq 4 \tag{4.4}$$

By comparing how the slopes and intercepts taken from the lines in Figures 4.1 and 4.2 vary with B, values for most of the various c's and k's can be estimated. The system of equations so derived follows:

$$\begin{aligned} c_r &= 6.25 \\ c_s + c_\delta &= 25.65 \\ c_s + c_r &= 12.67 \\ k_r &= 5449 \\ k_s + k_\delta + k_0 = &= 6502 \\ k_r + k_s &= 6057 \\ k_0 &= 4438 \end{aligned}$$

Solving it yields:

$$c_r = 6.25, \quad k_r = 5449$$

$$c_s = 6.42, \quad k_s = 608$$

$$c_\delta = 19.23, \quad k_\delta = 2627$$

$$k_0 = 4438$$

Substituting these values into Equations 4.3 and 4.4 with $C_e = 0$ gives:

$$T_{B_e} = (6.25B + 25.65)l + 5449B + 6502, \quad B \leq 4 \tag{4.5}$$

$$= 12.67Bl + 6057B + 4438, \quad B \geq 4 \tag{4.6}$$

Figure 4.8 shows the observed values of T_{B_e} and the lines produced from the previous equations. While their agreement bodes well for a predictor based on work schedules, some interpretation of these values is in order.

Note, for instance, that $k_s \ll k_r$. Sadly, this is expected. k_r, the per-message overhead incurred when receiving a message, includes both a context switch from OCARD's reader process into the driver's client Lisp program[2] and times to field packet interrupts. In contrast, there are at most two context switches from the driver program to the OCARD writer no matter how many messages are sent,[3] and k_s, the per-message overhead incurred

[2]The reader's main loop awaits network I/O and then iterates over reading an available packet and sending it to the client. If the inter-arrival gap between packets is such that the reader can finish delivering a packet to the client before another packet arrives, the kernel will put the reader to sleep awaiting a packet and awaken the client to accept the previously delivered packet. In the worst case, this behavior can result in one context switch per packet.

[3]At most 8 kilobytes are sent, twice the size of the buffer reserved by SunOS for interprocess communication between the client and the writer, so at most 2 context switches occur when the writer sends packets to the servers.

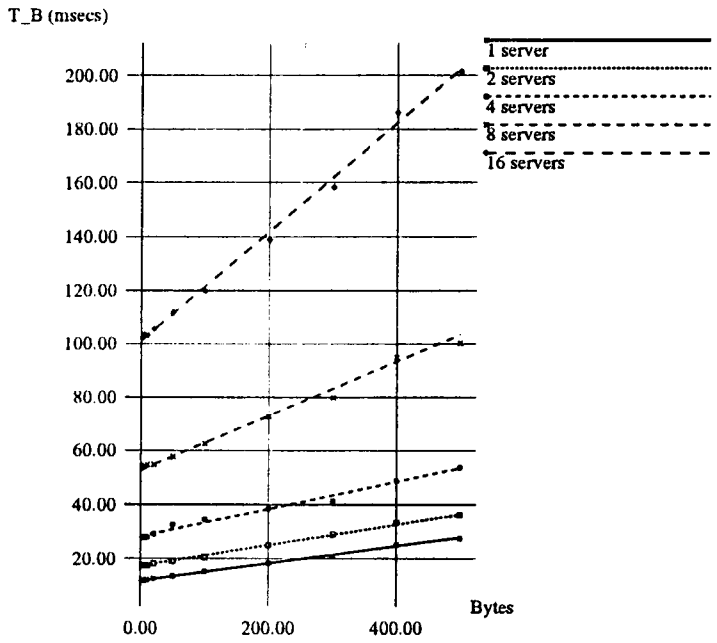

Figure 4.8: Observed and Predicted T_{B_e} versus B

when sending a message, does not include any times attributable to interrupts.

That $c_s > c_r$ is not surprising, given that c_s, the per-byte overhead associated with sending a message, includes delays incurred while messages are saved for potential retransmission. Note that for all $2 \leq l \leq 500$, $6.42l + 608 < 6.25l + 5449$, *i.e.*, , $T_s < R_r$, the previous arguments justify the earlier assumption that $T_s < R_r$.

The value of c_δ includes times while a server both receives and sends a message, in addition to two transmissions of a message via UDP. Since the same code implementing OCARD runs on both drivers and servers, c_δ should be nearly $c_s + c_r$ plus UDP transmission times. The lower bound on UDP transmission times is the data rate of the underlying hardware, 1.25 bytes per μsecs in the case of Ethernet, so one would expect c_δ to be at least $6.25 + 6.42 + 2.5 = 15.17$. The computed value of 19.23 is close enough to this lower bound that one may conclude c_δ is consistent with the other constants of proportionality.

While k_0 includes the time to switch once from the driver's Lisp program to OCARD's reader, it is likely dominated by garbage collection times, some of which will not be proportional to either B or l. Since no timings are made until RCalc-Lisp is initialized, k_0 does not include times for such things as establishing a connection to a nameserver or starting the writer or reader processes.

4.2.3 Results of Experiment 3

Experiment 3 seeks to determine how T_B varies with C_e, the strength of the bag calculation, and B, the number of servers (and, hence, the size of the bag). The TTA is used as the bag calculation in a bag-of-work style parallel algorithm implemented in RCalc using different numbers of servers. The average of five minutes of samples are shown in Figure 4.9,

demonstrating a linear relation between C_e and T_B, where the slopes of the lines are, as expected, very nearly one.

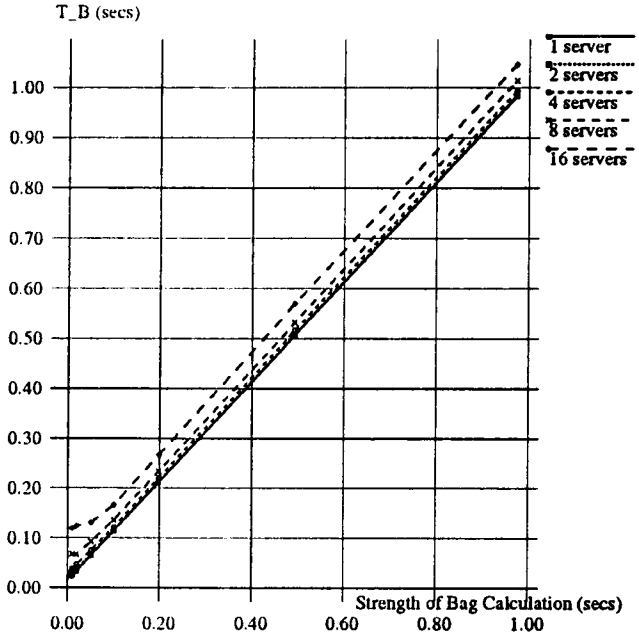

Figure 4.9: $\overline{T_B}$ versus C_e for Different B

The points to the left of Figure 4.9 on the lines for 8 and 16 servers show a slight non-linearity. This non-linearity cannot conclusively be accounted for, though it may be a result of a difference in the order in which the kernel's scheduling algorithm chooses processes. The effect is small enough that it is considered outside the scope of this study, and discovering its cause is left for future research.

4.3 Experimental Results to Validate Improved Predictor

4.3.1 Results of Experiment 4

Experiment 4 seeks to determine the strengths of the different bag calculations in the PSA-TSP. Recall that the PSA-TSP bag calculation randomly perturbs a given configuration until each city has been moved a number of times dependent on the number of participating servers. Figure 4.10 shows the average values of C_e for the PSA-TSP as the number of moves changes.

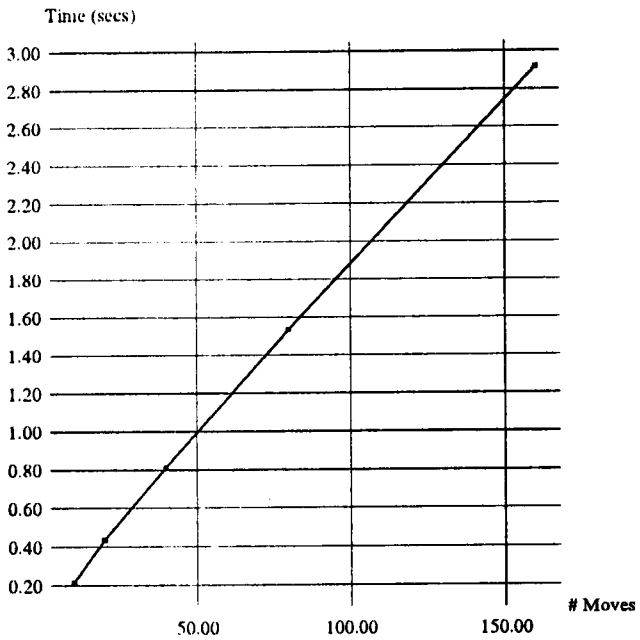

Figure 4.10: Average Time to Complete Bag Calculation of PSA-TSP vs. Number of Moves to Completion

It should be pointed out that the linearity of this curve is wholly a property of solving the TSP using SA, and thus has no effect on how these data are used in the testing methodology nor on how the data are used in the analytic predictor. No conclusions need be made on the data in Figure 4.10, and only one observation is relevant to this study: the bag calculations of the PSA-TSP are precisely those of a serial implementation of a simulated annealing TSP-solver. Therefore, these results also give empirical values of s, used in the computation of E (*cf* Section 3.4).

4.3.2 Results of Experiment 5

Experiment 5 gives values for C_r of the PSA-TSP. This experiment corresponds to applying Step 1 of the testing methodology to the PSA-TSP. Figure 4.11 shows average times to refill the bag of PSA-TSP versus the number of participating servers. Again, these times are properties of the PSA-TSP, and need not be dwelt upon.

4.3.3 Results of Experiment 6

Experiment 6 times the PSA-TSP when different number of servers, B, are used. Figure 4.12 shows the average of five minutes of run times of the PSA-TSP, plotted versus B. These values are taken to be p when computing E, the efficiency of the PSA-TSP.

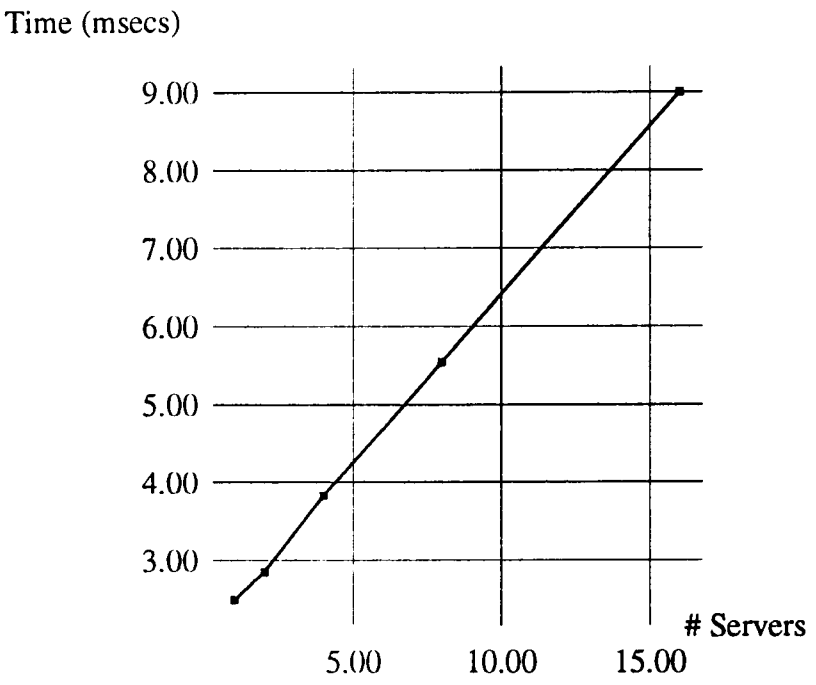

Figure 4.11: C_r for PSA-TSP versus B

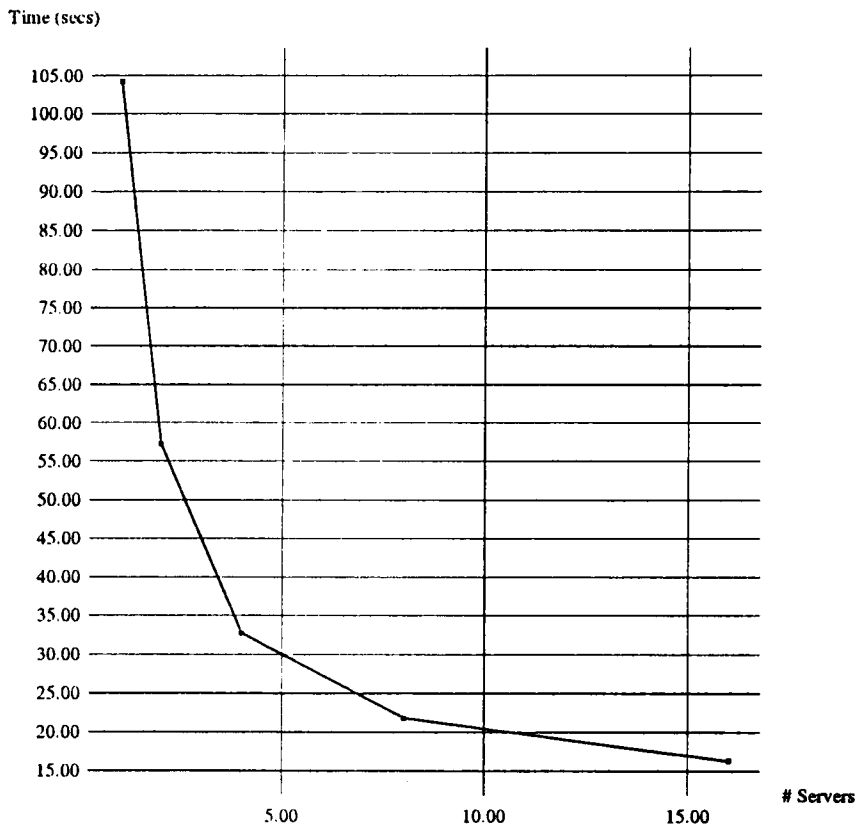

Figure 4.12: Average Run Time of PSA-TSP vs. B

4.4 Answers to Research Questions

In the following sections, the research questions of Section 1.3 are explicitly addressed.

4.4.1 QUESTION ONE: Derive a Predictor of Efficiency

Restated from Chapter 1, Research Question One is:

> To what extent can a predictor for the efficiency of bag-of-work algorithms as realized using RCalc be analytically derived?

Equation 3.2

$$E_{\text{upper}} = \frac{B(C_e + \overline{p}) + C_r + \overline{\phi}}{(B+1)(C_e + \overline{p} + C_r + \overline{\phi})}$$

gives an upper bound on efficiency. Equation 3.6

$$E = \frac{C_r + \overline{\phi} + B(C_e + \overline{p})}{(B+1)(C_r + \overline{\phi} + C_e + \overline{p} + c_l(l_T + l_R) + k_0) + (B^2 - 1)c_s l_T + (B+1)^2 k_B}$$

is the derived analytic predictor of efficiency.

Both are derived from a model of bag-of-work computations meeting the following criteria, repeated here from Section 3.3:

- The algorithm iterates through the steps of first distributing all work in the bag across server processes, and then refilling the bag with new work based on the results. All the outstanding work must complete before the bag can be refilled with new work.
- The algorithm's bag tasks are all similar in nature. In particular, they each require the same amount of CPU time, and both their CPU requirements and the CPU time necessary to convert their results into new tasks do not change from one iteration of emptying and refilling the bag to the next.

- The size of the bag is not changed by the algorithm.

The restrictions on using RCalc for the analytic predictor to apply are:

- A single driver host is solely responsible for emptying and refilling the bag, and it never performs any of the work in the bag.
- At the start of the algorithm, there are exactly as many tasks in the bag as there are server processes.
- The server and driver hosts are of equal power, and the only processes that execute on them are those involved with either RCalc or the algorithm.

The assumptions used in the derivation of the of Equation 3.6 are:

- Message passing delays are linear in the length of messages.
- Sending and receiving `START` messages with RCalc incurs the same type of delays as sending and receiving `RESULT` messages.
- Multiprogramming delays incurred on the server while calculating **`RESULT`** messages have low deviation and are less than delays introduced by RCalc.

4.4.2 QUESTION TWO: Test and Modify the Predictor

Restated from Chapter 1, the first part of Research Question Two is:

> To what extent can a testing methodology be devised so that the analytic predictor of efficiency may be applied to bag-of-work algorithms as implemented in RCalc?

The second half of Research Question Two is:

> To what extent can the analytic predictor of efficiency be modified to account for the results of applying the testing methodology to RCalc?

Q2-A has been answered in Section 3.5. The testing methodology presented there is outlined here:

Perform the following steps for each algorithm.

1. Determine how the strength of the algorithm's bag calculation, C_e, and the strength of the calculation required to refill the bag, C_r, both vary with the algorithm's input.
2. Determine how the lengths of the algorithms' **START** and **RESULT** messages vary with the algorithm's input.

Perform the following steps once per experimental environment.

3. Determine how the TTA's $\overline{T_B}$ varies with the number of servers, B, and message length.
4. Using results of the previous step, determine constants that characterize RCalc's performance.

The results of applying the methodology and using the analytic predictor on the TTA follow.

The upper bound given in Equation 3.2 is plotted along with the maximum efficiencies obtained in Experiment 3 in Figure 4.13. The tightness of the bound increases as C_e

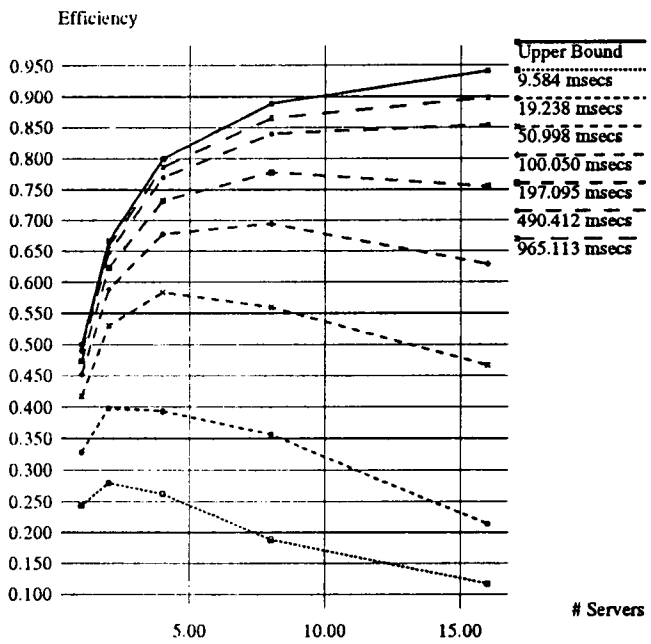

Figure 4.13: Upper Bound and Maximum Observed Efficiencies of TTA versus B for Different C_e

increases, that is, as the overhead of RCalc becomes negligible with respect to the computations performed by the servers. Since the bound was produced based on the assumption of no communication delays, this increasing tightness is only expected. Equation 3.2 is an accurate upper bound.

The assumptions made in Section 3.4 to derive the analytic predictor of Equation 3.6, however, have been shown to be flawed: they do not predict the saturation effect seen in Figure 4.2. To do so, two new equations for p, the time to perform a bag-of-work calculation in parallel, are derived based on Equations 4.3 and 4.4. One, for p_{unsat}, applies in the absence of saturation; the other, for p_{sat}, in its presence. These are used to derive corresponding equations for E, and thus a better analytic predictor of E.

$$\begin{aligned} p_{\text{unsat}} &= \frac{N}{B}\overline{T_B} \\ &= \frac{N}{B}(\overline{T_{B_c}} + C_c + \overline{\rho} + C_r + \overline{\phi}) \\ &= \frac{N}{B}((c_s + c_\delta)l + B(c_r l + k_r) + k_s + k_\delta + k_0 + C_r + \overline{\phi}) \end{aligned}$$

The efficiency when the driver remains unsaturated is thus

$$\begin{aligned} E_{\text{unsat}} &= \frac{S_{\text{unsat}}}{(B+1)} \\ &= \frac{s}{p_{\text{unsat}}(B+1)} \\ &= \frac{N(\frac{(C_r+\overline{\phi})}{B} + C_c + \overline{\rho})}{\frac{N}{B}(B+1)((c_s + c_\delta)l + B(c_r l + k_r) + k_s + k_\delta + k_0 + C_c + \overline{\rho} + C_r + \overline{\phi})} \\ &= \frac{B(\frac{(C_r+\overline{\phi})}{B} + C_c + \overline{\rho})}{(B+1)((c_s + c_\delta)l + B(c_r l + k_r) + k_s + k_\delta + k_0 + C_c + \overline{\rho} + C_r + \overline{\phi})} \end{aligned}$$

$$= \frac{C_r + \overline{\phi} + B(C_e + \overline{p})}{(B+1)((c_s + c_\delta + c_r)l + k_s + k_\delta + k_r + k_0 + C_e + \overline{p} + C_r + \overline{\phi}) + (B^2 - 1)(c_r l + k_r)} \quad (4.7)$$

While quantitatively different from Equation 3.6, the qualitative analyses of Equation 4.7 and Equation 3.6 yield similar results. If, as in Section 3.4, the limit of Equation 4.7 as $B \rightarrow \infty$ is taken, the B^2 term in the denominator dominates, and, again as in Section 3.4, $E_{\text{unsat}} \rightarrow 0$. One can conclude that effort spent improving RCalc should be concentrated on reducing c_r and k_r, the per-byte and per-server/message overhead associated with receiving messages. Given the assumption that $R_r > T_s$, this conclusion amounts to "reduce the most expensive per-byte and per-server/message overhead in RCalc," not at all different from the previous suggestion based on Equation 3.6 of reducing per-byte and per-server/message overhead. And, again, for a given number of server hosts, B, Equation 4.7's maximum is reached as $C_e \rightarrow \infty$ at $\frac{B}{B+1}$, and its minimum of $\frac{1}{B+1}$ is reached as $C_r \rightarrow \infty$. The suggestion that efficiency can be improved by giving the servers involved in a calculation as much work to perform as possible while giving the driver as little work as possible is exactly that drawn by observing the extrema of Equation 3.6, and it is still as obvious.

In the saturated case, $p_{\text{sat}} = \frac{N}{B}(B((c_s + c_r)l + k_s + k_r) + k_0 + C_r + \overline{\phi})$ and

$$\begin{aligned} E_{\text{sat}} &= \frac{s}{p_{\text{sat}}(B+1)} \\ &= \frac{N(\frac{(C_r + \overline{\phi})}{B} + C_e + \overline{p})}{\frac{N}{B}(B+1)(B((c_s + c_r)l + k_s + k_r) + C_r + \overline{\phi} + k_0)} \\ &= \frac{C_r + \overline{\phi} + B(C_e + \overline{p})}{(B+1)((c_s + c_r)l + k_s + k_r + k_0 + C_r + \overline{\phi}) + (B^2 - 1)((c_s + c_r)l + k_s + k_r)} \end{aligned} \quad (4.8)$$

It is clear that Equations 4.8 and 4.7 are qualitatively similar, and none of the suggestions derived from the qualitative analysis of Equation 4.7 nor of Equation 3.6 are contradicted:

the model parameters having the greatest impact on efficiency are B, C_e, and C_r, while of the properties of RCalc, the per-byte and per-server/message overhead dominate E.

Using Equation 4.7, the unsaturated case, and the previously derived values of various c's and k's, the efficiency of the TTA is predicted. The efficiencies so predicted are plotted in Figure 4.14 versus the number of servers participating in the TTA for different strengths of bag calculations. In Figure 4.15, the efficiencies observed in Experiment 4 are similarly

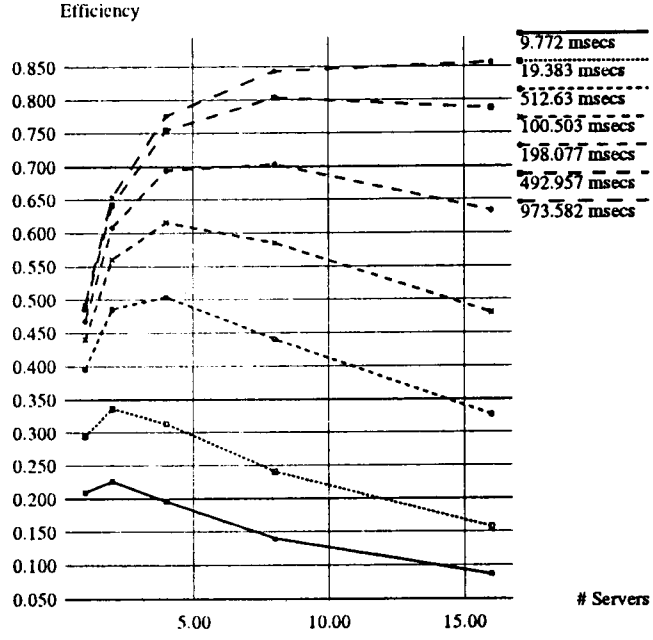

Figure 4.14: Predicted Efficiency of the TTA versus B for Different C_e

plotted. Figure 4.16 shows the ratios of the predicted efficiencies over the the observed efficiencies plotted versus the strength of the bag calculations. As can be seen, Equation 4.7 generally underestimates the efficiency of the TTA, and its predictions worsen as

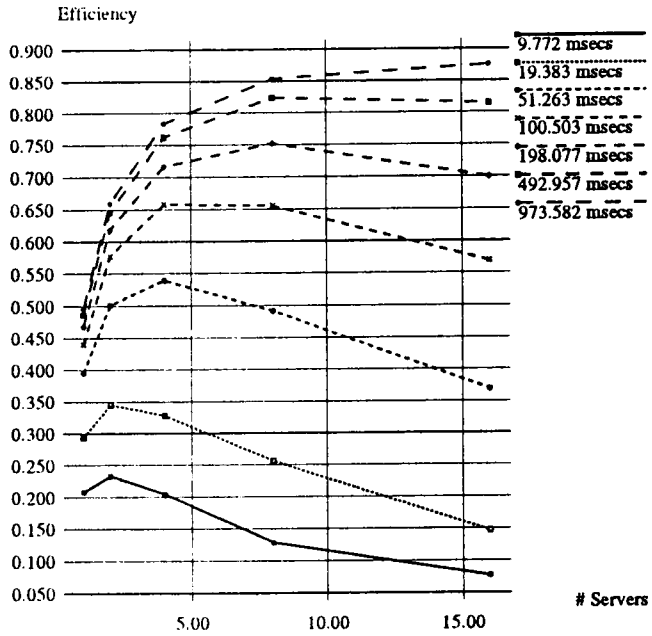

Figure 4.15: Observed Efficiency of the TTA versus B for Different C_e

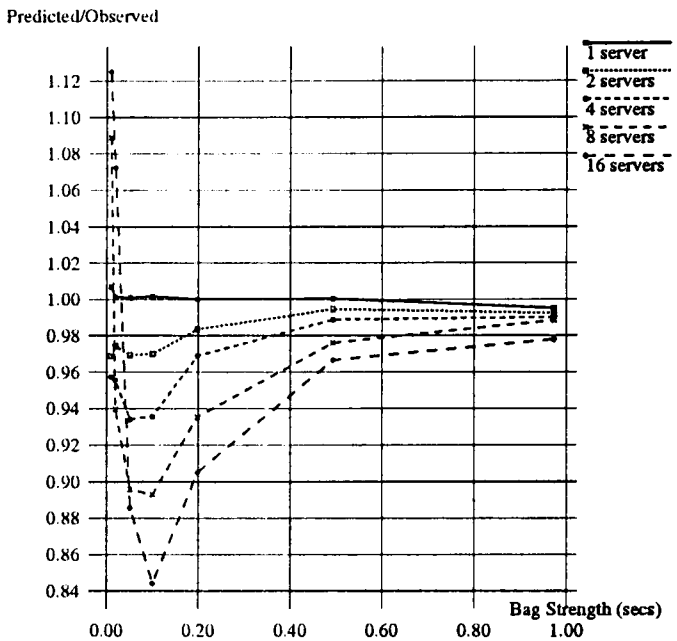

Figure 4.16: Predicted over Observed Efficiencies of the TTA versus C_e for Different B

the number of servers increases. However, Equation 4.7 is rarely wrong by more than ten percent, and it is never wrong by more than sixteen percent. Clearly, the error is some function of C_e. The shape of the curves are similar to those found in analyzing critically damped vibration, suggesting that the error is an exponential function of C_e. The predictor is at its most erroneous where C_e is near zero and near one-tenth of a second, near the edges of the Unix scheduler's time quanta. The location of these extrema, and the resemblance of the curves in Figure 4.16 to those involving exponential decay used by the Unix scheduler to calculate a process's priority, suggests that part of the error in prediction likely reflects an error in modelling multiprogramming delays. The multiprogramming delays incurred during Experiment 3, for example, were modelled as the sum of the average delays incurred during Experiments 1 and 2. However, the calculations in Experiments 1 and 2 involve I/O,[4] during which time they relinquish the CPU to a runnable process. The average multiprogramming delay experienced by the two sets of processes is then not the sum of the average times to execute each set: it is less than the sum, and the degree to which it is less than the sum is controlled by the kernel's exponential-based scheduling policy.

Figure 4.17 shows this same ratio of predicted efficiency over observed efficiency, plotted versus the number of servers for different bag strengths. While some regularity exists in these data, the points in the upper right corner of the graph are anomalous. They arise from data previously (*cf* Section 4.2.3) defined to be outside the scope of this study. If one excludes these points, the curves of Figure 4.17 appear to be some function of B.

[4]This is true for any process: even if no explicit I/O is done by a process, its least recently used memory pages will be paged out to backing store, and newly referenced ones will be read into memory.

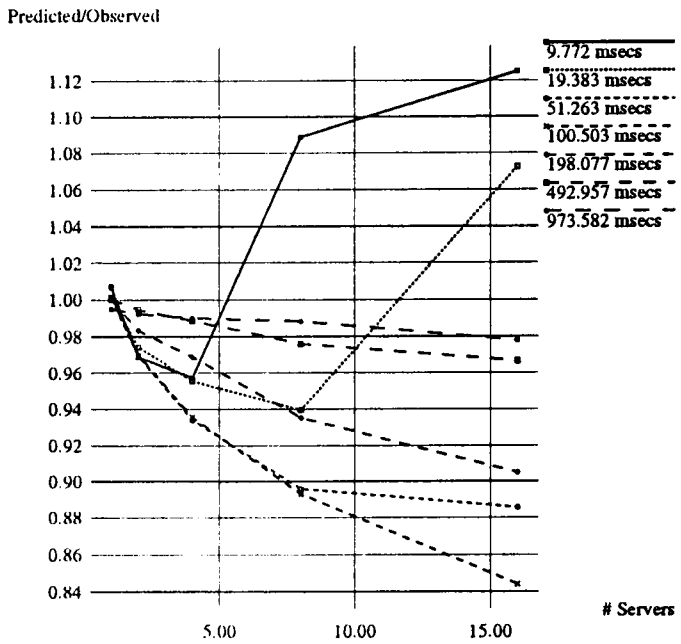

Figure 4.17: Predicted over Observed Efficiencies of the TTA versus B for Different C_e

Plotting $\frac{E_{\text{pred}}}{E_{\text{obs}}}B$ versus the number of servers for different bag strengths, as in Figure 4.18, uncovers this function: the percentage error in prediction appears to be proportional to the number of servers. Excluding the previously specified points, this means

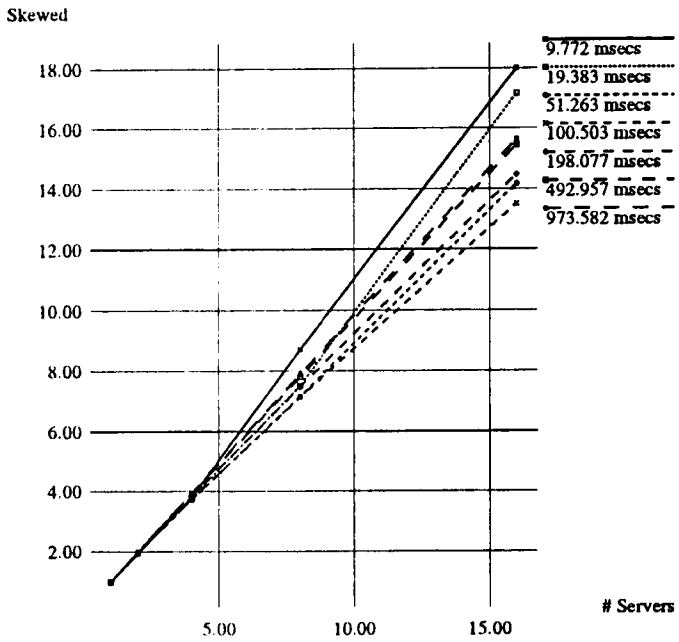

Figure 4.18: $\frac{E_{\text{pred}}}{E_{\text{obs}}}B$ versus B

$$\begin{aligned} \frac{E_{\text{pred}}}{E_{\text{obs}}}B &\propto B \\ &= cB + k \\ E_{\text{obs}} &= \frac{B}{cB+k}E_{\text{pred}} \end{aligned}$$

Figure 4.16 suggests that c and k are exponential functions in C_e.

4.4.3 QUESTION THREE: Validate the Predictor

Research Question Three, restated from Chapter 1, is:

> To what extent is the combination of the analytic predictor and the testing methodology validated when applying it to an example bag-of-work algorithm?

Using Equation 4.7 to predict E, Figure 4.19 shows observed and predicted efficiencies for the PSA-TSP plotted against the number of servers. Figure 4.20 plots the ratios of the predicted efficiency to the observed efficiency versus the number of servers. The error ratio increases with the number of servers to a maximum of twenty-one percent. If, as in Figure 4.21 $\frac{E_{\text{pred}}}{E_{\text{obsv}}}B$ is plotted versus the number of servers, a linear relation similar to that in the case of the TTA can be seen. Figure 4.22 plots the error ratio versus bag strength, and, again as in the case of the TTA, an exponential relation between the error ratio and C_e is suggested.

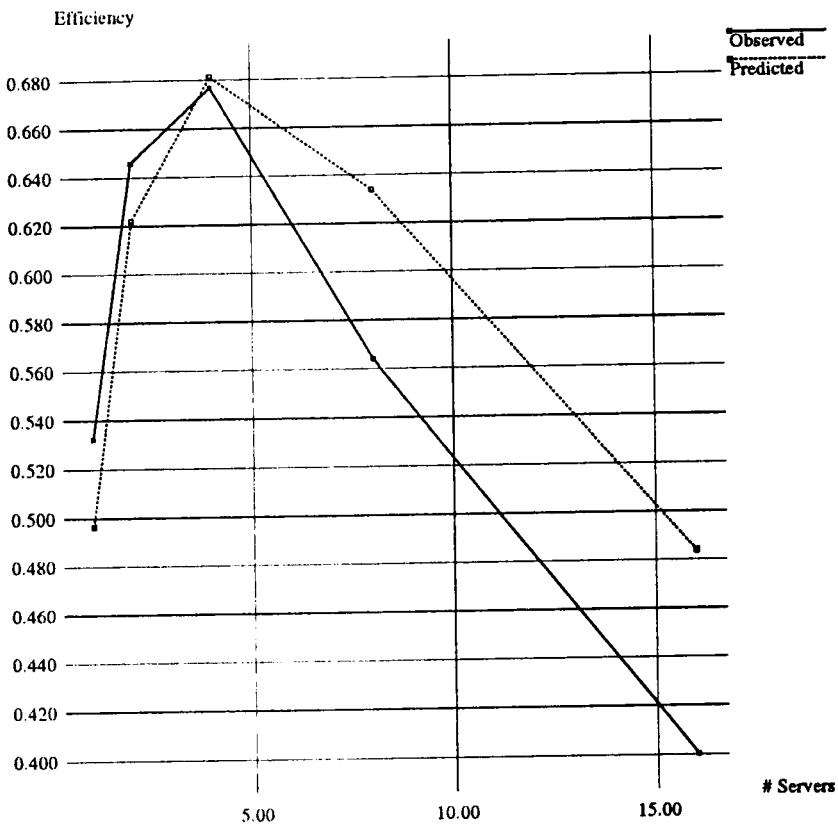

Figure 4.19: Efficiency of the PSA-TSP versus B

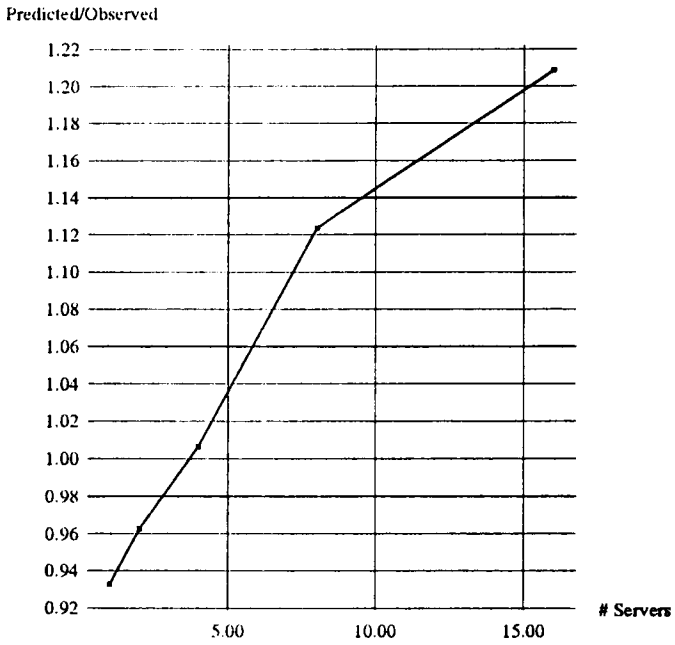

Figure 4.20: Predicted Efficiency over Observed Efficiency versus B

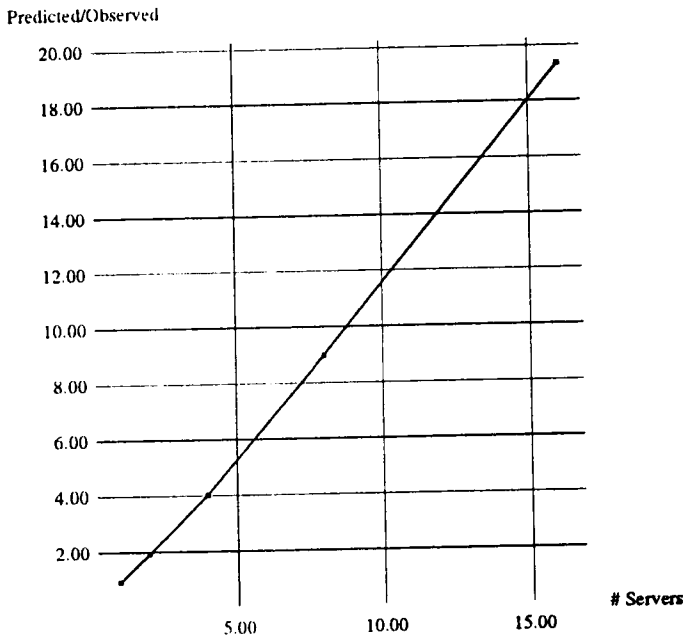

Figure 4.21: $\frac{E_{\text{pred}}}{E_{\text{obs}}} B$ versus B for the PSA-TSP

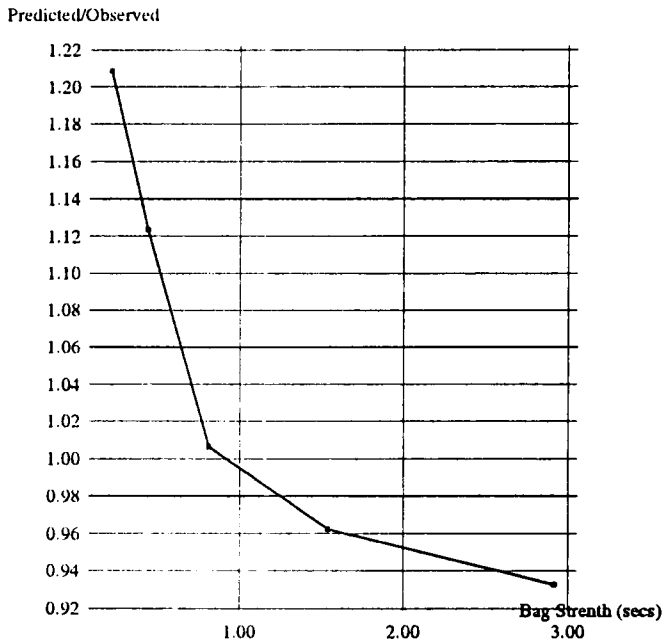

Figure 4.22: Predicted Efficiency over Observed Efficiency versus C_e for the PSA-TSP

Chapter 5

Summary, Conclusions, and Recommendations

This Chapter summarizes the study and its results, draws some conclusions based the study's results, and makes some recommendations for future research.

5.1 Summary

Four results have been presented. The first is RCalc-Lisp, a MIMD-LAN framework that may be used to implement parallel algorithms in Common Lisp given a collection of Unix workstations that run the TCP/IP protocol suite. The second result is an analytic predictor of parallel efficiency that applies to any MIMD-LAN system used to implement certain types of bag-of-work parallelism. The third result, closely related to the second, is a testing methodology that can be used to both obtain performance characteristics needed to apply

the analytic model to a MIMD-LAN system and to judge the accuracy of the analytic model. The final result is illustrative, reporting on the application of both the predictor and the methodology to RCalc-Lisp and the experiences gained in doing so.

5.1.1 RCalc-Lisp

RCalc-Lisp uses a UDP-based transport, OCARD, to allow a single *driver* host to reliably distribute work in the form of Lisp S-expressions across a collection of *server* hosts. Using simple Lisp functions, the *driver* presents these S-expressions to the *servers*, which asynchronously return the results of evaluating the S-expressions to the *driver*. Parallelism is achieved when more than one *server* can simultaneously evaluate different S-expressions. While not constrained to implement any particular type of parallelism, RCalc-Lisp has been shown to be suited to bag-of-work style parallelism.

RCalc-Lisp has been used to implement a parallel Simulated Annealing engine based on the work in [DLS93], and this engine has been applied to solving a 30 city Travelling Salesman Problem.

The most current version of RCalc-Lisp has only been tested on Sun-4 workstations running SunOS-4.1.3 using Austin Kyoto Common Lisp. The data gathered regarding RCalc-Lisp's performance pertain only to SparcStation IPXs running SunOS-4.1.3.

5.1.2 Analytic Predictor

As presented, the analytic predictor is constrained to modelling bag-of-work style parallelism where a single CPU, known as the driver, alternates between two states, that of emptying the bag by distributing B symmetric pieces of work over a collection of B server CPU's, and

that of refilling the bag by collecting the results of the B servers' efforts and using them to calculate new work for the bag. It is further restricted to the case where the lengths of the messages used to distribute work and return results are the same. The MIMD-LAN system in question is assumed to be well-behaved, in that delays associated with message-passing are linear in the length of the message. Conceptually, the length restriction, the constraint that the bag size be equal to the number of servers, and the restriction that the work in the bag be symmetric can be lifted without changing the model of computation on which the predictor is based; concretely, the algebraic derivation of the predictor presented earlier makes use of these restrictions.

To predict E using the analytic model, the following must be known about the bag-of-work algorithm and about the hosts participating in the calculations:

- C_e, the strength, in CPU-microseconds, of the pieces of work distributed over the servers;
- $\overline{\rho}$, the average multiprogramming delay, in microseconds, experienced by a server while performing a work item;
- C_r, the strength, in CPU-microseconds, of the calculation that utilizes the results returned by the B servers to refill the bag of work; and,
- $\overline{\phi}$, the average multiprogramming delay, in microseconds, experienced by the driver while refilling the bag.

By trivially modelling the delays associated with the hardware and software interconnecting the driver with the servers as being non-existent, an upper bound on E has been shown to

be

$$E_{\text{upper}} \approx \frac{B(C_e + \overline{\rho}) + C_r + \overline{\phi}}{(B+1)(C_e + \overline{\rho} + C_r + \overline{\phi})}$$

To better model the effects of the hardware and software connecting the driver and servers, the length l of the messages sending work its results must be known. In addition, the following characteristics of the MIMD-LAN framework are required:

- c_s and k_s, the per-byte and per-message overhead on the driver, respectively, associated with sending a piece of work to a server;
- c_r and k_r, the per-byte and per-message overhead on the driver, respectively, associated with receiving the result of a piece of work from a single server;
- c_δ and k_δ, the per byte and per-message overhead, respectively, of sending a work item and receiving the results of performing it that is not attributable to delays on the driver, including, for example, the overhead of the LAN hardware; and,
- k_0, the per-bag overhead of using the framework.

The testing methodology can be used to obtain these characteristics. Once they are known, there are two equations for E that comprise the analytic predictor. The first applies when the servers return work while the driver is still emptying the bag. This condition, termed saturation, occurs when C_e is on the order of the communication delays, in particular, when $\frac{C_e + c_\delta l + k_\delta}{c_s l + k_s} < B - 1$. When saturation occurs,

$$E_{\text{sat}} = \frac{C_r + \overline{\phi} + B(C_e + \overline{\rho})}{(B+1)((c_s + c_r)l + k_s + k_r + k_0 + C_r + \overline{\phi}) + (B^2 - 1)((c_s + c_r)l + k_s + k_r)}$$

When the driver is not saturated,

$$E_{\text{unsat}} = \frac{C_r + \overline{\phi} + B(C_e + \overline{\rho})}{(B+1)((c_s + c_\delta + c_r)l + k_s + k_\delta + k_r + k_0 + C_e + \overline{\rho} + C_r + \overline{\phi}) + (B^2 - 1)(c_r l + k_r)}$$

A qualitative analysis of both equations yields the following intuitively pleasing conclusions

1. Effort spent in improving a particular MIMD-LAN system should focus on reducing per-message delays.

2. To improve efficiency, the servers should be given as much work as possible and the driver should be given as little work as possible.

5.1.3 Testing Methodology

There are two parts of the testing methodology. The first applies to every algorithm tested. The second applies to the MIMD-LAN system and the environment in which it is used.

The methodology tests a MIMD-LAN system using the Trivial, Tunable Algorithm (TTA), which simply loads a value, g, into a hardware register and decrements it to zero. The times to complete a single bag iteration using a given MIMD-LAN system are measured as B, l, and g, are all varied. C_ϵ and $\overline{p}$ are determined as functions of g. The timings are plotted as functions of $C_\epsilon + \overline{p}$, and, using least-squares curve fitting techniques, values for c_s, k_s c_r, k_r, c_δ, k_δ, and k_0 can be estimated.

5.1.4 Results and Experiences Gained

The analytic predictor was checked by applying the testing methodology to a simple, well-understood bag-of-work algorithm. The insight gained in doing so were applied to improving the accuracy of the analytic predictor.

While applying the tests to RCalc-Lisp, a timing methodology, termed the ***map-clock***

method, was developed that, when used on SparcStation 1+'s, has five times less overhead than the the traditional Unix method of utilizing the *gettimeofday()* system call. Though inherently non-portable, other researchers may find its speed useful when working with Sun-4 machines running SunOS-4.1.x.

The experiments also uncovered the erratic effect of the Lisp Garbage Collector on timings of the TTA as implemented in Lisp. These effects were not lessened by granting the memory allocator access to more storage. Modifications in the code implementing the TTA were therefore made so that it would not require the services of the Garbage Collector (GC). Similar modifications to other parts of the Lisp code implementing RCalc-Lisp were made. These efforts were successful to a lesser degree. Rather than modelling GC, samples during which GC occurred were discarded.

The parameters of the TTA used in the testing methodology were varied in two experiments. In the first, C_e and C_r were fixed at 0, l was varied from two to 499, and B ranged from one to sixteen. The saturation effect mentioned previously was first observed during this experiment, and the analytic predictor was modified to account for it. The newly modified predictor therefore forecasted the results of this experiment with great accuracy.

In the second experiment, l was fixed at ten, C_e varied from 9.7 milliseconds to nearly one second, and B again ranged from one to sixteen. The analytic predictor forecasted the results of this experiment with varying degrees of success, typically underestimating the observed efficiency. Usually within ten percent of the observed value, the predicted value was off by at most sixteen percent. The error was shown to behave regularly with respect to C_e and B. The correction in C_e appears to be exponential. Since the Unix process scheduler uses exponential decay when calculating process priorities, it is possible that the

error in C_e reflects an error in modelling multiprogramming delays. The correction in B was shown to be $\frac{B}{cB+k}$, and no plausible explanation for the error in B was given. In any event, the predicted value for E matches the observed value within ten percent for values of $C_e > 0.2$secs.

The analytic predictor was then used to forecast the efficiency of the PSA-TSP as implemented in RCalc-Lisp and applied to a 30 city TSP. B ranged from one to sixteen servers, while l, C_e, and C_r, all properties of the PSA-TSP, were derived experimentally. All predictions were within twenty-one percent of the observed value. Again, the correction in C_e was shown to be exponential, and the correction in B was shown to be of the form $\frac{B}{cB+k}$.

5.2 Conclusions

5.2.1 Garbage Collection is Problematic

Modelling the behavior of a Lisp GC is problematic. Its effect is seen as a large variation in the run times of simple algorithms. Despite much effort–including allocating more memory for the Lisp–GC's effects on timings were neither eliminated nor satisfactorily modelled.

5.2.2 Sometimes, Use Powerful Drivers

Given a heterogeneous collection of workstations involved in a driver/server bag-of-work computation, there are some instances where one of the more powerful workstations should be chosen as the driver, rather than using it as one of the servers. Since the analytic predictor derived in this work does not apply to a heterogeneous collection of workstations,

these cases must be discovered empirically.

5.2.3 Though Useful, RCalc-Lisp May Be Obsolete

RCalc-Lisp can be used to obtain measurable parallelism in a Common Lisp environment. However, other work, such that reported in [PD91], has likely superseded it.

5.2.4 The Empiric Model As MIMD-LAN Test Harness

The tunability of the TTA's granularity allows one to readily explore communication bottlenecks in MIMD-LAN systems. Given the similarity between the nature of the error in the analytic predictor's forecast of the TTA's efficiency and the nature of its error in predicting the PSA-TSP's efficiency, one may tentatively conclude that the testing methodology itself can predict, with some modification, the behavior of bag-of-work algorithms as implemented under a MIMD-LAN system. This generality suggests that the TTA-based testing methodology may be more useful as a benchmark for MIMD-LAN systems than as a predictor for specific algorithms.

5.3 Recommendations for Future Research and Further Improvement

5.3.1 Implement Other Bag-of-Work Algorithms

Other bag-of-work algorithms should be implemented in RCalc-Lisp so that the suitability of both the analytic predictor and the testing methodology may be further explored.

5.3.2 Analyze Other MIMD-LAN Systems

Likewise, the accuracy of the predictor and the tests should be further explored by applying them to other MIMD-LAN systems.

5.3.3 Improve Predictive Power

Clearly, the accuracy of the analytic predictor may be improved. Some indications already exist as to which avenues one might pursue to gain a better understanding of how and why the current model's error varies in both B and C_e. In particular, the distributions of multiprogramming delay introduced by the Unix scheduler should be better modelled.

5.3.4 Adapt to Asymmetric Environments

The predictor and the tests are restricted in their applicability, the analytic predictor more so than the testing methodology. Of the following restrictions on the applicability of the analytic predictor

- the number of work items in the bag when it is full must be equal to the number of server hosts
- the messages sent by the modelled algorithm must all be the same length
- the calculations performed by the servers must all be the same strength, and
- the statistical distributions of the micro-programming delays on each host involved in a computation are the same

the first three could most easily be lifted when expanding its domain.

Changing the testing methodology to lift these restrictions amounts to altering how certain functions are called in RCalc-Lisp. While these changes are trivial to implement, no predictions on their effects are possible without associated changes in the analytic model.

5.3.5 Alter Testing Methodology for Use as MIMD-LAN Benchmark

Work should be done on adapting the testings methodology as a benchmark for MIMD-LAN systems. Towards that goal, some effort should be spent in incorporating a mix of integer and floating point computations into the TTA.

Cited References

Cited References

[ABB+86] M. Acetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for Unix development. In *Proceedings of the 1986 USENIX*, pages 93–112. USENIX, 1986.

[Amd67] G. M. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilitites. *Proc. AFIPS*, 30:483–485, 1967.

[AOC+88] Gregory R. Andrews, Ronald A. Olsson, Michael Coffin, Ewing Elshoff, Kelvin Nilsen, Titus Purdin, and Gregg Townsend. An overview of the sr language an implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, 1988.

[Bal91] H. E. Bal. A comparative study of five parallel programming languages. In *Proceedings of the EurOpen Spring 1991 Conference on Open Distributed Systems*, pages 209–228, May 1991.

[BCJ+90] K. Birman, R. Cooper, T. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood. *The ISIS System Manual.* The ISIS Project, second edition, September 1990.

[BD80] A. Brak and P. Downey. Distributed processor scheduling and user countermeasures. Technical Report 80-12, University of Arizona, May 1980.

[BDG+93] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, and Keith Moore. HeNCE: A heterogeneous network computing environment. Technical Report CS-93-205, University of Tennessee, Knoxville Computer Science Department, Knoxville, Tennessee, August 1993.

[BKT92] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[CFA85] Clement T. Cole, Perry B. Flinn, and Alan B. Atlas. An implementation of an extended file system for unix. In *Proceedings of the Summer 1985 USENIX Conference*, pages 131–149, El Cerrito, CA, June 1985. USENIX Association, USENIX Association.

[CG88] Nicholas Carriero and David Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, November 1988. Describes some parallel programming paradigms, and introduces Linda.

[Che88] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[Che92] Fah-Chun Cheong. *OASIS: An Agent-Oriented Programming Language for Heterogeneous Distributed Environment.* PhD thesis, The University of Michigan, 1992.

[CM87] Clement T. Cole and Craig J. Mathias. The extended backplane. *Unix Review*, 5(10):70–71, 73, 75–82, October 1987.

[CS92] Clemens H. Cap and Volker Strumpen. The PARFORM – a high performance platform for parallel computing in a distributed workstation environment. Technical Report 92.07, University of Zurich Information Institute, June 1992.

[DLM+88] Terence H. Dineen, Paul J. Leach, Nathaniel W. Mishkin, Joseph N. Pato, and Geoffrey L. Wyant. The network computing architecture and system: An environment for developing distributed applications. In *Digest of Papers: COMPCON Spring 88*, pages 296–299, Washington, D.C., 1988. IEEE Computer Society, IEEE Computer Society Press.

[DLS93] R. Diekmann, R. Luling, and J. Simon. Problem independent distributed simulated annealing and its applications. In Dr. Rene V. V. Vidal, editor, *Applied Simulated Annealing*, pages 17–44. Springer-Verlag, 1993.

[DS92] Ashish Deshpande and Martin Schultz. Efficient parallel programming with Linda. *IEEE Computer*, Dunno 1992.

[Gai85] Jason Gait. A distributed process manager with transparent continuation. In *Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 422–429, Silver Springs, MD, USA, May 1985. IEEE Computer Society and Institute de recherche et d'automatique and Information Processing Society of Japan and US Army Ballistic Missile Defense Advanced Technology Center, IEEE Computer Society Press.

[GC92] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):96–107, April 1992.

[GGM+] Timothy J. Gardner, Isabell M. Gerard, Carla R. Mowers, Evi Nemeth, and Robert B. Schnable. DPUP: A distributed processing utilities package.

[Gus88] John L. Gustafson. Re-evaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

[GZ90] A. Goscinski and W. Zhu. The development and performance study of the rhodos reliable datagram protocol. In S. Ramani, H. Shrikumar, and S. V. Raghavan, editors, *Proceedings of the 10th International Conference on Computer Communication*, pages 388–396, New Delhi, India, November 1990. ICCC, Narosa Publishing House.

[Hol75] J. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, Michigan, 1975.

[ISO87] Specification of abstract syntax notation one (ASN.1). International Standard 8824, International Organization for Standardization, December 1987.

[JLHB88] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[Kil88] Stephen L. Kilgore. A distributed hypercube simulator on UNIX workstations in a TCP/IP based network. Master's thesis, University of Tennessee, Knoxville, December 1988.

[KR88] Brain W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall Software Series. Prentice Hall, second edition, 1988.

[Lel90] William Leler. Linda meets Unix. *IEEE Computer*, February 1990.

[LMN88] Klaus-Peter Löhr, Joachim Müller, and Lutz Nentwig. Daphne: Support for distributed applications programming in heterogeneous computer networks. In *Eighth International Conference on Distributed Computing Systems*, pages 63–71, Washington, DC, June 1988. IEEE, IEEE Computer Society Press.

[LP93] Miron Livny and Jim Pruyne. Scheduling PVM on workstation clusters using condor. Invited talk presented at the 1993 PVM Users' Group Meeting, May 1993.

[LS92] Michael Litzkow and Marvin Solomon. Support checkpointing and process migration outside the UNIX kernel. In *Proceedings of the 1992 Winter USENIX Conference*. USENIX, The Usenix Association, January 1992.

[Mas90] W. Anthony Mason. VMTP: A high performance transport protocol. *Connexions: The Interoperability Report*, 4(6):2–10, June 1990.

[MRRT53] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, and A. H. Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physis*, 21(6), June 1953.

[NA93] B. Clifford Neuman and Steve Seger Augart. The Prospero protocol, version 5. Technical report, Information Sciences Institute, University of Southern California, 1993.

[Nel79] Bruce Jay Nelson. *The Remote Procedure Call.* PhD thesis, Carnegie Mellon University, 1979.

[NHSS87] David Notkin, Norman Hutchinson, Jan Sanislo, and Michael Schwartz. Heterogeneous computing environments: Report on the ACM SIGOPS workshop on accommodating heterogeneity. *Communications of the ACM*, 30(2):132–140, February 1987.

[PD91] Mark P. Pearson and Partha Dasgupta. CLiDE: A distributed, symbolic programming system based on large-grained persistent objects. In *Proceedings of the 11th International Converence on Distributed Computing Systems*, May 1991.

[Pet87] John W. Peterson. Distributed computation for computer animation. In *Fourth USENIX Computer Graphics Workshop*, pages 24–36, Berkeley, CA, October 1987. USENIX Association, USENIX Association.

[Pos81] J. Postel. User datagram protocol. RFC 768, Information Sciences Institute, September 1981.

[RH85] Jr. R. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(5):501–538, October 1985.

[RSL92] Martin C. Rinard, Daniel J. Scales, and Monica S. Lam. Heterogeneous parallel programming in jade. *IEEE Computer*, 23:245–256, 1992.

[SDW92] W. Timothy Strayer, Bert J. Depsey, and Alfred C. Weaver. *The Xpress Transfer Protocol.* Addison-Wesley, July 1992.

[Sep93] Douglas J. Sept. The design, implementation and performance of a queue manager for PVM. Master's thesis, The University Of Tennessee, Knoxville, 1993.

[Sun87] Sun Microsystems, Inc. technical staff. XDR: External data representation standard. RFC 1014, Sun Microsystems, Inc., June 1987.

[Sun88a] Sun Microsystems, Inc. Remote procedure call programming guide. In *Network Programming*, chapter 3, page 58. SMI, Mountain View, CA, May 1988.

[Sun88b] Sun Microsystems, Inc. RPC: Remote procedure call protocol specification, version 2. RFC 1057, Sun Microsystems, Inc., June 1988.

[Sun90] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[Sur89] Rao Surapaneni. Parallel best-first search: Optimizing the number of expansions per iteration on slaves. Master's thesis, University of Tennessee, Knoxville, May 1989.

[TvRvS+90] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990.

[VHS84] David Velten, Robert Hinden, and Jack Sax. Reliable data protocol. RFC 908, Bolt, Beranek and Neuman Communications Corporation, July 1984.

[Wat81] R. W. Watson. Timer-based mechanisms in reliable transport protocol connection management. *Computer Networks*, 5(1):47–56, February 1981.

[WL88] Robert A. Whiteside and Jerrold S. Leichter. Using linda for supercomputing on a local area network. In *Supercomputing '88*, pages 192–199, Washington, DC, November 1988. IEEE and ACM, IEEE Computer Society Press.

Appendix: RCalc-Lisp Man Pages

NAME

rcalc-lisp, rcalc:init, rcalc:stop, rcalc:send-msg, rcalc:get-msg, rcalc:make-server, rcalc:make-driver, calc-serve – AKCL specific Lisp interface to the RCalc module

SYNOPSIS

(load "/silver/homes/jepeway/Cuchulainn/2/rcalc-lisp")

(in-package 'rcalc)

(defun make-driver (host-file))

(defun make-server (exec-name &key monolithic))

(defun init (&optional host-file &key calc-side))

(defun stop ())

(defun send-msg (message))

(defun get-msg (&optional secs &optional usecs))

calc-serve [−m] [−n *niceness* **] [−x** *maxcalcs* **]** *calculator*

DESCRIPTION

The *RCalc* package is a set of C routines and a collection of executables that implement a simple bag-of-work model of distributed calculation. The *rcalc-lisp* package provides a Common Lisp interface to to the *RCalc* C routines. This interface is specific to Austin Kyoto Common Lisp, akcl, running on a Unix system.

Using *RCalc*, there are two types of hosts involved in a calculation: calculation ***drivers*** and calculation ***servers***. *drivers* send *messages* requesting that *servers* start a particular calculation. ***servers*** respond to requests with *messages* either containing the result of a calculation or indicating some failure on the *server*'s part. Once *messages* are sent, it may be assumed that they are delivered; no special action need be taken by the user to assure delivery. However, *servers* may choose not to perform calculations, or, as stated, may fail at attempting them. Reasons why either failures or refusals may occur are discussed below.

Under *rcalc-lisp*, the calculation a *driver* requests a *server* to perform is the evaluation of a Lisp s-expression.

Calling *make-driver* establishes the type of the calling host as a *driver* and determines which hosts will be involved in calculations as *servers*. The *host-file* parameter is a string that names a file of newline-separated hostnames. The default value of *host-file* is "**calc-hosts**". *make-driver* will return the number of hosts in *host-file*. Alternatively, one can call *init*. The *calc-side* key is an atom, either **DRIVER** or **SERVER**, specifying whether the host calling *init* is a *driver* or a *server*. *calc-side* is not optional; it is an error if it is not present. Ordinarily, *init* isn't called. To create a *server* host as they are described herein, see the description of *make-server*, below.

send-msg and *get-msg* send and receieve *messages*. A *message* is a list of the form *(host type id body)*. *host* is an integer index into the *host-file*. 0 designates the first host named in the file; 1, the second; etc. When sending messages, *host* specifies the destination host. When receiving them, it specifies the host that sent the *message*. *type* is an atom, one of **NULL-MSG**, **S-EXP**, **VALUE**, **FAILED**, **RESET**, or **DIE**. It defines the semantics of the remainder of the *message*. *id* is an integer that, for some messages, uniquely identifies the *body* of the message. Its primary use is to pair *messages* of type **S-EXP** with those of type **VALUE** or **FAILED**. *body* is usually an arbitrary s-expression.

A **NULL-MSG** contains no useful information. It should never be explicitly sent using ***send-msg***, and it will never be returned by *get-msg*. The interpretation of a **NULL-MSG**'s *id* and *body* is undefined. A user of *rcalc-lisp* should never see a **NULL-MSG**; it is included only for the sake of completeness.

An **S-EXP** *message* is a request sent from a *driver* to a *server* that the *server* evaluate the *body* of the message. The *server* responds with either a **VALUE** or a **FAILED** *message*. *id* provides a mapping between **S-EXPs** and their responses; therefore, each (*host*, *id*) pair should be unique in the set of **S-EXPs** for which there are outstanding responses. *id*s must be in the range **0** to **MAXINT−1** as defined in the file **/usr/include/values.h**. *rcalc-lisp* places no further restrictions on *id*s, so care must be taken that the user

RCALC-LISP(LOCAL)

can associate **S-EXP**s with the appropriate **VALUE**s or **FAILED**s. *servers* never send **S-EXP** *messages*.

A **VALUE** *message* is returned from a *server* in response to an **S-EXP** *message* from a *driver*. *id* is the same as that in the **S-EXP** *message* to which the *server* responds. *body* is the s-expression the *server* produced by evaluating the *body* of the **S-EXP** *message*. Only the first value of a multiple-value will be placed in the **VALUE** *message*. *drivers* should never send a **VALUE** *message*.

A **FAILED** *message* indicates to a *driver* that a *server* failed to evaluate an s-expression. Among the reasons a *server* might respond with a **FAILED** *message* are

resource exhaustion –
: the *server* may be running the maximum number of calculations allowed to it

a bad s-expression –
: evaluating the *body* of the **S-EXP** may have resulted in an error, or may have produced a value that cannot be read by **akcl** (e.g., something containing "#<")

an unreachable host –
: the *driver* host may not be able to contact the requested *server* host.

drivers should never send **FAILED** *messages*.

A *driver* sends a **RESET** *message* to a *server* telling it to abort all outstanding evaluations and return to its initial state (see the discussion of calc-serve, below, for futher information about state). Both *id* and *body* must be present, though they are essentially unused. *servers* never send **RESET** *messages*.

A **DIE** *message* tells the *server* to report all outstanding evaluations as **FAILED** and to terminate. The *server* host will no longer be reachable by subsequent calls to *send-msg*. *body* and *id* are treated as for **RESET** *messages*. *servers* never send **DIE** *message s*.

To create a *server* reachable by *rcalc-lisp*, call *make-server*. After starting **akcl**, load the Lisp code to be resident when the *server* evaluates *message* bodies, then **load** *rcalc-lisp*, and, finally, call *make-server*. *make-server* takes two arguments. *exec-name* is the filename of the executable *make-server* will produce. *monolithic* is a boolean key that must be present. It determines the *server*'s state between *messages*. If *monolithic* is non-**NIL**, the *server* will maintain its state from *message* to *message*. If *monolithic* is **NIL**, each *message* is evaluated in the environment extant at the time *make-server* is called. Note that **akcl** terminates once it saves its state, so calling *make-server* successfully should return you to the shell.

The program **calc-serve** makes executables built by *make-server* reachable by *drivers* using *rcalc-lisp*; running **calc-serve** on a host named in the *host-file* argument of *init* turns that host into a *server*. *calculator* is the name of the executable file created by *make-server*.

OPTIONS

-m
: The server is monolithic.

-n *niceness*
: *server* processes will run at a *nice* value of *niceness*. The default *nice* value if **-n** is not given is 0.

-x *maxcalcs*
: The maximum number of unfinished evaluations the *server* will allow is *maxcalcs*. The *server* returns a **FAILED** *message* in response to any **S-EXP** sent it once this limit is reached. The default value of *maxcalcs* is 5.

EXAMPLES

For the following examples, two hosts named "server-host" and "driver-host" will use *rcalc-lisp*. Text in **bold** is produced by the system. Text in roman is typed by the user.

To create a *server* loaded up with code from the file named "soweto.lsp":

```
server-host% akcl
>(load "soweto.lsp")
Loading soweto.lsp
Finished loading soweto.lsp
T

>(load "/silver/homes/jepeway/Cuchulainn/2/rcalc-lisp")
Loading /silver/homes/jepeway/Cuchulainn/2/rcalc-lisp.lsp
```

```
Finished loading /silver/homes/jepeway/Cuchulainn/2/rcalc-lisp.lsp
T

>(make-server "beat" :monolithic t)
server-host%
```

To start the *server*:

```
server-host% echo driver-host > calc-hosts
server-host% /silver/homes/jepeway/Cuchulainn/2/calc-serve -m beat &
server-host%
```

To start the *driver* and ask the *server* to add 1 to 99:

```
driver-host% echo server-host > calc-hosts
driver-host% akcl
>(load "/silver/homes/jepeway/Cuchulainn/2/rcalc-lisp")
Loading /silver/homes/jepeway/Cuchulainn/2/rcalc-lisp.lsp
Finished loading /silver/homes/jepeway/Cuchulainn/2/rcalc-lisp.lsp
T

>(make-driver "calc-hosts" nil)
1

>(send-msg '(0 s-exp 0 (1+ 99)))
NIL

>(get-msg)
(0 VALUE 1 100)

>(bye)
Bye.
driver-host%
```

FILES

/silver/homes/jepeway/Cuchulainn/2	parent directory of *rcalc-lisp* files
rcalc-lisp.lsp	Austin Kyoto Common Lisp code to load *rcalc-lisp*
./calc-hosts	default host file

SEE ALSO

Austin Kyoto Common Lisp User's Guide, *nice(1)*

DIAGNOSTICS

get-msg **returns NIL when there are no** *messages* **available.** ***send-msg*** **and** ***stop*** **always return NIL.** ***init*** **and** ***make-driver*** **return the number of hosts found in** *host-file*. ***make-server*** **never returns.**

BUGS

For the moment, the only operating system under which *rcalc-lisp* **runs is SunOS-4.1.3.**

Ordered delivery of *messages* is not guaranteed. Further, unordered delivery is a common occurrence.

Note that *messages* are delivered to *driver* hosts uninterpreted. This means that *drivers* can send *messages* to other *drivers* and ignore all of the semantics mentioned above. Consider this a feature to be exploited with care.

RCalc binds statically to its network connection, currently UDP port 23532. If there are other processes bound to that connection, *RCalc* will not start. If there are other processes using that port but not bound to it, *RCalc* may see alien packets or may never see packets sent to it.

Errors occurring in the C library to which *rcalc-lisp* provides an interface are often not propagated back to akcl, though they are reported on *stderr*. This often results in certain system resources' not being fully

released. To do a thorough clean-up after seeing an unrecoverable error message printed on the terminal, kill all processes named "r-ocard" and/or "w-ocard ."

Vita

Born in Chicago, Illinois on 7 August 1963, Chris Markland Jepeway has tramped around the country pursuing sundry educational whims. Given a firm grasp of the basics by the public schools of Orangeburg, South Carolina, he graduated from The Baylor School located in Chattanooga, Tennessee in 1981. He received a BS in Electrical Engineering from Rice University at the May 1985 commencement exercises in Houston, Texas. After a brief stay in southern California, where what he learned can't be put into writing, he moved to Knoxville, Tennessee to attend Graduate School. By the time you can obtain this document from a library, he hopes to have earned an MS in Computer Science from the University of Tennessee, Knoxville. What he's doing while you're reading these words is anyone's guess. If you're the gambling type, though, bet that he's no longer in school.