Linda on Distributed Memory Multiprocessors

Robert D. Bjornson

YALEU/DCS/RR-931
November 1992

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Linda on Distributed Memory Multiprocessors

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by

Robert D. Bjornson
May 1993

Abstract

Linda on Distributed Memory Multiprocessors

Robert D. Bjornson
Yale University
May 1993

The coordination language Linda [Gel85] is a convenient and powerful model for parallel and distributed computing. By permitting the expression of parallel algorithms in an architecture-independent manner, it supports truly portable programming [BCGL88]. In order to be practical, Linda must be efficient. Previous work [Car87], [Lei89] demonstrated that Linda can be made efficient on shared memory multiprocessors, bus-based distributed-memory multiprocessors, and local area networks. In this dissertation, we show that Linda can be made efficient on scalable distributed-memory multiprocessors.

We present a design for implementing Linda's Tuple Space on such machines, and claim that the design results in an efficient implementation. As an existence proof, we characterize the performance of the design as implemented on an Intel iPSC/2 hypercube. We go on to describe a number of runtime optimizations, and investigate their effect on the performance of several synthetic programs that stress communication, as well as eight non-trivial applications.

Finally, we discuss the extensibility of the design to massively parallel machines.

Copyright © 1993 by Robert D. Bjornson
All Rights Reserved

Acknowledgments

The preparation and writing of this dissertation has been a long process, and there are several people I need to thank. David Greenberg, Mike Factor, and Rick Mohr, apart from making my time as a graduate student more enjoyable, also contributed many hours spent in front of a white board. Ed Segall used the system and provided useful data on its performance. Ken Musgrave provided his erosion program.

I owe an enormous debt of gratitude to Nick Carriero. In addition to his guidance along the way, without which I would have floundered, he also volunteered for the onerous task of proof-reading the first version.

My committee, Drs. David Gelernter, Stan Eisenstat, and Lothar Borrman provided useful comments. I would like to especially thank Dr. Eisenstat for his careful reading of this dissertation. Finally, I would like to thank the people at Scientific Computing Associates for their patience.

Richard Carpenter, my undergraduate advisor, started me along the path to this dissertation through his encouragement and enthusiasm.

On a more personal note, I would like to thank Scott for sharing an office with me through this process and for being a good friend. I'd also like to thank Linda for her encouragement and support. Finally, Frau Hahn, ich hab's endlich geschafft!

This work was supported in part by an IBM scholarship and by ONR grant N0014-86-K-0310 and NSF grants DCR-86019620 and DCR-8657617.

This dissertation is dedicated to my mother, Verlene Bjornson.

Contents

1 Introduction **1**

2 Linda **5**
- 2.1 Introduction 5
- 2.2 The language 5
  - 2.2.1 Tuples 5
  - 2.2.2 Tuple Space 6
  - 2.2.3 The operations 6
- 2.3 Compile-time analysis 8
  - 2.3.1 Partitioning 8
  - 2.3.2 Classification 9
- 2.4 Other Linda systems 11
  - 2.4.1 Positive broadcast 11
  - 2.4.2 Negative broadcast 12
  - 2.4.3 Bi-polar multicast 13
- 2.5 Summary 13

3 Designing a Linda system for Distributed-Memory Multiprocessors **15**
- 3.1 A new kernel design: Distributed Hashing 15
- 3.2 Tuple Space servers 21
- 3.3 Request types 22
- 3.4 Mapping tuples and templates to rendezvous nodes 23
  - 3.4.1 Description of the problem 23
  - 3.4.2 Requirements for hashing 24
  - 3.4.3 Experiment 27
  - 3.4.4 Final remarks about hashing 31
- 3.5 The low level structure of Tuple Space 31
  - 3.5.1 Tuple data structures 31
  - 3.5.2 Organization of tuples into Tuple Space 36
- 3.6 Summary 41

4 Basic performance measures **43**
- 4.1 Single process communication latencies 43
- 4.2 Single process communication loads 45

4.3 Dissection of a Linda communication. 46
4.4 Synthetic programs 48
4.4.1 Ping-pong 49
4.4.2 Token ring 49
4.4.3 Barrier 49
4.4.4 BLP 52
4.5 Performance of Eval 57
4.6 Summary 64

5 Optimizations **65**
5.1 Introduction 65
5.2 Tuple rehashing 65
5.2.1 Cost of rehashing 67
5.2.2 Performance 69
5.2.3 An improved rehashing analysis 73
5.3 In/Out collapse 75
5.3.1 Performance 77
5.4 Memory caching 77
5.4.1 Previous approaches 80
5.4.2 Our approach 80
5.4.3 Performance 81
5.5 Local data caching 82
5.5.1 Performance 83
5.6 Tuple broadcast 83
5.6.1 A digression concerning correctness 84
5.6.2 Performance 85
5.7 Randomized tuple bag 86
5.7.1 Randomized queue set representation 86
5.7.2 How much does the randomized method cost? 87
5.7.3 Performance 89
5.8 Interactions between optimizations 91
5.8.1 Memory caching 93
5.8.2 Local data caching 94
5.8.3 Tuple broadcast 94
5.8.4 Random queue 94
5.8.5 Inout collapse 95
5.8.6 Rehashing 95
5.9 A peek ahead 95

6 Applications **99**
6.1 Introduction 99
6.2 General discussion of the experiments 101
6.3 DNA sequence comparison 104
6.3.1 Granularity 105
6.3.2 Performance 106

6.3.3 Summary . . . 110
6.4 Block matrix multiply . . . 113
6.4.1 Granularity . . . 113
6.4.2 Performance . . . 113
6.4.3 Summary . . . 115
6.5 Clumped matrix multiply . . . 119
6.5.1 Granularity . . . 119
6.5.2 Performance . . . 119
6.5.3 Summary . . . 120
6.6 2D fast Fourier transform . . . 124
6.6.1 Granularity . . . 125
6.6.2 Performance . . . 125
6.6.3 Summary . . . 126
6.7 LU decomposition . . . 130
6.7.1 Granularity . . . 131
6.7.2 Performance . . . 131
6.8 Erosion . . . 144
6.8.1 Granularity . . . 144
6.8.2 Performance . . . 146
6.9 Intelligent Cardiovascular Monitor . . . 148
6.10 Freewake . . . 151
6.10.1 Granularity . . . 151
6.10.2 Performance . . . 151
6.11 Overview of application granularity and efficiency . . . 154
6.12 Summary . . . 155

7 Massive Parallelism **159**
7.1 Introduction . . . 159
7.2 Increasing communication latency . . . 159
7.3 Memory utilization . . . 159
7.4 Node CPU contention . . . 161
7.5 Link contention . . . 161
7.6 Contention vs. Latency . . . 162
7.7 Alternate hybrid and mess set representation . . . 162
7.8 Summary . . . 164

8 Related Work **165**
8.1 Introduction . . . 165
8.2 Virtual shared memory . . . 165
8.3 Object-oriented models . . . 167
8.4 Semi-automatic data parallelism . . . 168
8.5 Functional languages . . . 168
8.6 Logic Programming . . . 170
8.7 Portable message passing . . . 170
8.8 Other Linda projects . . . 171

9 Conclusions **173**

A An overview of the iPSC/2 **175**
A.1 Introduction 175
A.2 Hardware 175
A.3 Software 176
A.4 Performance 177

B Application Source Code **181**
B.1 DNA 182
B.2 Block Matrix Multiply 191
B.3 Clumped Matrix Multiply 196
B.4 LU Decomposition 200

C Benchmarks **205**

D Implementation of Eval **213**

E Degenerate hybrid sets **217**

F Performance Data for LU and 2DFFT **219**

G Glossary **221**

List of Figures

1.1 Spectrum of connectivity for MIMD multiprocessors. 2

3.1 Schematic of Distributed Hash Design for Tuple Space. 16
3.2 A comparison of total bandwidth requirements for small (100 byte) tuples . 19
3.3 A comparison of total bandwidth requirements for large (10000 byte) tuples 20
3.4 Hashing experiment: functions 28
3.5 Hashing experiment: input domains 29
3.6 ptp data structure. 34
3.7 ptp using absolute vs. relative pointers to aggregates. 35
3.8 ST data structure. 37
3.9 A bucket in Tuple Space. 39
3.10 The queue, hybrid, and mess set Tuple Space structures. 40
3.11 The Hash set Tuple Space structure. 42

4.1 Out broken down 46
4.2 In broken down 47
4.3 Ping_pong.cl 50
4.4 Ping_pong.c 51
4.5 Timeline for Ping_pong. 52
4.6 Ring.cl 53
4.7 Ring.c 54
4.8 Barrier.cl 55
4.9 Barrier Synchronization: Cost per synch versus machine size 56
4.10 Blp.cl 58
4.11 BLP: total execution times versus task size 59
4.12 BLP: communication overhead versus task size 60
4.13 BLP: efficiency versus task size 61
4.14 Blpeval.cl 62
4.15 BLP using evals. 63

5.1 Example forwarding chain before notification. 68
5.2 Example forwarding chain after notification. 68
5.3 Multitoken-ring.cl 72
5.4 Barrier Synchronization: Cost per sync. versus machine size 74
5.5 stblp.cl 78

5.6 Single tuple BLP, communication overhead per task for in/out vs. inout 79
5.7 Expected probes using a randomized queue 90
5.8 Overhead for Randomized vs. Non-randomized method 91
5.9 Modified BLP program 92
5.10 Overhead for Randomized vs. Non-randomized method 93
5.11 Synopsis of interactions between optimizations. 94
5.12 A preview of the performance of the optimizations 96

6.1 Example graph 102
6.2 Schematic of DNA hybrid algorithm 105
6.3 DNA Database Search (1000x40 blocking) 107
6.4 DNA Database Search (40x40 blocking) 108
6.5 DNA Database Search (3x500 blocking) 109
6.6 DNA Database Search. Single large comparison 111
6.7 DNA Database Search (full DB, 1000 by 40 blocking) 112
6.8 Block matrix multiplication (120x120) Total time 114
6.9 Block matrix multiplication (120x120) Net time 116
6.10 Block matrix multiplication (240x240) Total time 117
6.11 Block matrix multiplication (240x240) Net time 118
6.12 Clumped Matrix Mult. (400x400) 121
6.13 Clumped Matrix Mult. (240x240) 122
6.14 Clumped Matrix Mult. (400x400) Varied clumping 123
6.15 Schematic of 2D-FFT global transpose. 124
6.16 2D-FFT, dimension 256x256 127
6.17 2D-FFT, dimension 1024x1024 128
6.18 2D-FFT, Native versus Linda 129
6.19 LU decomposition (256x256) 132
6.20 LU decomposition (400x400) 133
6.21 LU decomposition (400x400), including message passing 134
6.22 LU decomposition (128x128), Linda vs. Crystal 136
6.23 LU Decomposition, Native versus Linda 137
6.24 LU decomposition model 138
6.25 LU decomposition model 100x100) 139
6.26 LU decomposition model 256x256) 140
6.27 LU decomposition model 400x400) 141
6.28 LU decomposition model 1000x1000) 142
6.29 LU decomposition model 10000x10000) 143
6.30 communication pattern for erosion simulation. 145
6.31 Erosion simulation (200x200) 147
6.32 Intensive Care Monitor 150
6.33 Freewake on iPSC/2. 2048 wake elements 152
6.34 Freewake on Intel i860 hypercube. 8192 wake elements 153
6.35 Efficiency vs. granularity for applications. 156

A.1 Message latency for various message sizes on the iPSC/2 178

A.2 Message latency for various path lengths on the iPSC/2 179

C.1 Hybrid DNA Database Search. 206
C.2 Block Matrix Multiply. 207
C.3 Clumped Matrix Multiply (400x400). 208
C.4 Clumped Matrix Multiply (240x240). 209
C.5 2D-FFT 210
C.6 LU decomposition. 211

D.1 Eval Server 215

Chapter 1

Introduction

Over the past decade, parallel computers have developed from exotic research machines to commercial products available from a number of vendors in a bewildering assortment of architectures and programming environments. Looking across the field, one sees multiple vs. single instruction streams, closely vs. loosely coupled connection schemes, shared vs. distributed memory, etc. Ironically, all of these architectures share one characteristic: idiosyncratic, machine-specific programming environments.

Each machine supports its own method of realizing parallelism: message passing, shared data pages with locks, automatic compilation tools, implicit data parallelism, etc. The programs produced using explicit methods are non-portable, and usually difficult to write and debug, while implicit parallel models often yield disappointing performance. Because of this, many researchers have proposed new, high-level parallel programming models, promising portability, programming ease, and code efficiency[BST89]. Linda[1] is one such model.

Why Linda? The case has been argued extensively elsewhere [BCGL88], [CG89]. We see the following characteristics as decisive:

Familiarity. The vast majority of coding remains in the programmer's familiar idiom. Many high-level parallel programming models present the programmer with a new and unfamiliar environment; examples include functional, logic, or object-oriented programming. We don't argue that such models are necessarily bad, only that programmers should not have to completely change the way they think, and throw away their old code, simply in order to use parallelism.

Parsimony. Only six new operations need to be understood. In comparison, a recent release of the operating system for the Intel iPSC hypercubes provides 42 system calls for managing processes and passing messages[Int90]. One commercial portable message-passing product, *Express* from ParaSoft[Par90], provides more than 150.

Decoupling. Linda processes exist decoupled from one another in time and space. In comparison, most message-based models require that the sender and recipient co-

[1]Registered trademark of Scientific Computing Associates, Inc., of New Haven.

exist in time. Examples include Concurrent Smalltalk[YT86], Remote Procedure Call[Nel81], and CSP/Occam[Hoa78][Int84].

Portability. Linda programs are portable across the gamut of multiple instruction stream computers.

Linda's elegance notwithstanding, programmers are concerned (quite reasonably) with the efficiency of their programs. Indeed, naive implementations of Linda can be quite inefficient. Considerable effort has gone into developing compilation techniques that reduce the complexity of Linda operations at run-time. Using these techniques, Carriero [Car87] demonstrated Linda's efficiency on two different machines: a bus-based, shared memory multiprocessor[2], and a bus-based, distributed memory multiprocessor[3].

These two machines represent the near and middle ground, respectively, of the spectrum of connectivity for multiple-instruction-stream multiprocessors (Figure 1.1). This dissertation completes the proof of Linda's practicality by addressing the far end of the spectrum: Distributed Memory Multiprocessors connected by some sort of network. The challenge is greatest here since communication costs are the highest relative to computation speeds. In return, however, the rewards of Linda on such systems are also potentially the greatest, since these machines are generally agreed to be the most extensible to very large machines[FF88]. A prime example of these machines is the hypercube.

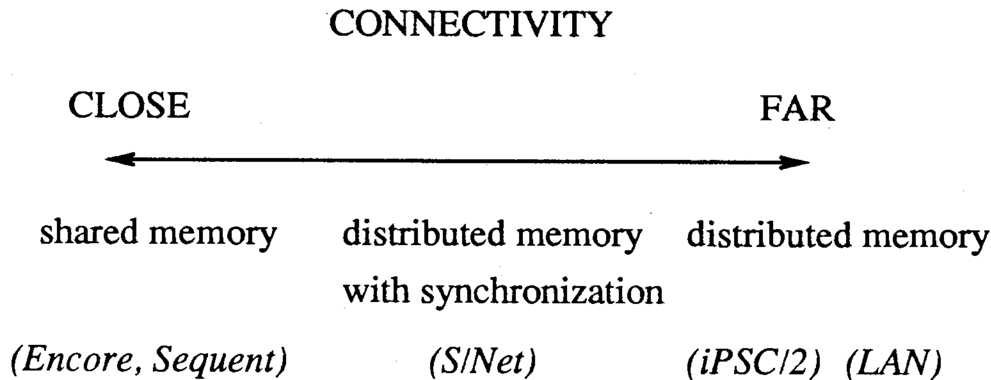

Figure 1.1: Spectrum of connectivity for MIMD multiprocessors.

In this dissertation, we show that Linda can be efficiently implemented on such machines.[4] The rest of the dissertation is organized as follows:

Linda Linda is introduced as a coordination language. We examine a compilation analysis method developed by Carriero as part of his dissertation. Finally, we look at some other Linda runtime systems, in order to judge their applicability to Distributed Memory Multiprocessors.

[2]The Encore Multimax.

[3]The S/Net, built by Sidhur Ahuja of Bell Labs, Holmdel NJ.[Ahu83]

[4]The thesis of the work.

Design of a Linda System for Distributed Memory Multiprocessors We present a general design strategy for this class of machines.

Basic Performance We determine the costs of various Linda operations using an implementation on the Intel iPSC/2. We break down the cost of an individual operation into its components. Finally, we look at the performance of several synthetic programs designed to stress communication.

Optimizations We describe a number of optimizations that improve upon the performance described in the previous chapter. We quantify the effect of these optimizations on the cost of basic operations. We also look at their effect on the performance of the synthetic programs discussed in the previous chapter.

Applications We present a number of Linda applications, and examine their performance, both for different problem sizes and under varying degrees of optimization.

Massive Parallelism We examine issues of large scale parallel machines, and predict how the Linda design we developed would perform on such a machine. We identify problem areas and discuss possible solutions.

Related Work We survey a number of alternative approaches to portable programming models for distributed memory multiprocessors.

Major results of this work:

- A new Tuple Space design for Distributed Memory Multiprocessors.
- A number of optimizations upon the basic design.
- Using the optimized design, much more application experience, on larger machines, than for any previous Linda implementation.

Interestingly enough, the methods developed proved themselves to be extremely effective on another, often overlooked, class of easily extensible parallel computer: networks of workstations. Such networks can have an aggregate performance well in excess of any "supercomputer", yet this potential is generally wasted. The methods developed in this work resulted in a network Linda that performs very well.

Chapter 2

Linda

2.1 Introduction

In this chapter we will set the stage for discussing Distributed Memory Multiprocessor Linda (DMM-Linda). First, we present an overview of the Linda language, defining the fundamental Linda objects *tuples* and *Tuple Space* and the operations that manipulate them. We then turn to the compile-time analysis developed by Nicholas Carriero. This analysis is important because it motivates several concepts that will be used throughout this dissertation. Finally, we will look at some other Linda implementations.

2.2 The language

Linda [BCGL88] is a set of six operations forming a *coordination language*. Adding Linda to some base language creates a Linda dialect of that language, e.g. Fortran-Linda or C-Linda. As a coordination language, Linda provides an elegant, machine- and language-independent method of expressing interprocess communication and synchronization, leaving to the base language the more mundane chores such as arithmetic, I/O, etc. We shall see in Section 2.3 that the six Linda operations are not mere system calls; considerable compilation effort is expended to render their runtime performance as efficient as possible, in part by understanding each operation in the context of the others. The six operations manipulate *tuples* in *Tuple Space*. We turn first to these two fundamental Linda concepts, then see how the Linda operations manipulate them. This dissertation is mostly concerned with C-Linda, although the runtime kernel is largely language independent, and efforts are proceeding elsewhere to produce a Fortran-Linda compiler that will make use of our DMM-Linda kernel.[1]

2.2.1 Tuples

A tuple[2] is the fundamental Linda data object. It consists of an ordered collection of typed data objects or place holders, called *elements*. In general, the types of elements allowed are

[1] Scientific Computing Associates, Inc., of New Haven.

[2] Pronounced "*too'*-ple"

those allowed by the base language, although some restrictions may apply.[3] By convention, the first field of a tuple is usually a string. This helps both programmer and compiler pick out related operations, but is not required.

Elements having values are called *actuals*; in C they correspond to r-values. Actuals may be constants, variables, or expressions. Elements acting as place holders are indicated by a question mark; they correspond to l-values. Such elements are called *formals*, because they are similar to formal procedure parameters. Formals may also be type names rather than variables, in which case no data copying will occur. Such fields are termed *anonymous*.

```
("test", 3, i, ? j, ? int)
```

In this example, the first three elements are actuals. The fourth and fifth are formals, as indicated by the question marks. Furthermore, the fifth is anonymous. A tuple may have any number of elements, although most current implementations restrict the number of elements to 16.

2.2.2 Tuple Space

Tuple Space is where tuples reside. It is an associative, logically-shared tuple memory. Tuple Space is associative because tuples are referenced via their contents rather than by address. It is logically shared because any process can reference any tuple, regardless of where (or how) the tuple is actually stored. Of course, this doesn't describe the underlying hardware; Tuple Space might be implemented on a shared memory machine, a tightly-coupled distributed memory machine, or across a network. The job of the implementer is to provide the illusion of a shared, associative tuple memory, and do it efficiently.

2.2.3 The operations

Of the six Linda operations, two produce tuples and four consume them.

Generative operations (*Out* and *Eval*)

`Out` and `eval` produce tuples and insert them into Tuple Space. They differ in the way their elements are evaluated. `Out` evaluates the elements in an unspecified order and dumps the tuple to Tuple Space once evaluation is complete. `Eval` dumps the unevaluated tuple (called an active tuple) and creates new threads of control, one to evaluate each element. When all elements have been evaluated, the active tuple quiesces to a passive tuple, indistinguishable from one created by `out`.[4] For example:

[3]For instance, in C-Linda, pointer types are not allowed as elements. Having received the pointer from Tuple Space, a process might reasonably expect to be able to dereference it. Allowing this would necessitate also communicating the data pointed to, since a pointer is generally meaningless outside of its home address space. Unfortunately, in C, since pointers may be nested, or concealed in unions, and also because C is so cavalier about distinguishing between pointers and arrays, it is decidedly non-trivial to determine just what data might eventually be referenced via that pointer, and thus, what to send. In C-Linda, pointers are allowed if followed by a colon and optional length specifier. Such a field specifies an array of the type pointed to, of length elements.

[4]Presently, although active tuples are considered to be present in Tuple Space, only passive tuples are available for matching, since C contains no type for "active" values, e.g. "active int". One way to provide an

```
out("foo", i, foo(j))
```

The process performing the out evaluates "foo", i, j, and foo(j), packs the results into a tuple, and dumps it to Tuple Space.

```
eval("foo", i, foo(j))
```

The process performing the `eval` dumps the unevaluated tuple to Tuple Space, creates three new threads of control[5], and continues with the statement following the `eval`. The threads will each evaluate their element, put the result in the tuple, and terminate. When all threads working on a given active tuple have terminated, the tuple becomes passive.

An `eval`ed function operates in an environment that consists of the values of all statically initialized variables and any parameters passed to it when invoked. The function does *not* see the environment of the process that created (`eval`ed) it. Eval is actually implemented by transforming it into a series of `in`s and `out`s. This transformation is described in Appendix D.

Synchronous extraction operations (*In* and *Rd*)

These two operations take a template and compare it element by element against tuples in Tuple Space. A template looks much like a tuple, but is used to extract tuples from Tuple Space. In order for a template to match a tuple, the tuple and template must:

1. Agree on the number of elements.
2. Agree on the types of corresponding elements.
3. Agree on the values of corresponding actuals.
4. Have no corresponding formals.

For instance:

```
int i=3;
double k;
in("foo", i, ? k)
```

would match (assuming 3 is of type `int` and 4.3 is of type `double`)

```
("foo", 3, 4.3)
```

but would fail to match

```
("foo", 3)           (rule 1)
("foo", 3.0, 4.3)    (rule 2)
("foo", 2, 4.3)      (rule 3)
("foo", 3, ? i)      (rule 4)
```

active type would be to treat it like a Multilisp *future* [RHH85], i.e. a process would block when attempting to use the value. This would also provide a way of suspending a process by `in`ing the `eval` containing it.

[5] Actually, most implementations will "inline" evaluation of simple elements, such as "foo" and i, in order to avoid the overhead involved with process creation.

When a match is found, any formals present in the template receive the corresponding value from the tuple. In the above example, *k* is set to 4.3 as a side effect of the `in`. Following a match, `in` withdraws the tuple from Tuple Space, whereas `rd`[6] simply reads the data, leaving the tuple in Tuple Space. If an `in` or `rd` fails to find a match, the process blocks until a matching tuple appears in Tuple Space.

Asynchronous extraction operations (Inp and Rdp)

These two operations are non-blocking predicate forms corresponding to `in` and `rd`. If `inp` or `rdp` find a match they return true in addition to performing the actions of `in` or `rd`. If no match is found they immediately return false.[7]

2.3 Compile-time analysis

Before we look at the the Linda run-time system, we need to understand the Linda compile-time analysis. We will only present an overview of the compilation process; for a detailed examination of the issues, see [Car87].

Analysis of Linda programs consists of two main phases: *partitioning* and *classification*. The goal of partitioning is to separate all Linda operations into disjoint *sets*, such that no tuple from one set can possibly match a template in any other set. This greatly reduces the search space for a given template at runtime. After partitioning is complete, the classifier takes each set in turn and chooses an efficient data structure and storage paradigm to represent the operations found in that set. This further reduces the number of matches, and makes those matches that are still required more efficient.

2.3.1 Partitioning

Organizing Linda operations into sets involves performing a pre-match on all the operations found in the program. This pre-match is semantically very similar to matching tuples and templates; the main difference being that values of actuals are not available during compilation. Compare with the rules presented on Page 7. We will still require that they:

1. Agree on the number of elements.

2. Agree on the types of corresponding elements.

---

[6] Rd is pronounced "reed"

[7] Inp and rdp are somewhat problematic. Consider the following program fragment:

Process 1	Process 2
`out("A")`	`in("B")`
`out("B")`	`inp("A")`

Assuming that no other Linda operations interfere, can we predict the success or failure of `inp("A")`? The naive programmer might think so, believing that an ordering existed such that `out("A")` must precede `inp("A")`. This is not the case; out is asynchronous; in particular, it does not guarantee that a tuple is available in Tuple Space by the time it returns. Thus, in this case, the result of `inp("A")` is unpredictable. Leichter [Lei89] discusses this and other problems of inp and rdp. Because of these difficulties, inp and rdp may not appear in future versions of Linda.

3. Agree on the values of corresponding *constants*.
4. Have no corresponding formals.

However, we will assume that corresponding non-constant actuals match. Thus, the pre-match is strictly weaker than true tuple matching.

The partitioning proceeds as follows. First, an initial partitioning is performed on the basis of the number and types of elements. Next, each of these partitions is examined using the pre-match to compare all **outs** against all **ins**. If the pre-match succeeds, the **out** is added to the **in**'s list of matchables, and vice versa. After all partitions have been processed, we begin labeling the Linda operations with set identifiers. We choose some unlabeled operation and label it. Then, we recursively label all the operations in its list of matchables. When no more new operations can be marked, we choose some new unlabeled operation and begin again with a different label. When all operations have been labeled, we stop. A *set* consists of all operations with the same label. This corresponds to finding the equivalence classes in the transitive closure of the pre-match relation.

Because the pre-match is strictly weaker than true matching, we know that if two operations fail to pre-match, the tuple and template they produce cannot possibly match. Thus, we have restricted the search space at run-time to just those tuples in the same set as the template.

2.3.2 Classification

Once Linda operations have been organized into disjoint sets, we need to choose a storage paradigm for each set, appropriate to the types of operations actually found in the set. Carriero developed five different paradigms. Care must be taken to distinguish the classification, which is an abstract property of the set, from an actual run-time storage regimen, which may vary for different machine implementations.

Queue paradigm

The queue paradigm is chosen if no runtime tuple-matching is ever required. This occurs when no possible **out-in** pair within the set has corresponding actuals (excepting constants). Thus, any tuple is guaranteed to match any template. True to their name, queue sets are usually stored in a LIFO queue (i.e. a stack).[8]

Example:

```
out("test", i)
in ("test", ? j)
```

Counting semaphore paradigm

The semaphore paradigm is actually a special (degenerate) case of the queue paradigm. In such a set, not only is no matching required, but no data needs copying either. A good representation of this set is a simple counting semaphore.

Example:

[8] We eventually changed to FIFO queues for templates only; see the footnote in Section 4.4.4

```
out("test")
in ("test")
```

Hash paradigm

The hash paradigm is chosen if some field is always actual (but not always constant) across all operations in the set. Because this field always has a value, we can use it as a *key*. Hash sets are typically stored in one large hash table, using the key and set id as the input to the hash function. At present, if more than one field could be used as a key, we (arbitrarily) choose the left-most. [9] In the following example, the second field would be chosen as the key.

Example:

```
out("test", i)
in ("test", j)
```

Hybrid paradigm

The hybrid paradigm is chosen if none of the above paradigms apply, and furthermore, some field is always actual across all **outs** in the set, while that same field is sometimes actual and sometimes formal across the **ins**. We can use that field as a *partial key*. All tuples will have a key. Some templates will have a key as well, which we can use to restrict matching. Others will have no key, and we may need to search the entire set for a match. Two different storage schemes have been tried for the hybrid paradigm: an ordered tree and a private hash table. The private hash table has been found to be generally superior [Car87].

Example:

```
out("test", i)
in ("test", j)
in ("test", ? j)
```

Mess paradigm

The mess paradigm serves as a catch-all for any set that fails to meet the criteria of the four other paradigms. In order to be classified as mess, a set must have no element that is always actual across **outs** (thus ruling out *hash* and *hybrid*), and have at least one field requiring a match (ruling out *queue*). Such sets seem to be very rare [Car87]. The accepted representation of mess is a simple list, although unlike the queue paradigm, templates may need to be matched against all tuples in the list. This representation is a potential source of inefficiency, although no problems have ever been actually observed. Section 7.7 presents an alternative mess representation that is more efficient.

Example:

```
out("test", ? i)
out("test", i)
in ("test", j)
```

[9] [Seg91] discusses multiple keys, including partial keys.

In Chapter 3 we will discuss how best to implement each of the paradigms on DMMs, including as a concrete example the Intel iPSC/2. The concepts of set and key will be important when we discuss mapping tuples to nodes of the DMM.

2.4 Other Linda systems

Before we examine how DMM-Linda was designed to work efficiently on that class of machine, we want to examine previous efforts on other architectures and consider their implications for a Distributed Memory Multiprocessor design.

Any Linda design must answer the following questions: When I perform an **out**, where does the tuple go? Likewise, when I perform an **in** or **rd**, where does the template go, and how does it find a matching tuple?

If the machine has shared memory, we can use it for Tuple Space. The main problem will be avoiding contention on particular pieces of Tuple Space; toward that end careful use of blocking mechanisms is vital.

If no true shared memory is available, we have to locate tuples and templates on specific processors. Performing an **out** will cause a copy of the tuple to be sent to some subset of the nodes. Call this the *out-set*. Likewise, some subset (*in-set*) will receive a particular template. So long as we guarantee that the out-set will intersect the in-set, we can be sure that tuples and templates will find one another. We can vary the size of the out-set and in-set, as well as the way they are chosen, giving us a number of different possibilities.

More formally, let O_{ps} and I_{ps} represent the out- and in-sets for a particular process p **out**ing or **in**ing a particular tuple belonging to set s. Then:

$$\forall p, s : O_{ps} \cap I_{ps} \neq \emptyset$$

Since none of the mappings presented in this section make use of s, we can ignore it and simply look at O_p and I_p.

2.4.1 Positive broadcast

If we let the out-set encompass all nodes of the system and the in-set be the local node, we have what is termed a positive broadcast Linda kernel:

$$O_p \equiv P$$

$$I_p \equiv \{p\}$$

This is the strategy that Carriero used on the S/NET [Car87]. Using positive broadcast, **out**s require a broadcast, which is reasonably inexpensive on the S/NET machine since its nodes are connected by a single, high-speed bus. Nevertheless, every tuple interrupts all CPU's, and because all tuples are replicated on each node, memory usage for Tuple Space on each node is proportional to the total size of Tuple Space, so that no benefit can be realized from the increase in aggregate memory as the machine grows.

When performing an **in**, the searching takes place locally. Once a candidate tuple is found, a delete protocol is needed to ensure that all copies of the tuple are deleted

atomically. The node that generated the tuple is sent a delete request by each process desiring the tuple. Only one such request is granted, and all other nodes are ordered to delete their copy. Rds are very cheap, since no communication is required.

2.4.2 Negative broadcast

This is the counter-part to positive broadcast. Using this scheme, tuples are stored locally. Ins and rds first search locally; if no match is found, the template is broadcast throughout the machine.

$$O_p \equiv \{p\}$$

$$I_p \equiv P$$

Here, outs are cheap and ins and rds are usually expensive.[10]

This is the scheme used by Leichter for his LAN Linda system [Lei89]. As he pointed out, there are some subtle advantages to broadcasting templates rather than tuples.

1. Most importantly, the protocol can be easily designed such that multiple copies of templates present no logical difficulty. Because of this, we can use an unreliable broadcast mechanism for the templates, retransmitting when not receiving a reply within an appropriate timeout period.

2. There is an upper bound on the number of concurrent in or rd operations, and thus the number of templates present in Tuple Space, namely the number of processes (evals). In contrast, the number of concurrent tuples is unbounded. Replicating templates is therefore generally preferable to replicating tuples from the standpoint of memory usage, since there are generally far fewer templates than tuples present in Tuple Space at any given time.

3. Once generated, tuples can wander among the nodes of the system. This allows considerable flexibility for balancing memory usage. If a particular node's memory becomes overburdened, some tuples can be easily transferred to other nodes with no record-keeping necessary.

One disadvantage of negative broadcast is that ins and rds are largely serialized, since all nodes are typically involved in each search, assuming that no local match can be found. In contrast, positive broadcast requires only a local search, so many different ins and rds can search concurrently on different nodes.

[10] If we could predict the likely consumer of some tuple, we could preemptively store it on that node, rather than locally. This is similar to the rehashing optimization discussed in Chapter 5. Leichter[Lei89] discusses several possible methods for guessing the consumer, thereby increasing the proportion of local matches, although none were implemented.

2.4.3 Bi-polar multicast

This is an intermediate method between the two extremes of positive and negative broadcast. Here, both tuples and templates are multicast to a subset of the nodes. It is particularly advantageous to construct the sets such that the intersection is guaranteed to be one node, since this obviates the need to arbitrate between matches found by multiple nodes.

As an example, consider an $i \times j$ mesh of processes p_{ij}. Let the out-set be the row in which the outing node resides and let the in-set be the column in which the ining node resides:

$$O_{p_{ij}} \equiv p_{ik} | k \in 1 \ldots j$$

$$I_{p_{ij}} \equiv p_{kj} | k \in 1 \ldots i$$

This method was used by Krishnaswamy for the Linda Machine [Kri91]. It has the following characteristics:

- Both tuples and templates require multicasts.
- In requires a delete protocol.

In general, this method is a compromise. Tuples are replicated, but typically only by $\sqrt{n}$ rather than n, where n is the number of processors. Furthermore, multicasts are used rather than broadcasts. Unfortunately, in the process some of the advantages of each are lost. Tuples are no longer free to wander about, as they were with negative broadcast, and the multicast of tuples, at least, must be reliable. Because tuples are copied, a delete protocol is necessary. Krishnaswamy shows how this can be done very efficiently on his Linda Machine using bus protocols.

2.5 Summary

In this chapter, we described Linda and the role of tuples and Tuple Space. We also presented a compile-time analysis used to facilitate efficient matching. In the final section, we discussed three different implementations of Tuple Space. All three implementations share the characteristic that the location of a tuple or template depends only on the location of the process that creates it, rather than any quality of the tuple or template itself. Since matching depends on characteristics of the tuple rather than its creator, this storage rational complicates the matching process. In the next chapter, we will develop a very different design, one that is very well suited to distributed memory machines.

Chapter 3

Designing a Linda system for Distributed-Memory Multiprocessors

In this chapter we shall come to the heart of the matter; the design of a Linda runtime system that will be effective on distributed memory multiprocessors (DMMs). The methods used are suited to DMM's generally.[1] We tested the design by implementing it on an Intel iPSC/2[2].

3.1 A new kernel design: Distributed Hashing

Section 2.4 presented several designs for implementing Linda runtime kernels. All were judged to be inappropriate for large DMM's for two main reasons: they replicate either tuples or templates, and they require multicasts or broadcasts. Both are problematic for even small DMMs, and as the number of nodes increases, the cost in terms of memory and communication is prohibitive.

In general, these machines lack virtual memory, and thus the available real memory must be guarded cautiously. For that reason, tuple replication *a la* positive broadcast was quickly rejected. Template replication, *a la* negative broadcast, was somewhat more promising, since the number of templates that can co-exist is bounded by the number of `eval`s. Still, even assuming only one `eval` per node, the memory cost per node could grow linearly with the machine size, and even this was considered too high a cost.

More importantly, using broadcast or even multicast for each communication would have quickly saturated the machine, both in terms of communication bandwidth, as well as the load on the individual processors, which would be interrupted for every communication. This is especially true for machines with individual node-to-node links, such as

[1] For instance, the code was recently moved to a collection of Sun Sparcstations. Only the low level communication routines needed to be changed to allow for unreliable communication.

[2] Readers unfamiliar with the Intel iPSC/2 may want to read the overview of the machine presented in Appendix A before continuing with this chapter.

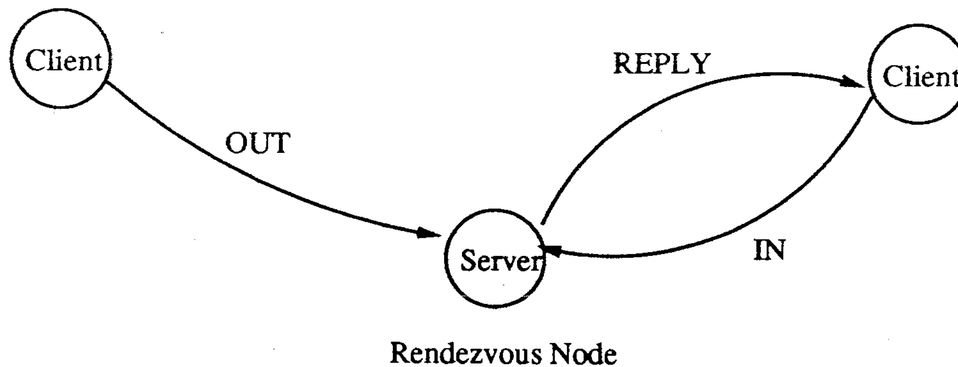

Figure 3.1: Schematic of Distributed Hash Design for Tuple Space.

hypercubes.[3]

The Distributed Hashing scheme has the required characteristics. In this scheme, **each tuple or template is directed, via a hash function, to some node in the system.** This node is the *rendezvous node* for that particular tuple. A Tuple Space server on that node will be responsible for finding tuples and templates that match. See Figure 3.1.

A typical Linda communication, where one node **outs** a tuple and another **ins** it, consists of a constant number of communications (usually three), regardless of the size of the machine. Only one copy of the tuple and template are made, so the memory requirements with respect to the size of the machine are constant as well.

Table 3.1 compares the total bandwidth, CPU and memory requirements of the four methods discussed, including distributed hashing, for a single **out** followed by an **in** on a machine with n nodes. O_{len} and I_{len} refer to the length of tuple or template, respectively. O_{cpu} and I_{cpu} refer to the CPU cost of handling an **out** or **in** request. Bandwidth refers to the total communication bandwidth consumed by the operations. For purposes of this comparison, we consider the CPU cost of tuple deletion (necessary in both positive broadcast and bipolar multicast) to be an additional I_{cpu} on each node involved; this is in addition to any communication that is required to transmit the delete request. For

[3]On interconnects where all communication is effectively serialized anyway, such as busses or Ethernet, the cost in terms of communication bandwidth is the same for broadcasts or point to point sends. Such machines are necessarily restricted in size, however, since communication bandwidth does not grow gracefully with the number of compute elements. Furthermore, although bus or Ethernet connects always "broadcast" in the sense that the data is potentially available to all nodes, a Linda design that forced all nodes to handle every communication would impose a load on each node proportional to the total number of nodes, up to a maximum imposed by the bandwidth of the machine. If network bandwidth is low compared to CPU speed, or if communication coprocessors are available, this load may not be prohibitive. However, we wanted a design that did not depend on any specialized hardware, nor degrade with improvements in communication bandwidth. For these reasons, we believe that our design is better suited to such architectures than broadcast-based schemes.

Method	Bandwidth (Hypercube)	Bandwidth (Bus/LAN)
Pos Broadcast	$nO_{len} + (n + 2\log_2 n)I_{len}$	$O_{len} + 3I_{len}$
Neg Broadcast	$\log_2 nO_{len} + nI_{len}$	$O_{len} + I_{len}$
Bipolar Multicast	$(\sqrt{n} + \log_2 \sqrt{n})O_{len} + 2\sqrt{n}I_{len}$	$2O_{len} + 2I_{len}$
Distributed Hash	$2\log_2 nO_{len} + \log_2 nI_{len}$	$2O_{len} + I_{len}$

Method	Cpu Load	Memory Load
Pos Broadcast	$nO_{cpu} + nI_{cpu}$	$nO_{len} + I_{len}$
Neg Broadcast	$O_{cpu} + nI_{cpu}$	$O_{len} + nI_{len}$
Bipolar Multicast	$\sqrt{n}O_{cpu} + 2\sqrt{n}I_{cpu}$	$\sqrt{n}O_{len} + \sqrt{n}I_{len}$
Distributed Hash	$O_{cpu} + I_{cpu}$	$O_{len} + I_{len}$

Table 3.1: A comparison of four Tuple Space designs.

comparison, we include the bandwidth required on a bus or LAN; the main distinction being that they allow multicasts and broadcasts to be done in a single communication event, saving considerable bandwidth.[4]

The equations for hypercube bandwidth deserve some explanation. First, recall a few characteristics about communicating on an N node hypercube:

- Sending a message between two nodes requires no more than $\log_2 n$ links.
- A broadcast requires n links using a spanning tree[5].

Positive Broadcast The initial broadcast of the tuple requires nO_{len}. When the **in** is performed, the match occurs locally. When a match is found, a request for the tuple is sent to the owner node ($\log_2 nI_{len}$), the owner replies ($\log_2 nI_{len}$) and broadcasts the delete request (nI_{len}).

Negative broadcast The **out** incurs no communication. The **in** causes the template to be broadcast (nI_{len}). Finally, the node holding the tuple sends it to the requesting node ($\log_2 nO_{len}$).

Bi-modal multicast The **out** requires a multicast to the out-set ($\sqrt{n}O_{len}$); similarly the **in** requires $\sqrt{n}I_{len}$ to send the template to the in-set. Sending the matching tuple

[4]However, the total bandwidth *available* on current LANs is much lower, so that the LAN will reach saturation much sooner than a hypercube such as the iPSC/2. A 64 node iPSC/2 has a total bandwidth of approximately 128 * 2.2 Mbytes/sec, or 2.8 Gbytes/sec, compared to a single ethernet having an effective rate of 0.5 to 1.0 Mbytes/sec total. Of course, achieving such high performance on an iPSC/2 is possible only with certain communication patterns that avoid node and link contention while still using all of the connections. Ethernet is less sensitive to the communication pattern, and will reach its maximum throughput more readily.

[5]This is done by using a binary tree to span the hypercube. The tree can be embedded in the hypercube so that adjacent nodes in the tree are also nearest neighbors in the hypercube.

to the requesting node requires $\log_2(\sqrt{n}O_{len})$. Finally, the delete request requires another multicast $(\sqrt{n}I_{len})$.

Distributed Hash Sending the tuple to the rendezvous node requires $\log_2 nO_{len}$, sending the template requires $\log_2 nI_{len}$, and sending the reply requires another $\log_2 nO_{len}$.

From the table it is evident that the various loads imposed by the distributed hash method grow much slower than for the other methods. The costs associated with distributed hashing grow at most by logarithmic factors; all of the others have some costs that grow as polynomials as n increases. Figure 3.2 displays how the total bandwidth predicted by the model for a single small (100 bytes) **out** and **in** (O_{len} 100 bytes, I_{len} 50 bytes) grows on an increasingly large hypercube. Bandwidth is in units of link-bytes; for instance, broadcasting 100 bytes on an n-cube would require $100n$ link-bytes assuming that the broadcast was done using a spanning tree. The logarithmic growth of the distributed hash method contrasts sharply with the polynomial growth of the other three methods.

Figure 3.3 shows the bandwidth for a large (10000 byte) tuple (O_{len} 10000 bytes, I_{len} 50 bytes). Here again the distributed hash method is clearly superior to the other three methods. The lower case letters refer to optimized versions of three of the protocols in which the large chunk of data in the tuple is not needed for the match (as is typically the case). In that event, we can arrange to leave the data behind on the node of origin, and forward it directly to the destination after the match is resolved. This leads to substantial savings for large tuples for three of the protocols; the distributed hash method remains superior. Section 5.5 discusses this optimization in the context of distributed hashing.

Table 3.1 illustrates why the basic distributed hashing paradigm is much more scalable than the other three methods on DMMs, particularly hypercubes. Its memory and cpu loads grow linearly with the machine size, allowing the available memory and compute power to keep pace. The bandwidth requirements grow slightly faster than the available bandwidth, but only by a log factor.

The reader may well remain concerned about the fact that each node's bandwidth requirements will grow logarithmically, while the per-node bandwidth will remain constant as the machine grows. We respond to this concern with three observations:

- For many programs, this is the best that we can do. If the program uses communication patterns without much locality, the lengths of these communications will necessarily grow with the diameter of the machine.

- If there is some locality, the Linda program may be able to exploit it and hold the bandwidth requirements constant as the machine grows. This is done by mapping the processes onto the machine so as to keep the distances between communicating nodes short, and then allowing the rehashing optimization (see Section 5.2) to send the data directly to the consumer. Obviously, it would be nice to provide this mapping automatically; at present, the user must map **evals** to nodes by hand. This is done by **eval**ing a vanilla process on each node; the vanilla process then determines what node it is running on and executes accordingly.

- Logarithmic growth is very slow!

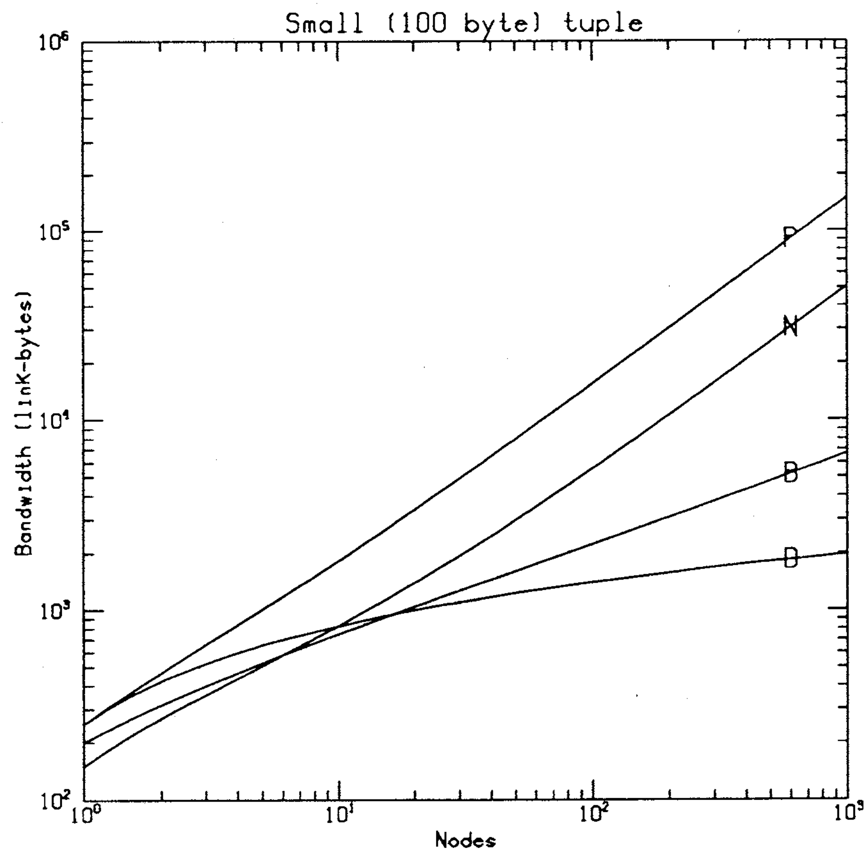

Figure 3.2: A comparison of total bandwidth requirements for **out** and **in** of a small (100 byte) tuple on a hypercube using four different methods: Positive Broadcast (P), Negative Broadcast (N), Bipolar-Multicast (B), and Distributed Hash (D)

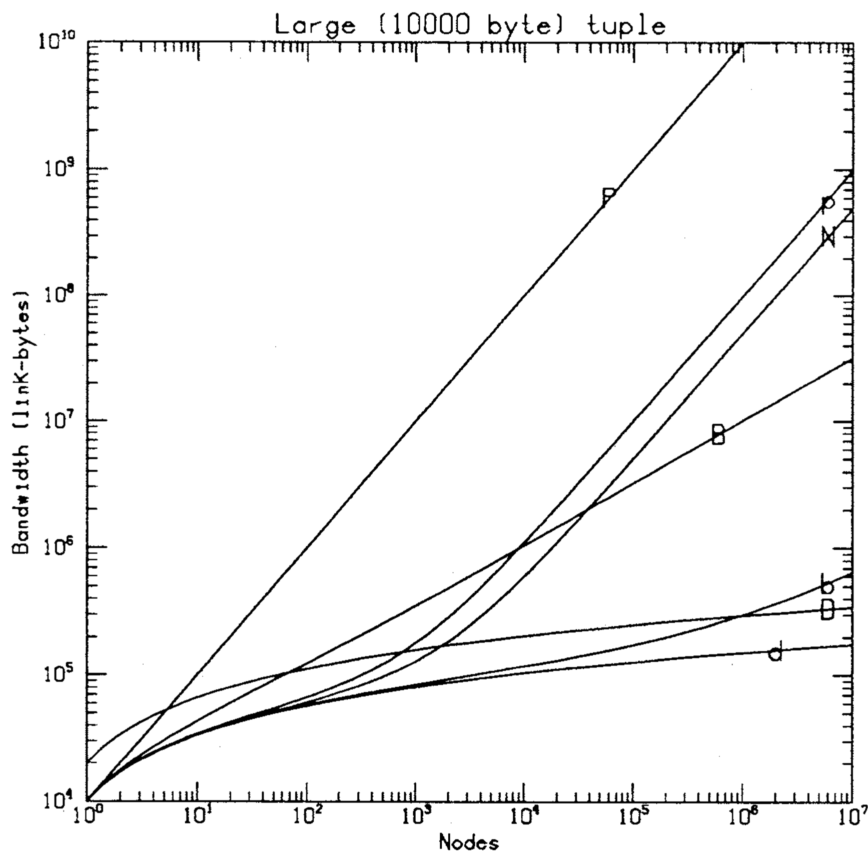

Figure 3.3: A comparison of total bandwidth requirements for **out** and **in** of a large (10000 byte) tuple on a hypercube using four different methods: Positive Broadcast (P), Negative Broadcast (N), Bipolar-Multicast (B), and Distributed Hash (D). Lower case letters refer to optimized protocols; see the text.

3.2 Tuple Space servers

The fundamental structure of the design implements Tuple Space by a collection of Tuple Space servers, each servicing requests to a particular disjoint part of Tuple Space. How Tuple Space is divided among the servers is discussed in Section 3.4. Processes performing Linda operations request services from the Tuple Space server responsible for the appropriate slice of Tuple Space. Several constraints were imposed on the Tuple Space servers. We will discuss each in turn.

Servers should run at higher priority than clients

We want to service requests as fast as possible, since another process may be waiting, unable to proceed until the request is serviced. Speedy request service is also important in order to minimize the amount of message buffering needed.[6]

Servers may not generate requests requiring a reply (i.e. block)

This is a necessary condition for deadlock avoidance. Assume two Tuple Space servers A and B each receive a request simultaneously. Assume further that A's request causes A to make a blocking request of B, and vice versa. Neither will be able to proceed, since both are awaiting service from the other.

It is important to prove that, if only clients can block on requests, deadlock cannot result. Such a proof is trivial: any request generated in the system will eventually begin being serviced, and will eventually complete, possibly generating requests of its own.[7]

Servers must be able to handle requests from clients with an imperfect view of Tuple Space

We want to be able to adjust the mapping of Tuple Space to physical processors, e.g. in order to take advantage of locality or to balance memory- or CPU-loading. As the mapping changes, nodes will have inconsistent views of Tuple Space, some still using the old mapping, some the new. This condition is acerbated by the fact that we want the option of notifying nodes gradually, as they actually need to know, rather than preemptively, since in many cases only a few nodes will ever need to know of the change.

[6]The manner in which we organize clients and servers will vary depending on the capabilities of the machine in question. If a machine supports multiple processes per node (nCUBE2 [nCU90], iPSC/2 [Int90]), we can have one distinguished Tuple Space server and a number of clients on each node. If each node supports only one process, but does have message interrupt facilities (iPSC/860 [Int90]), we can let one process act as client and server. If the machine does not support asynchrony on a single node, we will need to segregate the nodes into client and server nodes. Essentially, this implies determining a reasonable static ratio of compute to communication load *a priori*, which may not be easy to do in any particular case, and probably impossible in general.

[7]The system could reach a form of livelock if handling a request caused an infinite number of new requests to be generated.

A given unit of Tuple Space has only one owner

In order to coordinate the orderly transfer of pieces of Tuple Space, each mappable unit has one owner. The owner is solely responsible for changing the manner in which that part of Tuple Space will be handled. (Most of these decisions concern optimizations, which will be described in Chapter 5). For instance, only the owner may force a change of ownership by specifying a new node as owner. As servers hand off pieces of Tuple Space to other servers, they retain forward pointers to the new location. These pointers are necessary because out-dated requests may continue to be directed to the old server during the rest of the computation. Subsequent requests to the old server cause the request to be forwarded, and the requester is informed of the change.

3.3 Request types

Several types of requests are allowed in the system. The most important are those corresponding to the Linda operations themselves. When an **out** is performed, the process (client) makes a tuple space request to the appropriate server. Included in the request is the tuple itself. Similarly, when doing an **in**, the template is included in the request for service. Servicing an **in** request will eventually require a reply to the requester (i.e. the matched tuple). Thus, there are two basic types of communication in the system: requests, which go to Tuple Space servers, and replies, which go to clients. Tuple Space servers may also generate requests for other servers, in effect acting as clients. However, in order not to violate constraint 2, such requests for service must not wait for a reply. No assumptions are made about the order in which requests arrive on the destination node. This facilitates implementing the design on top of an unreliable communication mechanism; we need only insure that the message *eventually* arrives.

In addition to **in** and **out** requests, several other requests may be made to Tuple Space servers:

Redirect requests This request informs a node of dynamic changes to the Tuple Space mapping.

Statistics requests Each node maintains a variety of runtime statistics. Other nodes can request this information.

Randomize requests The random queue paradigm uses several request types in order to inform nodes of conversions to and from the random state (The random queue paradigm is an optimization; see Section 5.7).

Large data elements Sometimes it is advantageous to store large data elements separately from the tuple itself, either to equalize memory usage or to avoid multiple sends. Once a match occurs, a request must be made to the process holding the data, causing it to be sent to the recipient of the tuple. See Section 5.5.

3.4 Mapping tuples and templates to rendezvous nodes

An important aspect of any Linda implementation is organizing Tuple Space such that as few matches as possible are performed. As mentioned in Section 2.3, Carriero's compile-time analysis usually reduces the number of potential matches considerably. Some tuples still require run-time matching, however. In addition, in a distributed memory implementation, the tuples need to be dispersed as evenly as possible across the machine, in order to equalize the demands on memory and CPU from Tuple Space requests. In this section we will examine schemes for mapping tuples and templates into Tuple Space.

3.4.1 Description of the problem

The goal of the hashing is to map tuples and templates to nodes, and in some cases (hash and hybrid) to partitions (buckets) within a node. There are many ways of performing this mapping; however, we wish to place a restriction on the mapping that makes searching for matches much easier:

Matching tuples and templates must be mapped to the same node.

This restriction to the mapping provides a number of benefits:

A Tuple Space search is limited to one node. If no match is found there, none exists, and searching may stop until more tuples arrive.

Communication overhead is predictable. A given Tuple Space request is only sent to one node. Of course, other requests use different nodes.

Synchronization requirements are minimized. Only one node participates in searching for a particular match.

Concurrent Tuple Space access is improved. Since any given Tuple Space access only involves one node, many of them may operate fully independently in parallel.

Information concentration is facilitated. Since a node is solely responsible for the tuples mapped to it, it can build up knowledge about those tuples, becoming a sort of "expert".

We will use a hash function to perform the mapping of tuples to nodes. We must chose the inputs to the hash functions carefully, in order to satisfy the above restriction. Each tuple belongs to some set s. In addition, all tuples from *hash* sets and some tuples from *hybrid* sets have a key k. Obviously, any matching tuple/template pair must belong to the same set and have the same key value. Therefore, using them as inputs to the hash function will certainly satisfy the restriction. It can be shown that this is the best we can do while still obeying the restriction that only one node need be searched. The set identifier contains all of the information available during compilation. At runtime, the only additional information we have is the value of the fields; only fields that are always present as actuals in all tuples and templates (keys) may be used as inputs, since using any other fields would violate the restriction. To see why this is so, consider hashing a template with a formal field. If the corresponding actual field was used in hashing a matching tuple, we

have no way to guess the value, and thus no way to provide the same inputs to the hash function in order to reach the same node.

In the distributed hashing method, a tuple's storage location is a function $\mathcal{H}$ of s and k.

$\mathcal{H}$ returns the node, termed the *rendezvous node*, and bucket within that node (if applicable), where that tuple should be stored. Thus, if $\mathcal{N}$ is the set of nodes, $\mathcal{B}$ is the set of buckets within each node, and $\mathcal{S}$ the tuple sets:

$$\mathcal{H}(s,k) \Rightarrow\ n \in \mathcal{N}, b \in \mathcal{B}$$

3.4.2 Requirements for hashing

What characteristics should a good mapping have?

1. As mentioned above, only one node should need to be searched for a match.
2. If possible, tuples should be smeared evenly across all the nodes. This is important in order to equalize both the memory requirements of Tuple Space and the searching load on the CPUs.
3. Some sets will need to be confined to a single node. Such sets should be mapped to different nodes to the extent possible (if $|S| < |N|$, otherwise, each node will support more than one set). Again, we want to equalize the load as best we can.
4. Within a given node, the tuples should be spread as evenly as possible throughout that chunk of Tuple Space (where applicable). This reduces the search space for a given match.

In order to obey condition 1, a set classified as queue, hybrid, or mess must be confined to a single node, since there is no way to divide such a set so that the rendezvous node for a matching tuple is unique and deterministic. This is because none of these sets have dependable keys (queue and mess have no keys; hybrid has only partial keys). The only acceptable input to the hash function is the set identifier, so all tuples of a given set will map to the same node.

In a hybrid set, all tuples have keys, and we could distribute them among different nodes based on the key. Some templates have keys as well, and these templates could be directed to the correct node. Templates without keys are the problem, since such templates might have to search all the nodes looking for a match. Because of this, we presently confine each hybrid set to a single node. Chapter 7 presents a possible alternative implementation and Appendix E sketches an augmented data structure for private hash tables that reduces the cost of exhaustive search.

Because each of these sets has a different set identifier, they will be mapped individually to different nodes, which will tend to even out the load (condition 3). Furthermore, although hybrid sets are mapped to a single node, we will then use the partial key to map into a private hash table on that node (condition 4).

Tuples belonging to sets classified as hash have both set identifier and a key value, and are thus scattered across the entire machine. If we treat the collection of local hash tables

as one global hash table, $\mathcal{U} \equiv \mathcal{N} \times \mathcal{B}$, then determining the destination of a particular tuple involves first mapping to a particular $u \in \mathcal{U}$, and then breaking that address into node and bucket. We let the number of buckets per node be a power of two, so decomposing u into node and bucket can be done cheaply using bit manipulation.

We handle each case below:

Queue and Mess

These sets have no key fields, so $\mathcal{H}$ is simply a function of the set identifier. Set identifiers are consecutive small positive integers assigned by the compiler, so we can use a very simple function:

$$\mathcal{H}_{node}(s) \equiv s \bmod n$$

Hybrid

All tuples and some templates of this set have a key, but the key cannot be used to determine the node, because we want the entire set to be stored on the same node. Thus, we use the same function to determine the node that we used for queue and mess. For tuples with keys, we determine the bucket within the node via a standard hash function.

$$\mathcal{H}_{node}(s) \equiv s \bmod n$$

$$\mathcal{H}_{bucket}(k) \equiv \mathcal{H}_f(k)$$

Note that because all tuples in a private hash table share the same set index, there is no reason to include s in $\mathcal{H}_f$.

Hash

This is the most interesting case. We want to smear each set across all the buckets in all the nodes. Again, we want a function H:

$$\mathcal{H}(s,k) \Rightarrow n \in \mathcal{N}, b \in \mathcal{B}$$

We can approach this problem in two steps:

$$\mathcal{G}(\mathcal{H}_1(s), \mathcal{H}_2(k)) \Rightarrow u$$

$$\bar{\mathcal{G}}(u) \Rightarrow n \in \mathcal{N}, b \in \mathcal{B}$$

$\mathcal{G}$ is a combining function, while $\bar{\mathcal{G}}$ separates the hash output into node and bucket.

How shall we choose $\mathcal{H}_1$ and $\mathcal{H}_2$? There is a great deal of existing research on hash functions; assuming that s and k are "normal" inputs for hash functions, we can simply choose "good" hash functions. Unfortunately, it isn't obvious that they are "normal" inputs; as stated before, s belongs to the set of small positive integers. Furthermore, Linda programs tend to use keys in hash sets in a linear way, since they often function as indices:

$$k_i \in \mathcal{K} = c_1 + ic_2$$

Because they occur so often, we want to make sure that our hash functions handle these sorts of input domains well. We considered two different approaches and performed a small experiment to judge the effectiveness of each on various input domains.

- Use a standard *universal* hash function[8], of the sort $ak + b \bmod m$, where k is the input and a, b, and m are primes.
- Use a handcrafted function that is optimized to perform well on the sort of linear combinations of keys described above.

Universal hash function

One example of a universal class of hash functions is similar to a linear congruential random number generator, $c_1 k + c_2 \bmod m$. We will use this hash function to map from $< s, k >$ to a large range $\mathcal{U}$, which we will then decompose into node and bucket. This decomposition can be done in a straightforward manner because the output of the hash function has the property that, on average, all bits are equally significant. It is important to note that the property of the universal hash class holds for the class as a whole, not for each member. We will still need to be careful choosing c_1, c_2, and m. Knuth [Knu73] recommends making c_1, c_2, and m relatively prime, and m a prime.

We still need to decide how to compose the two inputs to the hash function (s and k). Carter and Wegman describe a method of extending universal hash functions to very long keys. They split the key into two components, apply a universal hash function to each, and compose the result with exclusive or. They go on to prove that such a composition of two universal classes of hash functions is also universal. We used this technique, hashing s and k separately with different values of c_1 and c_2, combining the two results with exclusive or[9]. In addition, we can also split k into low and high halves, k_0 and k_1, and similarly hashed each half separately with a different function. This helps to avoid arithmetic overflow when computing $c_1 k$.

Handcrafted hash function

These functions are unabashed attempts to optimize hashing of linear keys. Consider the entire collection of nodes and buckets to represent one large hash table $\mathcal{U}$. In the following discussion, we assume that both $|N|$ and $|B|$ are powers of two. The basic function is:

$$H_U \equiv ((2s+1)|B|+1)k + s|B|$$

$$b = H_U(s,k) \bmod |B|$$

$$n = (H_U(s,k)/|B|) \bmod |N|$$

For successive $k \in \mathcal{K}$ our function will take a step of size $(2s+1)|B|+1$ buckets, which corresponds to stepping through an odd number of nodes (based on s), plus one bucket. The addition of $s|B|$ causes each set to begin on a different node.

---

[8] A universal class of hash functions is a set of hash functions H mapping from A to B, such that no pair of distinct keys collides under more than $1/|B|$ of the functions. For a good discussion of universal hash functions, see Carter and Wegman [CW79]. Such functions have the property that the average performance on *any* input is comparable to the performance of a single function constructed with knowledge of the input.

[9] This method would also work well for combining keys in tuples that contain multiple keys. Ed Segall [Seg91] at Rutgers University has looked into multiple key hashing.

Such a function has several nice properties. Because the step $(2s+1)|B|+1)$ is relatively prime to any power of two, and the size of $\mathcal{U}$ ($|N||B|$) is a power of two, the least common multiple of the number of buckets stepped and $|\mathcal{U}|$ is their product, so that H_U describes a permutation of $1 \ldots |\mathcal{U}|$. We are guaranteed that consecutive keys will map to all nodes before any repetitions.

Similarly, because the number of nodes stepped $(2s + 1)$ is relatively prime to the number of nodes (a power of two), for consecutive keys we step through each node before returning to the same. Finally, because step size depends on s, each set visits the buckets in a different permutation.

Although the above discussion presumed consecutive keys (i.e. striding by one), the properties described hold for any stride relatively prime to $|\mathcal{U}|$, i.e. any odd number. In order to deal with even strides, particularly powers of two, an additional $k/|N||B|$ was added to the function, which has the effect of bumping the function by one bucket each time k strides though $|\mathcal{U}|$.

3.4.3 Experiment

We devised an experiment to compare eight versions of the hand-crafted hash function and six versions of the linear congruent hash function.

These hash functions were tested against 26 different sequences of set/key pairs, using a small driver program that simulated evaluating the functions on a machine with 64 nodes and 1024 buckets per node. Most of the domains to be hashed represented linear set/key pairs, of the type already described as very common in Linda programs. We also included a random domain generated by the Unix library call `random()`, and a domain formed from entries in a dictionary file[10]

For each test, a total of 65536 pairs were hashed, the same as the total number of buckets. Four criteria were used to judge a hash function's performance on a particular pair domain:

Maximum hits on a single bucket The expected number of hits on a bucket was one; this criteria looks at the most unbalanced bucket.

Number of unbalanced nodes Each node expects 1/64th of the total pairs; this measure counted the number of nodes off by more than ± 10%.

Successive hashes to the same node Given that keys often function as indices, we felt it important that consecutive key values not show patterns such as mapping to different buckets within the same node many times consecutively. This measure simply counts the number of consecutive hits on the same node.

[10]The motivation for this domain was to present at least one "normal" set of inputs; i.e. one without any discernible structure, yet not random either. We formed the domain by using words from the Unix file `/usr/dict/words`. We used exclusive-or to combine the first two 32 bits (four characters) of each word into one 32-bit key. Unfortunately, `/usr/dict/words` only held 25000 words. In order to generate enough keys (65536) we generated two more keys from each key described above by capitalizing the second and third letters of the word, respectively.

1. $u = ((2s+1)h+1) + sh + k, \text{node} = u/h, \text{bucket} = u$
2. $u = ((2s+1)h+1) + sh + k + k/ch, \text{node} = u/h, \text{bucket} = u$
3. $u = ((2s+1)h+1)k + sh, \text{node} = u/h, \text{bucket} = u$
4. $u = ((2s+1)h+1)k + sh + k/ch, \text{node} = u/h, \text{bucket} = u$
5. $u = ((2s+1)h+1)k + sh + k/ch + k, \text{node} = u/h, \text{bucket} = u$
6. $u = ((2s+1)h+1)k + sh + k/ch + s, \text{node} = u/h, \text{bucket} = u$
7. $u = ((2s+1)h+1) + sh, \text{node} = u/h, \text{bucket} = k$
8. $u = ((2s+1)h+1) + sh + k/ch, \text{node} = u/h, \text{bucket} = k$
9. $u = ((251k + 277) \wedge (271s + 257)) \bmod 65537, \text{node} = u, \text{bucket} = u/n$
10. $u = (((251k_1+277)\wedge(263k_0+269)\wedge(271s+257))) \bmod 65537, \text{node} = u, \text{bucket} = u/n$
11. $u = ((251k) \wedge (271s)) \bmod 65537, \text{node} = u, \text{bucket} = u/n$
12. $u = ((7001k) \wedge (5003s)) \bmod 65537, \text{node} = u, \text{bucket} = u/n$
13. $u = ((((251k_1 + 277) \wedge (263k_0 + 269) \wedge (271s + 257))))^2/16, \text{node} = u, \text{bucket} = u/n$
14. $u = (((251k_1) \wedge (263k_0) \wedge (271s))) \bmod 65537, \text{node} = u, \text{bucket} = u/n$

Figure 3.4: Hashing experiment: functions

Number of empty buckets For a truly random input, the expected ratio of empty buckets is about 0.32, or ≈ 24000.

Figure 3.4 shows the hash functions tested. In functions 10, 13 and 14, k_1 and k_0 refer to the most and least significant bits of the key, respectively. In all cases, we first map into $\mathcal{U}$, and then break this value into a node and bucket. This last step is done using modulo arithmetic, which is omitted from the figure. Figure 3.5 shows the pair domains. In these figures, n is the number of nodes, h is the number of buckets per node, both powers of two.

We scored the performance of each hash function by first finding its mean performance for each criteria across all input domains. See Table 3.2. We then ranked each hash function based on the mean, and finally formed an overall ranking from the category rankings. See Table 3.3.

The functions performing best were 12, 11, 9 and 14, all linear congruent hash functions. The best hand-tailored function, 6, shared 5th place, although the top six hash functions performed well and were fairly closely grouped, with the remaining eight performing poorly.

```
for i=0 to 65535:
```

1. $< 0, i >$
2. $< i, 0 >$
3. $< i \bmod 2, i/2 >$
4. $< 0, 2i >$
5. $< i \bmod 3, i/3 >$
6. $< 0, 3i >$
7. $< i \bmod 5, i/5 >$
8. $< 0, 5i >$
9. $< i \bmod 15, i/15 >$
10. $< 0, 15i >$
11. $< i \bmod 16, i/16 >$
12. $< 0, 16i >$
13. $< i \bmod 17, i/17 >$
14. $< 0, 17i >$
15. $< 0, ih >$
16. $< 0, ih - 1 >$
17. $< 0, ih + 1 >$
18. $< 0, ic >$
19. $< 0, ic - 1 >$
20. $< 0, ic + 1 >$
21. $< 0, ihc >$
22. $< 0, ihc - 1 >$
23. $< 0, ihc + 1 >$
24. $< 0, random() >$
25. $< 0, word >$

Figure 3.5: Hashing experiment: input domains

Hash func.	Max. hits	Unbal-anced	Same node	Empty buckets
1	39.69	20.54	35548	37718
2	7.81	20.08	35552	22907
3	37.58	8.31	14585	35824
4	6.81	1.00	14504	19891
5	7.46	1.12	14031	31552
6	3.04	1.00	14506	18561
7	70.81	61.54	47891	62930
8	67.31	61.23	44921	61553
9	4.38	0.04	5790	22494
10	3.65	0.42	15887	12828
11	3.77	0.04	5864	12855
12	4.35	0.00	104	22180
13	65.04	23.31	5959	25818
14	5.85	0.00	5756	23599

Table 3.2: Mean hash function performance on test domains.

Hash func.	Max. hits	Unbal-anced	Same node	Empty buckets	Overall
1	11	11	11	12	12
2	9	10	12	7	9
3	10	9	9	11	11
4	7	6	7	4	7
5	8	8	6	10	8
6	1	6	8	3	5
7	14	14	14	14	14
8	13	13	13	13	13
9	5	3	3	6	3
10	2	5	10	1	5
11	3	3	4	2	2
12	4	1	1	5	1
13	12	12	5	9	9
14	6	1	2	8	3

Table 3.3: Hash function ranking on test domains.

3.4.4 Final remarks about hashing

In retrospect, it was probably unwise to make $|\mathcal{U}|$ a power of two; a prime would have been a better choice, since division and remainder operations on powers of two ignore some subset of the available bits, thus consistently losing information.

Although the previous discussion has assumed that the key values will be integers, as indeed they often are, we need to be able to handle arbitrarily typed keys by first converting them into an integer and proceeding as before. For most types a simple type-converting assignment will serve; for aggregates (arrays and structures) some sort of combining is necessary; we handled them by exclusive-or'ing the length and the first word of the data.

The function actually implemented on the iPSC/2 was the hand-tailored version. Most of the programs run thus far on the system have used linear keys, and the function has performed very well. Still, it is potentially more vulnerable to degenerate keys than the universal function; for that reason, we recommend that future Linda implementations use universal hash functions instead.

3.5 The low level structure of Tuple Space

As has been previously mentioned, each node in the system maintains some piece of Tuple Space and handles requests for those tuples. We now examine the data structures that, together, organize Tuple Space. We can divide the data structures into two groups: those that represent the actual tuples, and those that organize the tuples into a coherent Tuple Space.

3.5.1 Tuple data structures

A tuple is really just a collection of information. This information can be separated into three classes pertaining to:

this particular tuple e.g. values of the fields, the hash values, runtime length, and place of origin.

all tuples generated by the same Linda operation e.g. element polarity (formal vs. actual) and operation type (`in out rd`).

all tuples belonging to the same set e.g. type signature, number of fields.

Initially we planned to use three separate data structures, one for each class outlined above. This would have maximized the data sharing, with each set represented by one set structure, each operation represented by one operation structure (and containing a pointer to the appropriate set structure), and each tuple represented by a tuple structure in turn pointing to an operation structure. In the end we decided to use only two structures: a dynamic structure (called a *proto-tuple-packet* or `ptp`) that contains all of the tuple-specific data and a static structure (called an `st`) that is a combination of the set and operation structures. One `st` is produced by the Linda compiler for each Linda statement found in the program. Folding the set data into the operation structure does result in it being

replicated; however, the total data involved is small enough (14 bytes at present) that the simplification was considered worthwhile, in that it reduced the amount of indirection necessary to access set-specific information. In general, the number of textual operations and sets is very small compared to the total number of tuples generated; the most important consideration is to avoid replicating everything for each tuple.

When transmitting a tuple from node to node, only the `ptp` is sent. Each node maintains a copy of all of the `sts`. This results in substantial savings in communication bandwidth (since less data is sent) and memory (since all tuples generated by the same Linda statement residing on a particular node share a single `st`)[11]. Figure 3.6 shows the structure of a `ptp`. We describe the fields below:

scratch A volatile field used for various purposes, most notably specifying the request type. The request handler uses the value of this field to dispatch the request to the correct routine.

succ A `ptp` pointer used to link `ptps` together in chains. We used singly-linked chains (discussed below) so only one pointer is needed.

tid A (reasonably) unique identifier given to each tuple. This id is most useful when performing a delete protocol to ensure that exactly the same tuple is deleted system-wide. The tid is produced by concatenating the id of the process generating the tuple with a operation counter specific to that process.[12]

orig_node, orig_pid Specifies the source of the tuple or template. Replies will be returned to this process.

dest_node, dest_pid Specifies the ultimate destination of the tuple.

st_ptr Pointer to the `st` structure associated with this tuple. On the iPSC/2 we were guaranteed that memory was laid out identically on each node, so we simply used a pointer. On most machines it is necessary to make `st_ptr` an index into a table of pointers to `st` structures, with the table correctly initialized on each node.

in_st_ptr After a match occurs, the `ptp` representing the `out` has this field set to the `st` of the `ptp` representing the `in`. The reason is a little obscure: in addition to being returned to the `in`ing process, the `ptp` may be sent to some other node as well. That node may need to know the identity of both `in` and `out`.

length The total runtime length of the tuple in bytes. This is the static length (see the description of the `st` length field, below) plus the length of any data elements that were concatenated to the end. This is the number of bytes that will actually be sent.

[11]Currently, a k element tuple (ignoring array or structure elements) consumes $38 + 8k$ bytes for the ptp and $30 + 8k$ bytes for the ST, giving us a memory savings of almost 50%. While the bandwidth savings are overshadowed by the startup-cost on present machines, the technique *does* allow us to remain below the 100 byte threshold (which is important on the iPSC/2, see Appendix A) until a tuple reaches 8 elements, compared to 2 or 3 without it.

[12]Although the counter could roll over and thus produce a non-unique tid, this would take a minimum of 9 hours on a 64 node iPSC/2 (assuming 2 Linda operations per node per millisecond, and 26 bits dedicated to the counter). Obviously, using a few more bits would relieve any worry one might have about this.

hash_value Each tuple or template has a hash value that is used to determine its storage location and as a quick match filter. In order to access this value quickly, we store it explicitly.

local_field Some large data elements may be stored locally on the node of origin rather than packed into the ptp. The n_{th} bit set indicates that this was done with the n_{th} element. See Section 5.5 for discussion of local field storage.

field This array holds the values of actual tuple elements and pointers for formals. Aggregate data objects (structs and arrays) are handled somewhat differently since they will not fit directly into the slot provided (8 bytes). In this case we use a descriptor (LBINT) consisting of a length and a pointer. Length specifies the length of an actual element in bytes. The pointer can be either an absolute pointer or a relative pointer. We discuss this below.

A ptp can exist in two slightly different forms, having to do with the representation of aggregates[13]. Aggregates are represented by descriptors that can use either absolute or relative pointers to the data. Initially, the data pointers in the descriptors (ptp.field[].b) are absolute pointers into the client's address space. To satisfy Linda's semantics, the data is copied out of the client's space into Tuple Space buffers. In order to reduce the number of communication startups, when doing this copying it is usually best to pack the aggregates together with the ptp proper into one contiguous buffer[14]. Once this is done the ptp uses a pointer relative to the start of the buffer in order to allow the entire buffer to be readily relocated without having to adjust the pointers. Figure 3.7 shows the layout of a ptp with two aggregate fields, before and after conversion. Note that the representation using relative pointers is contiguous and easily relocatable[15].

Figure 3.8 shows the structure of an st. In the descriptions following, the starred (*) items refer to data that is constant for the entire set, and could have been stored in a separate, set-specific data structure to save additional storage.

set_id Set to which the operation belongs.

op_type Type of Linda operation (in, out, etc.)

inout_func Pointer to a function used by the in/out collapse optimization. See Section 5.3. Normally a null pointer.

num_fields* Number of elements in the tuple.

length* Static length of the ptp that will be generated by the Linda operation. This length excludes the length of any aggregates that may be concatenated to the ptp, since their lengths can only be determined at run-time. The static length is simply the size of an ST_TYPE less unused ST_FIELDs, based on the value of num_fields.

---

[13] In C, structs or arrays.

[14] Effectively trading off extra copying for less communication startup cost, a very effective strategy on most DMMs, where communication startup costs are high, and communication bandwidth is considerably less than the in-memory copy speed.

[15] Relative pointers are not used for fields that will be stored locally due to the optimization described in Section 5.5.

```
/* dynamic tuple structure.  Filled each time operation is performed */
typedef union ptp_field {  /* The value of a particular tuple element */
    double  d;             /* double (actual) */
    double  *dp;           /* double (formal) */
    float   f;             /* float  (actual) */
    float   *fp;           /* float  (formal) */
    char    ic;            /* char   (actual) */
    char    *icp;          /* char   (formal) */
    short   is;            /* short  (actual) */
    short   *isp;          /* short  (formal) */
    int     ii;            /* int    (actual) */
    int     *iip;          /* int    (formal) */
    long    il;            /* long   (actual) */
    long    *ilp;          /* long   (formal) */
    LBINT   b;             /* BLOCK  (actual or formal) */
} PTP_FIELD;

/* Block data structure.  Used for struct and array elements */
typedef struct {
  long size;               /* length in bytes */
  union {
    char *abs;             /* absolute data pointer */
    long rel;              /* relative data pointer */
  } data;
} LBINT;


typedef struct ptp_type {
  long   scratch;        /* volatile field for various purposes.  IT MUST
                            BE FIRST IN THIS STRUCTURE! */
  struct ptp_type *succ; /* for linking ptp's */
  long  tid;             /* unique id for this tuple */
  short  orig_node;      /* id of origination node */
  short  orig_pid;
  short  dest_node;      /* id of destination node */
  short  dest_pid;
  ST_PTR st_ptr;         /* key into statement array */
  ST_PTR in_st_ptr;      /* st of in matching this out */
  long   length;         /* runtime length of tuple (including long fields) */
  long   hash_value;     /* hash value */
  long   local_field;    /* long fields that were stored locally:  the nth */
                         /* bit signals for the nth field */
  PTP_FIELD field[NUM_FIELDS];
} PTP_TYPE;
```

Figure 3.6: ptp data structure.

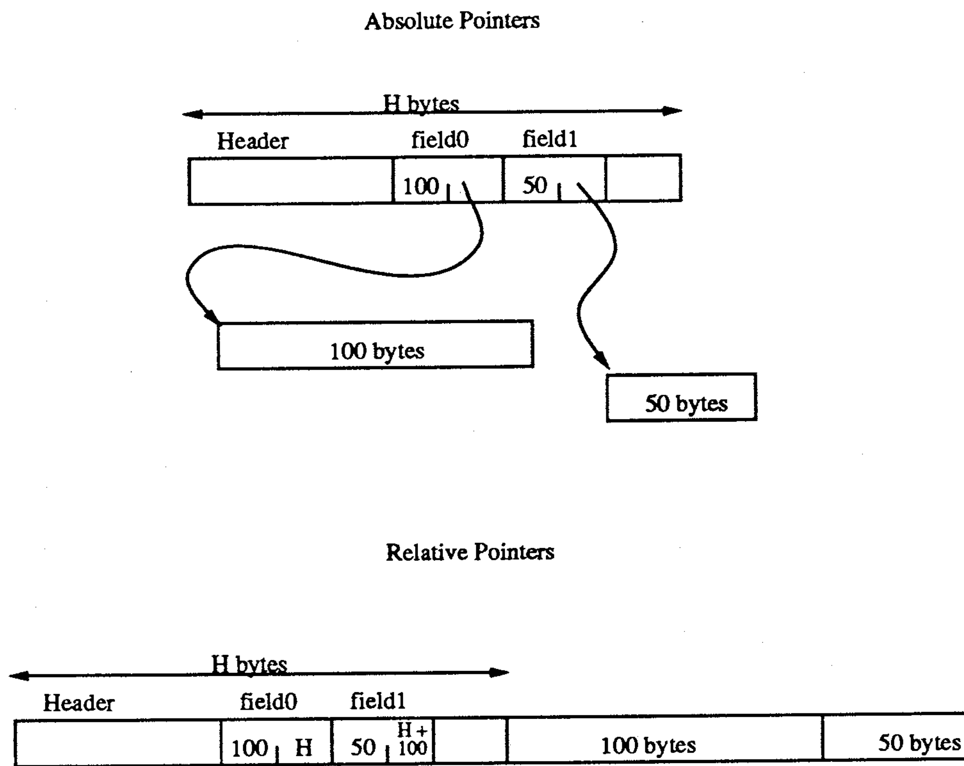

Figure 3.7: ptp using absolute vs. relative pointers to aggregates.

hash_type* The storage paradigm used for this operation (queue, hash, etc.)

hash_field The index of the tuple element serving as a key, if any.

copy_flag Indicates whether or not any data copying will be needed following a match. Copying is necessary if the template contains any non-anonymous[16] formal fields. This flag saves us the cost of calling the copy routine if no copying is actually needed.

long_flag* Indicates the presence of aggregate fields. These fields often incur special handling, so it is useful to have a simple way to check for their presence.

rd_flag* Indicates that the set contains at least one `rd`. This is used by the tuple broadcast optimization, see Section 5.6.

start_line, debug_info These fields are for debugging, and allow tools to reconstruct the original source line of the Linda operation.

field This array is analogous to the `field` array in the `ptp`. It provides static information about each element, including the polarity and type.

3.5.2 Organization of tuples into Tuple Space

This section describes the data structures that organize `ptps` into a coherent Tuple Space.

Each of the storage paradigms (hash, hybrid, mess, queue) organize their `ptps` into buckets, each containing two singly-linked chains, one of tuples and another of templates. See Figure 3.9. For the hashing paradigms (hash and hybrid), chaining hash tables were used instead of collision-resolving, linear hash tables. There are several reasons for this:

No collision resolution is necessary. It is very common to install a number of tuples in Tuple Space with identical hash function inputs. Traditional collision resolution techniques work best when resolving collisions between distinct inputs that happen to map to the same hash value. These techniques would tend to break down when presented with large sets of identical inputs.

No rehashing is necessary. As Tuple Space grows, linear hash tables would eventually require some sort of table expansion, probably involving rehashing the entire table. Chaining hash tables have no fixed limit; they tend to degrade gracefully as the chains increase in length, in marked contrast to linear hash tables, which experience marked degradation in performance as the table fills. Rehashing the table is still possible in order to shorten the chains, but never required.

Grouping related data. Tuples or templates organized into a single bucket are generally closely related. This allows us to easily manipulate the entire set of tuples that have been mapped to one bucket. An example of this occurs during the rehashing optimization, when we want to send all tuples of a particular type to another node.

[16] Recall that anonymous formals are specified by `?<type>`. They require that the type match, but incur no data copying.

```
/* Static tuple structure.  Initialized by code generated by the compiler */
typedef struct {
  short  polarity; /* formal or actual */
  short  type;     /* data type of field */
  short  control;  /* field representation */
  short  class;    /* anonymous field? */
} ST_FIELD;

typedef struct {
  short  set_id;      /* set id number */
  short  op_type;     /* type of operation */
  void   (*inout_func)();  /* pointer to inout function */
  short  num_fields;  /* number of fields in tuple */
  short  length;      /* length of used part of ptp, in bytes (send length) */
  short  hash_type;   /* paradigm for tuple */
  long   hash_field;  /* field in tuple used for hashing */
  short  copy_flag;   /* copying needed (not used) */
  short  long_flag;   /* contains extra data */
  short  rd_flag;     /* contains rds (but no inout) */
  short  start_line;  /* line in which call occurs in source text */
  char   *debug_info; /* contains copy of call text (well, sort of) */
  ST_FIELD  field[NUM_FIELDS];
} ST_TYPE;
```

Figure 3.8: ST data structure.

Isolation. Chained hash tables have the feature that each chain is independent of all others. This allows easy concurrent access to a hash table, including insertions and deletions, so long as each access is mapped to a different chain. At present, distributed memory Linda does not involve any concurrent access, but future versions may. Another advantage of this isolation is (code) robustness; it sets up a logical "firewall" between each chain, so that problems in one hash chain will not tend to cause the entire table to self-destruct the way that a linear table will.

Uniformity. Chains can be used for all of the storage paradigms, including queue. This results in implementations of the paradigms that are very similar, allowing considerable code sharing, as well as simplifying maintainance.

The chain is traversed forwardly, with a back pointer maintained to facilitate deleting `ptp`s. This allows each `ptp` to contain only one pointer field. New `ptp`s are added to the head of the chains. This gives Tuple Space a decidedly last-in-first-out (LIFO) nature, which, although consistent with the semantics of Linda, can cause extreme unfairness when satisfying requests. Because of this, we chose to make the template chains FIFO by adding one pointer for each bucket that points to the end of the chain. When templates are added to such chains, they are added to the end in constant time using this pointer. This means that older requests tend to be fulfilled before newer ones.

Matching involves comparing a newly-arrived tuple or template against members of the opposite chain of a bucket. Normally, each queue set and each mess set consist of a single bucket. However, in the case of queue sets the length of the chain is not a matter of concern, as the first element of the chain will always match.

A hybrid set is represented by a private hash table containing a number of buckets all on one node, into which all tuples, and all templates with keys are hashed. One additional bucket is used to contain templates without a key. Arriving tuples are compared against all templates in the appropriate keyed bucket as well as the unkeyed bucket. Arriving templates *with* keys are compared against all tuples in the same (keyed) bucket; those *without* keys must be checked against all tuples in the entire table. See Figure 3.10 for a schematic of queue, mess and hybrid set structures. Fortunately, in most cases unkeyed templates will match the first tuple they encounter. Section 7.7 discusses hybrid sets in greater detail. The present implementation uses private hash tables with 128 buckets; this is large enough to spread tuples out, but small enough that exhaustive search is not prohibitively expensive.

All hash sets share one large common hash table, represented by a number of buckets on every node. We chose to use one large common table rather than a number of private tables (as was done with hybrid sets) for two main reasons. First, hash sets never require any exhaustive search, so the only drawback to a very large table is the memory required for the chain headers[17]. Secondly, different hash sets will have radically different key ranges, and thus impose very different loads on a hash table. Giving each set a private, standard sized table would result in some sets getting too much space and others too little. In order to be maximally flexible it is better to pool all the sets together into one large table. See Figure 3.11 for a schematic of the hash set structure.

[17] On the iPSC/2, 12 bytes per chain.

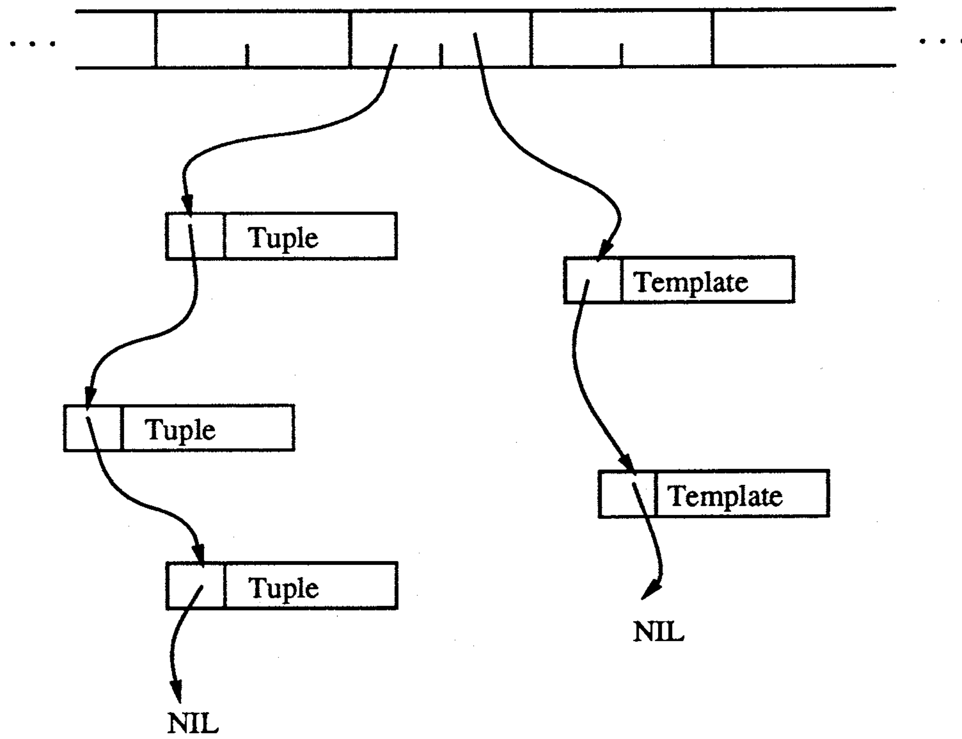

Figure 3.9: A bucket in Tuple Space.

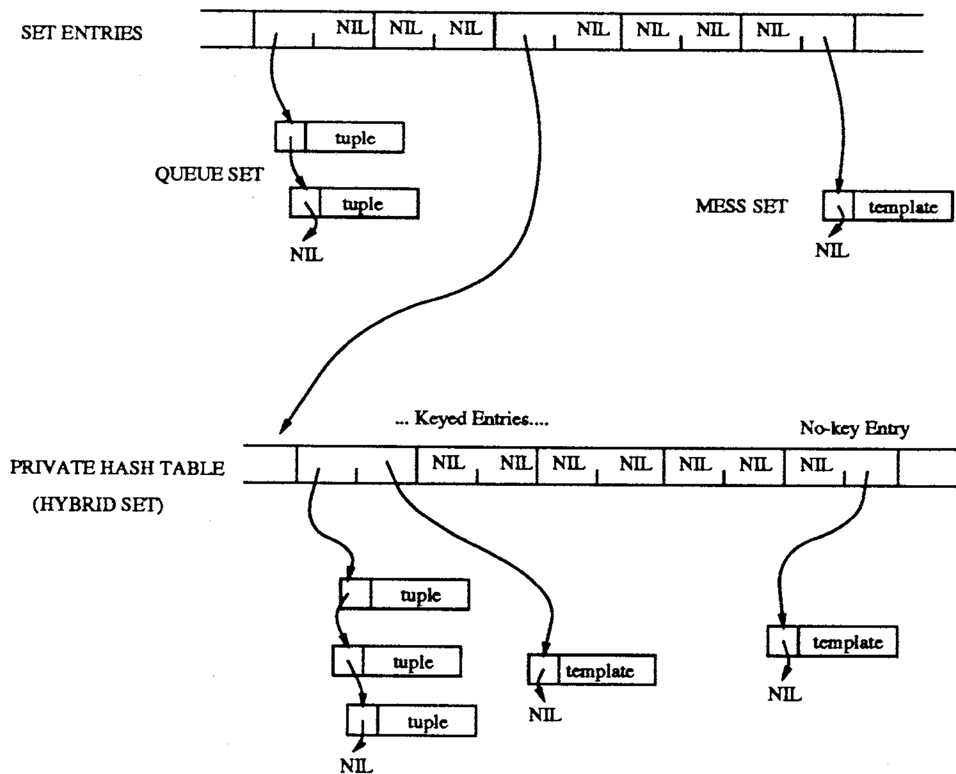

Figure 3.10: The queue, hybrid, and mess set Tuple Space structures.

3.6 Summary

To recapitulate, our DMM-Linda design consists of a number of server processes, each handling requests on some part of Tuple Space, and a number of client processes, each computing evals, during which they perform other Linda operations and make requests of various Tuple Space servers. This is a clean conceptual framework; its actual implementation on a particular machine may vary[18].

Tuple Space is distributed among all nodes of the system via a hash function that operates solely on characteristics of individual tuples and templates. This allows us to guarantee that searching for a match can be confined to a single node. We showed that a relatively simple hash function provides good scattering of tuples across the machine.

Finally we described the data structures that organize Tuple Space and provide efficient access to tuples. In the next chapter, we begin to examine the actual performance of the design as implemented on the Intel iPSC/2.

[18] For instance, on the iPSC/2, running multiple processes on a single node was considered impractical. There are two main reasons for this. First, the context switching overhead was considered excessive. Second, there is no fast method of communication between intra-node processes. At present, shared memory on a node is not supported, so processes have to communicate using message passing, which is roughly as expensive as inter-node message passing. The fact that intra-node communication is no faster than inter-node greatly reduces the benefits of recognizing and fostering locality, an issue in which we were very interested, making such a design unattractive from the point of view of experimenting with locality. Thus, we were faced with either dividing the nodes into server nodes and client nodes, or forcing a single process to serve as both Tuple Space server and client. The former might well have resulted in a cleaner design, but at the cost of considerable performance penalty, partly because some nodes would be permanently unavailable for eval computation (the real point of the whole exercise, after all), and partly because local Tuple Space access would not be possible (Tuple Space servers and clients would not share nodes, and thus clients could not benefit from locality of tuple reference.) The later option, combining Tuple Space server and client in one process (using an interrupt mechanism) resulted in a more complicated mechanism, but one that allows all nodes to take part in the computation, and allowed local accesses to Tuple Space to be optimized, as will be discussed in Chapter 5.

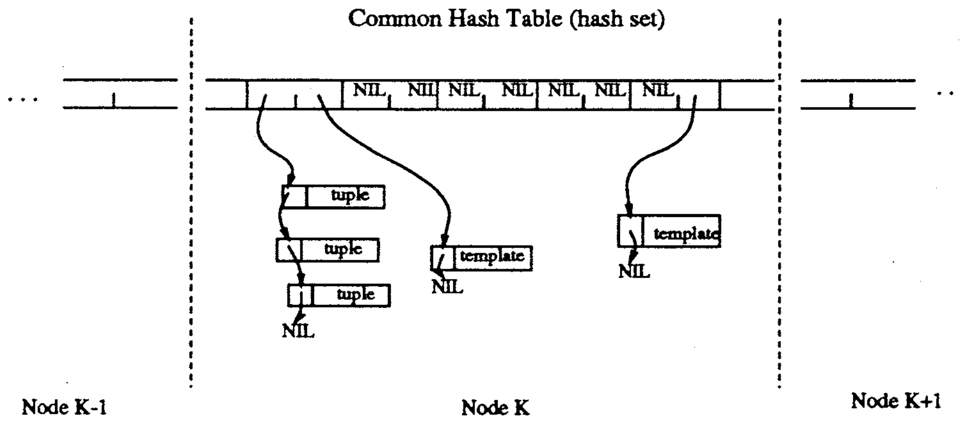

Figure 3.11: The Hash set Tuple Space structure.

Chapter 4

Basic performance measures

This chapter describes the low-level performance of iPSC/2 Linda. We give performance measurements for Linda operations on various types of tuples. We then dissect Linda operations into their component parts and show where time is consumed. Next, we give performance results for several simple programs, including a tuple ping-pong, a token ring, a logarithmic synchronization, and the Basic Linda Program (BLP). Finally, we look at the performance of `eval`.

We measure the cost of Linda operations in two different ways. By *Latency*, we mean the delay a node experiences between starting and ending a Linda operation. By *Load*, we mean the cost of an operation to a particular node's CPU. Since a Linda communication can involve multiple nodes[1], we are interested in the load incurred on each of them.

There are two important reasons why we measure both load and latency. Some of the latency may not show up as load, because it is incurred by DMA[2] instead of by the CPU. On the other hand, some of the load may not show up as latency, because it is work done in parallel (an example of this is preparing to receive a reply just after sending the request. See Section 5.2.2).

We will also be interested in the portion of the cost of a Linda operation not attributed to actual communication delays. This isolates the "Linda overhead", and represents the portion over which we (as Linda implementors) have the most control.

4.1 Single process communication latencies

In this section we present latencies for `rd`s, and for `out`s followed immediately by `in`s on various types of tuples. In the case of `rd`, the tuple was already present in Tuple Space. For these experiments, one process performed all of the communication on a 16-node cube (the rest of the nodes simply acted as Tuple Space servers). The size of the machine is relatively unimportant for this test since only one Linda operation is performed at a time,

[1] The normal distributed hashing paradigm typically involves three nodes, and some of the optimizations described in Chapter 5 can involve even more.

[2] Direct Memory Access

Operation		Latency (μsec) local	non-local
out("test")	in("test")	486	1470
out("test", i=1,1000)	in("test", ? i)	502	1490
out("test", j=1)	in("test", j=1)	582	1560
out("test", buf1:len)	in("test", ? buf2:len)		
$100 \leq len \leq 5000$		455+.46/byte	2230+1.01/byte
$len > 5000$		1806+.38/byte	2852+.30/byte
rd("test")		258	1000
rd("test", ? i)		267	1002
rd("test", i)		267	1008
rd("test", ? buf2:len)			
$100 \leq len \leq 5000$		294+.19/byte	1385+.45/byte
$len > 5000$		1574+.21/byte	2224+.22/byte
raw comm (small out and in)			1015
raw comm (small rd)			782

Table 4.1: Latencies for selected communications. (`i`, `j` are integers, `buf1` and `buf2` are byte arrays.)

so contention for rendezvous nodes is not an issue[3]. For the non-local tests, the rendezvous node was distinct from the node performing the operations, whereas for the local tests one node performed both functions.

Each communication was performed 1000 times in a tight loop. See Table 4.1. For both `out/in` and `rd` four different types of matches were tested:

- Minimal tuple requiring no matching and no copying at runtime.
- Single simple element copy.
- Single simple element match.
- Single aggregate element copy of varying length.

For further comparison, the raw communication costs of `out`ing and then `in`ing or `rd`ing a small tuple are included. These times were gathered by constructing a skeletal "mock-kernel" that performs exactly the same message passing events as the true Linda kernel without performing any of the non-communication actions of the Linda kernel. The difference between the two corresponds to the Linda processing overhead.

From Table 4.1 we can see that the Linda processing overhead for a small `rd` is $\approx 220\mu$sec, and for an `out/in` pair $\approx 450\mu$sec. The Intel 80386, the CPU used in the iPSC/2, has a peak rating of 4 million instructions per second (MIPS); at that rate the Linda overheads correspond to 880 and 1800 instructions, respectively. The actual instruction counts are probably somewhat lower. The times correspond well with the cost of local

[3]The cost of non-local communications will tend to rise slowly as the machine grows, adding a factor of about $30 \log_2 n$ μsec for an `out/in` pair.

Operation	Rend. Node Load (μsec)
out("test")	607
rd("test")	749
in("test")	771
out("test", bufl:len)	
$100 \leq len \leq 5000$	811+.18/byte
$len > 5000$	690+0.0/byte
in("test", ? bufl:len)	
$100 \leq len \leq 5000$	1268+.32/byte
$len > 5000$	1148+0.0/byte
rd("test", ? bufl:len)	
$100 \leq len \leq 5000$	1217+.32/byte
$len > 5000$	1066+0.0/byte
raw comm. (OUT)	481
raw comm. (RD)	649
raw comm. (IN)	649

Table 4.2: Loads for selected communications.

Linda operations, for which most of the communication overhead is eliminated. As tuple size increases, the Linda overhead becomes proportionally smaller, as one would expect. The discontinuity for tuples containing large data items is due to an optimization that will be explained in Chapter 5.[4]

4.2 Single process communication loads

In this section we examine the CPU load incurred by Linda communication on the rendezvous node. To gather this data, we performed the following experiment. The source node did repeated `ins`, `outs` or `rds`, separated by a short, calibrated delay loop in order to let the processing for one operation complete before beginning the next. To calculate the load on the rendezvous node, we had it perform a long calibrated loop twice, once while it was also handling the Linda communications generated by the source node, and once while no Linda operations were occurring. Handling the operations delayed the completion of the first loop; by measuring the difference between the two, we obtained the time spent handling Tuple Space requests. For comparison, we again include times for just the raw communication. For results, see Table 4.2. Again, the discontinuity for tuples containing large data items is due to an optimization that will be explained in Chapter 5. This optimization is also the reason that the per byte cost of `ins` and `rds` is zero for elements over 5000 bytes. By comparing the raw communication and Linda costs, we see that the Linda overhead in terms of CPU load at the rendezvous node is 100-120 μsec for small tuples.

[4]The reason for excluding array fields of less than 100 bytes from Table 4.1 is that the iPSC/2 demonstrates a performance discontinuity at 100 bytes. Messages below 100 bytes require one packet and acknowledgement, whereas messages over 100 bytes cause a circuit to be set up, almost doubling the communication startup cost. See Appendix A.

4.3 Dissection of a Linda communication.

In this section we break a typical Linda operation into its component parts, examining the cost of each part. Keep in mind that the cost of a Linda operation can depend on many variables; e.g. whether the tuple or the template arrives first.

Figure 4.1 shows the components of an `out`, and Figure 4.2 does the same for an `in`. A `rd` is almost identical to the `in`, the only difference being that the `free tuple` step is omitted. Both figures assume that a corresponding tuple/template is already waiting. If this isn't the case, the tuple or template is enqueued after the `perform match` step. The remaining steps will be performed when the matching template/tuple arrives. The time for the interrupt mechanism to fire is included in the sending time. We explain each step below:

Source node	**Rend. node**
build PTP (56)	
send PTP to rend. node (440)	
continues	
	interrupt recv. fires
	lookup tuple (24)
	search chain (43)
	perform match (35)
	free template (16)
	send reply (430)
	record match (26)
	check for requests (35)
	reset interrupt recv (156)
	return from interrupt (43)

Figure 4.1: `Out` broken down (μsec)

build PTP The PTP is converted into a package that can be conveniently sent by packing it together with any long data items found in the tuple or template. In C-Linda, long data items are either structures or arrays. The key value, if any, is copied in a known location for quicker access. The destination of the tuple or template is determined by invoking the hash function.

send PTP **to rend. node** If the rendezvous node differs from the sending node, a message send occurs; otherwise we simply call the handler directly after disabling interrupts. The time shown is for a non-local send.

prepare to recv. reply After sending the template to the rendezvous node, a node per-

Source node	**Rend. node**
build PTP (56)	
send PTP to rend. node (440)	
prepare to recv. reply (120)	
	interrupt recv. fires
blocks	lookup template (24)
	search chain (43)
	perform match (35)
	free tuple (16)
	send reply (430)
	record match (26)
poll for reply (36)	check for requests (35)
recv. reply (10)	reset interrupt recv. (156)
	return from interrupt (43)

Figure 4.2: `In` broken down (μsec)

forming an `in` or `rd` awaits the reply[5].

poll for reply We have to poll for replies, due to a quirk of the iPSC/2. The time shown is the minimum time, that for a single poll.

recv. reply Once the reply tuple for an `in` or `rd` is received, the data from the tuple is copied into the formal elements of the `in` or `rd`. The time shown is for a simple tuple with only one formal.

interrupt recv. fires In addition to computing some `eval`, each node always maintains an interrupt receive that waits for any kind of tuple request. When a message of the correct type arrives, the operating system interrupts the user's process and transfers control to the Tuple Space request handler.

lookup tuple/template Because the sending node may have had an outdated notion of where the tuple or template belongs, the correct destination is rechecked before handling the request. If the receiving node is no longer responsible for the tuple/template, it is forwarded to the new rendezvous node, and the sender notified of the change. The time shown is for the lookup only.

search chain This time accounts for the general activities of the request handler, excluding the actual matching, freeing, and recording (accounted for separately). Included

[5]Unfortunately, we have to poll actively on the iPSC/2. This is because the iPSC/2, once blocked awaiting a message, does not recognize some other events. In particular, messages arriving for interrupting receives are not handled. In our context this would mean that a node, once blocked on an `in`, would cease handling Tuple Space requests until the `in` succeeded, which would lead to deadlock problems.

here are such things as calling the appropriate handler, finding the correct chain, doing a prematch, looking for data to copy, etc.

perform match Of course, the actual time to perform a match varies according to the complexity of the tuple. This time is for a relatively simple tuple of three elements: a constant string and two integer elements.

free template/tuple This includes the time to extract the tuple/template from the linked list, and free it to the memory allocator. This time is for a storage cache "hit", see Section 5.4.

send reply The tuple is returned to the process that did the `in` or `rd`. Again, if the node determines that it is the recipient, we avoid actually doing the send.

record match The kernel keeps track of who is interested in each type of tuple. This routine updates the record each time a match occurs. The data collected will be used by the rehashing optimization described in Section 5.2.

check for requests The interrupt mechanism is fairly expensive, both to set up and to invoke. Instead of immediately returning from the interrupt once the request is finished, the interrupt handler checks to see if any additional requests have arrived in the meantime, and handles them immediately. Only when no more requests are pending does the handler reset the interrupt and return. This is in concord with the notion that Tuple Space handling should take priority over `eval` computation.

reset interrupt This reactivates the interrupt mechanism.

return from interrupt The node resumes computing its current `eval`.

4.4 Synthetic programs

In this section, we will present some simple programs that demonstrate tuple transmission speed and tuple contention. We will examine much more complex programs in Chapter 6; we choose to examine simple, synthetic programs first for two reasons. First, simple programs are easier to understand and dissect. We ought to be able to determine exactly what actions are performed and what they cost. Secondly, these programs are extremely communication intensive, doing essentially no computation. This means that the effects of communication will not be masked by computation, giving us a clearer picture of the communication and contention costs.

We begin by comparing Linda and native message passing versions of three communication patterns. In the first, `ping-pong`, two processes trade tuples back and forth. In the second, `ring`, a ring of processes trade a single tuple around the ring. In the third, `barrier`, a number of processes perform a barrier synchronization. We then examine the cost of several processes contending for a single tuple using a simple program called the Basic Linda Program. The results from this test will allow us to determine the coarsest granularity at which we would expect to see performance degradation due to communication. Finally, we examine the costs of Linda's `eval` operation.

These synthetic programs represent the worst case for Linda, in that they are easily and efficiently implemented in message passing and consist of only communication. We hope to show that the Linda versions are not unacceptably more expensive than message-passing. In general, since each Linda `out/in` pair requires three messages, the communication cost will be roughly three times that for message passing. Chapter 5 will discuss optimizations that bring the Linda cost closer to that for message passing.

4.4.1 Ping-pong

This program consists of three processes: a master, a pinger, and a ponger. The master `eval`s the pinger and the ponger. The pinger loops, creating **pings** and consuming **pongs**, while the ponger does the reverse. See Figure 4.3 for the Linda program, Figure 4.4 for the message passing version.

The Linda version took 2120 μsec per ping/pong, compared to 746 μsec for the native message passing version. The main reason for the difference is that each ping or pong requires three messages in Linda but only one with message passing.

The Linda time is considerably less than twice the single node `out/in` latency reported in Section 4.1. To see why, consider Figure 4.5, which shows how the pinger and ponger interleave. Based on this picture, we would expect the time for one complete ping/pong to be roughly the latency for one `out/in` pair, plus the load for one out. This sum, $1470 + 607 = 2077$, is indeed very close to the observed time.

4.4.2 Token ring

This program builds a ring of processes and passes a tuple around the ring. See Figure 4.6 for the Linda program, Figure 4.7 for the message passing version. The Linda version running on 16 nodes, and sending the tuple around the ring 1000 times took 1080 μsec per step. In comparison, the native message passing program took 395 μsec. The timeline for this program is similar to that for ping-pong, in that there is considerable concurrency among Linda operations. Again, forwarding the tuple along each segment of the ring requires three messages for the Linda program, versus one for the native program. Section 5.2.2 examines the dynamics of this program in more depth.

4.4.3 Barrier

As the last synthetic program, we turn to a barrier synchronization. A barrier is a standard parallel programming tool; upon reaching the barrier, each process blocks. When all processes have reached the barrier, all are free to continue.

An efficient way to implement a barrier on hypercubes is using two spanning trees, one fanning in to collect blocked processes, and one fanning out to free the processes. This requires $2(n-1)$ communications, but each spanning tree is only $\log_2 n$ in height. See Figure 4.8 for the Linda program. We do not present the message passing version of this program.

The barrier was performed 10000 times on various numbers of nodes. Two non-Linda variants were also run for comparison. In one, we replaced the Linda communications with

```
real_main(argc, argv)
int argc;
char *argv[];
{
  int loops;
  int ping(), pong();

  loops = atoi(argv[1]);

  start_timer();
  eval("ping", ping(loops));
  eval("pong", pong(loops));

  in("ping", ? int);
  in("pong", ? int);

  timer_split("done.");
  print_times();
}

ping(loops)
int loops;
{
  while (loops--) {
    out("ping");
    in("pong");
  }
}

pong(loops)
int loops;
{
  while (loops--) {
    in("ping");
    out("pong");
  }
}
```

Figure 4.3: Ping_pong.cl

```
#include <cube.h>
#define LOOPS 10000

main()
{
  int node;
  int i;
  int dummy;
  int loops;

  loops = atoi(argv[1]);

  node = mynode();
  start_timer();
  if (node == 0) {
    for (i=0; i<loops; ++i) {
      csend(0, &dummy, sizeof(dummy), 1, 0);
      crecv(0, &dummy, sizeof(dummy));
    }
    timer_split("done");
    print_times();
  }
  else if (node == 1) {
    for (i=0; i<loops; ++i) {
      crecv(0, &dummy, sizeof(dummy));
      csend(0, &dummy, sizeof(dummy), 0, 0);
    }
  }
}
```

Figure 4.4: Ping_pong.c

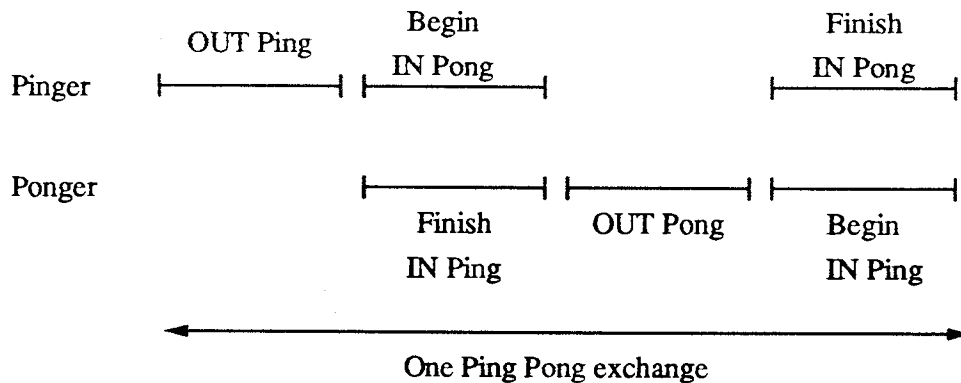

Figure 4.5: Timeline for Ping_pong.

iPSC/2 native message passing. In the other, we simply used Intel's **gsync** library call, which is an optimized barrier.

Figure 4.9 compares the performance of the three versions on machine sizes ranging from 4 to 64 nodes. When plotted on a logarithmic axis, the time per synchronization is close to linear due to the logarithmic cost complexity of the algorithm. The Linda cost is consistently about three times higher than the message passing version, as expected. Section 5.2.2 will show an optimized Linda result closer to that for message passing.

The Linda barrier takes roughly 1290 μsec per edge of the spanning tree. This communication is actually quite similar to one step of the ring program, in that the **in** occurs before the **out** (and is partly not on the critical path; see Section 5.2.2). The ring program took 1070 μsec per step; the difference is due to contention, since only one node communicates at a time in the ring program, but many communicate at the same time in the barrier. The contention arises for two reasons. First, the spanning tree was not mapped to be "nearest neighbor" on the hypercube, although it could be. Secondly, each tuple's rendezvous node is at an arbitrary location, necessitating multihop message sending. For both of these reasons, contention occurs in the Linda version of the barrier. Section 5.2.2 closely examines the optimized performance of the Linda program.

4.4.4 BLP

The Basic Linda Program (BLP) is a standard Linda benchmark used to estimate the smallest reasonable granularity supported by a particular Linda implementation. It simulates a master/worker program with varying task granularity.

The master creates a number of workers. The workers **in** a task tuple and mimic performing the task by looping for some determined period (the "granularity") and then **out**ing a result tuple. The master begins by loading Tuple Space with an initial allotment of task tuples and then loops, consuming one result and creating another task, until all

```
#define next(i) (i+1<ring_size?i+1:0)

real_main(argc, argv)
int argc;
char *argv[];
{
  int i, len;
  int ring_size;
  int loops;
  int ring();

  loops = atoi(argv[1]);
  ring_size = numnodes();
  for (i=1; i<ring_size; ++i) eval(ring(ring_size, loops, i));
  start_timer();
  out("token", next(0));
  ring(ring_size, loops, 0);
  timer_split("done");
  print_times();

}

ring(ring_size, loops, id)
int ring_size, loops, id;
{
 int i;
 int next_id;
 next_id = next(id);
 for (i=0; i<loops; ++i) {
    in("token", id);
    out("token", next_id);
  }
}
```

Figure 4.6: Ring.cl

```
#define LOOPS 10000

main()
{
  int id, my_node, dummy, num_nodes, next_node, i, size;
  char buff[2000];

  size = 4;
  my_node = mynode();
  num_nodes = numnodes();
  next_node = (my_node+1) % num_nodes;

  /* start the ball rolling */

  if (my_node == 0) {
    csend(1, buff, size, next_node, 0);
    start_timer();
  }

  /* do ring */
  for (i=0; i<LOOPS; i++) {
    crecv(1, buff, size);
    csend(1, buff, size, next_node, 0);
  }

  /* finish up */
  if (my_node == 0) {
    timer_split("done");
    print_times();
  }
}
```

Figure 4.7: Ring.c

```
real_main(argc, argv)
int argc;
char **argv;
{
  int nodes, id, max_dim = 0, num_synchs;
  nodes = numnodes() - 1;
  num_synchs = atoi(argv[1]);

  /* determine machine dimension */
  while(nodes >> max_dim) max_dim++;

  start_timer();
  for (id = 0; id < nodes; id++)
     eval("worker", worker(nodes, id, max_dim - 1, num_synchs));
  worker(nodes, nodes, max_dim - 1, num_synchs);

  for (id = 0; id < nodes; id++) in("worker", ? int);
  timer_split("done");
  print_times();
}

worker(nodes, id, max_dim, num_synchs)
int nodes, id, max_dim, num_synchs;
{
  int j;
  for (j = 0; j < num_synchs; j++) synchronize(nodes, id, 0, max_dim);
}

synchronize(nodes, id, dim, max_dim)
unsigned nodes, id, dim, max_dim;
{
  if ((id >> dim) & 01) {
      /* if my dim-th bit is set, get a message from my counterpart
         with that bit clear, then recurse. */
      in("fanning in", (id & ~(01 << dim)));
      if (dim < max_dim) synchronize(nodes, id, dim + 1, max_dim);
      out("start fanning out", (id & ~(01 << dim)));
  }
  else {
      out("fanning in", id);
      in("start fanning out", id);
  }
}
```

Figure 4.8: Barrier.cl

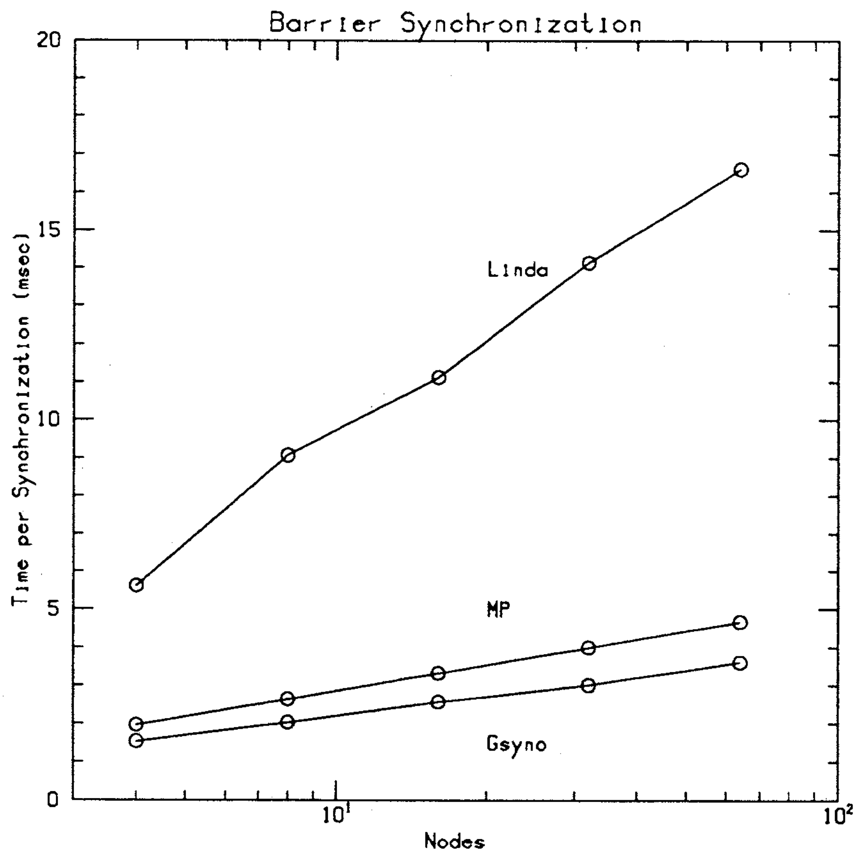

Figure 4.9: Barrier Synchronization: Cost per synchronization versus machine size for Linda, message passing, and Gsync versions.

tasks are complete. See Figure 4.10 for the Linda program[6].

By varying the task size, we can determine the point at which contention for task tuples arises. For instance, we can determine the point where efficiency is reduced to 50%[7].

Figure 4.11 shows the total execution time required for 5000 tasks of varying size, using different numbers of processes. The dotted line shows the ideal time, i.e. total "work" divided by the number of workers. This time rises to the right, since we are holding the number of task constant at 5000 and increasing the task size.

Figure 4.12 shows the average communication overhead for a task. By overhead, we mean the time it takes a worker to complete a task, over and above the time to perform the loop simulating the work. This overhead is due to the communication for the task and result tuples, and rises as contention increases. The overhead remains constant at roughly 1.5 msec until contention arises, at which point it rises linearly. Figure 4.13 shows the efficiency for various task sizes. Notice how efficiency drops sharply as contention arises. On the iPSC/2, we reach 50% when the task size is roughly $600w$ μsec, where w is the number of workers.

4.5 Performance of Eval

Obviously, the time needed to compute an **eval** is dependent on the functions it calls. We are interested in the additional overhead incurred by an **eval**, as compared to calling the function in-line and outing the result. In order to determine that overhead, we ran a program that **eval**ed and **out**ed tuples containing calls to a trival function **f()** and then **in**ed them again. Table 4.3 presents the results. From this data, we can see that on the iPSC/2 an **eval** has a base overhead of about 2.2 msec over the cost of an equivalent **out**, plus 2 msec per function call. The number of parameters to a function is not very significant to performance. Appendix D details the implementation of **eval**; essentially, it involves a transformation of the **eval** into a number of **in**s and **out**s. To be precise, we use $2n+3$ **out**/**in** pairs for an **eval** with n function calls.

In order to determine the minimum reasonable granularity for an **eval**, we turned once again to the BLP. This time, instead of **out**ing task tuples, we simply **eval**ed the mock task directly. See Figure 4.14 for the program. Figure 4.15 shows the results for a 32 cube. Notice how the efficiency drops as the task size decreases; compare with Figure 4.13. On 32 nodes, the granularity at which 50% efficiency is reached is about 46 msecs, or 1.44 msec/node. This compares with 600 μsec for **out**, indicating that the finest reasonable granularity for an **eval**ed task is roughly 2.5 times greater than for an **out**ed task.

[6] When this program was first run, we noticed a very strange behaviour: some workers received no tasks, while others received theirs with no apparent contention. As the granularity was decreased, the number of "starving" workers increased. This marked unfairness was due to the way templates were stored in Tuple Space: last-in-first-out queues. As contention arose, some templates languished at the bottom of the queue, and the workers that had issued them starved. Partly motivated by this experience, we chose to store templates first-in-first-out, making **in**s and **rd**s much fairer.

[7] Efficiency equals *speedup/#workers*.

```
real_main(argc, argv)
     int argc;
     char *argv[];
{
  int i, num_workers, task_size, start, tasks;

  task_size = atoi(argv[1]);
  tasks = atoi(argv[2]);
  num_workers = atoi(argv[3]);
  start = atoi(argv[4]);

  out("task size", task_size);
  for (i = 0; i < num_workers; ++i) eval("worker", worker());
  for (i = 0; i < num_workers; ++i) in("started");

  start_timer();
  /* seed Tuple Space */
  for (i = 0; i < start; ++i) out("task", 0);
  /* main loop */
  for (i = 0; i < tasks-start; ++i) {out("task", 0); in("result"); }
  /* cleanup */
  for (i = 0; i < start; ++i) in("result");
  for (i = 0; i < num_workers; ++i) out("task", 1);
  timer_split("done");
  print_times();
}

worker()
{
  int stop, task_size;
  rd("task size", ? task_size);
  out("started");
  while(1) {
    in("task", ? stop);
    if (stop) break;
    /* this is a calibrated delay loop */
    delay(task_size);
    out("result");
  }
}
```

Figure 4.10: Blp.cl

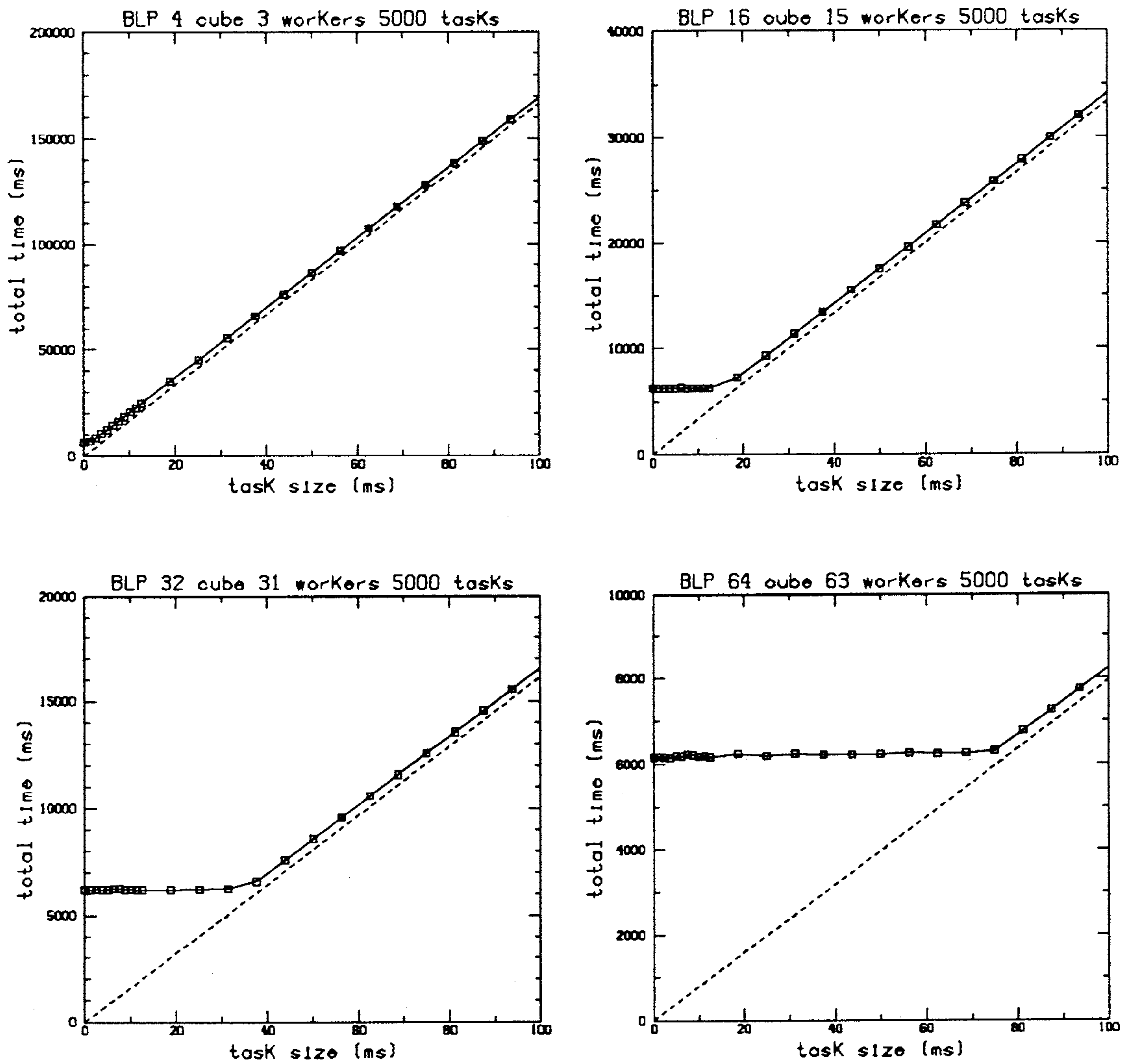

Figure 4.11: BLP: total execution times versus task size for various numbers of workers.

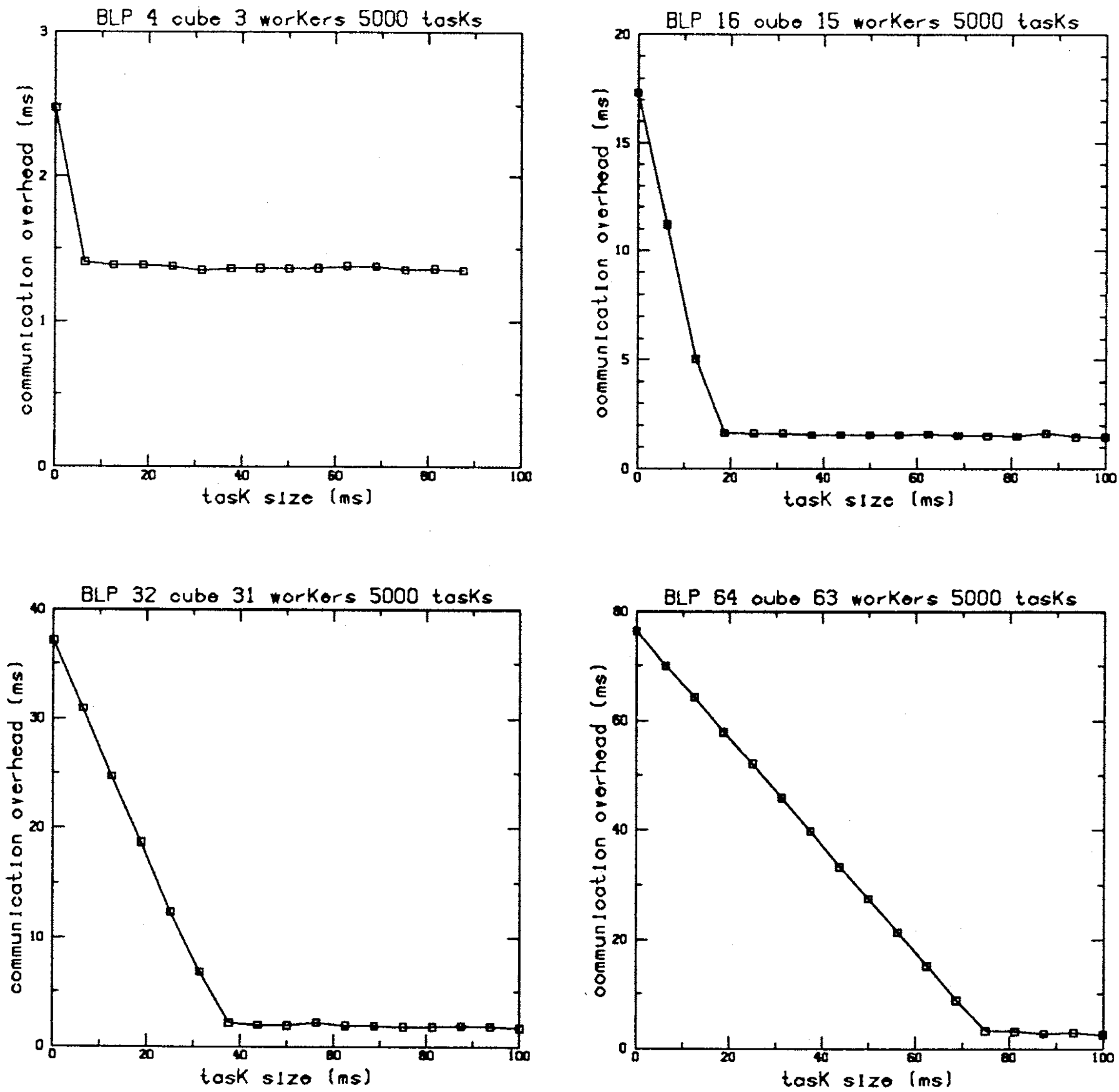

Figure 4.12: BLP: communication overhead per task versus task size for various numbers of workers.

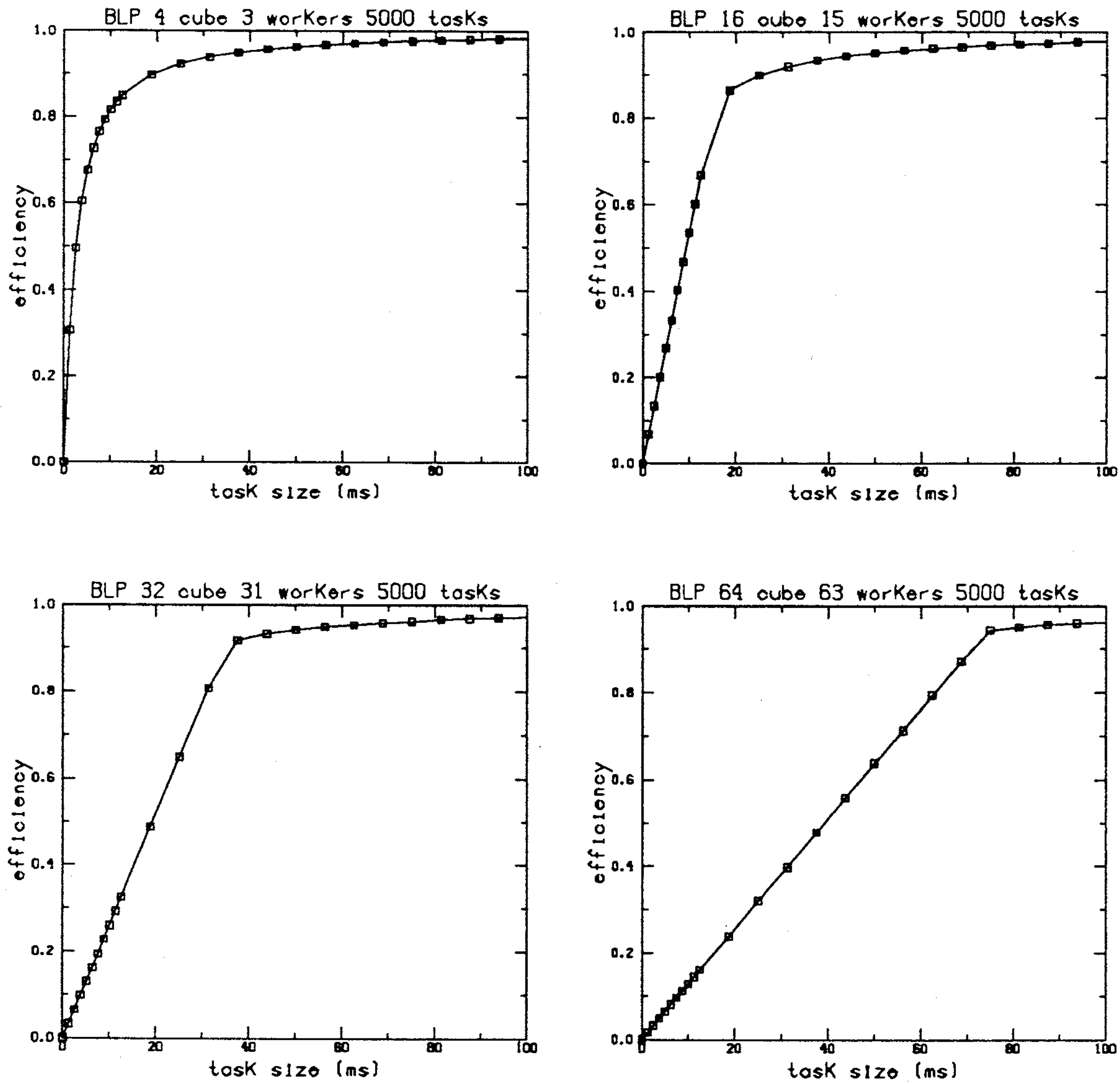

Figure 4.13: BLP: efficiency versus task size for various numbers of workers.

```
real_main(argc, argv)
     int argc;
     char *argv[];
{
  int i, task_size, tasks, start;

  task_size = atoi(argv[1]);
  tasks = atoi(argv[2]);
  start = atoi(argv[3]);
  start_timer();
  /* seed Tuple Space */
  for (i = 0; i < start; ++i) eval("result", task(task_size));
  /* main loop */
  for (i = 0; i < tasks-start; ++i) {
    eval("result", task(task_size));
    in("result", ? int);
  }
  /* cleanup */
  for (i = 0; i < start; ++i) in("result", ? int);
  timer_split("done");
  print_times();
}

int task(task_size)
{
  int task_size;
  /* this is a calibrated delay loop */
  delay(task_size);
}
```

Figure 4.14: Blpeval.cl

Operation	time (msecs)
`eval(f()); in(?i)`	5.74
`eval(f(i)); in(?i)`	5.74
`eval(f(i,i)); in(?i)`	5.74
`eval(f(), f()); in(?i, ?i)`	7.76
`eval(f(), f(), f()); in(?i, ?i, ?i)`	9.96
`out(f()); in(?i)`	1.52
`out(f(i)); in(?i)`	1.52
`out(f(i,i)); in(?i)`	1.52
`out(f(), f()); in(?i, ?i)`	1.56
`out(f(), f(), f()); in(?i, ?i, ?i)`	1.58

Table 4.3: Comparison of costs of **eval** and **out**.

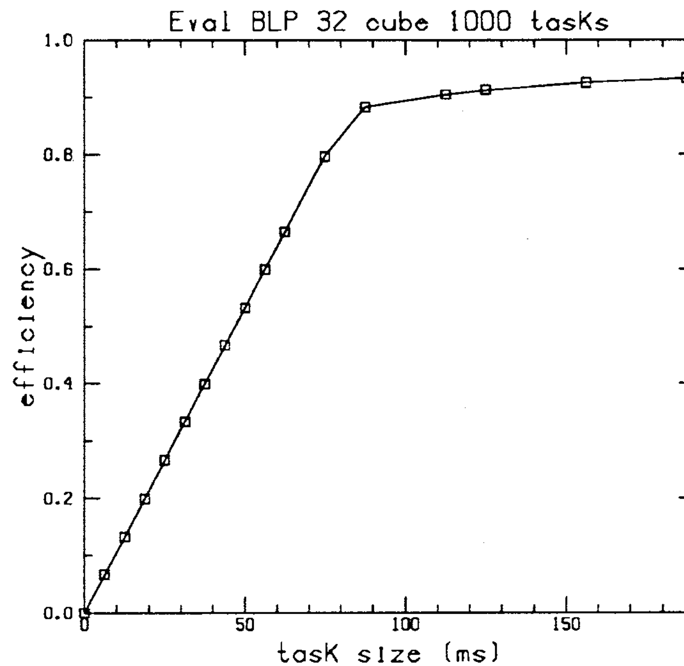

Figure 4.15: BLP using **evals**. Efficiency versus task size (on 32-cube).

4.6 Summary

This chapter investigated the performance of the unoptimized kernel running on the iPSC/2, by timing individual operations in isolation, and also in small synthetic programs. We saw that the Linda overhead (apart from communication) on the source node was about 200 μsec; the cost to the rendezvous node was less, about 100 μsec. These times are considerably less than the communication startup times on the iPSC/2, and thus should be acceptable. As long as the processor speed increases at least as fast as the interconnect speed, the actual communication portion of a Linda operation will dominate the total cost. Intel's next generation machine supports this view; the Intel iPSC/860 hypercube [HGD90] has only slightly faster communication but much faster CPUs. Likewise, the performance of workstations has far outpaced the speed of their interconnects.

In Chapter 5 we will take another look at a number of the synthetic programs, in order to see the effect of a number of optimizations. Many of these optimizations are designed to reduce the amount of communication necessary to support a Linda operation.

Chapter 5

Optimizations

5.1 Introduction

In previous chapters, we described the design of a Linda runtime system for DMM's and tested its performance on simple synthetic programs. In this chapter, we will present several modifications (or optimizations) upon the basic design that improve performance substantially. We will give performance figures showing the improvement achieved on basic operations, as well as some of the synthetic programs presented in Chapter 4. In the next chapter, we will show each optimization's effect on a number of applications.

The six optimizations presented are:

1. Tuple rehashing
2. In/Out collapse
3. Memory caching
4. Local data caching
5. Tuple broadcast
6. Randomized tuple bag

5.2 Tuple rehashing

As described earlier, the purpose of hashing tuples to nodes of the machine is to spread them out evenly, in order to equalize the load on each node and prevent hot spots. The hash function previously described maps the tuples arbitrarily; no attempt is made to map tuples to any particular node. In some cases, however, placing tuples more deliberately can result in improved performance, for example when a particular type of tuple is consumed primarily by one process. By a particular type, we mean the unit hashed by the hash function, i.e. an entire queue, hybrid, or mess set, or, in the case of a hash set, one particular hash key.

The runtime kernel can keep track of the identity of consumers of each type of tuple; if it detects that a particular type is consumed by only one process, it can amend the hash function, rehashing that tuple type to the consumer node. Assuming that the guess was right, and the same process will continue to consume that tuple type, further operations on this type will only require communication for outs, since in will be local, and two messages will be saved.

The protocol is as follows: Each Tuple Space server maintains two tuple/consumer association tables. One is of fixed size, and contains an entry for each queue, hybrid, or mess set. We only need one entry per set on each node since the entire set will be remapped as a unit. The other table is shared by all hash sets. It is initially large, and can grow. It contains one entry for every set-key pair ever seen for any hash set, allowing us to rehash each key separately. Since keys have unpredictable values, this table is managed as a linear probing hash table, using the set and key as inputs. The queue/hybrid/mess and hash association tables have identical entries: each contains three fields: the identity of the last consumer (`last`), the number of consecutive `ins` from the same consumer without intervening `ins` from any other process (`count`), and the current Tuple Space server owning that set (`owner`). Each time a server satisfies an `in`, it updates the entry as follows (in pseudo C):

```
if (consumer == LAST) {
      COUNT = COUNT + 1
      if (COUNT >= THRESHOLD) {
            rehash the set to LAST
      }
}
else {
      COUNT = 1
      LAST = consumer
}
```

The table actually performs two distinct but closely related functions. First, it accumulates data on the consumers of tuples. Only the entry in the current owner is used for this purpose. In addition, the table records rehashing events. Before a node sends a request, it will first check the table to see if that particular tuple type has been rehashed. This, in contrast, occurs on all Tuple Space servers, not just the owner.

The process of actually rehashing a set is as follows:

- After servicing the `in` or `out` that initiated the rehashing event the old server notifies the new server of the change. At this point, the new server becomes the official owner of that type.
- The old server forwards any other tuples or templates remaining to the new server.
- As subsequent requests arrive at the old server, they are forwarded to the new server. In addition, a notification message is sent immediately to the requester, informing it of the change.

5.2.1 Cost of rehashing

This protocol has several costs. Whenever a tuple is generated, the table is consulted, and each time a tuple transaction is completed, the table entry must be updated. This is relatively cheap, costing about 40 μsec for the hash paradigm on the iPSC/2, about 10% of the cost of sending a message. The cost is much less for paradigms other than hash, since they avoid the overhead of looking up the entry in the hash table. [1]

The tables can also consume considerable memory. Only one entry is required for each queue, hybrid, or mess set, but a hash set may make unbounded demands, as each key value is tracked separately. The simplest solution to this problem would be to restrict the table size; additional key information would be discarded.[2] An alternative would be to discard old entries. Only entries that had never been rehashed could be discarded, however; the entries for rehashed tuples serve as forwarding pointers and are logically necessary. In fact, hash sets generating huge sets of keys are probably not very amenable to rehashing optimization, since any given key is not encountered many times. Therefore, restricting the size of the table would probably not harm the optimization very much.

The other main cost of the protocol is the cost of rehashing a tuple. In the following discussion we will measure the cost in terms of the number of messages generated, ignoring any additional computation involved. This is reasonable, since the computation is small compared to the cost of a message. The total cost of a particular rehash is the cost of notifying the new rendezvous node, plus forwarding all the tuples currently present, plus the cost of all eventual request forwarding and return notifications for this rehash.

We are interested in the worst case amortized cost of rehashing, i.e. the additional number of messages divided by the number of requests. We shall examine the initial cost of forwarding tuples separately from the costs eventually generated by requests to the rehashed set.

Under the current protocol, the amortized cost of forwarding the tuples cannot be bounded, since a constant number of **ins** could cause an unbounded number of tuples to be forwarded.

Giving a bound on the costs generated by requests is more complicated. As stated above, this cost has two components; forwarding requests that have been sent to old rendezvous nodes, and returning notification messages to nodes that made the out-of-date requests. For the protocol we used, the two costs are symmetric; each forward generates one notification.

One might imagine that amortized cost of handling requests is a constant, since (clearly) each request either requires a notification or not, the worst case having each request generate a notification. In reality things are not quite so simple, since the "lazy" notification scheme we use can allow long forwarding chains to build up. A node might continue to be informed of a particular rehash long after the set has been rehashed yet again, even if the node never makes a reference to the set itself. In Figure 5.1, after many rehashes, we see

[1] We could reduce the cost of looking up hash entries in the association table, at the price of slightly reduced benefit from the optimization, by not resolving collisions. Both consumer and rehashing information can be safely discarded; without the consumer information the set will simply not be rehashed, and without the rehashing information tuples will continue to be sent to old locations.

[2] This is the approach that we used on the iPSC/2. The rehashing threshold was 3.

that a chain of forwarding pointers connects the old rendezvous nodes (A-D) to the present one, labeled E. 1 is ignorant of any of the changes, and its request will be forwarded four times. After the forwarding and notification procedure takes place, the situation will be as shown in Figure 5.2. Notice how nodes A-C are informed of an old change, even if they never actually requested that tuple set. However, there is good news as well: each node's distance to the rendezvous node has been halved. A given node can never be more than $n - 2$ steps out of date, and $\log_2 n$ requests will bring it up to date.

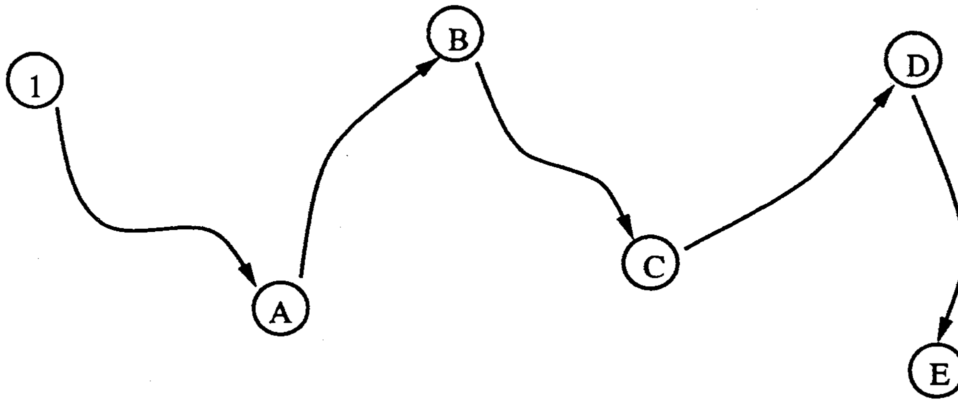

Figure 5.1: Example forwarding chain before notification.

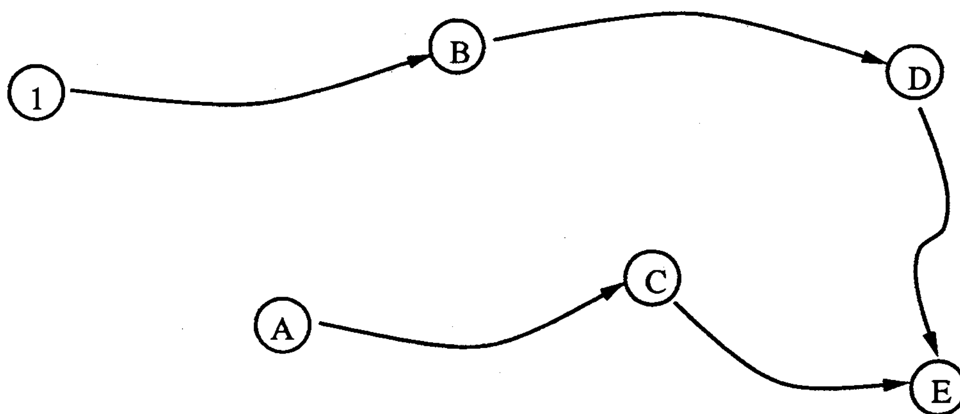

Figure 5.2: Example forwarding chain after notification.

In fact, the true situation is somewhat worse, since outs are asynchronous. Working quickly, a node could continue to generate or forward several requests before the return notification finally arrived. The rendezvous node should keep track of the nodes it has notified, in order to avoid generating multiple notifications for each request coming from the same node.[3]

To summarize, in theory the rehashing method we implemented cannot be bounded in the number of messages for three reasons:

- When rehashing a set, no attention is paid to the number of tuples already present, each of which will have to be forwarded to the new rendezvous node.
- Because outs are asynchronous, a node may manage to generate large numbers of out-of-date requests before receiving a correction.
- No record is kept of notifications already made. If several similar out-of-date requests are made by the same node, each will be notified separately.

Despite the lack of a good worst case bound, empirically, the current heuristic works well and is simple. Section 5.2.3 makes two reasonable assumptions and proves a worst case bound.

Several things could be done to improve its typical performance. At present, a tuple type may be rehashed again and again, regardless of the savings realized. Some hysteresis is desirable. One solution is to increase the rehashing threshold on each subsequent rehash. More interesting would be to keep an additional statistic on each rehashed type: the number of local vs. non-local requests, indicating whether the *last* rehash paid off. If not, the system could be more suspicious about yet another rehash.

Another weakness of the current implementation is that, since data is kept separately for each key value of a particular hash set, no overview of the set as a whole is possible. Often, although each key is used by a different process, the general behavior of the entire set is very similar in terms of amenability to rehashing. Keeping track of the set as a whole might allow for a more intelligent response to new or infrequent keys.

5.2.2 Performance

The simplest measure of performance improvement due to rehashing is to compare the latency of a local `rd` vs. a non-local `rd`. (Ins are only slightly more expensive in both cases.) These times were presented in Figure 4.1. For simple tuples, the latency was reduced from 1000 to 258 μsec. It proved difficult to design a simple program that would truly reflect the performance of the rehashing optimization. For instance, the ring program described in Chapter 4 actually showed only an 11% improvement in performance; see Table 5.1.

The reason is connected to the concept of ***critical path***. The critical path consists of those actions that contribute to the total latency of the computation. Activities that are not part of the critical path are concurrent to actions in the critical path, and do not

[3]Our iPSC/2 implementation did not do this.

Message Passing	395 μsec
Communication only	680 μsec
Linda with rehashing	959 μsec
Linda without rehashing	1075 μsec

Table 5.1: Performance of ring.cl with and without rehashing. Times are for each hop. The program was run for 10000 circuits on a 16 cube. The time for communication only corresponds to performing just the communication that the (rehashing) Linda program performs.

contribute to the latency. In this case, the critical path consists of all of the processing and sending involved with the `out`, but only some of the processing and sending involved with the `in`. For example, the packaging and sending of the `in` is done while the token is being passed further down the ring, and is thus out of the critical path. Because of this, the advantage of performing this send locally is not reflected in the critical path. In addition, several calls, including the resetting of the interrupt receive, are included in the critical path for local `in`s, but not for non-local `in`s. This cost largely offsets the advantage gained by eliminating the reply communication. Although the difference in latency is quite small, there are real benefits in terms of the load. The activities not on the critical path are still consuming CPU cycles, which would be important in an real application.

Table 5.2 contrasts the two execution paths, with components not on the critical path shown in italics. Events directly associated with communication are underlined. The predicted times agree quite well with the observed behaviour.

In order to test the benefit to load of the rehashing optimization we used a revised version of the ring program. In this version, each process initially dumps a token into Tuple Space. On a 16-cube, there will be 16 tokens traveling around the ring, rather than just one. See Figure 5.3. This keeps each node busy most of the time, and is thus a more realistic test. We ran the test on a 16-cube, letting each token travel around the ring 1000 times. This means that each node passed on 16,000 tokens, or 256,000 total. Table 5.3 presents the results. This test shows substantial benefit from rehashing, reducing the hop time by almost 50%. These times are higher than the previous times because there is no longer much opportunity for concurrency; the critical path encompasses almost all of the CPU effort.[4]

As another simple test, we tried the barrier synchronization described in Section 4.4.3. Figure 5.4 compares the performance of the Linda version with and without rehashing to the message passing and `gsync` versions on machine sizes ranging from 4 to 64 nodes.

Again, on 32 nodes each barrier is 10 communications in length so the optimized Linda program is taking 1010 μsec per communication, vs. 323 μsec for message passing. To see why, refer again to the optimized time-line in Figure 5.2. Of the total estimated 945

[4] We also decided to check for link contention by mapping the ring to the hypercube using gray codes, such that ring neighbors were nearest neighbors in the hypercube. This would guarantee that the rehashed ring program would send each tuple the shortest distance possible, and would not have to complete for that link. The program ran 1% faster, which is attributable to just the shorter path lengths. Thus, on 16 nodes at least, link contention does not seem to be a factor for this program, which is extremely communication intensive.

With Rehashing (local in)

Outing node	**Ining (and rend.) node**
build PTP (56)	
send PTP to rend. node (440)	
	interrupt recv. fires
	lookup tuple (24)
	search chain (43)
	perform match (35)
	free template (16)
	send reply (local) (25)
	record match (26)
	check for requests (35)
	reset interrupt recv.(156)
	return from interrupt (43)
	poll for reply (36)
	recv. reply (local) (10)
	Total (945)

Without Rehashing (non-local in)

Outing node	**Rend. node**	**Ining node**
build PTP (56)		
send PTP to rend. node (440)		
	interrupt recv. fires	
	lookup tuple (24)	
	search chain (43)	
	perform match (35)	
	free template (16)	
	send reply (non-local) (430)	
	record match (26)	poll for reply (36)
	check for requests (35)	recv. reply (10)
	reset interrupt recv.(156)	
	return from interrupt (43)	
		Total (1090)

Table 5.2: A comparison of the critical paths during one step of the token ring program. The underlined steps are those directly involving communication, those in italics are not part of the critical timing path.

```
#define next(i) (i+1<ring_size?i+1:0)

real_main(argc, argv)
int argc;
char *argv[];
{
  int i, len;
  int ring_size;
  int loops;
  int ring();

  loops = atoi(argv[1]);
  ring_size = numnodes();
  loops *= ring_size;
  for (i=1; i<ring_size; ++i) {
    eval(ring(ring_size, loops));
    in("live");
  }
  start_timer();
  ring(ring_size, loops);
  timer_split("done");
  print_times();
}

ring(ring_size, loops)
int ring_size;
int loops;
{
  int id;
  int i;
  id = mynode();
  out("live");
  out("token", next(id));
  for (i=0; i<loops; ++i) {
    in("token", id);
    out("token", next(id));
  }
}
```

Figure 5.3: Multitoken-ring.cl

With rehashing	1486 μsec
No rehashing	2780 μsec

Table 5.3: Performance of variation on ring.cl with and without rehashing. Times are for each hop. The program was run for 1000 circuits on a 16 cube.

μsec latency, only 210 μsec are actually Linda overhead.[5] The remaining 735 μsec are costs associated directly with communication; these costs are double that for a simple message send because we are forced to use the inefficient interrupt-receive mechanism on the iPSC/2.

5.2.3 An improved rehashing analysis

In this section we show that, by making two additional assumptions about the rehashing mechanism, we can bound the worst case amortized cost of rehashing by $2 + 2n/k$, where n is the number of nodes, and k the rehashing threshold. This proves that the "lazy" notification scheme is only a constant factor worse in the worst case than an "eager" scheme, which is also clearly n/k, since we immediately broadcast the remapping which takes n messages[6].

The assumptions are:

1. If a set contains t tuples, we set the rehashing threshold k to be at least t. This adds hysteresis to reduce "tuple sloshing".
2. We piggyback rehash notifications on the low-level message acknowledgement. This gets the notification back to the node that generated an out-of-date request before it can generate another request.

Let r be the number of requests between rehashing events. The first assumption allows us to bound the tuple forwarding cost by a constant, since:

$$r \geq k, k \geq t, \text{ so } t/r \leq 1$$

The second assumption allows us to bound the notification cost by n/k.[7]

Proof

We use an amortization proof using tokens to allocate cost. Each notification will consume a token, and each rehash will generate new tokens. We have to show that the tokens will suffice.

In order for a node to forward a request, that node must have been the rendezvous node at some time in the past. We will give each new rendezvous node n tokens. The node

[5] `Build PTP, lookup tuple, search chain, perform match, free template, record match,` and `recv reply`.

[6] Although on a hypercube the broadcast is more efficient, since it is performed as a nearest neighbor spanning tree.

[7] We suspect that a better bound is possible for this algorithm, since the above bound ignores the often widespread reduction in the depth of the tree caused by a notification.

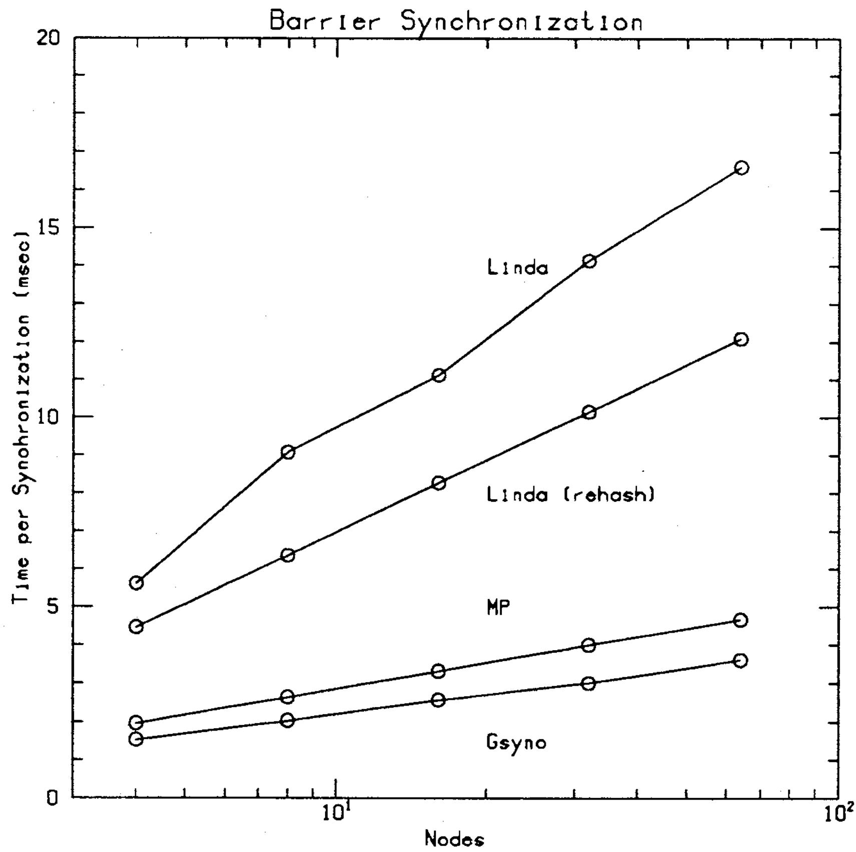

Figure 5.4: Barrier Synchronization: Cost per synchronization versus machine size for Linda, message passing, and Gsync versions.

will spend one token each time it forwards a request along the chain. Thus, we must show that the n tokens suffice.

How many requests can it forward? The entire set of nodes forms a tree rooted at the rendezvous node, with each directed edge connecting a node with the node it believes is the current rendezvous node. Once a rendezvous node hands off the responsibility to some other node, the number of nodes in the subtree it roots will drop monotonically by at least one for each request it forwards, since the node immediately preceding it in the forwarding chain will now point to the node below it.

Once the node becomes a leaf, it no longer forwards any requests, until it once again becomes a rendezvous node. Thus, the forwarding of requests associated with each rehash costs at most n, each also generating a return notification. Since the rendezvous node can change as often as every k requests the amortized cost is $2n/k$ and the total amortized worst case cost of rehashing is:

$$1 + 1 + 2n/k$$

5.3 In/Out collapse

A common usage in Linda is to first `in` a tuple, change one or more fields, and `out` it again. Using the distributed hashing protocol described previously, such a pair of operations requires three messages, assuming that no locality can be exploited. Under some conditions, however, these two operations can be combined into a single operation `inout` that can be thought of as a `rd` with modification. `Inout` returns a copy of the original tuple and modifies the tuple in place in Tuple Space. This saves one message, as well as the cost of freeing a tuple and then generating and installing a largely identical tuple. For instance:

```
in(''value'', ? i);
out(''value'', i+1);
```

could be transformed to:

```
inout(''value'', incr(?i));
```

where `incr` returns one greater than its argument.

Such an optimization requires both compiler and runtime support. The compiler attempts to discover pairs of `in`s and `out`s that can be safely combined. Wang [Wan90] wrote a preprocessor for the Linda compiler that parses the Linda program, looking for suitable pairs and generating Linda code with `inout`s. His rules for collapse are:[8]

1. The operations must be tightly adjacent. Nothing may appear between the `in` and `out`.

2. The operations must be in the same block level.

[8] As concerns this dissertation, we were primarily concerned with the runtime behavior of collapsed operations, and thus considered the restrictions imposed by the compiler acceptable. More research is needed into methods of relaxing the restrictions. Our work demonstrates that such research is justified, because the runtime savings that result are considerable.

3. No free variables are allowed in the modification functions.

4. Corresponding actual fields may not be altered.

5. No array or structure elements may be present.

In order to guarantee that the inout operation is semantically identical to the original operation, we have to ensure that no identifiers changed values between the in and the out. Eventually, we hope to determine this via an analysis of the program, including its flow behaviour; for this initial work, restrictions 1 and 2 are sufficient.

Restriction 3 refers to the fact that the function altering the tuple cannot count on having the environment surrounding the inout available to it, since for many implementations, including those for DMMs, the tuple is modified remotely on the rendezvous node. Again, program analysis should allow us to lift the restriction somewhat by identifying free variables and sending them along. At present, inout pairs that reference free variables are not eligible for collapse.

Restriction 4 prevents modification to the key field, which would necessitate reinstalling the tuple in Tuple Space. This is overly cautious since the Linda analyzer knows which field is the key, and can allow other corresponding actual fields to be altered. However, for simplicity Wang built his inout collapse analyzer as a pre-processor. At this stage the key field has not yet been determined, so he assumed that any field might be a key.

Restriction 5 is purely for expediency. Inouts that only modify an element of an array or structure without changing its length could be easily supported. Modifications that alter the length of the array or structure are more difficult. Since tuples are stored in contiguous memory, expanding a tuple element might necessitate copying it into a larger buffer.

The transformation we actually used for the iPSC/2 was slightly different than outlined above, solely for expediency.[9] The syntax for inout specifies a function identifier positioned after the closing parenthesis. This function is used to modify a ptp in place. Wang's preprocessor converts the discovered pairs:

```
in(''value'', ?i);
out(''value'', i+1);
```

into:

```
inout(''value'', ?i) foo;

foo(ptp_ptr)
PTP_PTR ptp_ptr;
{
ptp_ptr->fields[0].ii = ptp_ptr->fields[0].ii+1;
}
```

[9] We originally collapsed in/out pairs by forcing the user to write the inout and modifier function by hand; when Wang's preprocessor was available, we simply adapted its output to match the handwritten syntax.

Operation	Latency μsec	Rend. Load μsec
IN & OUT	1600	1270
INOUT	1000	680

Table 5.4: Latency and load of `in/out` vs. `inout`

Function `foo`, generated by the compiler, will be called after the matching succeeds and a copy of the tuple is returned to the `in`ing process. `Foo` is passed a pointer to the tuple `ptp` structure; its body performs the required changes to the `ptp` in place in Tuple Space.

The Linda parser was changed to accept the new operation **inout**, which is handled by the parser just like a `rd` except that it has an additional identifier trailing the parenthesis. This identifier is the name of the function that is applied to the tuple.

The runtime kernel was changed to handle the inout operation. `Inout` is handled as a `rd`, but after the reply message is sent the function is applied to the tuple in place, side-effecting some of the fields.

5.3.1 Performance

Table 5.4 presents the costs of an `inout` vs. `in` followed by `out`. Each `inout` requires two messages rather than three. We would expect the latency to be roughly two thirds of the uncollapsed pair; the timings confirm this estimate. The rendezvous node handles one request rather than two, and indeed, the observed load was reduced by nearly half.

As an example of its effect on performance in a program, we used a modified version of the Basic Linda Program (BLP). This test differs from the BLP in two ways. First, workers do not generate result tuples after performing each task. Secondly, only one task tuple is generated, rather than one per task as with the BLP. This single tuple is `in`ed, updated, and `out`ed by the workers. This program eliminates all other communication and concentrates on access to a single task tuple. See Figure 5.5 for the Linda program.

Figure 5.6 shows how the inout collapse optimization changes the point at which the task granularity becomes too fine. Contention occurs when the rendezvous node becomes saturated with requests, i.e. the load imposed on it exceeds its capacity.

If the cost of handling a request is l seconds, then the rendezvous node can handle at most $1/l$ requests per second. If w processes each generate requests every t seconds, then contention will arise if $w/t > 1/l$. Rearranging, we see that contention occurs when $t < wl$.

Using the costs from Figure 5.4, for 63 workers we would expect to see contention arise at $t = 43$ msec for `inout`, versus $t = 80$ msec for the uncollapsed version. These predictions agree well with the observed behaviour in Figure 5.6.

5.4 Memory caching

As tuples are produced and consumed, tuple space servers are constantly concerned with memory management. Managing memory effectively and cheaply is vital to an efficient implementation of Tuple Space in terms of both speed and memory use. Previous systems

```
real_main(argc, argv)
int argc;
char *argv[];
{
  int delay, tasks, workers, i;

  delay = atoi(argv[1]);
  tasks = atoi(argv[2]);
  workers = atoi(argv[3]);

  for (i=0; i<workers; ++i) {
    eval("worker", worker(delay));
  }

  out("task", tasks);

  for (i=0; i<workers; ++i) in("done");
}

worker(delay)
int delay;
{
  while (1) {
    /* get a task */
    /* the following ops could be collapsed */
    in("task", ? task);
    out("task", task-1);
    if (task <= 0) {
      out("done");
      return;
    }
    /* calibrated delay loop */
    delay(delay);
  }
}
```

Figure 5.5: stblp.cl

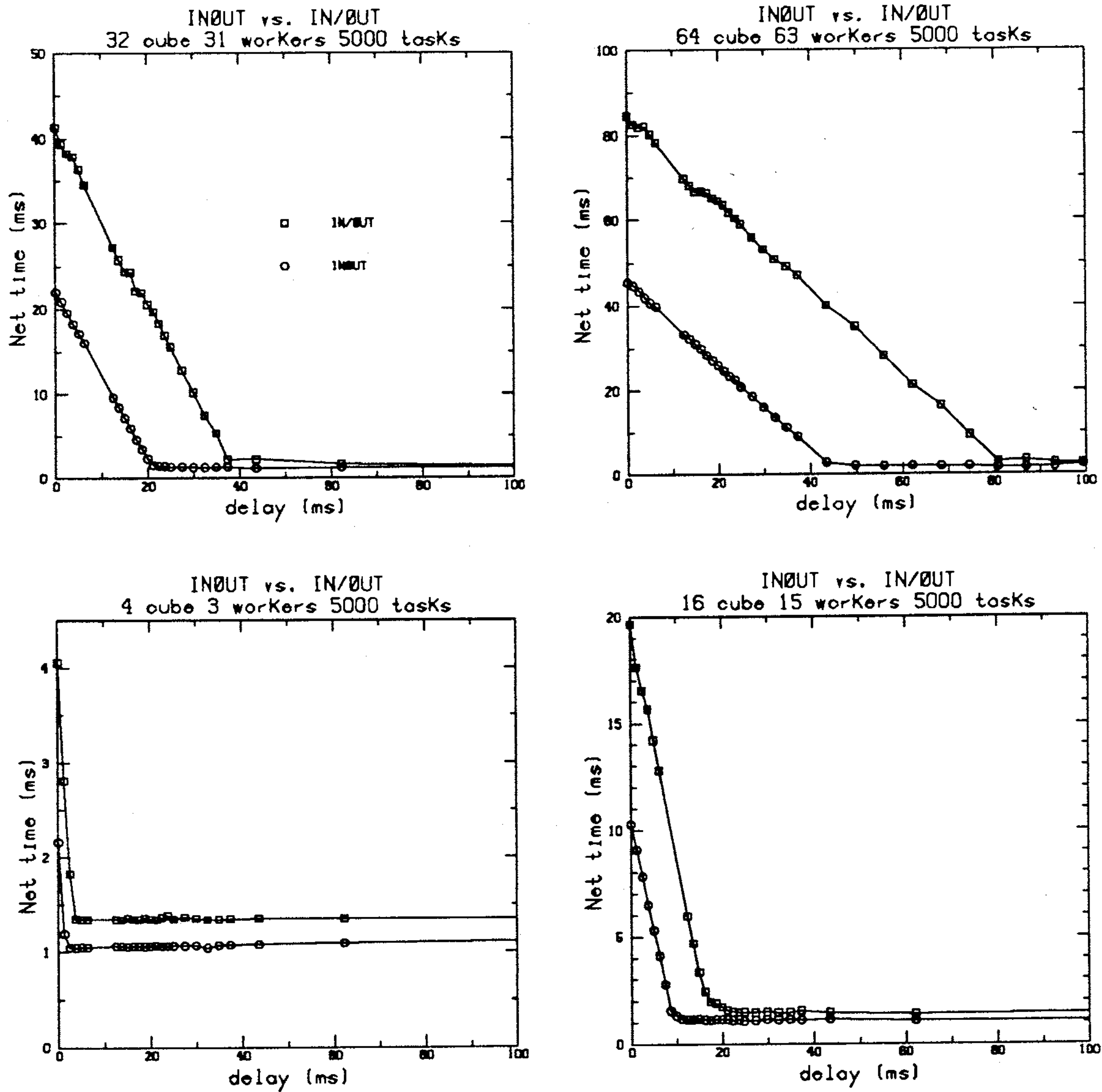

Figure 5.6: Single tuple BLP, **in/out** vs. **inout** for various numbers of workers.

have tried different approaches, with varying degrees of success. We chose a new approach that combines both speed and effective memory use and is particularly well-suited to machines with message passing communication.

5.4.1 Previous approaches

The first Linda runtime systems (See Carriero [Car87]) stored tuples in fixed-size blocks. A tuple with no array elements could be stored in one block. Longer data fields, e.g. arrays, were stored in one or more additional blocks linked to the main block. A large number of blocks were allocated to a free list during initialization. This method made allocating and freeing memory very cheap, at the cost of poorer memory utilization due to the fixed size. In addition, matching tuples was made somewhat more complicated by the discontinuities. More importantly, from the aspect of message passing machines, the resulting tuple structure was non-contiguous, consisting of a number of blocks linked together.

This worked well on the S/Net where communication is based on packet switching[10], with the packetization performed by the user's code. Carriero chose the blocks to be of a size that was reasonable to use directly as packets. However, most modern DMMs present a different communication model. The message passing primitives expect an address and a length and send the entire chunk of data. If packetization occurs, it is invisible to the user. Furthermore, many machines, the iPSC/2 included, use circuit switching[11] for large messages. In either case, we are strongly rewarded for storing tuples in contiguous memory, rather than in discontiguous blocks.

Leichter's runtime system [Lei89] took an entirely different tack, storing tuples in a garbage-collected tuple heap. This allowed for cheap allocations and frees of arbitrary length at the cost of doing an occasional garbage collection. The garbage collection was straightforward, since a table of all allocated blocks, whether allocated or freed, was kept.[12]

5.4.2 Our approach

Our approach provides (generally) fast allocation and freeing of blocks of arbitrary length. The method is based on the observation that most Linda programs `out` tuples of the same sizes repeatedly, thus causing the Tuple Space server to allocate and free the same sized blocks again and again. We provide a general purpose allocator, along the lines of C's `malloc`, but in addition provide a caching mechanism. For the cache we used a hash table.

[10]Packet switching breaks messages into pieces, or packets, which are directed separately to the destination, where they are reassembled again. If the path requires multiple hops, packets will be buffered on intermediate nodes.

[11]Circuit switching builds a continuous route (circuit) between the source and destination. Once the circuit is built the message is streamed to the destination without the need for any intermediate buffering.

[12]Leichter claimed that the cost of GC was negligible. We remain skeptical on this point – the cost of his GC is dependent on the ratio of live to dead tuples in Tuple Space, which itself is very application-specific. During a GC, each live tuple must be compacted (copied) to the bottom of the heap. If an application keeps considerable state in Tuple Space, the live portion will be large, and the copying costs will be high. GC is most useful in situations where it is difficult to determine whether a block can still be referenced when freeing a pointer to it; in the case of Tuple Space each block is referenced only once, so this property is not needed.

Each element of the hash table contains a pointer to a list of free blocks of a particular size, plus a field indicating the size.

When allocating a block of size s:

1. Round up s to a multiple of 16 bytes $s\prime$ (to improve cache hits).
2. Hash on $s\prime$ into the hash list. (The hash function ignores the bottom 4 bits.)
3. Check the size entry for this list. If it differs from $s\prime$, this is a hash collision. Get a block from the general purpose allocator and return it.
4. If the block list of the entry is non-nil, remove the first block from the list and return it.
5. If the block list is nil, no blocks of size $s\prime$ are cached. Get a block from the general purpose allocator and return it.

When freeing a block of size s (necessarily a multiple of 16):

1. Hash on s into the hash list. (The hash function ignores the bottom 4 bits.)
2. If the size field for that entry matches s, free the block to that list.
3. If the size field differs from s, but the list is nil, resolve the collision by reassigning the entry to size s by setting the size entry to s and hanging the block on the list.
4. If the size field differs but the list is not nil, free the block to the general purpose allocator.

If at any point the general purpose allocator runs out of memory, we flush enough cached buffers to satisfy the request.

The way hash collisions are resolved could be improved. Currently, if an uncommon size is freed to the cache and never reallocated, it would prevent a common size that happened to be hashed to the same location from being cached. We should probably "time-out" size allocations to the table; if a particular size showed no activity over a long period, we could allow a collision to flush it from the cache.

This caching strategy is similar to several developed elsewhere. Oldehoeft and Allan [OA85] used a working set method to save frequently used sizes in a front-end cache. Bozman [Boz84] used a lookaside buffer to provide quick access to frequently used sizes in a linked list.

5.4.3 Performance

The effectiveness of this optimization depends on the extent to which programs reuse the same size blocks. Table 5.5 shows the savings realized when the caching succeeds. The times are a result of allocating and freeing the same size block repeatedly.

General purpose allocator	135.3 (μsec)
Caching allocator	31.3 (μsec)

Table 5.5: Performance of memory caching optimization.

5.5 Local data caching

This optimization improves the transfer of large amounts of data through Tuple Space. Recall the basic communication paradigm: the tuple is **out**ed to a rendezvous node, the template is sent to the same node, and the tuple is returned to the **in**ing node. If the tuple contains a large amount of data, that data is sent twice. If the element is not needed for matching, we can arrange to send it only once.

We first modified the Linda compiler to provide the runtime system with an additional piece of information about each element of a tuple: whether or not that element will ever be involved in matching. Then, when the runtime system **out**s a tuple, it checks each element of the tuple against two criteria:

1. Is the length of the element greater than a chosen threshold?
2. Is the element never needed for matching?

If both criteria are met, the element is stored locally in the **out**ing node. A pointer to it is inserted in the tuple, along with a note that the element is stored locally. The tuple is set to the rendezvous node without that element. Upon matching, an additional message is sent back to the originating node, requesting it to send the element directly to the **in**ing node. The **in**ing node, after receiving the tuple from the rendezvous node, waits for any such local elements to be sent to it.

Let us compare the latency for a tuple with one long element of k bytes, with and without the optimization, where b is the fixed overhead, and a is the additional per-byte cost of sending long tuples.

Without optimization:

Tuple sent to rendezvous node.	$ak+b$
Template sent to rendezvous node.	b
Tuple sent to **in**ing node.	$ak+b$

With optimization:

Tuple sent to rendezvous node.	b
Template sent to rendezvous node.	b
Tuple sent to **in**ing node.	b
Request sent to originating node.	b
Long element sent to **in**ing node.	$ak+b$

Thus, we have $2ak + 3b$ vs. $ak + 5b$. We should choose the threshold value based on the break-even point of these two equations.[13]

[13]On the iPSC/2, b_{small} is $\approx 400\mu$sec for messages under 100 bytes, and $b_{large} \approx 750\mu$sec for larger

5.5.1 Performance

We refer the reader to Figures 4.1 and 4.2. The lower per-byte costs of sending tuples greater than the threshold (5000 in this case) is due to this optimization. Notice also the larger fixed cost, which reflects the fact that more messages are sent.

Under some conditions this optimization may actually degrade performance. In particular, if one node is responsible for outing most or all of the tuples optimized, that node may run out of memory. In addition, that node will have to service all the requests for the long elements. This is particularly troublesome for hash sets, since the standard paradigm spreads the tuples across Tuple Space, equalizing the memory and CPU loads. It may be better to restrict this optimization to sets other than hash.

5.6 Tuple broadcast

The distributed hash paradigm was chosen as the basis of DMM-Linda for reasons described in Chapter 3. There we argued that broadcasting tuples or templates would be prohibitively expensive as the machine grew. Under certain circumstances, though, a different approach is preferable. In particular, tuples that are read by many processes perform better under a replicated tuple paradigm using broadcast such as Carriero used on the S/Net [Car87].

Determining the relative frequency of `rd`s to `in`s for each process at compile time is extremely difficult; adapting at runtime may well lead to missed opportunities. For example, in a typical scenario we `out` a tuple just once, and each process reads it once. It would be difficult to optimize such accesses using runtime adaptation alone. We chose a simple, yet reasonably effective compile-time approach: any set containing a `rd` is stored under the replicated tuple paradigm.

When an `out` belonging to such a set is performed, the tuple is broadcast to all nodes rather than just the rendezvous node. The rendezvous node, which is determined in the usual manner, is still special since it is the ultimate owner of the tuple and is responsible synchronizing multiple `in`s. On each node, the tuple is stored in the normal manner. When a `rd` is performed, the process checks only locally; if a matching tuple exists anywhere, a local copy will eventually be present. If the search fails, the template is stored locally awaiting a matching tuple.

When an `in` is performed, the template is sent directly to the rendezvous node. As soon as a matching tuple is found, the `in` is satisfied by sending the reply and marking the tuple as deleted. The `in`ing node receives the reply and extracts data from it in the usual fashion. It then broadcasts a delete message, which contains a description of the tuple to delete. When nodes receive the delete message, they search for that tuple, and delete it if found. The delete message carries enough information with it (the set identifier, key, and unique tuple identifier) that only one tuple chain is scanned. It is possible not to find the tuple, not because the tuple has already been deleted (`in`s are arbitrated by the rendezvous node, so the tuple can only be deleted once), but because the delete message

---

messages, while $a \approx 0.35\mu$sec/byte. For tuples that exceed 100 bytes, the cost is $2ak + b_{small} + 2b_{large}$ without the optimization, versus $ak + 4b_{small} + b_{large}$ with. The breakeven point is $k \approx 5600$. We actually set the threshold at 5000 bytes.

may arrive before the tuple. In this case, the delete request is retained in a chain-specific list. All broadcast tuples that arrive are checked against the appropriate list.[14]

The effect of the replicated tuple paradigm is to make `rd`s very cheap, requiring no communication, while `in`s and `out`s are rendered more expensive, each requiring a broadcast. Another potential drawback is increased memory usage due to tuple replication.

This simple scheme could be improved in several ways. Most importantly, nodes should be free to decline or flush copies of replicated tuples if they run short of memory. Only the rendezvous node's copy of the tuple must be retained. Nodes having flushed the tuple would simply divert the query to that node.[15]

5.6.1 A digression concerning correctness

It is important to guarantee that Tuple Space remain consistent regardless of the presence of multiple copies of tuples. Intuitively, we need to insure that no process is able to `in` or `rd` a tuple after another process has `in`ed it. `In`ing is clearly not a problem under the described scheme; using the rendezvous node to synchronize `in`s prevents two processes from `in`ing the same tuple.

The situation with `rd` is more complicated. Temporal ordering has a well-defined meaning in Linda. Events are partially ordered by Linda dependencies; an event can precede, be co-temporal, or follow another event based on the partial ordering.

The axioms forming the partial ordering are:

1. Within a single process, an `in` or `rd` precedes any Linda operation that occurs after it in the control flow of that process.

2. An `out` precedes the matching `in` or `rd`.

Note in particular that an `out` does *not* automatically precede a Linda operation that occurs after it in the control flow of a single process. This is because `out`s are asynchronous; the `out` may return before the tuple actually exists in Tuple Space.

Now the requirement for consistency can be clearly stated. We need to insure that a `rd` that follows an `in` according to the partial ordering will not find the same tuple. For example:

```
Process 1   Process 2
in("A")     in("B")
out("B")    rd("A")
```

Using the axioms, we can establish that `rd("A")` must follow `in("A")`[16], and thus

[14] These out-of-order deletes were observed to be fairly rare, so the iPSC/2 implementation uses a simple linear search. A number of other options are possible, including hashing on the tuple identifier, or a binary tree.

[15] We did not implement flushing of copies because flushing makes it difficult to determine what a node should do if it performs a `rd` and finds no match. Using flushing, there are two possible reasons that no match is found: either no matching tuple exists anywhere, or one does exist but the local copy has been flushed. If it exists nowhere, the node should keep waiting locally; if the copy was flushed, it should immediately send the template to the rendezvous node. Unfortunately, distinguishing these two cases is difficult unless records are kept of tuples that have been flushed, which could be expensive.

[16] `in("A")` precedes `out("B")` by axiom 1, `out("B")` precedes `in("B")` by axiom 2, and `in("B")` precedes

rd("A") must block.

It is the case that any chain of events establishing that an in precedes a rd will necessarily involve at least one event occurring in the control flow of the ining process after the in. To see why, consider the possibilities. There are two classes of precedence chains that can establish in("A") precedes rd("A"). The in and the rd can occur in the control flow of the same process. In this case the following event is the rd itself. Alternatively, they can occur in different processes, in which case there must be some out/in pair with the out (the following event) occurring on the process performing the in("A").

In either case, consistency will be preserved if we can guarantee that all nodes see the the delete message *before* they see the following event. On machines such as the iPSC/2, where messages are reliable and ordering is preserved, this property holds if both the delete message and the following event emanate from the same node, i.e. the ining node. This is the reason for broadcasting the delete message from the ining node rather than from the rendezvous node.[17] On machines without this property, the delete message will have to be acknowledged by every node.

5.6.2 Performance

Table 5.6 shows the costs of tuple operations using tuple broadcast. The costs are determined for three different nodes:

- latency at source node
- load on rendezvous node
- load on other nodes

The times shown are for a simple tuple without any long elements. The experiment used to determine these times was similar to that used for normal operations in Chapter 4. Latencies were measured by averaging over a large number of the operations. Loads were determined by comparing the time required to perform a calibrated loop while also handling Tuple Space traffic. On the iPSC/2, the cost of a broadcast depends on the dimensionality of the machine. The times in Figure 5.6 reflect the fact that outs require a broadcast by the source node, and ins require one by the rendezvous node (the delete message). The cost to other nodes for out and in reflects the cost of handling the tuple and delete message, respectively. As we would expect, rd is very cheap, incurring only a local Tuple Space access.[18]

---

rd("A") by axiom 1.

[17]This consistency problem was discovered after the iPSC/2 implementation was completed; in that implementation, the rendezvous node performed the delete message broadcast, which could have allowed inconsistency.

[18]The times in Table 5.6 differ in several ways from those presented in Tables 4.1 and 4.2 for non-broadcast operations. The base latency for out is $\approx 100\mu$sec higher. This is because the source node, after broadcasting the tuple, also has to install it locally (a broadcast on the iPSC/2 does not send to the sender).

Operation	source node latency	rend. node load	other node load (μsec)
out	$416 + 84d$	698	576
in	1550	$879 + 86d$	530
rd	191	0	0

Table 5.6: Costs for operations using tuple broadcast (simple tuple). d is the dimensionality of the machine.

5.7 Randomized tuple bag

As the number of nodes increases, a major concern is that rendezvous nodes might become a bottleneck. Because hash sets are spread out over the machine, a bottleneck will only arise if many processes want tuples that happen to be mapped to the same node. Queue sets are a different matter since the standard paradigm maps the entire set to one node. This may result in that node becoming a bottleneck, even on relatively small machines.

The randomized tuple bag optimization addresses this problem by spreading queue sets over multiple nodes, the exact number depending on the number of tuples in the bag. We let queue sets have two possible representations: normal and randomized.

5.7.1 Randomized queue set representation

The randomized queue representation spreads out the handling of a queue set across a number of nodes, thus reducing node contention. Out of the total set of nodes N, some subset S of the nodes are chosen. All tuples of the set already in Tuple Space are divided among S. On a hypercube a natural choice for S is one of the subcubes of size $\sqrt{N}$ that contain the original rendezvous node, since multicasting to that subset is particularly convenient. When a node outs a tuple to a randomized set, it sends it randomly to a member of S. When a node requests such a tuple, it sends the request to a random member as well. If that node happens to have a tuple, the request is handled. If the node does not have a tuple, it forwards the request to some other, random member of S. If the request bounces some number of times k, the request fails, and the kernel returns to the normal (i.e. single node) representation.

Converting to and from the random paradigm is as follows:

Similarly, the latency for an **in** is roughly 500 μsec greater than the non-broadcast operation. It includes the cost of handling the delete message, which is 530μsec.

Rd latency is $\approx$ 70 μsec less than before, for two reasons. Matches on broadcast tuples are not recorded since such sets are never rehashed, saving roughly 25 μsec. In addition, when doing a **rd** we save two destination lookups, since we know the tuple to be local. This saves an additional 50 μsec.

The reported load for an **out** is less on the rendezvous node than the other nodes. This is actually an artifact of the testing procedure. For our test, the rendezvous node happened to be a non-leaf node of the broadcast, whereas the other node was a leaf. On the iPSC/2, non-leaf nodes have to forward the broadcast message.

Finally, the reported load for **in** on the other node is about 240 μsec less than reported in Table 4.2 for the rendezvous node. This is because Table 4.2 includes matching and returning a tuple, whereas in this test no match ever occurred on the other node.

1. Rendezvous node decides to convert.
2. Set S is chosen.
3. Every member of S is notified.
4. Rendezvous node spreads the existing tuples among S.

Converting back is analogous:

1. Some member of S handles a failed request (one that has bounced k times).
2. It informs the rendezvous node and bounces the request to it.
3. The rendezvous node notifies all members of S.
4. All members of S return their remaining tuples to the rendezvous node.

The advantage of this method is that the number of tuples present is not required to be explicitly known anywhere; we simply depend on probabilistic behavior to inform us when few tuples remain. In the next section we will examine the probability of requests failing; it turns out that the function turns sharply at a point close to $|S|$, going asymptotically to 0 or 1 as the number of tuples increases or falls, respectively.

How do other nodes know where to request tuples? We use the same strategy as with tuple rehashing: need-to-know. After the switch to the randomized method, other nodes will send their next request to the rendezvous node. The rendezvous node will forward it into S and send a message back to the node, informing it of the change. Likewise, after converting back to the standard method, each node will make its next request of some member of S. That node forwards the request to the rendezvous node, and notifies the requester of the change.

5.7.2 How much does the randomized method cost?

We want to examine the number of additional messages generated, amortized over the number of requests served. Note that the number of messages cannot be less than the number generated using the standard method. The randomized method reduces *contention*, not messages. However, we would like to ensure that in the worst case the number of messages is no more than a small constant factor greater than that for the normal method.

Let:

s = size of S.

t = number of tuples present in Tuple Space.

i = nodes that must be informed of a change.

k = maximum misses allowed.

The cost of the random method is the cost of converting to and from the method plus any additional cost incurred while randomized.

$$C_{Total} = C_{Conv} + C_{Rand} + 1 + C_{Conv}$$

C_{conv}, the cost (in messages) of conversion, is the same to and from the randomized method, ignoring the single message from the member of S that discovers failure to the rendezvous node informing it of the failure (the reason for the 1 in the formula). We need to inform the members of S, forward the tuples, and inform other nodes as they need to know.

$$C_{Conv} = s + t + i$$

Before we can define C_{Rand}, we need to define some other quantities: The chance that a single probe will fail is:

$$P_f = \left(\frac{s-1}{s}\right)^t$$

The easiest way to see this is to reverse the process: first, choose a node, then randomly place t tuples. Each has a $\frac{s-1}{s}$ chance of missing the chosen node, and t tuples have a $(\frac{s-1}{s})^t$ chance of missing. The chance that a single probe will succeed is:

$$P_s = 1 - P_f = 1 - \left(\frac{s-1}{s}\right)^t$$

The expected number of probes until success is:

$$E_p = 1/P_s = \frac{1}{1-(\frac{s-1}{s})^t}$$

This assumes independence. We could do somewhat better by keeping track of which nodes had been probed. So C_{Rand} is the sum of all additional probes:

$$C_{Rand} = \sum_{r \in R} (E_p(t(r)) - 1)$$

where R is the set of requests generated from the time the rendezvous node converts to the randomized method to the time the last node is notified of the conversion back to the normal method and $t(r)$ is the number of tuples present in Tuple Space when request r is made. Thus:

$$C_{Total} = s + t_0 + i + \sum_{r \in R} (E_p(t(r)) - 1) + s + t_1 + i$$

where t_0 is the number of tuples when converting to the randomized method, and t_1 the number when converting back.

What is the worst case? Since the algorithm is random, the actual worst case is that the first request fails, and we convert back immediately. However, this is very unlikely if s is chosen correctly. We are more interested in the average worst case.

First, two assumptions:

- $t_0 > 2t_1$ (thus, $t_0 + t_1 < 3t_0/2$ and $t_0 - t_1 > t_0/2$)

- $t_0 > s$

Clearly, $|R| \geq t_0 - t_1$. Also, $|R| \geq i$, since a node is only notified if it generates a request. $C_{Rand} \leq k|R|$, since no request may probe more than k times before failing. Thus, the average worst case amortized cost of the randomized method:

$$Ca_{Total} \leq \frac{2s + t_0 + t_1 + 2i + k|R|}{\max(|R|, t_0 - t_1, i)}$$

$$\leq \frac{2s}{t_0/2} + \frac{3t_0/2}{t_0/2} + \frac{2i}{i} + \frac{k|R|}{|R|}$$

$$\leq 4 + 3 + 2 + k$$

The average behaviour of the method is very good. Because E_p turns quite sharply at about $t = |S|$ (see Figure 5.7), the number of probes per request will remain very close to one until the turning point is reached, at which point the failure rate will rapidly increase and k will be exceeded quickly, causing a conversion back to the normal representation.

5.7.3 Performance

As previously mentioned, the performance of operations handled under the randomized method cannot be judged as simply as that of the previous optimizations. A single randomized operation done in isolation will *always* cost at least as much as the non-randomized, even ignoring the cost of conversion. This is because a random operation will require at least one probe. The true advantage of the randomized method only becomes apparent when contention becomes a problem.

To test the optimization, we returned to the Basic Linda Program (BLP). Figure 5.8 shows the communication overhead for each task using various task granularities and 31 workers. The two methods were very similar for task sizes greater than 23 msec, but below this, the randomized version did markedly worse. Statistics gathered while running the program explain this strange behaviour. First of all, the task tuples were never randomized for any task size because enough never accumulated in Tuple Space to exceed the threshold. The initial group of tuples was consumed immediately by waiting workers, and the watermarking algorithm used to feed tuples slowly into Tuple Space kept the number low thereafter.

The second reason for the poor performance was more interesting. Normally, the result tuples would be rehashed to the master node very quickly and would be entirely local from then on. Unfortunately, the number of result tuples present in Tuple Space seemed to hover around the randomizing threshold – the set converted back and forth more than one hundred times during one run. Not only did this incur substantial overhead, but the act of converting back and forth had the effect of diluting the rehashing information so that the set never rehashed (which would have prevented further thrashing.) We took care of the thrashing problem by raising the randomizing threshold each time a set converted back.

Another problem with the BLP in this context is that the result tuples can become a bottleneck since only one process consumes the tuples produced by many (31 in this case).

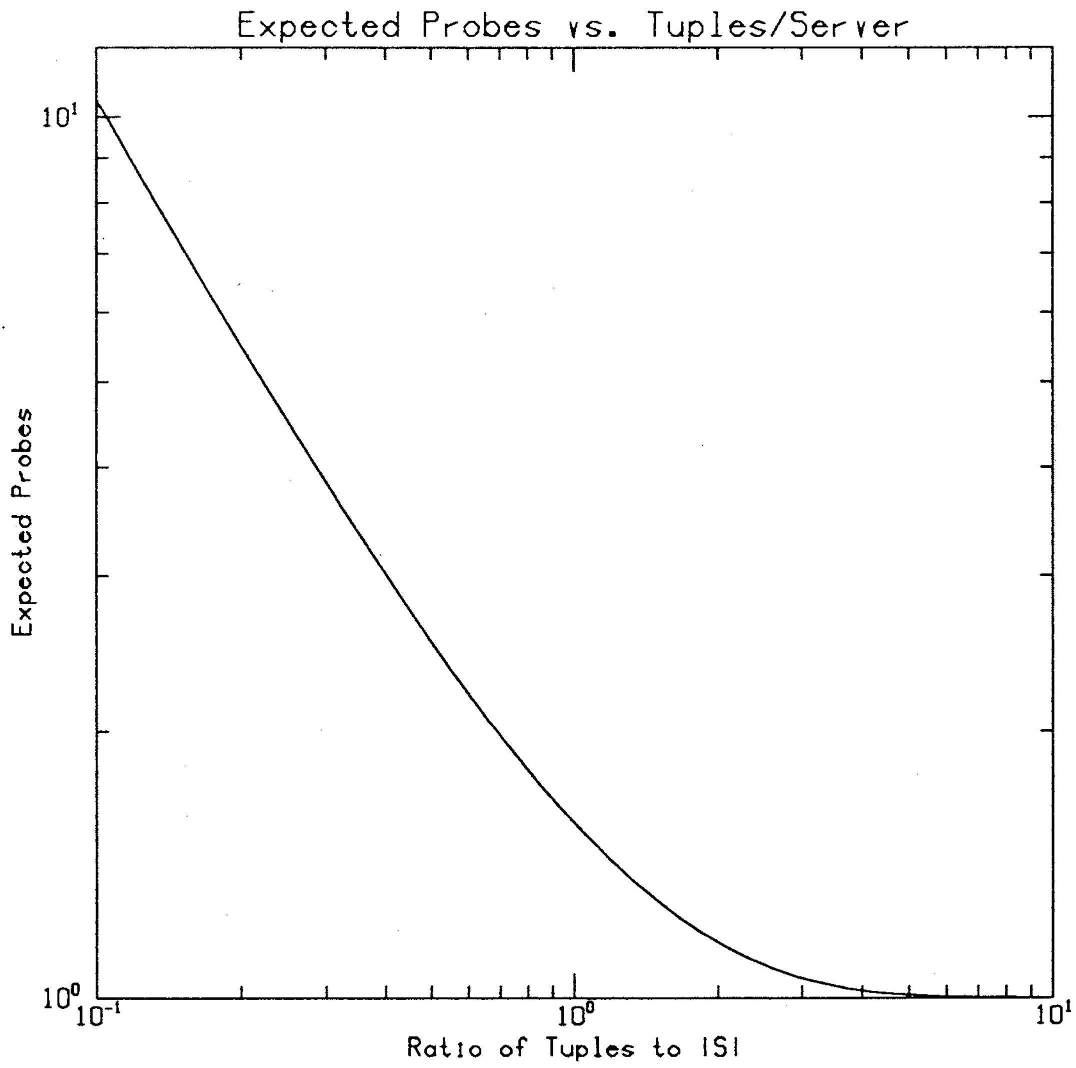

Figure 5.7: Expected probes versus ratio of tuples present to size of subset S.

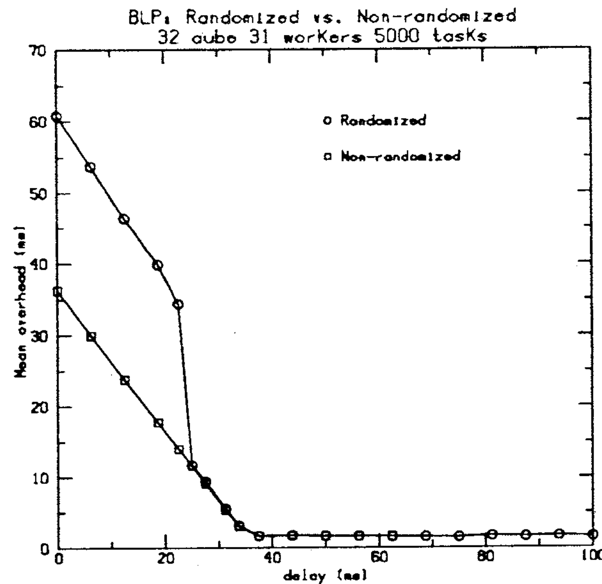

Figure 5.8: Average task communication overhead for Randomized vs. Non-randomized method for BLP on 32 nodes, varying task size.

If the master falls behind, the workers will have to wait a considerable amount of time for the poison pill task tuples after finishing all of the real tasks. In order to isolate just the effect of the randomizing optimization on the task tuples, we modified the BLP. This new BLP differed from the old in two respects. First, task creation was distributed to all of the workers. Upon being created each worker dumped its share of tasks into Tuple Space. Secondly, we ignored the time lost between the last successful task completion and the arrival of the poison pills. The effect of these two changes was to eliminate the bottleneck effect of the master, leaving just the contention for the task tuples.

See Figure 5.9 for the modified BLP program. Figure 5.10 shows the performance of this new program using the kernel modified as above to avoid thrashing. The effect is quite dramatic; using the random method, almost no contention is observed for even very small tasks.

The randomized queue representation can markedly reduce contention for a queue set by spreading it across several nodes. Moreover, making use of statistical properties allows the representation to operate with almost no conversation between the processors handling the set. Most requests will be handled with no additional latency.

5.8 Interactions between optimizations

In the previous discussion, each optimization was discussed independently, with no regard for the other optimizations. One might reasonably ask what effect the optimizations might

```
real_main(argc, argv)
     int argc;
     char *argv[];
{
  extern int atoi();
  int i, num_workers, task_size, tasks, worker();
  int           t1, t2, got, diff, tasks_per_worker;
  int           remainder, t;
  task_size = atoi(argv[1]);
  tasks = atoi(argv[2]);
  num_workers = atoi(argv[3]);
  out("task size", task_size);
  tasks_per_worker = tasks/num_workers;
  remainder = tasks%num_workers;
  for (i = 0; i < num_workers; ++i){
    t = tasks_per_worker + (remainder-- > 0 ? 1 : 0);
    eval("worker", worker(t));
  }
  for (i = 0; i < num_workers; ++i) in("done");
  out("go");
  for (i = 1; i <= tasks; ++i) in("result");
  for (i = 0; i < num_workers; ++i) {
    out("task", 1); in("stats", ? got, ? diff);
    printf("worker got %d time %d\n", got, diff);
}}

worker(t)
int t;
{
  int i, stop, task_size, t1, t2;
  int tasks=0, started=0;
  rd("task size", ? task_size);
  while (t--) out("task", 0);
  out("done");
  rd("go");
  while(1) {
    in("task", ? stop);
    if (!started) {started = 1; t1 = mclock();}
    if (stop) break;
    ++tasks;
    _l_delay(task_size);
    out("result");
    t2 = mclock();
  }
  out("stats", tasks, t2-t1);
}
```

Figure 5.9: Modified BLP program

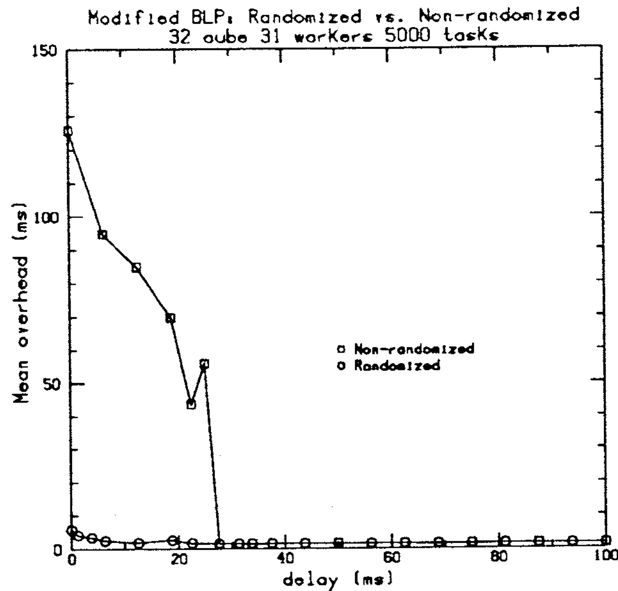

Figure 5.10: Average task communication overhead for Randomized vs. Non-randomized method for BLP modified to dump tasks first on 32 nodes, varying task size.

have on one another. These interactions fall into three general categories:

Positive (P) Such optimizations work well applied to the same set.

Negative (N) Such optimizations interfere with one another when applied to the same set.

Disallowed (D) Such optimizations are not allowed to be applied to the same set in the current implementation.

Figure 5.11 gives an overview of the interactions. We discuss each entry further below. Note that the discussion concerns applying multiple optimizations to a particular set, not to Tuple Space as a whole. All of the optimizations complement one another insofar as they are applied to different sets. We can and do run Linda programs with all optimizations enabled, with some subset of them eventually invoked automatically by the kernel on each set.

5.8.1 Memory caching

This optimization is orthogonal to all of the others, and thus combines well with all of them.

	Local Caching	Random Queue	Tuple Broad.	Inout	Rehashing
Memory Caching	P	P	P	P	P
Local Caching		P	P	P	P
Random Queue			N/D	P	N
Tuple Broad.				N/D	D
Inout					P

Figure 5.11: Synopsis of interactions between optimizations.

5.8.2 Local data caching

This combines well with tuple broadcast since it avoids the necessity of replicating the large data elements. However, in some cases the benefits of the tuple replication will be lost, since the node or nodes holding the long elements may become a bottleneck.

In general, the decision about what to do with large, non-matched tuple elements (the concern of this optimization) is largely independent of the decision of how to store the rest of the tuple (the ptp), which is the concern of the other optimizations. Whether or not this optimization is successful depends largely on how well the elements cached are distributed among the nodes of the machine, so as to prevent memory and CPU usage imbalance.

5.8.3 Tuple broadcast

This optimization is not currently applied to the same set as the rehashing, random queue or inout optimizations. In the case of rehashing, it makes little sense to bother to change the rendezvous node when the tuple will be copied to all nodes.

Combining tuple broadcast with the random queue method also makes little sense. True, it might distribute the rendezvous responsibility among several nodes, but this would be of little benefit since the delete protocol would still involve all of the nodes.

Updating replicated tuples using inout would be difficult to do correctly, since the global update would have to be atomic. To see why this is important, imagine two simultaneous inouts performed on the same tuple, one updating with function $f()$, the other with $g()$. Afterwards, some copies might be $f(g(t))$ and others $g(f(t))$.

5.8.4 Random queue

This optimization combines very effectively with the inout optimization since the tuples are distributed (evening out the load) and updated in place (decreasing the load.) For reasons already discussed, this optimization is incompatible with tuple broadcast.

At present, rehashed queues may then be randomized, and randomized queues that reconvert to the normal method may then be rehashed. However, both cases are less than ideal. If a queue is rehashed, the system is guessing that the new rendezvous node will consume most of the tuples. Randomizing it destroys the locality from which we were hoping to benefit. Furthermore, randomizing a set interferes with the collection of tuple usage data being gathered for rehashing. After reconverting, such information would have

to be gathered from all the submasters and integrated. At present we do not collect this data while the set is randomized.

5.8.5 Inout collapse

The inout collapse works well with all other optimizations except tuple broadcast. As we discussed previously, updating replicated tuples in place is a daunting task.

5.8.6 Rehashing

This optimization has been discussed in the previous sections. It works well applied to the same set as memory caching, local data caching, and inout collapse. It complements tuple broadcast, although it may not be applied to the same set. Finally, it can interfere with the random queue optimization.

5.9 A peek ahead

The next chapter will present results from running a number of applications on the iPSC/2 Linda system, including the effect of each of the optimizations. We would like to close this chapter with a quick preview of the effect of the optimizations discussed in this chapter on the applications, as presented in Table 5.12. Each application is described in Chapter 6. All runs were on 64 nodes of the iPSC/2.

In/rd local The percentage of tuple accesses satisfied locally. Note that on average, 1/64th (2%) would be local just by coincidence; the rest can be attributed to optimization.

Messages The total number of messages, compared with the unoptimized version. We charged a broadcast as 6 messages[19].

Random ops The percentage of total operations rendered random by the random queue optimization.

Cache hits The percentage of memory allocations that could be satisfied by the buffer cache.

Tuples examined The ratio of the total number of tuples examined to the total number of `in`s and `rd`s.

Full Match The ratio of match attempts to the total number of `in`s and `rd`s.[20]

[19] ($\log_2 64$)

[20] The number of match attempts is less than the number of tuples examined for two reasons. First, the queue paradigm does not require a match. Second, some potential matches are rejected quickly before invoking the full matching algorithm. The tuple examined ratio can be slightly less than one due to final `in`s and `rd`s that never find a match.

Statistic	Application						
	DNA search	Block. Matrix	Clump. Mat	LU decomp.	ICM	Erode	2DFFT
Rehashing							
In/rd local	30%	89%	2%	3%	37%	99%	95%
Messages	83%	42%	99%	99%	nr	nr	52%
Tuple Broadcast							
In/rd local	6%	4%	99%	96%	nr	nr	3%
Messages	97%	99%	4%	13%	nr	nr	99%
Random Queue							
Random ops	46%	0%	0%	0%	nr	nr	0%
Messages	107%	na	na	na	na	na	na
Memory Caching							
Cache Hits	51%	95%	35%	72%	98%	99%	96%
Match Ratios							
Tuples Examined	0.98	0.99	1.00	1.00	1.71	1.00	1.08
Full Match	0.31	0.95	0.99	0.98	1.27	0.99	0.99

Figure 5.12: A preview of the performance of the optimizations. Na means "not applicable", nr means "data not recorded"

None of the applications did significant numbers of **inouts**, so no figures are presented for that optimization. Also, none of the applications took advantage of the local data caching optimization when run on 64 nodes, although some did when running on fewer nodes. This is due to effective data decomposition; as the machine grew, the size of each node's communication shrunk below the local caching threshold.

For reference, the details of each program are listed below:

DNA 468 base test sequence vs. DB ($\approx$ 730 sequences, totaling $\approx$ 163000 bases). 1000$\times$40 blocking.

Block MM 240x240 matrices (single precision). 10 timesteps.

Clumped MM 400x400 matrices (single precision). A clumped by 1, B clumped by 1.

2D FFT 1024x1024 matrix (complex single precision).

LU 400x400 matrix (single precision).

Erosion 200x200 points, for 200 timesteps.

ICM 101 process nodes, median execution time per node 6.8 msec. 100 timesteps.

First of all, notice how well rehashing and tuple broadcast complement one another: all of the applications shown benefit substantially from one or the other, but not both. This is due to the fact that one particular pattern of communication accounts for most of

the communication cost in each of the programs, which in most cases was either rehashed or broadcast.

The two applications that rehashed well (Blocked matrix multiply and 2DFFT) reduced the total number of messages substantially. In the case of 2DFFT, the reduction was somewhat less than might be expected.[21] The extra messages were found to be due to a relatively large ratio of rehashing notifications.

The DNA program was the only one to randomize any queue sets. The overhead of additional messages was modest. About half of the additional 7% was the cost of flushing tuples during conversion, the other half was due to bounced requests.

The memory caching was a success for most of the programs. DNA showed less buffer length coherency mostly because some of its tuples varied in length according to the length of the DNA sequences. Clumped Matrix multiply had a poor buffer cache hit ratio because it didn't free many buffers during the course of the computation.

The match ratios show that the analysis and the hash functions are performing well, so that the kernel does not search many different tuples for a match.

[21] Local ins use no messages, so instead of three messages per rehashed communication we would expect just one for the out, so we would expect $(95\% * 1/3 + 5\% = 37\%)$ instead of 52%.

Chapter 6

Applications

6.1 Introduction

In Chapter 4 we presented basic performance figures for the iPSC/2 implementation of DMM-Linda using small synthetic programs to pin-point specific characteristics in isolation. In this chapter we will examine performance on several real programs. Our purpose is twofold: first, to observe how the system functions as a whole on real programs. Secondly, to examine the contribution of each of the optimizations to performance. Eight different applications were examined:

1. DNA sequence comparison.
2. Block matrix multiply.
3. Clumped matrix multiply.
4. 2D FFT.
5. LU decomposition.
6. Landscape erosion simulation.
7. Intensive care monitor.
8. Helicopter rotor wake simulation.

The eight applications presented in this chapter were not chosen because they were particularly well-suited to running under Linda on distributed memory multiprocessors. On the contrary, most of the programs were written by programmers without any knowledge of the Linda system implemented on the iPSC/2. The DNA, clumped matrix multiply, and lu codes were written several years ago for other Linda systems (shared memory multiprocessors and the S/Net, see [Car87]) with very different implementation characteristics. The 2D-FFT, intensive care monitor, and helicopter wake simulation codes were all written by relatively inexperienced Linda programmers with little or no detailed knowledge of any Linda runtime systems, including the design implemented on the iPSC/2. Only the block

Program	Code Size (lines)	Method	Granularity
DNA	445	M-W	$1.8N/P$ Ops/byte
Block MM	266	SYST	$0.25N/\sqrt{P}$ Flops/byte
Clumped MM	175	M-W	$0.5N/P$ Flops/byte
2D-FFT	1843	SYST	$5\log N/16$ Flops/byte
LU	296	M-W	$0.08N/P$ Flops/byte
Erosion	2164	SYST	$.75\sqrt{N}/P$ Flops/byte
ICM	21199	DF	*n/a*
Heli. wake	1609	M-W	$7N/P$ Flops/byte

Table 6.1: An overview of the Applications. M-W is Master-Worker, SYST is Systolic, DF is Data Flow. N denotes the problem size in terms of an NxN matrix. P is the number of processors. For the programs allowing arbitrary blocking (Clump MM and DNA) we assumed optimal blocking, i.e. the coarsest blocking allowing full parallelism.

matrix multiply and erosion codes were written by us with foreknowledge of the iPSC/2 implementation.

In addition, the applications represent a wide range of granularity and programming style. The next section will present actual granularity measured from running programs. Table 6.1 presents a more abstract view of each program, listing the size of the source code, a single-phrase description of the programming method, and a theoretical measure of granularity. Here the granularity measure considers only the ratio of arithmetic (integer or floating point) operations performed to bytes communicated for various problem sizes. The granularity ranges more than three orders of magnitude for a given problem size, from the fine grained LU and Clumped Matrix Multiply, to the coarse Helicopter Wake Simulation.

Likewise, there are three very distinct programming styles represented. Several of the programs are written in a "master-worker" style, in which a single master creates a number of identical workers and dumps tasks into Tuple Space. The workers pick up tasks at their own pace. This programming method has several attractive characteristics, including good load balancing and easy algorithmic scalability.

Others are written in a more "systolic" style, where each of a number of identical processes attends to a specific portion of the problem, pumping data to its logical neighbors. This style is more rigid than master-worker, and can often save communication, albeit at the cost of flexibility, since each process is typically allotted a fixed amount of computation.

Finally, one application (ICM) was written in a heterogeneous dataflow style. A large number of distinct processes coexist, asynchronously pumping data to one another. Data arriving on a given process causes it to pump data to some other processes.

6.2 General discussion of the experiments

The first five applications presented were subjected to extensive testing, in most cases including trying different problem configurations that varied the computation to communication ratios. We were interested in running the applications with relatively high rates of communication (smaller problem size, or finer blocking), since this way we could see the effect of the optimizations more clearly. We also ran each at a coarser granularity, in order to judge performance under more favorable conditions. In addition, each problem was run under at least eight of the following optimization levels:

1. No optimizations.
2. Memory caching only.
3. In/out collapse only.
4. Local data caching only.
5. Random queue only.
6. Tuple broadcast only.
7. Rehashing only.
8. All optimizations.
9. All optimizations except local data caching.

The last three applications are much larger programs, consisting of thousands of lines of code. All three were written in part or in entirety by programmers not expert in Linda programming. These applications were not subject to the same exhaustive testing; we simply wanted to see how they performed with and without optimization.

In all, over one thousand experiments were run. Initially, we ran each trial three times, finding the mean and standard deviation. Experience showed that the timings were remarkably consistent, never varying more than 1%, so subsequent experiments were run only once each.

Most of the results will be presented graphically in a standard format, showing both the wallclock times and the speedups on the ordinate, and the number of processors along the abscissa. See example Figure 6.1. The wallclock times are joined by solid lines falling from left to right, with the scale on the left. The speedups are joined by dashed lines rising from left to right. We will sometimes distinguish between *total time* and *net time*. Total time includes the entire wallclock runtime, from program startup to final Linda cleanup. Net time is also wallclock time, but excludes some portion of the initialization; this varies by application and will be explained in each case. Net time, when used, may be justified by one of the following reasons:

1. We want to compare the Linda program's performance to that of a native (message passing) version, and that version excluded the initialization.

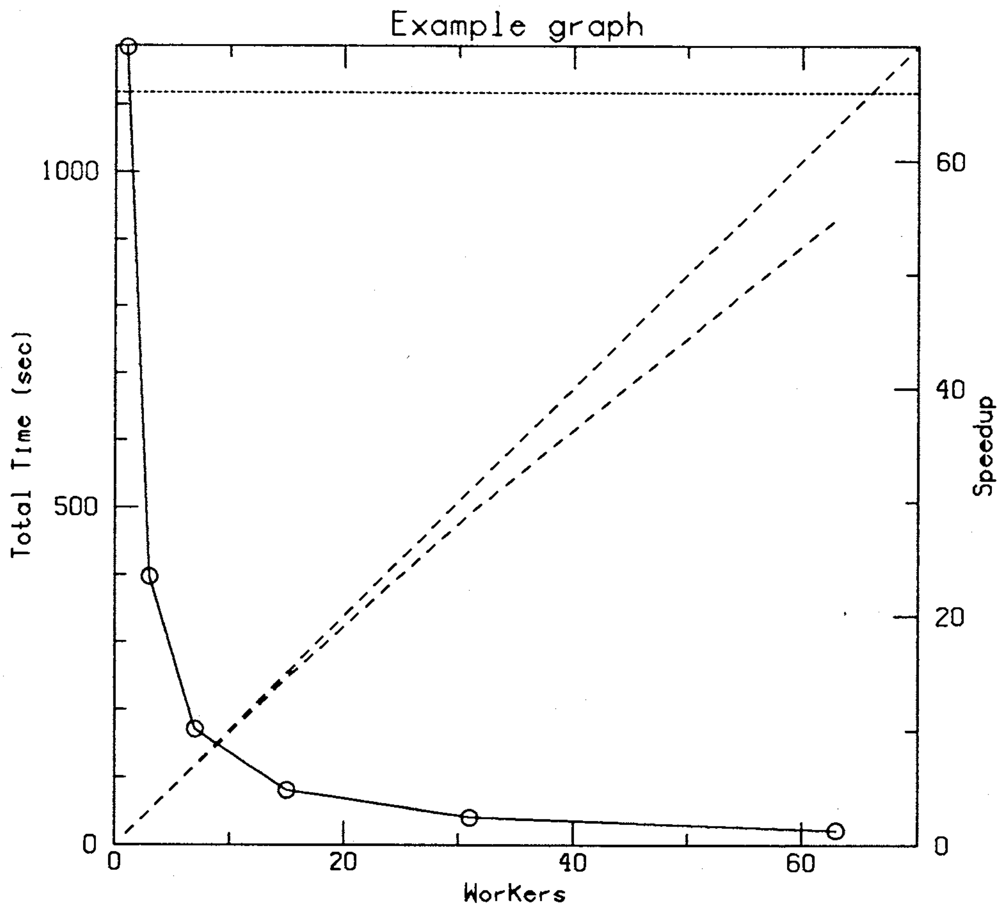

Figure 6.1: Example graph, showing wallclock time (solid line), speedup (dashed line), sequential time (dotted line).

2. The program could reasonably expect the data to be in Tuple Space already.

3. In one case (Clumped Matrix Multiply) initialization had to be slowed artificially, to avoid a quirk of the iPSC/2.

The number of processors is given either as workers or processors. Processors indicates the total number of processors involved in the computation, whereas workers is one less than the processors, since one processor is acting as the master. For comparison, perfect speedup is represented by a diagonal dashed line. Whenever possible, the speedups are in comparison with a sequential version of the code having a computational kernel as close as possible to that of the Linda program. In a few cases, we did use the time for the single worker Linda program. However, in each of these cases the amount of communication performed was insignificant, so the time obtained should be very close to a true sequential time. In each of the figures we indicate the basis for speedup. If available, times for a sequential version of the program are indicated by a horizontal dotted line. In several cases we also present times for hand-coded message-passing versions of the program.

The following sections concentrate on the fully optimized performance. Appendix C presents the full data, including each optimization run alone.

6.3 DNA sequence comparison

This algorithm takes a target DNA sequence and compares it against a data base of sequences, searching for the closest match. The comparison used is an $O(n^2)$ algorithm developed by Gotoh [Got82]. Computing the closeness of two sequences of length m and n involves computing an $m \times n$ similarity matrix, where each element of the matrix has a data dependency on its north, west, and northwest neighbors. The computation is performed entirely in integer arithmetic, and requires about 22 integer arithmetic operations per matrix element.

The data base used for the experiments was part of the GenBank[1] repository of genetic sequences from a variety of species. At the time, the full primate subsection of the database contained almost 2 million base pairs. For the experiments described below, we used a small portion of the primate subsection that contained 730 sequences totaling 163452 base pairs with a median length of 194. The longest sequence had a length of 468; the shortest 18. The test sequence we used had 468 base pairs.

An initial attempt to parallelize the sequential algorithm suggested two potential sources of parallelism. First, the target sequence could be compared to many sequences simultaneously. Secondly, individual comparisons could be parallelized. The first approach has a potential for good efficiency, since each comparison is independent, so communication will be minimized. However, load balancing can be a significant problem, due to the wide variation in lengths of sequences in the full data base. One of the primate sequences contains 70000 bases, or more than 3% of the total. Unless we can parallelize these long comparisons, our maximum speedup will be limited when using the full database.

We chose a hybrid master-worker approach combining the two approaches. We compute several comparisons in parallel, but if a comparison is expensive, we split it up into blocks, allowing several workers to attack it in unison. Good efficiency demands that the block size be small enough to allow significant parallelism, yet large enough that the cost of communicating edge values is acceptably low. Several partially computed matrices are kept in Tuple Space at all times by using a watermarking algorithm. Workers crawl over the matrices, computing blocks that are enabled based on their data dependencies. When a matrix is completed, the similarity value is communicated to the master. In this way, small comparisons are represented by single blocks minimizing communication, while larger comparisons containing many blocks are computed in parallel.

Figure 6.2 shows a schematic of the computation of one particular comparison. At any given time, the computation forms a diagonal wavefront across the similarity matrix. After a block is computed, the east and south neighboring blocks are enabled, and the edge values are communicated. In order to save communication, the process finishing a particular block always acquires the block to the east, if one exists, as the next task, which avoids the need of communicating the right edge values between the two. Thus a "task" actually consists of computing an entire row of blocks, although bottom edge values are communicated to the southern neighbor after each block.[2] Note that when we say "south neighbor" we mean the *logical* neighbor, i.e. the process that happened to pick up the task

[1] Maintained by the Department of Energy at Los Alamos National Laboratories.

[2] This hybrid algorithm was developed by Carriero[CG90].

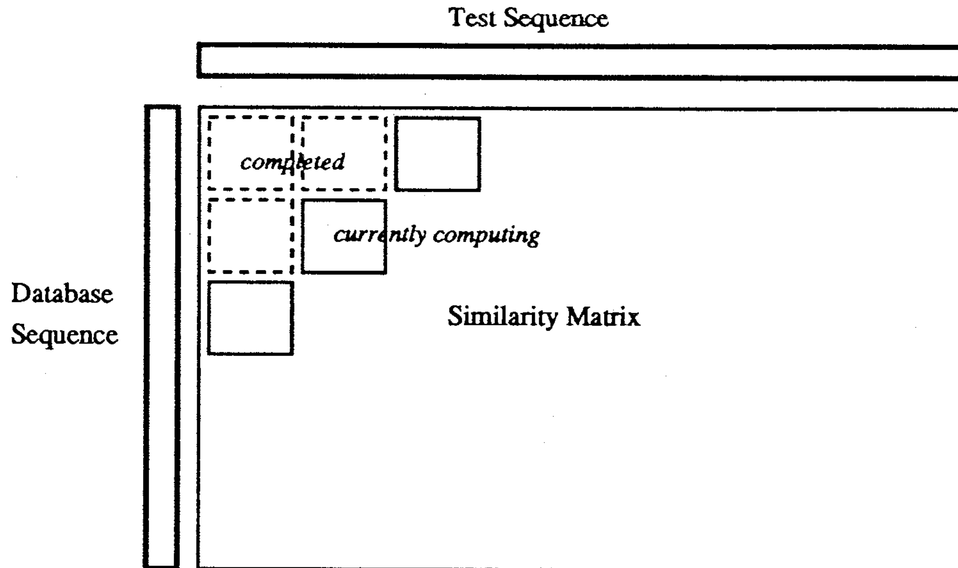

Figure 6.2: Schematic of DNA hybrid algorithm. Enabled blocks in solid, completed blocks dashed.

corresponding to the next row of blocks. We are not implying anything about the actual location of the that process, and in fact a process's neighbors will change for each task that it computes. The program is not in any way tuned to be particularly well-suited for a hypercube. Section B.1 presents the text of the program.

6.3.1 Granularity

When computing an $m \times n$ sequence comparison using blocks of size $x \times y$, computing one block requires $22xy$ integer operations. After finishing the block, the lower edge is communicated, requiring the sending of $12y$ bytes. This gives us a granularity of:

$$\frac{22xy}{12y} = 1.8x \text{ ops/byte} \approx 0.3 \text{ FP ops/byte}$$

In the special case that $x > m$, no vertical sub-blocking is done, so no bottom edges are sent. In this case, the only communication is the initial sequences and the final result, or $\approx n + m$ bytes. giving a granularity of:

$$\frac{22nm}{n+m} \text{ ops/byte}$$

6.3.2 Performance

We ran the program using three radically different blocking strategies in order to see the effects of different granularities. Figure 6.3 shows the performance of the program when blocking the database sequence by 1000 and the test sequence by 40 elements. This blocking size is such that no intra-comparison parallelism is possible, since the vertical blocking is greater than the maximum sequence length for our subsection of the database. This blocking results in coarse granularity, about 10,000 integer operations/byte. Most of the efficiency loss for 63 workers is due to a startup period of about about 2 seconds, during which the worker processes are starting up and the master is filling Tuple Space.

For this program we use the time taken by a sequential version of the code as a basis for speedup. In order to draw the fairest possible comparison, we ignore the time the sequential program spends reading the database. Because the iPSC/2 has a low I/O bandwidth, the sequential time would be significantly greater than the 1 worker parallel time since the master does the I/O in the parallel program. We also do not include the master in the number of nodes for purposes of speedup.

Figure 6.4 shows the performance of the program using blocks of 40 by 40 elements. This blocking size allows considerable intra-task parallelism but has a higher communication cost, reflected in the lower speedups reached than for the 1000 by 40 case. Here the granularity is about 73 integer operations/byte.

Figure 6.5 shows the performance for yet another blocking, namely 3 by 500 elements. This blocking greatly increases the inter-task communication, and in this particular case actually renders each comparison sequential, since the horizontal blocking (500) is greater than the length of the test sequence. The granularity for this blocking is 5.5 integer operations/byte.

Several optimizations contributed to the performance of the program by reducing communication cost, although this is only clearly visible in the most communication intensive example (3 by 500 blocking). In particular, this case shows considerable improvement from the random queue method[3]. See Figure 6.5. The sudden decrease in efficiency experienced by the unoptimized program is due to contention for task tuples, which is avoided by the randomized method. The full suite of runs, under varying levels of optimization, appears in Appendix C.

For this application, the rehashing optimization aids primarily in comparisons of two very long sequences. To see why, consider the communication pattern between a process working on a particular row of blocks with the process working on the blocks directly south of it. As each block is finished, the edge values are **out**ed by the first process, and eventually **in**ed by the second. This communication pattern will last only through one row of blocks; unless there are a reasonable number of blocks along a row, the system doesn't have time to recognize, rehash, and take advantage of the pattern before the pattern changes.[4] Since the target sequence was only 468 bases long, the effects of the optimization were not discernible in these experiments. In order to gauge the effect of the optimization, we ran a single comparison of two 6780 base sequences, with and without rehashing using 40×40

[3]The size of the random queue subset was $\sqrt{N}$ where N is the number of nodes.

[4]For all of our experiments, the rehashing threshold was 3.

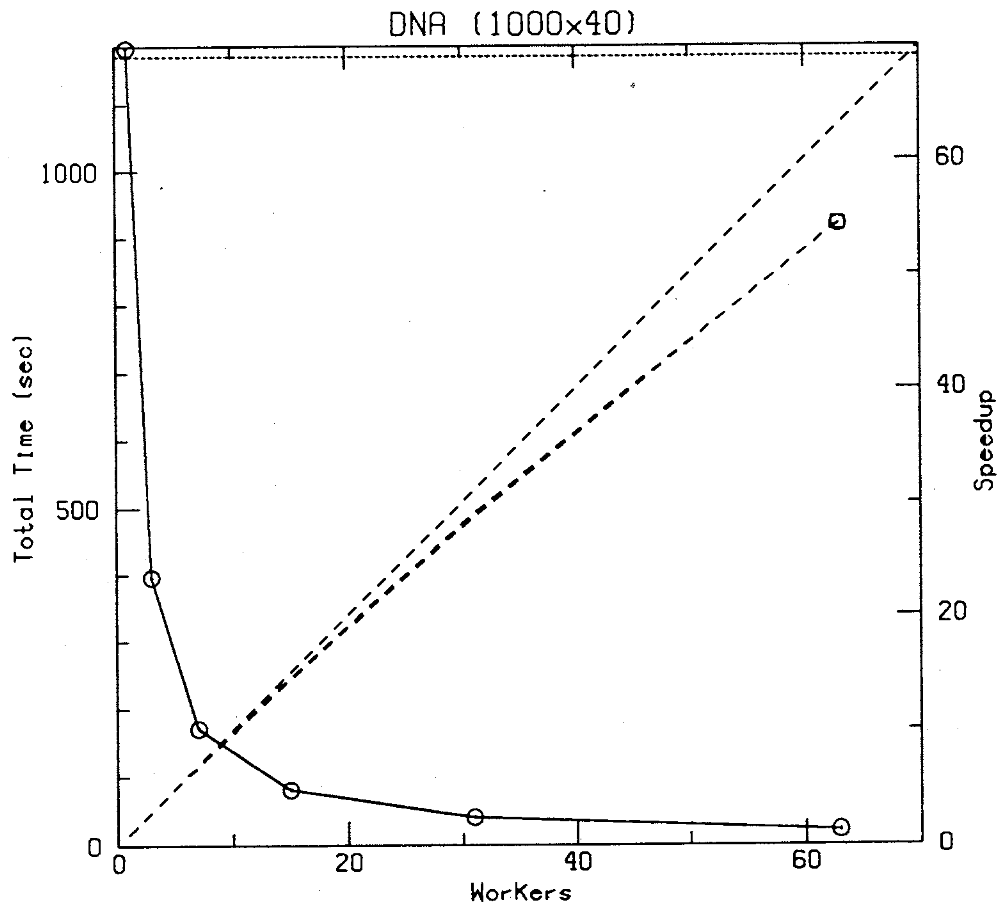

Figure 6.3: DNA Database Search. 730 sequences, totaling 163000 bases, compared against 468 base test sequence. 1000×40 blocking. Total time of computation, including setup. ○ represents all optimizations enabled, □ represents no optimizations enabled. Speedup relative to sequential program.

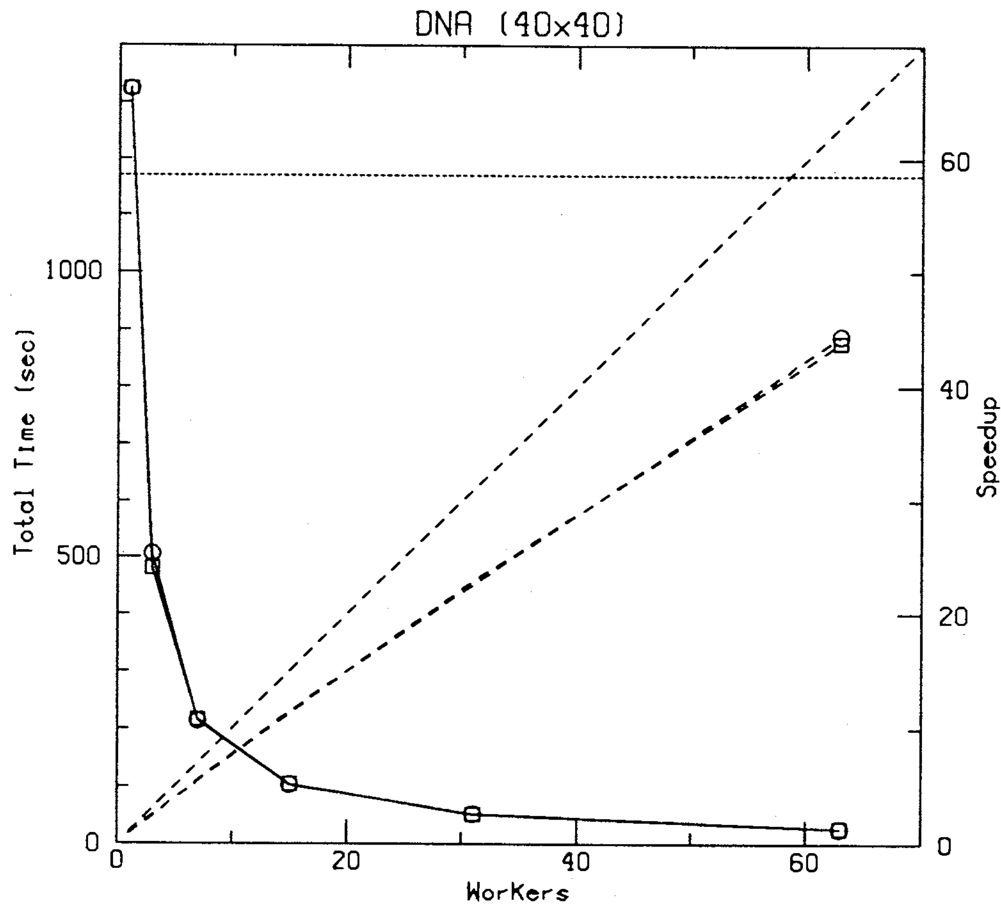

Figure 6.4: DNA Database Search. 730 sequences, totaling 163000 bases, compared against 468 base test sequence. 40×40 blocking. Total time of computation, including setup. ○ represents all optimizations enabled, □ represents no optimizations enabled. Speedup relative to sequential program.

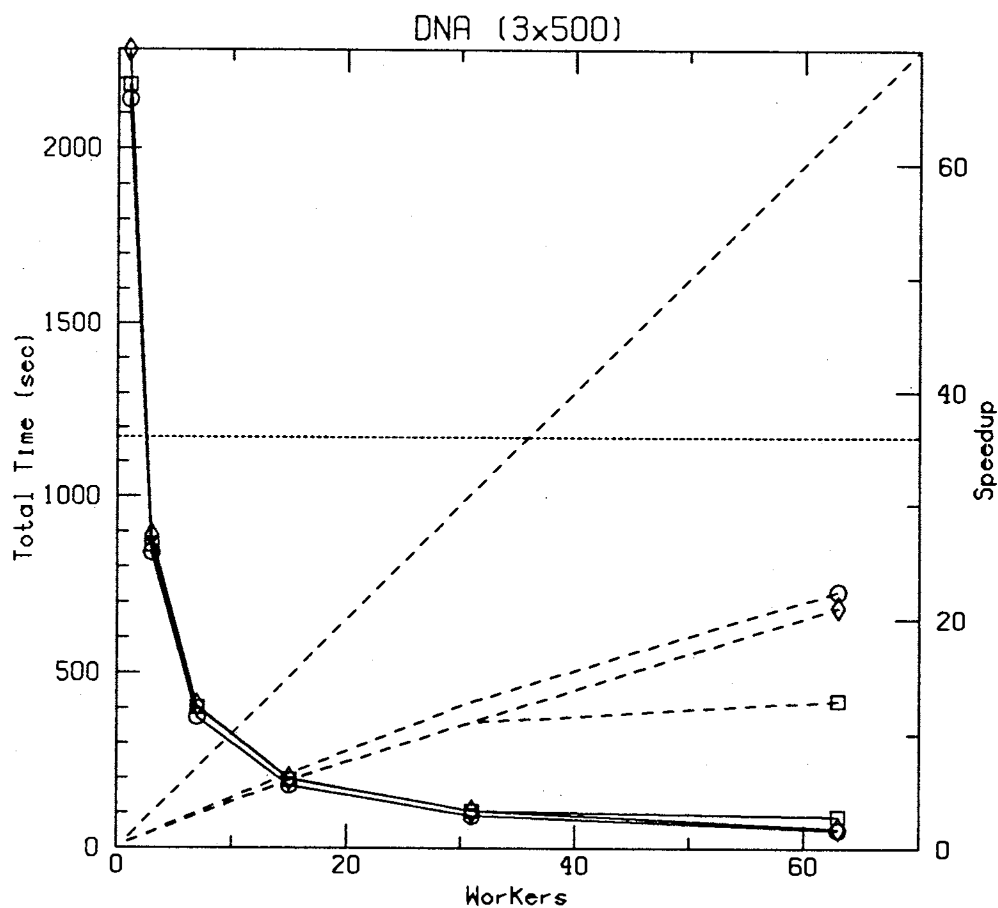

Figure 6.5: DNA Database Search. 730 sequences, totaling 163000 bases, compared against 468 base test sequence. 3×500 blocking. Total time of computation, including setup. ○ represents all optimizations enabled, ◇ represents just random queue enabled, □ represents no optimizations enabled. Speedup relative to sequential program.

blocking. Figure 6.6 shows the results.

Figure 6.7 shows the performance of the hybrid approach running fully optimized on the full primate database, including the very long sequences, using 1000 by 40 blocking. The granularity is finer for this case than the smaller database with the same blocking, since there are sequences in the database long enough to require the passing of block edges.

6.3.3 Summary

Linda using the hybrid approach gives excellent performance with reasonably coarse blocking. Finer blocking increases the communication overhead, and performance drops off somewhat. Very fine task blocking (the 3×500 case) causes the task tuples to become a bottleneck; the random queue optimization solves this problem. Large sequence comparisons (the 6780×6780 case) show good improvement from the rehashing optimization, since the communication patterns persist long enough to be recognized and used.

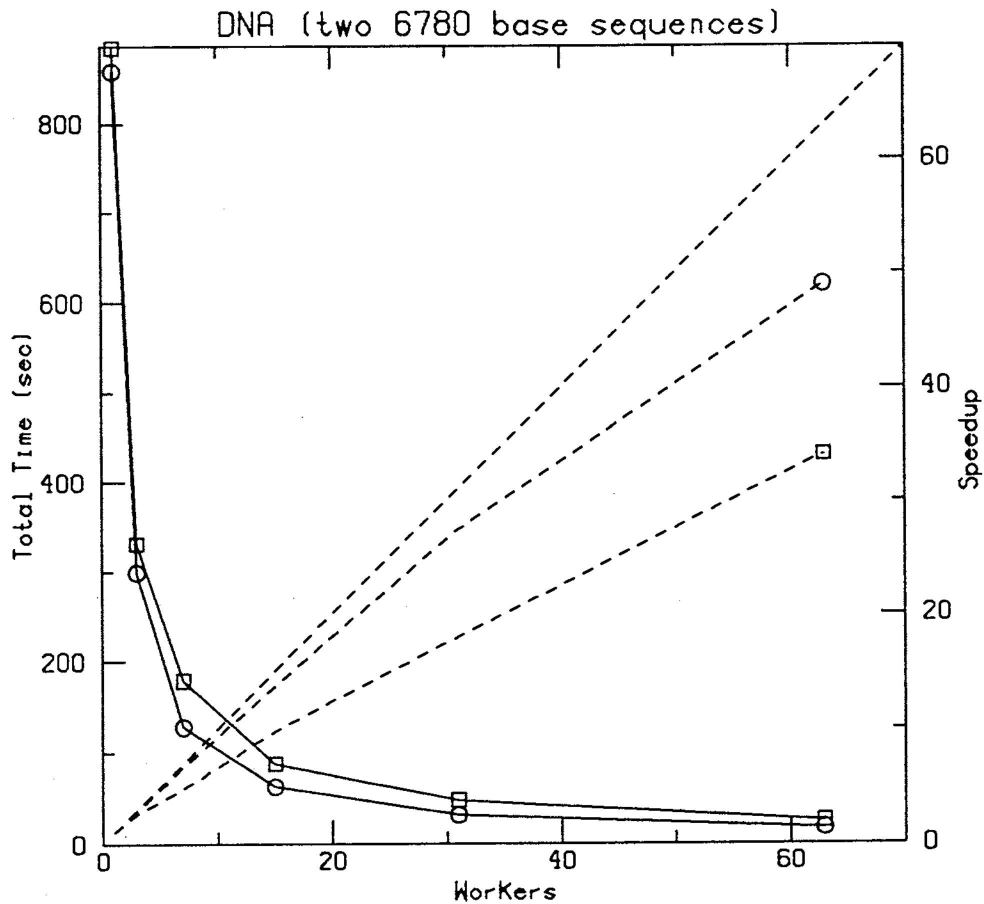

Figure 6.6: DNA Database Search, comparing two 6780 base sequences. 40×40 blocking. Total time of computation, including setup. ○ represents rehashing enabled, □ represents no optimization. Speedup relative to sequential program.

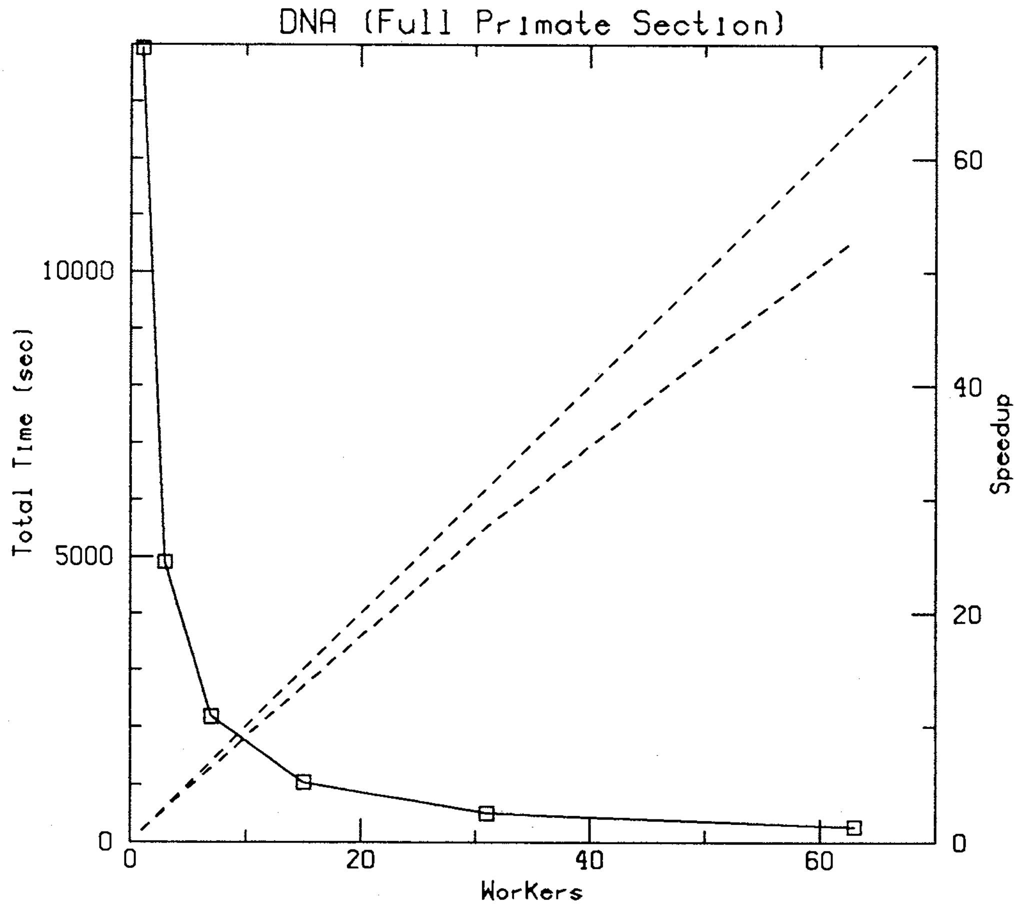

Figure 6.7: DNA Database Search. 1565 sequences, totaling 1930507 bases, compared against 468 base test sequence. 1000×40 blocking. Total time of computation including setup, with all optimizations enabled. Speedup relative to 1 worker Linda program.

6.4 Block matrix multiply

This algorithm (Dekel *et al.* [DNS81]) performs dense matrix multiplication, using an approach particularly well suited to two-dimensional mesh architectures.

Assume that we have two $n \times n$ matrices A and B (padding with zeros if necessary.) We map them onto an $n \times n$ mesh of processors P:

$$p_{i,j} \leftarrow a_{i,i \oplus j} \quad and \quad b_{i \oplus j,j}$$

where $\oplus$ signifies modulo n addition. Now we perform n steps. At each step, each node of the mesh computes the product of its elements and adds it to an accumulated sum. Then, both matrices are rotated on the mesh, A row-wise, B column-wise. $m_{i,j}$ sends its element of A to $p_{i \oplus 1,j}$ and its element of B to $p_{i,j \oplus 1}$. After the n steps, $p_{i,j}$ holds $c_{i,j}$ of the resultant matrix $C = AB$.

In order to increase efficiency, instead of mapping one element to a particular node, we map a subblock of size $n/k \times n/k$, where $k = \sqrt{\lceil P \rceil}$. Computing the product on each node now consists of performing matrix multiplication on the two subblocks, and the communication phase likewise sends subblocks, with a total of k steps. The Linda code is presented in Section B.2.

6.4.1 Granularity

With P processors, each processor multiplies two $N/\sqrt{P} \times N/\sqrt{P}$ subblocks, requiring $\frac{2N^3}{P^{3/2}}$ FP ops,. Each processor then sends each subblock, totalling $\frac{8N^2}{P}$ bytes. The granularity is:

$$\frac{2N^3P}{8P^{3/2}N^2} = \frac{N}{4\sqrt{P}} \text{ FP ops/byte}$$

6.4.2 Performance

Figure 6.8 shows the performance of this program on the iPSC/2, multiplying 120x120 matrices[5]. Because the time to perform such an operation is so small, roughly 5 seconds on a single processor, the multiply was repeated 10 times, effectively calculating $C = (\cdots(A \times B) \times B \times B)\cdots B)$[6]. The granularity for this problem on 64 nodes is 3.75 Flops/byte. We chose to run a relatively small problem with repeated multiplies since this emphasizes communication; a larger problem would have very good but uninteresting performance since it would be so coarse grained. We used single precision elements (floats) although the actual computations were promoted to double precision, as is always the case in C.

[5] Because of the nature of the program, the height of all blocks must be identical, as must the width. This means that the square root of the number of workers must divide each dimension of the matrices. The smallest number divided by the integers 1...8 is 840, far too large. Instead, we used 120 and 240, both divisible by all except 7. For 7 we used 119 and 238 respectively, scaling up the resulting times appropriately.

[6] Repeating the multiplication also enhanced the effect of the rehashing optimization, since the communication pattern persisted longer.

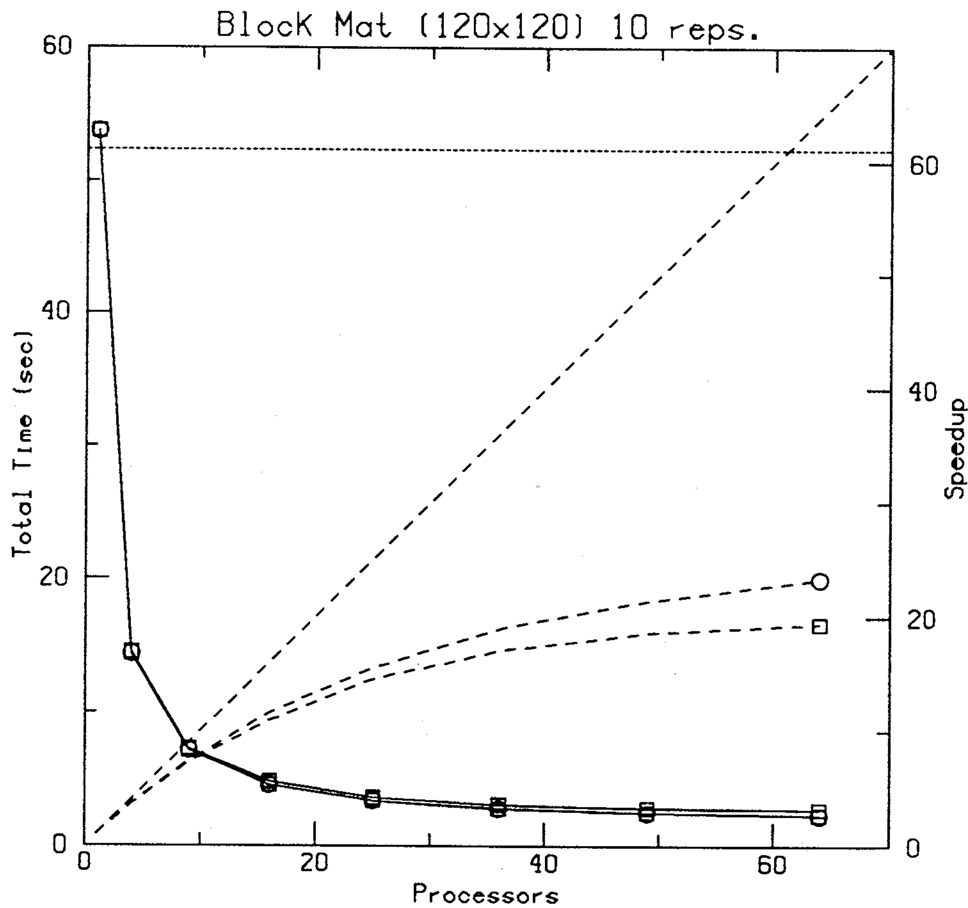

Figure 6.8: Dekel dense matrix multiplication on two 120x120 matrices, 10 repetitions, total time. All optimizations enabled ○ and no optimization □ . Speedup relative to sequential program.

The sequential time shown is for a program that performed the inner-loop dot products via the same computational kernel as the parallel code. In both cases, care was taken to ensure that the inner-loop variables went into registers. The loop was not unrolled, however. In this section, all speedups are relative to this sequential program.

For this particular program, the only optimization with a significant effect was rehashing. Curve ○ represents enabling all optimizations, while curve □ represents no optimization. Figure 6.8 shows the total run time, including the cost of generating and `out`ing the matrices and creating the workers. Figure 6.9 ignores this cost, beginning the timing after the matrices are `out`ed and the workers `eval`ed. Both times include `in`ing the results.

Figures 6.10 and 6.11 show the performance on a larger problem (240x240). The granularity for this problem on 64 nodes is 7.5 Flops/byte. The dashed curve in figure 6.11 (labeled ◇) shows the performance of a handcoded message-passing version of the program (written by us).

6.4.3 Summary

The Linda program performs very well, virtually identical in fact to the hand coded version on the 240×240 case. The kernel quickly recognizes the communication pattern. From that point on, almost all communications go directly to where they will be needed. Even the small 120×120 case still showed 50% efficiency with 64 processors; the 240×240 problem reached 85% and still larger problems should show even better performance, since the computation to communication ratio (i.e. granularity) will continue to rise. The rehashing optimization improved performance significantly, particularly on the smaller problem where a larger proportion of time was spent communicating.

In general, this algorithm has a well-behaved communication style that avoids hot spots. This is not surprising, since it was originally developed for mesh topologies. In the next section, we will examine an alternative dense matrix multiplication program that has a very different communication pattern.

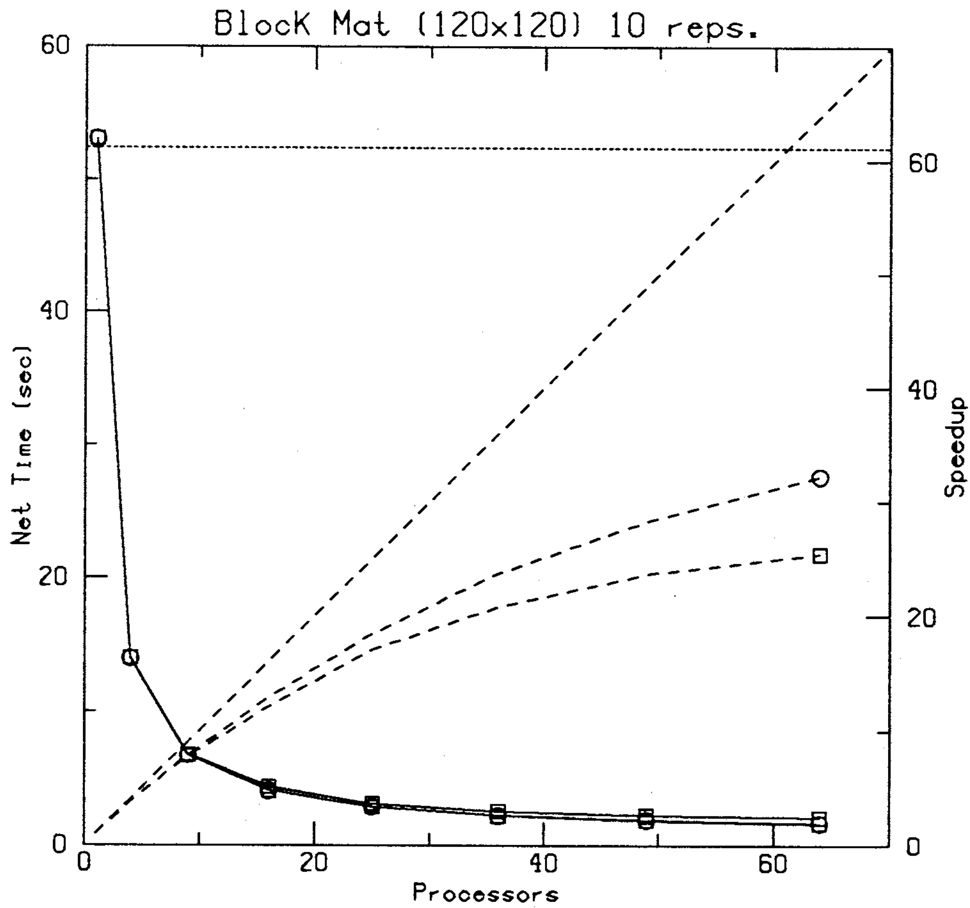

Figure 6.9: Block matrix multiplication on two 120x120 matrices, 10 repetitions, net time (ignoring initialization). All optimizations enabled ○ and no optimization □ . Speedup relative to sequential program.

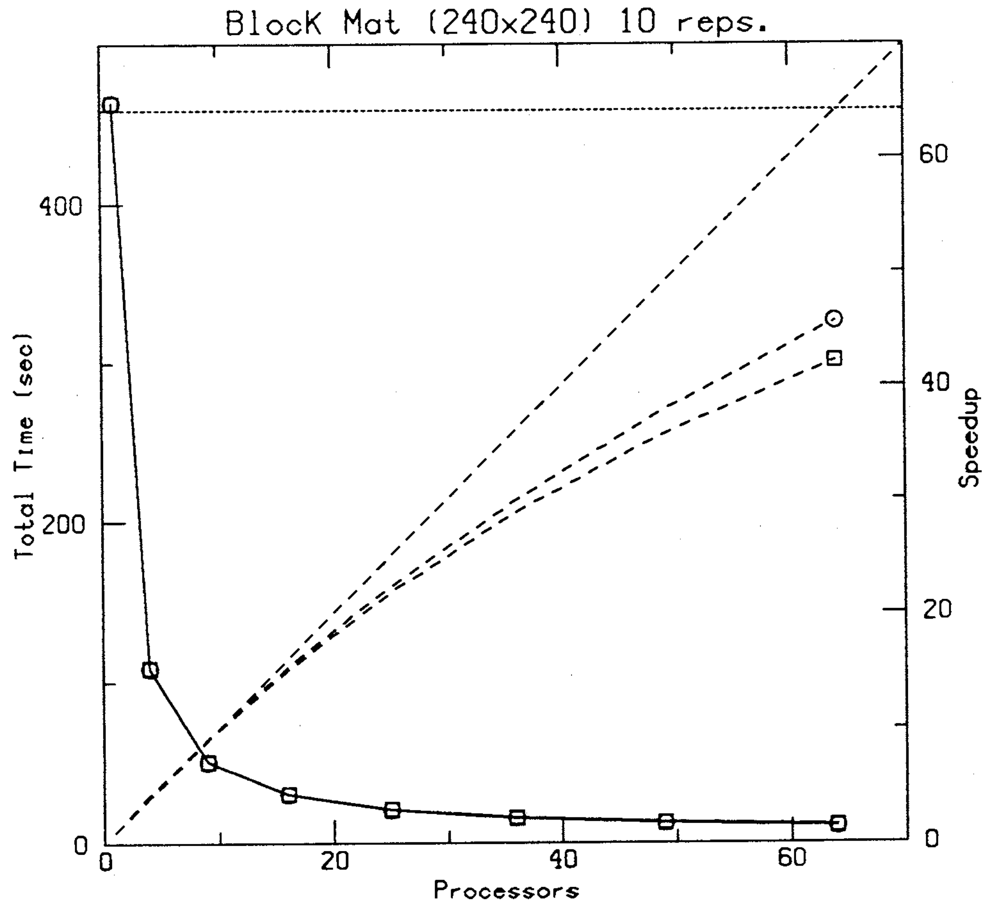

Figure 6.10: Block matrix multiplication on two 240x240 matrices, 10 repetitions, total time. All optimizations enabled ○ and no optimization □ . Speedup relative to sequential program.

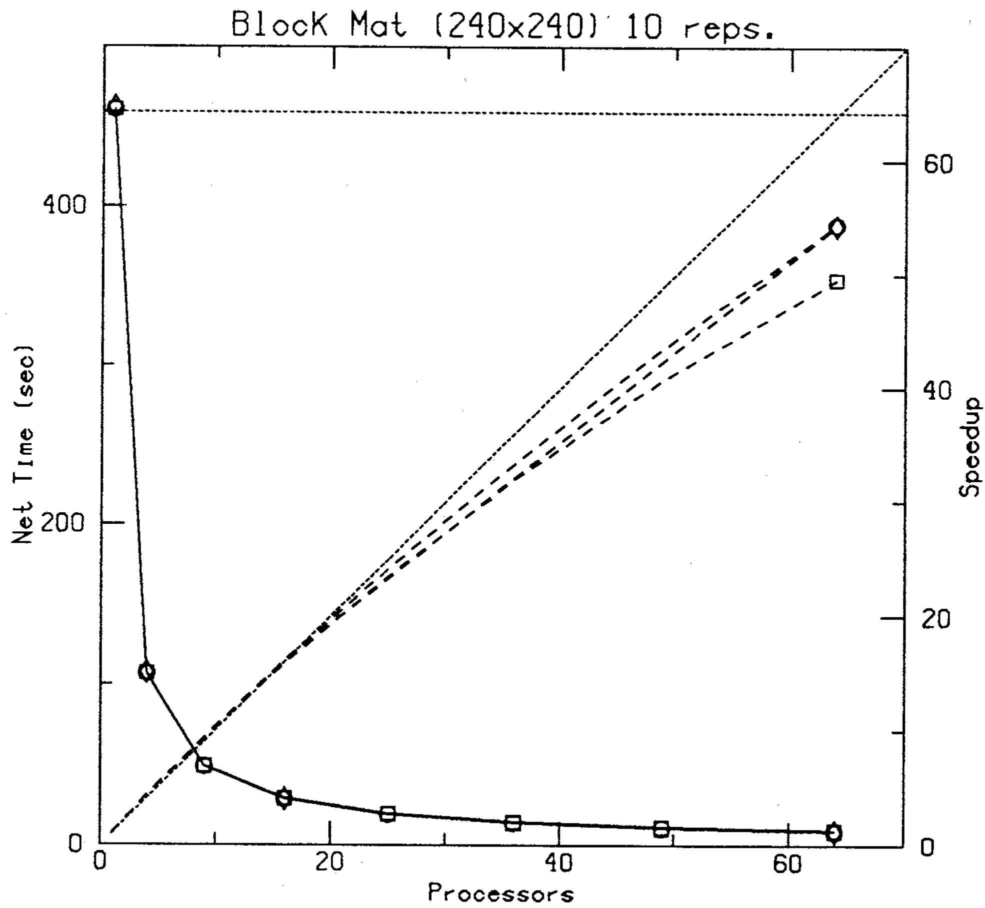

Figure 6.11: Dekel dense matrix multiplication on two 240x240 matrices, 10 repetitions, ignoring initialization. ○ represents all optimizations enabled. □ represents no optimization. ◇ represents a hand-coded message-passing version. Speedup relative to sequential program.

6.5 Clumped matrix multiply

In this section we present an alternate algorithm for dense matrix multiplication. Given that the Dekel algorithm showed good performance, why would we be interested in a different algorithm? First, as a systolic-style algorithm, the Dekel algorithm makes some strong assumptions about the hardware on which it runs. The number of active processors must be a square. In addition, for good results, each processor must operate at the same level of performance, since the algorithm is logically tightly coupled. The total performance will be limited by the performance of the weakest component.

The clumped algorithm, in contrast, uses the master/worker model and displays the uncoupling and inherent load balancing typical of such algorithms. It will run on any number of nodes and adjusts well to collections of nodes of varying compute strength. Both of these characteristics make it the better method for use on a network of workstations, for instance. Finally, running both algorithms, with their different communication patterns, allows us to compare Linda's performance on each one.

Assume that we are multiplying $A \times B$. We put the rows of A and the columns of B into Tuple Space. Instead of storing a single row of A or column of B in a tuple, we clump several together; the clumping factor is chosen to reduce communication cost while still allowing sufficient parallelism. A's clumping effects the total *amount* of data communicated, while B's clumping only reduces the number of communication startups. We then `eval` a number of workers and `out` a task tuple. A task consists of calculating one row-wise clump of the result matrix C (its clumping is identical to A's). Each worker loops, getting a task from the task tuple and `rd`ing the appropriate row clump of A. It then loops across B, `rd`ing each of B's clumps in turn and performing the appropriate inner products to form one clump of C. This clump is `out`ed, and the worker looks for more work. Section B.3 presents the Linda program.

6.5.1 Granularity

A task consists of multiplying one clump of A against all of B. This requires $2aN^2$ FP ops, and the communication of $4(aN + N^2)$ bytes, resulting in a granularity of:

$$\frac{2aN^2}{4aN + 4N^2} \approx \frac{a}{2} \text{ FP ops/byte}$$

6.5.2 Performance

We ran the program on two different problem sizes, 400x400 and 240x240. In addition, we used four different clumping combinations, clumping A by 1 and 4 rows, and B by 1 and 8 columns. When timing this program we excluded the time to `out` the matrices, due to a quirk of the iPSC/2.[7] All speedups is this section are relative to the same sequential

[7] According to an Intel technical document, [Nug] when the iPSC/2 runs out of message buffers, communication is "throttled", slowing down messages tremendously in order to allow buffers to cleared. Our experience is that once communication is "throttled", it tends to stay throttled. Some Linda programs cause message buffers to be exhausted, usually those that do many repeated outs, and especially if those outs are broadcast. The clumped matrix multiply program exhibited this problem when outing the rows

matrix multiply code described for the block matrix multiply in Section 6.4. The achieved efficiency on 64 nodes was 68%.

Figure 6.12 shows the performance of this program multiplying two 400x400 matrices, using the best clumping for this size: a=1 and b=8. This corresponds to a granularity of .5 Flops/byte. The optimized version performs much better than the unoptimized version, mostly due to the effect of the tuple broadcast optimization, which effectively provides each processor with its own copy of A and B. As A and B grow larger, replicating each in its entirety on each node becomes problematic. Section 5.6 discusses improvements to the tuple broadcast optimization that address this problem. We reran this program with 64 nodes on a 504×504 problem, blocking A by 4 and B by 8. With this selection of parameters, each worker gets two tasks, load balancing problems are largely eliminated, and the larger granularity improves efficiency. Under these conditions, with rehashing and tuple broadcast enabled, we achieved 84% efficiency, again measured against the sequential program.

Figure 6.13 shows the performance on a smaller problem, using the same clumping.[8]

Figure 6.14 contrasts the performance of the 400x400 problem for all four clumping combinations. All optimizations were enabled.

6.5.3 Summary

Because this algorithm had a much finer granularity, it generally performed more poorly than the blocked matrix multiply. It benefited greatly from the tuple broadcast optimization, since it rendered the `rd`s of the columns of B entirely local. The rehashing optimization also helped by directing the result tuples to the master process.

---

and columns of the matrix. In order to avoid this problem, a delay of 2 milliseconds was inserted between each out during the initialization phase. Because of this artificial slowdown of the initialization phase, we choose to exclude it in the figures.

[8]The astute reader will notice that the times given are roughly 1/10th those reported for Dekel on the same problem size. This is because Dekel did 10 repetitions.

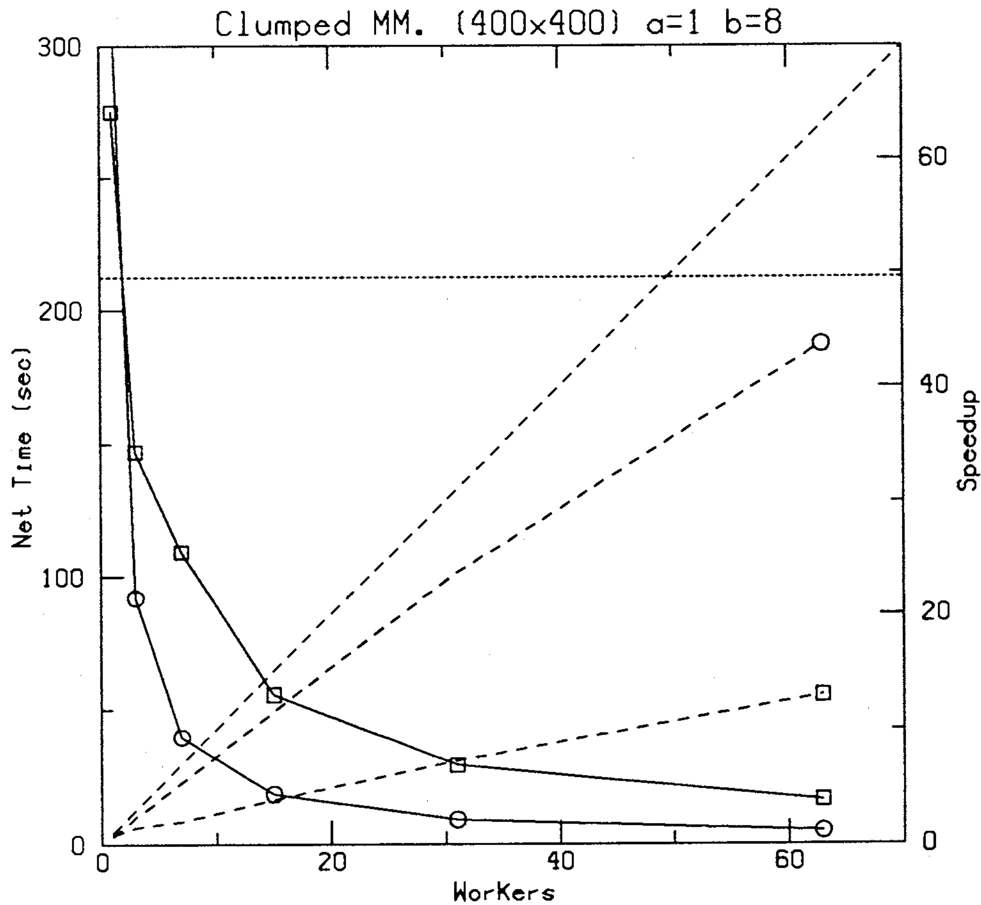

Figure 6.12: Dense Matrix Multiply, clumped version, dimension 400x400. All optimizations enabled ○ and none enabled □ . A was clumped by 1 and B by 8. Speedup relative to sequential program.

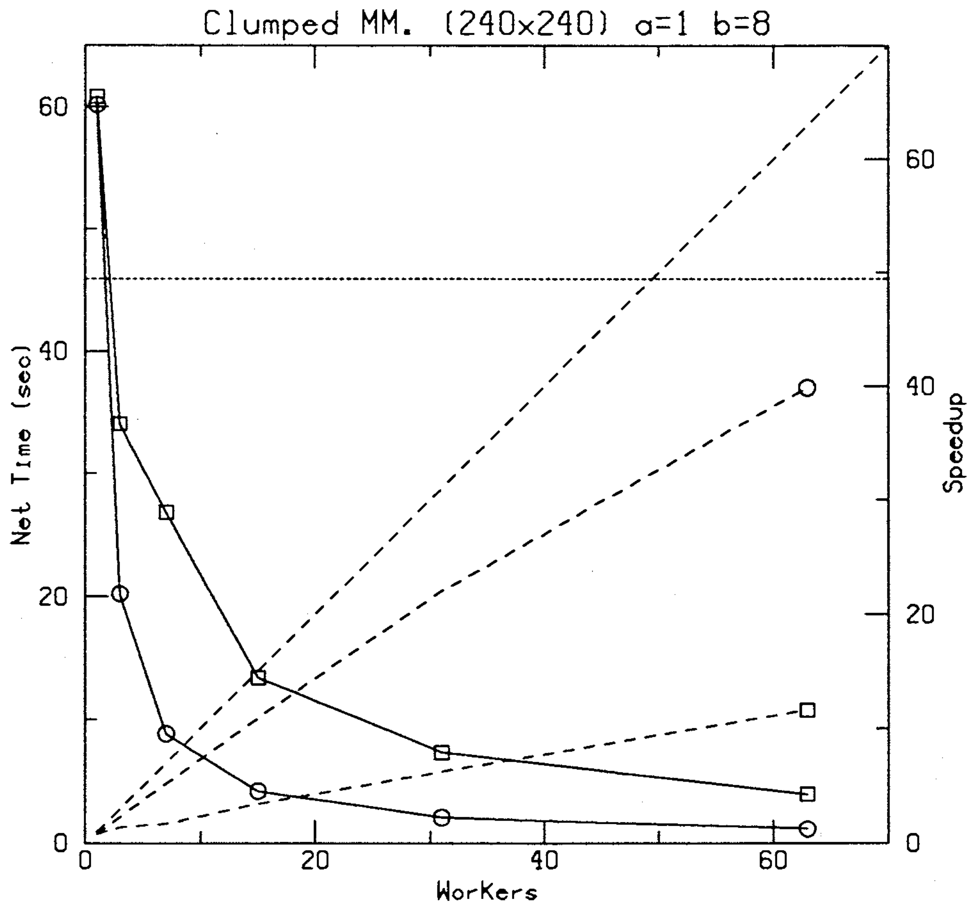

Figure 6.13: Dense Matrix Multiply, clumped version, dimension 240x240. All optimizations enabled ○ and none enabled □ . *A* was clumped by 1 and *B* by 8. Speedup relative to sequential program.

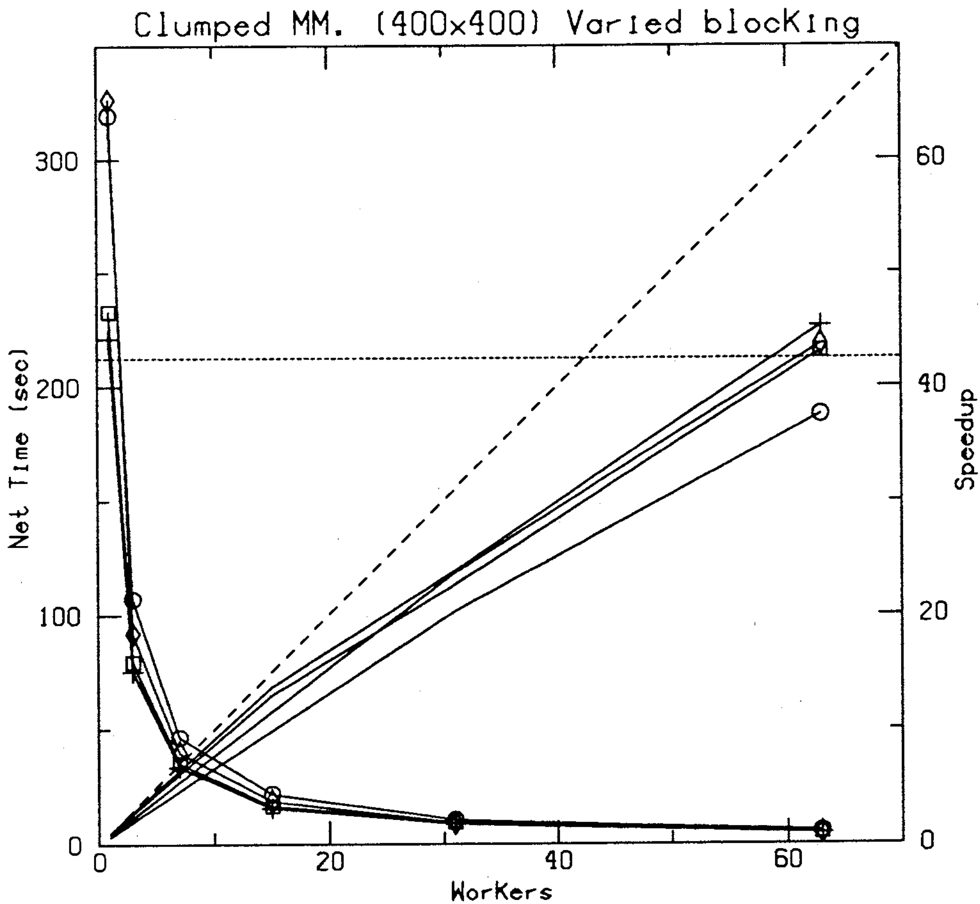

Figure 6.14: Dense Matrix Multiply, clumped version, dimension 400x400, showing effect of clumping on performance. + is a=4, b=8. ◇ is a=1, b=8. □ is a=4, b=1. ○ is a=1, b=1. All optimizations were enabled. Speedup relative to sequential program.

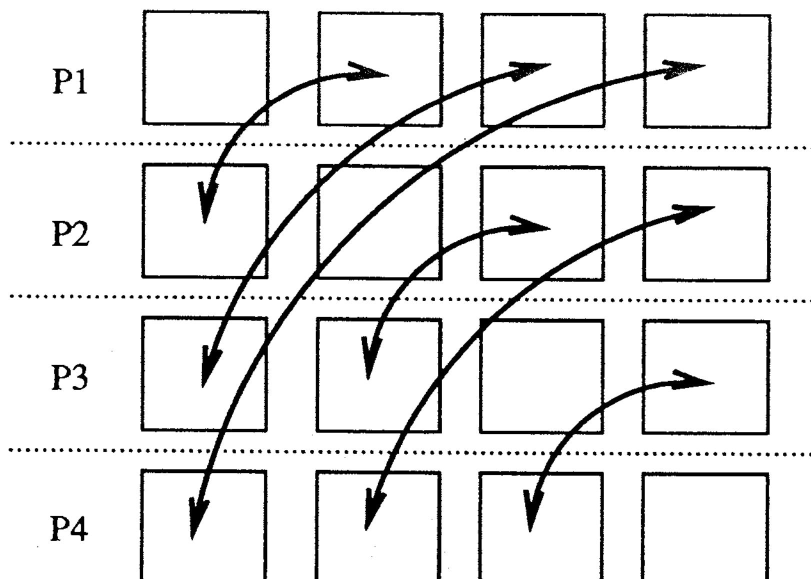

Figure 6.15: Schematic of 2D-FFT global transpose.

6.6 2D fast Fourier transform

This program performs a forward and backward fast fourier transform on a two dimensional matrix of single precision complex values. Multi-dimensional FFT's can be done by performing 1D-FFTs along each dimension in turn.

The algorithm was parallelized as follows: Assume an $n \times m$ matrix and p processes. Process i holds n/p rows of the matrix, starting with row $i * (n/p)$. These rows are further subdivided into p blocks, each $n/p \times m/p$. The following steps compute the 2D-FFT:

- Each process does a 1D-FFT on each row it holds.
- Transpose the matrix. This is performed as a number of pair-wise swaps of blocks of the matrix to get each block to the correct node, then local transposes of each block. A total of $(p^2 - p)/2$ paired exchanges are necessary, see Figure 6.15.
- Repeat the 1D-FFT step on the transposed matrix. Each process performs 1D-FFTs on each row it now holds. Each row corresponds to a column of the original matrix.
- Repeat the transpose, leaving each process with the FFT transform of its original rows.

This particular code began as a Fortran program written specifically for the iPSC/2 using message passing by an employee of Intel Scientific Computers, Inc., David Scott.

Ed Segall at Rutgers University rewrote the communication in Linda by creating small C-Linda communication routines that were called from the Fortran routines. Segall also used a 1D-FFT provided with Intel's math library. In order to make the Linda code portable to machines other than Intel hypercubes, we replaced this call with a public domain 1D-FFT code[9]. The performance of the netlib 1D-FFT was the same to within one percent of the Intel library routine. This was the only change we made to Segall's version.

6.6.1 Granularity

The computation is $\mathcal{O}(n^2 \log_2 n)$, since each 1D-FFT is $n \log_2 n$ and n must be done for each 2D-FFT. For the purpose of estimating granularity, we used 5 FP ops as the constant, or $5N^2 \log_2 N$ FP ops for a single pass across the entire matrix. Each pass requires that $8N^2 \frac{P-1}{P}$ bytes be transmitted. This gives a granularity of:

$$\frac{5N^2 \log_2 N}{8N^2 \frac{P-1}{P}} \approx \frac{5 \log_2 N}{8} \text{ FP ops/byte}$$

6.6.2 Performance

Our program actually did both a forward and a backward 2D-FFT, allowing us to compare the result against the original matrix. On the iPSC/2, for the 1024×1024 case on 64 nodes, the FFTs accounted for 83% and the transposes 17% of the total time. For the 256×256 case, the FFTs accounted for 37%, and the transposes 63% of the total time, of which the majority for the transposes was time spent communicating.

Figure 6.16 presents the performance on a 256x256 matrix, including a comparison with Scott's original message passing code. Speedup is based on the time of the message-passing version running on a single processor; however, this is virtually the same as a sequential time, since the matrix is represented as one large block and no communication occurs during transposes.

Figure 6.17 presents the performance on a 1024x1024 matrix. The 1024x1024 case could not be run on fewer than eight nodes due to memory limitations on the iPSC/2, so a sequential time was difficult to obtain. In this case speedup was based on an estimated sequential time of 862.4 seconds, calculated by taking eight times the eight processor message-passing case; this probably makes the speedup look somewhat better than it actually is, although this estimate is less (and thus produces lower speedups) than another estimate obtained by scaling the single processor 256×256 time (921.6 sec). Both of these timings exclude initialization, since that is how the original (message passing) version was written.

Figure 6.18 presents Segall's own data [PS91]. The graph shows closely the performance of the Linda version approached that for the native version. The Linda version performs at better than 90% for the 512x512 problem on 64 nodes and reaches almost 97% on the 1024x1024 problem..

[9]We used FFTPACK, a part of the netlib library available from netlib@ornl.gov. FFTPACK was written by Paul Swarztrauber of the National Center for Atmospheric Research.

6.6.3 Summary

The rehashing optimization was the most important for this application; it succeeded in reducing 95% of the communication to their point-to-point equivalents when running on 64 nodes. The Linda performance is very close to the native program's performance for problems of a reasonable size. The small difference remaining between the two is primarily due to the inefficient interrupt mechanism we were forced to use for Linda (see Chapter 4).

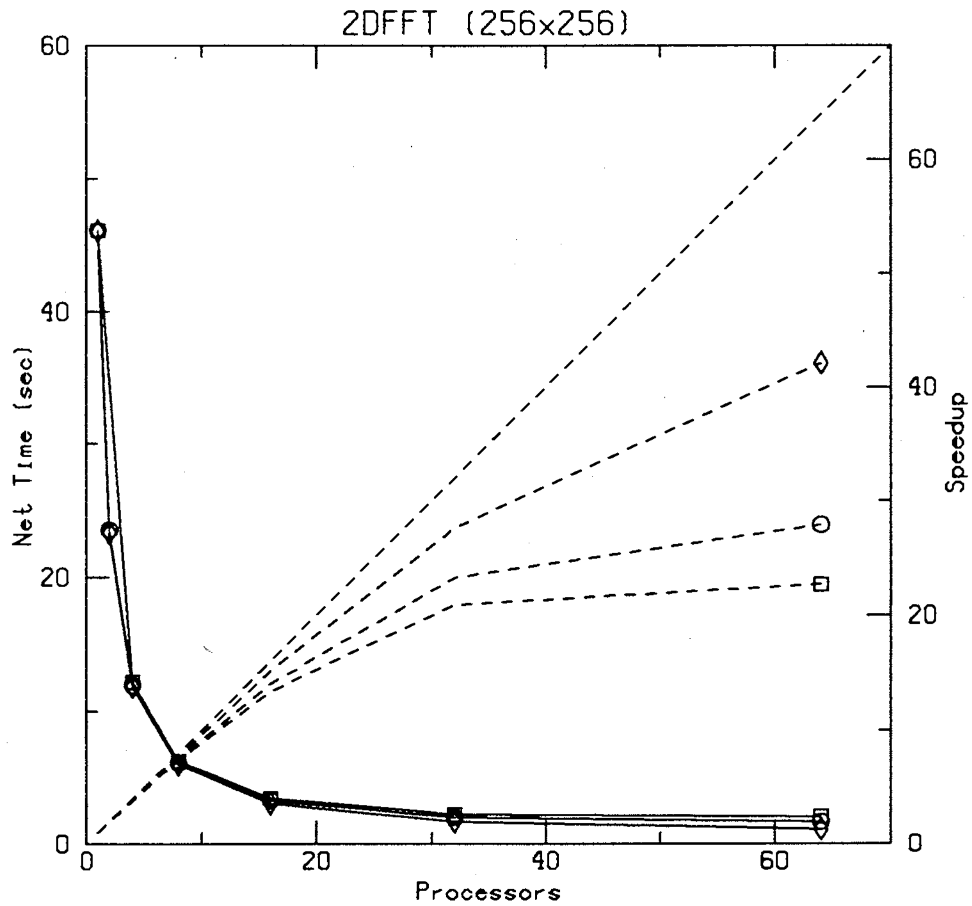

Figure 6.16: 2D-FFT, dimension 256x256. Net time, excluding initialization. All optimizations enabled ○ versus unoptimized version □ . ◇ is the native message passing version. Speedup is relative to the message passing one processor time.

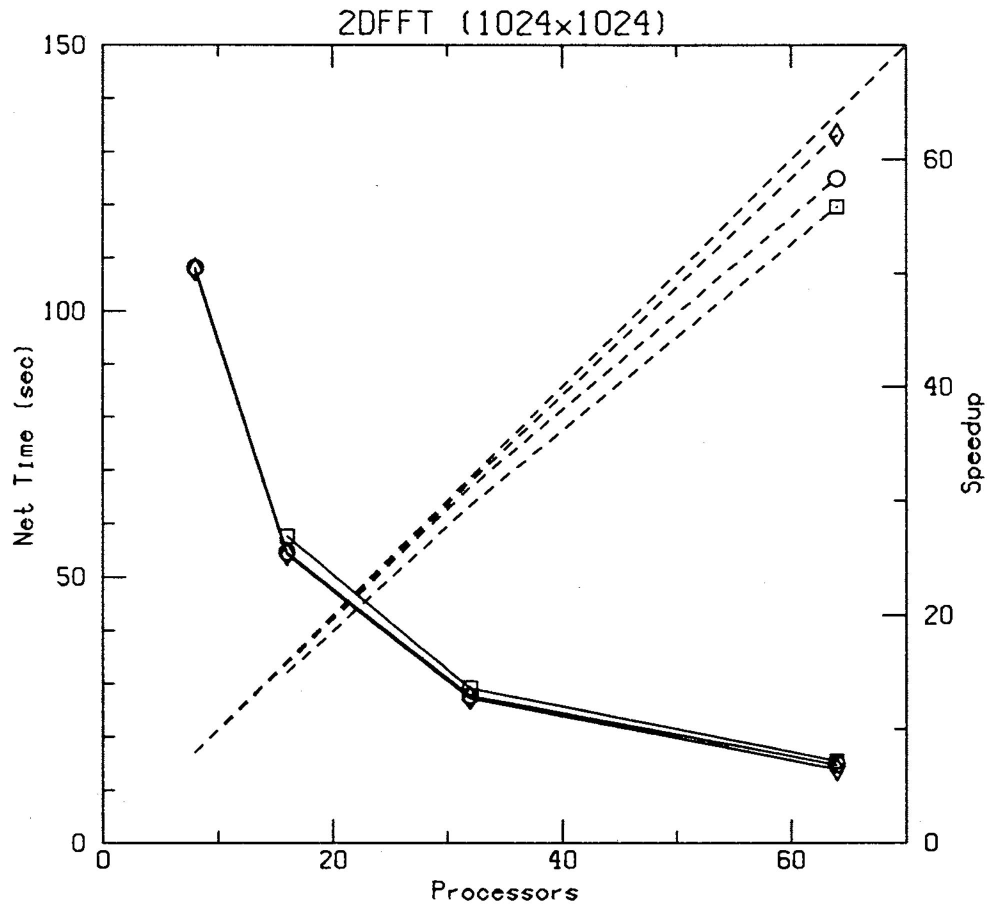

Figure 6.17: 2D-FFT, dimension 1024x1024. Net time, excluding initialization. All optimizations enabled ○ versus unoptimized version □ . ◇ is the native message passing version. Speedup is relative to the message passing eight processor time.

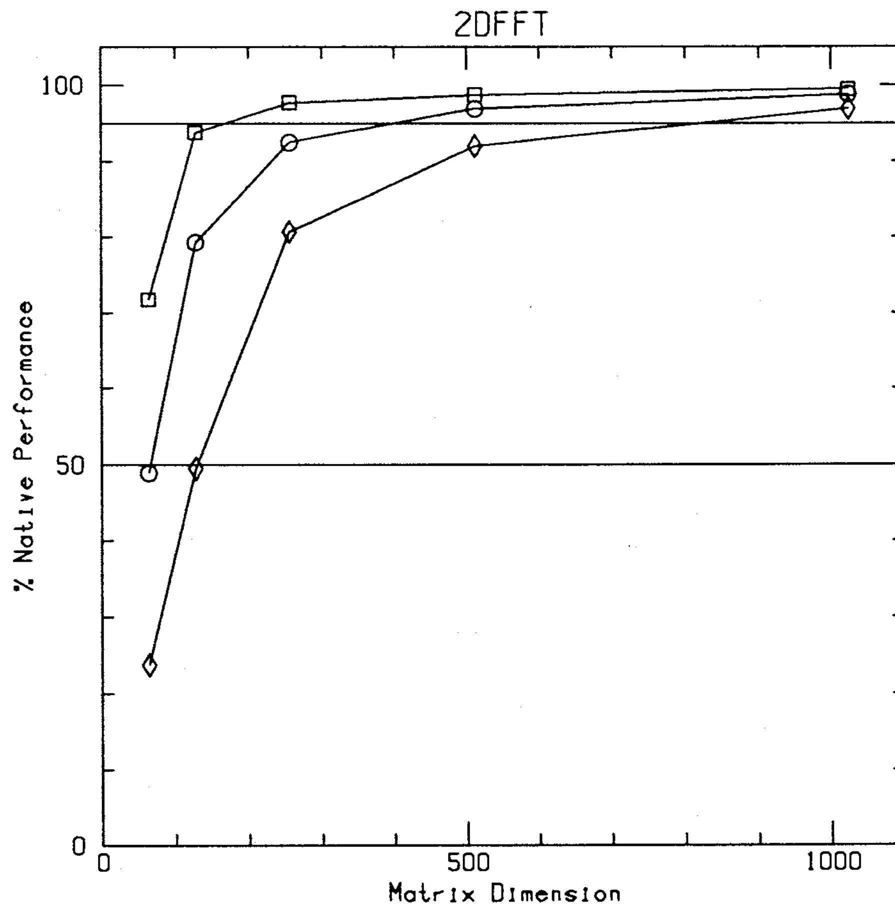

Figure 6.18: 2D-FFT, Ratio of Native and Linda Performance. □ = 16, ○ = 32, and ◇ = 64 processors. Both versions fully optimized. Net time, excluding initialization.

6.7 LU decomposition

LU decomposition separates a matrix A into two factors L and U such that $L \cdot U = A$, where L is *lower triangular* (has non-zero elements only on or below the diagonal) and U is *upper triangular* (has non-zero elements only on or above the diagonal). L and U are a useful representation of A because they allow very efficient solutions of linear systems. If we want to solve

$$A \cdot x = (L \cdot U) \cdot x = L \cdot (U \cdot x) = b$$

we can first solve

$$L \cdot \hat{x} = b$$

and then solve

$$U \cdot x = \hat{x}$$

The advantage of this method is that solving triangular sets of equations can be solved very quickly by simple substitution. Thus, if the same matrix A will be solved many times with different right hand sides b, LU decomposition is a very efficient method.

We used a Linda program based on the LINPACK benchmark program developed by Dongarra [Don87]. This routine solves systems of equations by first performing an LU factorization, then performing a forward and back solve. The factorization $\mathcal{O}(n^3)$ dominates the computation, since the backsolve is only $\mathcal{O}(n^2)$. Because of this, and because the solve is hard to parallelize, our code only performed the factorization step. The factorization is also interesting because it is not perfectly parallelizable.

The factorization proceeds as follows: For each step i, we add a multiple of row i to all rows $i+1 \ldots n$ such that each row's i^{th} element is zeroed. Obviously, if the ith element of row i (called the ***pivot***) is zero, we have a problem. In fact, it turns out that if the pivot is non-zero but very small compared to other values, the method is numerically unstable. For this reason, it is important to always use ***pivoting*** with this method. Pivoting rearranges the matrix at each step, to bring a relatively large absolute value to the pivot position. Full pivoting rearranges both columns and rows to bring the largest value in the matrix to the pivot position; partial pivoting, which we used, just rearranges columns to bring the largest value in the row to the pivot position. Partial pivoting has been shown to be almost as stable as full pivoting [GV73].

We now turn to our parallel algorithm. The Linda version of Dongarra's code was written by Nicholas Carriero [Car87], and is presented in Section B.4. The code originally ran on a shared-memory multiprocessor; no changes were made to adapt it to the iPSC/2.

First, the rows of the matrix are distributed evenly to the workers. Because the rows involved in the computation become fewer and fewer as it progresses, it is important to distribute the rows so that each worker holds some active rows as long as possible. Carriero interleaved the workers, such that worker k (of w workers) received rows k, $k+w$, $k+2w$, etc. At each step i the master `out`s the current multiplier row, the workers `rd` it, and perform eliminations on their rows. The $(i+1)^{th}$ row will form the next multiplier; as soon as the worker that holds that row has finished the elimination against it with the current multiplier, it `out`s the row, the master `in`s it, and begins preparing the next multiplier.

Partial pivoting is performed by swapping columns to bring the desired element to the pivot.

6.7.1 Granularity

The computation of the entire matrix requires

$$\sum_{i=1}^{N} i^2 \approx \frac{N^3}{3} \text{ FP ops}$$

The communication is dominated by the broadcast of the multiplier, which totals $4N^2P$ bytes. Thus, the granularity is:

$$\frac{\frac{N^3}{3}}{4N^2P} \approx \frac{N}{12P} \text{ FP ops/byte}$$

6.7.2 Performance

In Appendix C, we present the complete performance results for three different matrix sizes, 128x128, 256x256, and 400x400, with different levels of optimization. Figures 6.19 and 6.20 compare the program's performance fully optimized and unoptimized for the 256x256 and 400x400 cases, respectively. All speedups are relative to a sequential version that uses exactly the same computational kernels as the parallel version. The major contribution of the optimizations is the tuple broadcast optimization, which speeds the transmission of the multiplier row.

Figure 6.21 compares the Linda version of the LU code, running with all optimizations enabled, to a program written using the native message passing routines, written by us by replacing Linda operations with low-level message passing. The native version used broadcast to transmit the multiplier to all the workers. Both sets of timings exclude the time spent initializing the matrix. The optimizations bring the total number of messages that the Linda program needs very close to that for the native program. The multipliers are broadcast directly to the workers just as in the native program via the tuple broadcast optimization; the new pivot row is quickly remapped to the master node via rehashing. The main reason for the difference between the Linda program and the message passing version is the CPU overhead of Linda, both the handling of the requests themselves and the (already mentioned) interrupt mechanism. This overhead begins to show up because the computation is quite fine grained with 64 processors on a 400×400 problem (roughly .5 Flops/byte). Even so, the Linda program delivers fully 80% of the efficiency of the native program in this case.

We also compared our Linda program against a well-tuned hypercube version. John Gustafson [Gus92] of Iowa State University and nCUBE Corporation ported the NAS LU benchmark [BBLS91] to the Ncube hypercube. The NAS benchmark is written in Fortran and makes heavy use of the BLAS level 1 calls DSCAL, DAXPY and IDAMAX. We translated the nCUBE message calls to their Intel equivalents. Figure 6.23 compares the performance of the two versions. The Linda program performs within 90processors.[10]

[10] We might also note that porting Gustafson's nCUBE code to the Intel was a non-trivial exercise, even

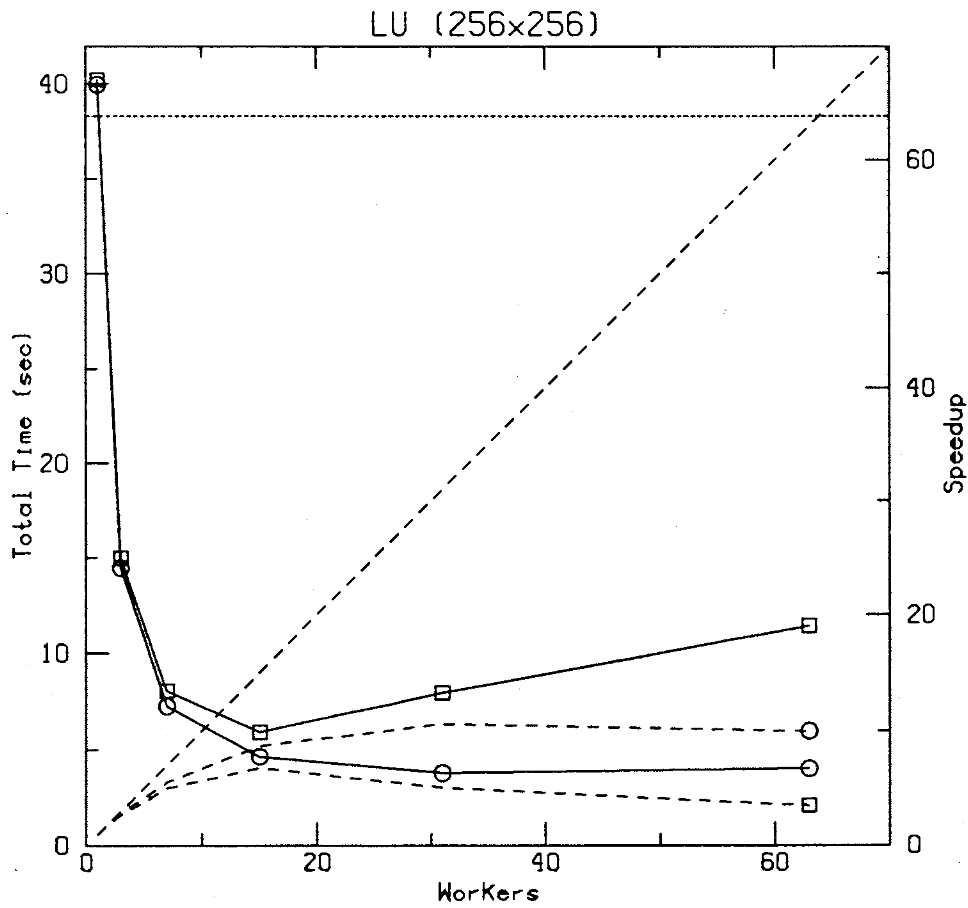

Figure 6.19: LU decomposition, dimension 256x256. All optimizations enabled ○ and none enabled □ . Speedup is relative to the sequential version.

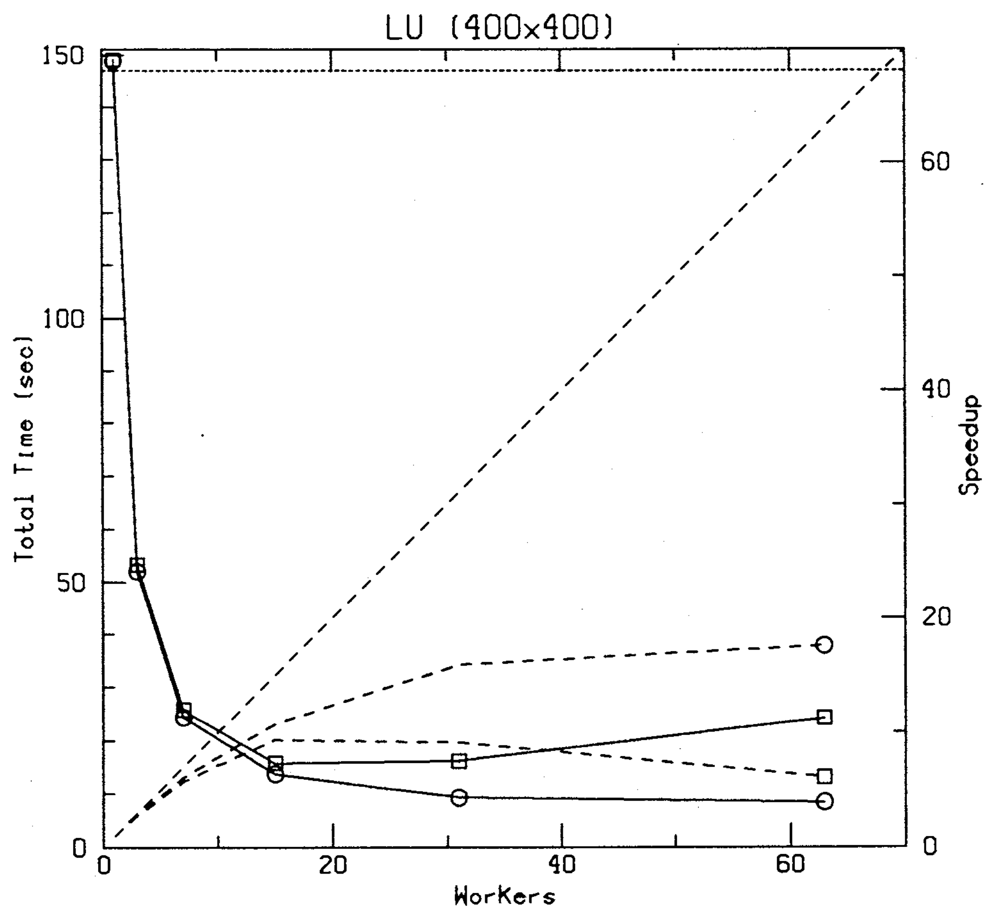

Figure 6.20: LU decomposition, dimension 400x400. All optimizations enabled ○ and none enabled □ . Speedup is relative to the sequential version.

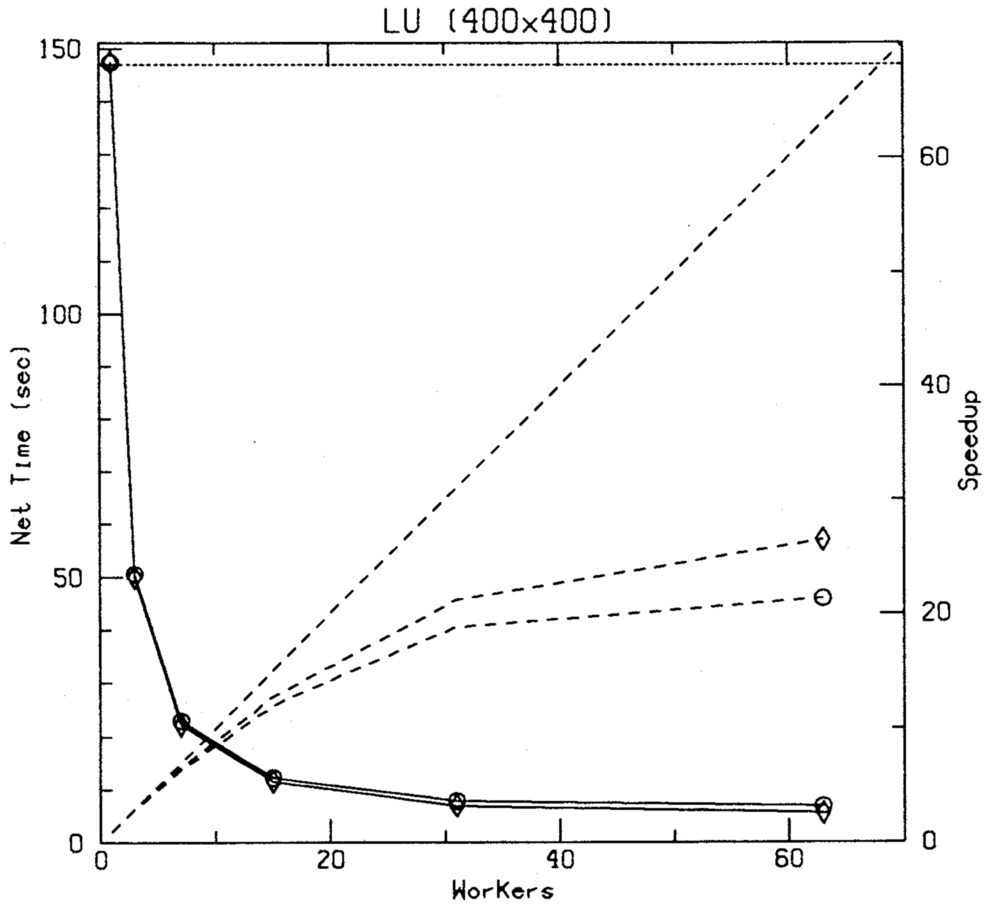

Figure 6.21: LU decomposition, dimension 400x400. Linda with all optimizations enabled ○ versus message-passing version ◇ . Speedup for Linda and our native version is relative to sequential version.

As a further point of comparison, Figure 6.22 compares the performance of the Linda program against a Crystal program[11] and handwritten message passing program written by JinKi Li [Li91]. For this comparison we count all processors involved in the computation, including the master in the Linda program.

Using our knowledge of the internals of the Linda kernel and the cost of different communications on the iPSC/2, we developed a model of the performance of the LU code. Figure 6.24 presents the model and the parameters we used. The model explains the performance degradation as the number of workers increases: at some point the master becomes a bottleneck. This happens much sooner without tuple broadcast. Figures 6.25, 6.26, and 6.27 compare predicted with observed times, both with and without tuple broadcast, for matrices of dimension 100, 256, and 400, respectively. The predicted and observed times agree fairly well considering that the execution times span two orders of magnitude; rather than adjust the model parameters to fit the data, we chose to use values determined by basic testing of individual components.

We then used the model to predict performance on much larger machines. Figure 6.28 predicts the performance on up to 256 nodes for a 1000x1000 matrix, while Figure 6.29 predicts performance for a 2024 node machine for a 10000x10000 matrix.[12] We see the speedup drop off (the "knee") at about $n/8$ for an $n \times n$ problem, a figure that is reasonably consistent with the speedup curves for actual times from 256x256 and 400x400 problems. It is reasonable that the maximum number of processors that can be effectively applied to a problem should increase as n; the computation increases as $\mathcal{O}(n^2)$, but the communication increases as $\mathcal{O}(n)$, so the increase in granularity is $\mathcal{O}(n)$.

The model shows how the DMM-Linda system could be expected to scale to a very large machine for at least one code. In this case the program scales well due to the tuple broadcast optimization until the point is reached where the work performed by the master (and thus not parallelized) begins to dominate.

---

though the central portion of the code (ignoring the BLAS routines) was only a few hundred lines long. Porting took about 10 hours of concerted effort for a programmer very familiar with the iPSC/2, and fairly familiar with the nCUBE. The message passing models presented by the two machines differ in subtle but important ways. The Linda code, on the other hand, can be run without change on an nCUBE by simply recompiling.

[11] for a brief description of Crystal, see Section 8.5

[12] We do not claim that the model predicts performance with a great degree of accuracy. Still, it should give us a fair idea of where the "knee" in the speedup curve exists, which is the most important datum in determining performance of a particular problem size on a particular machine size.

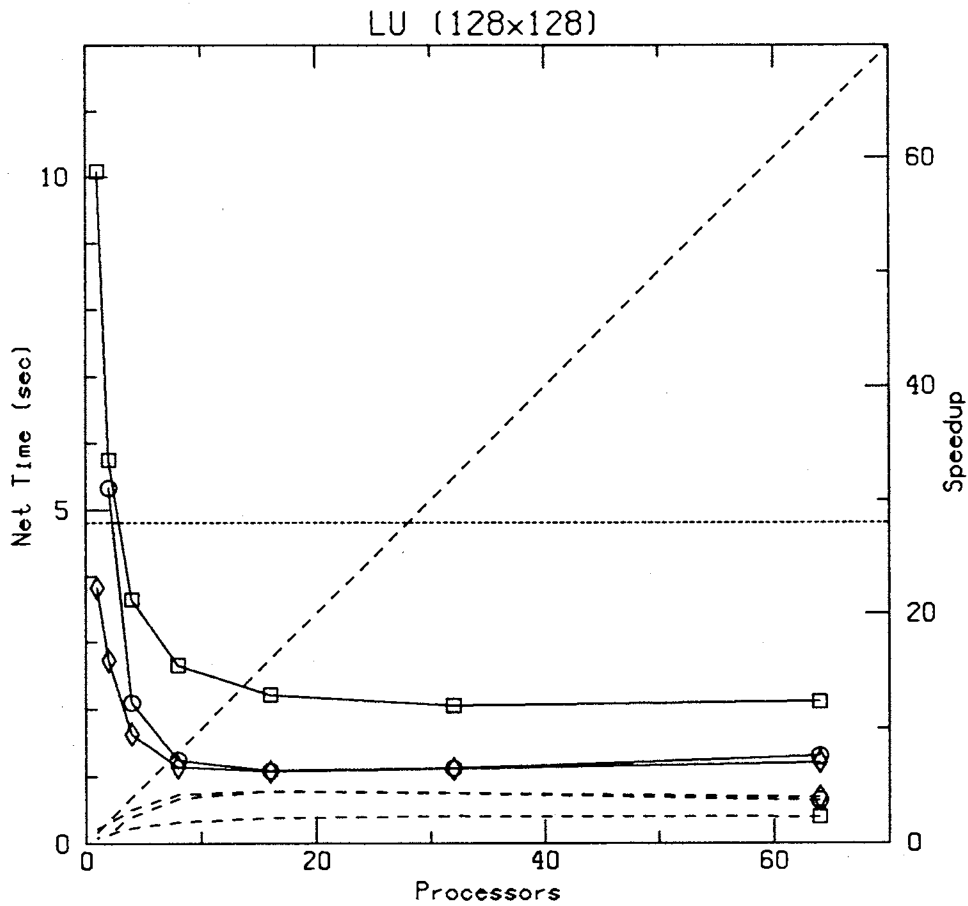

Figure 6.22: LU decomposition, dimension 128x128 Linda with all optimizations enabled ○ , Crystal □ , and a handwritten message-passing version written by Li ◇ . Speedup is relative to our sequential version.

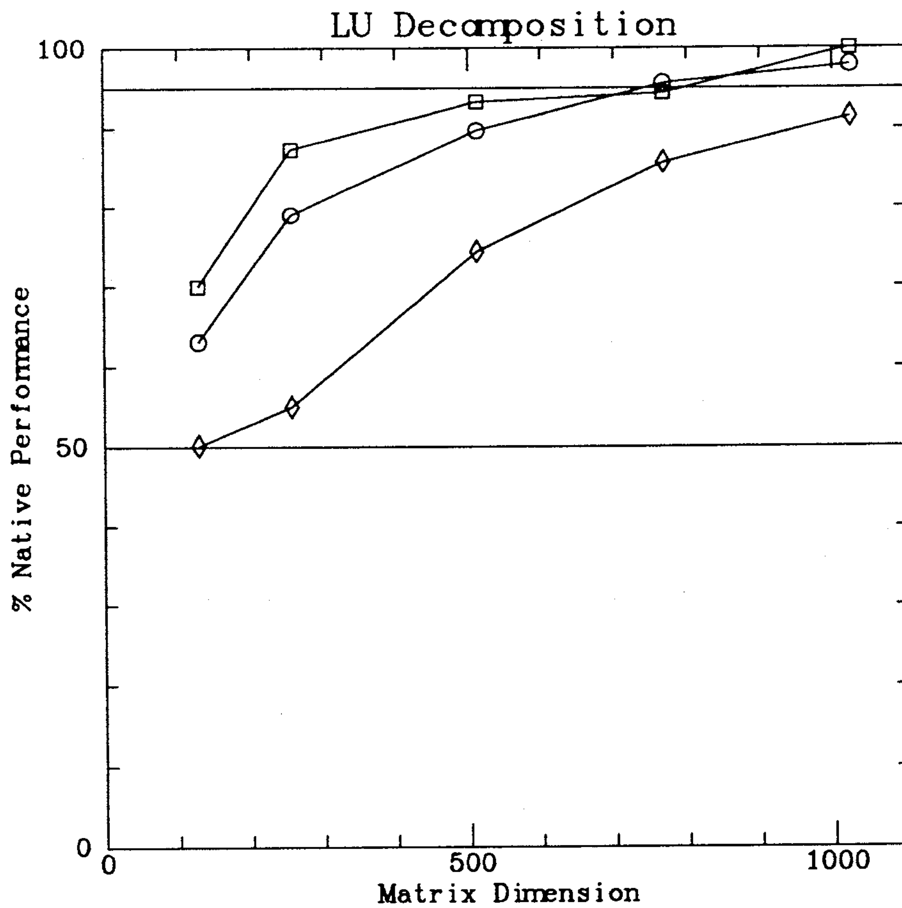

Figure 6.23: LU Decomposition, Ratio of Native and Linda Performance. □ = 16, ○ = 32, and ◇ = 64 processors. Both versions fully optimized. Net time, excluding initialization.

w workers
d dimension hypercube
$n \times n$ matrix
$AW_k = \min(w, n-k)$ (the number of active workers on iteration k)

For iteration k with $i = n - k$

Master's time m_k

out multiplier	$M_b + M_a i$	
	broadcast	$M_a = .186 + 4.27d$, $M_b = 2000 + 410d$
	no broadcast	$M_a = 3.6(w+1)/2))$, $M_b = 1000(w+1)/2)$
find max	$X_b + X_a i$	$X_a = 6.46, X_b = 68.2$
scale multiplier	$S_b + S_a i$	$S_a = 5.04, S_b = 7.27$
in pivot row	$P_b + P_a i$	$P_a = 4.4, P_b = 1000$

Workers' time w_k

rd multiplier	$L_b + L_a i$	$L_a = 1.76, L_b = 1000$
do daxpy's	$i(D_b + D_a i)$ / AW_k	$D_a = 6.66, D_b = 68.05$

$t_k = \max(m_k, w_k)$
$T_{tot} = \sum_{k=0}^{n-1} t_k$

Figure 6.24: LU decomposition model for iPSC/2 Linda. All times in μsec.

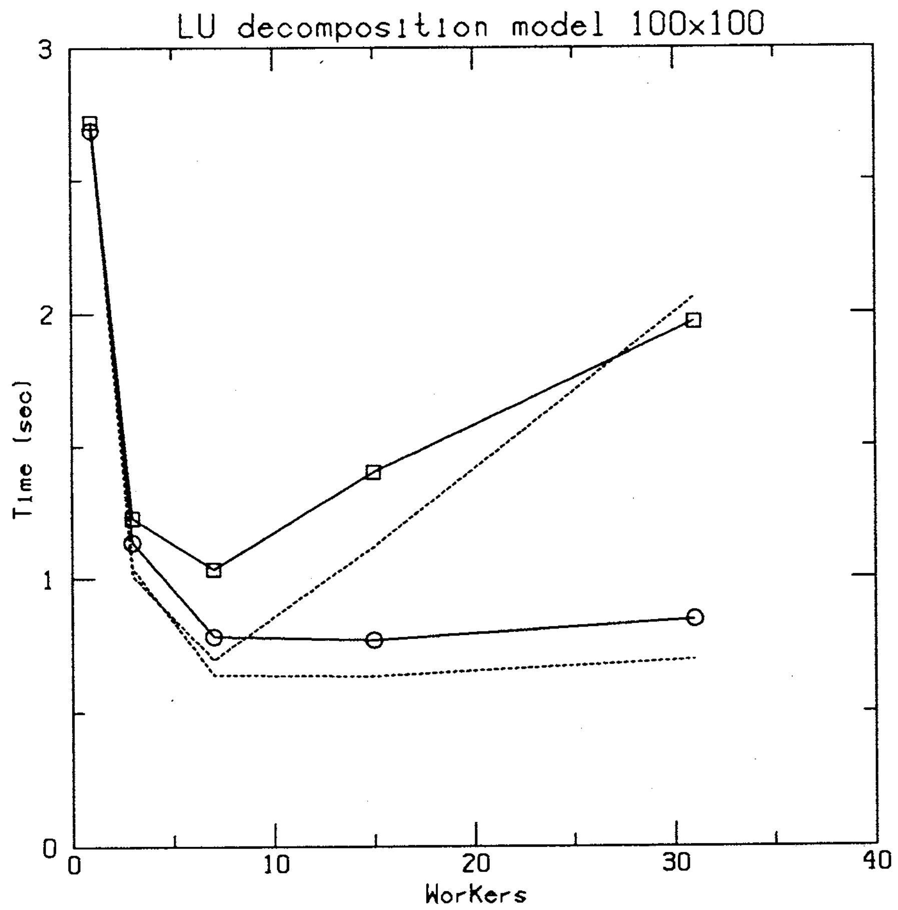

Figure 6.25: LU decomposition model, dimension 100x100. Tuple broadcast enabled ○ and disabled □ . Dashed lines indicate predicted times.

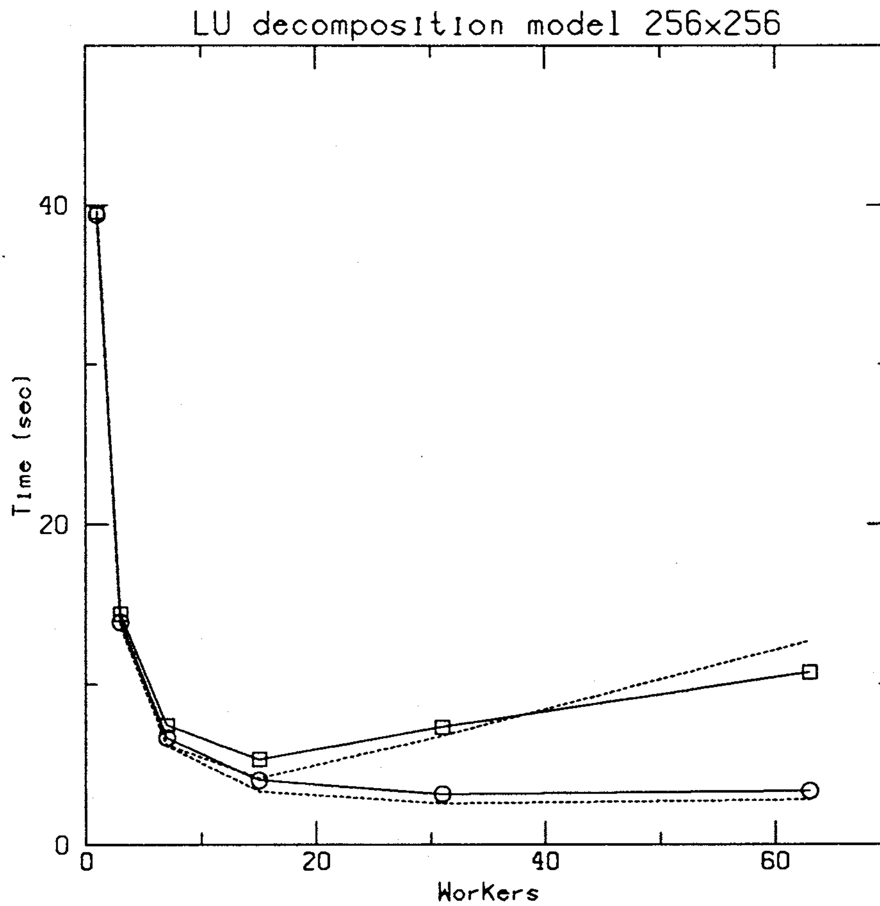

Figure 6.26: LU decomposition model, dimension 256x256. Tuple broadcast enabled ○ and disabled □ . Dashed lines indicate predicted times.

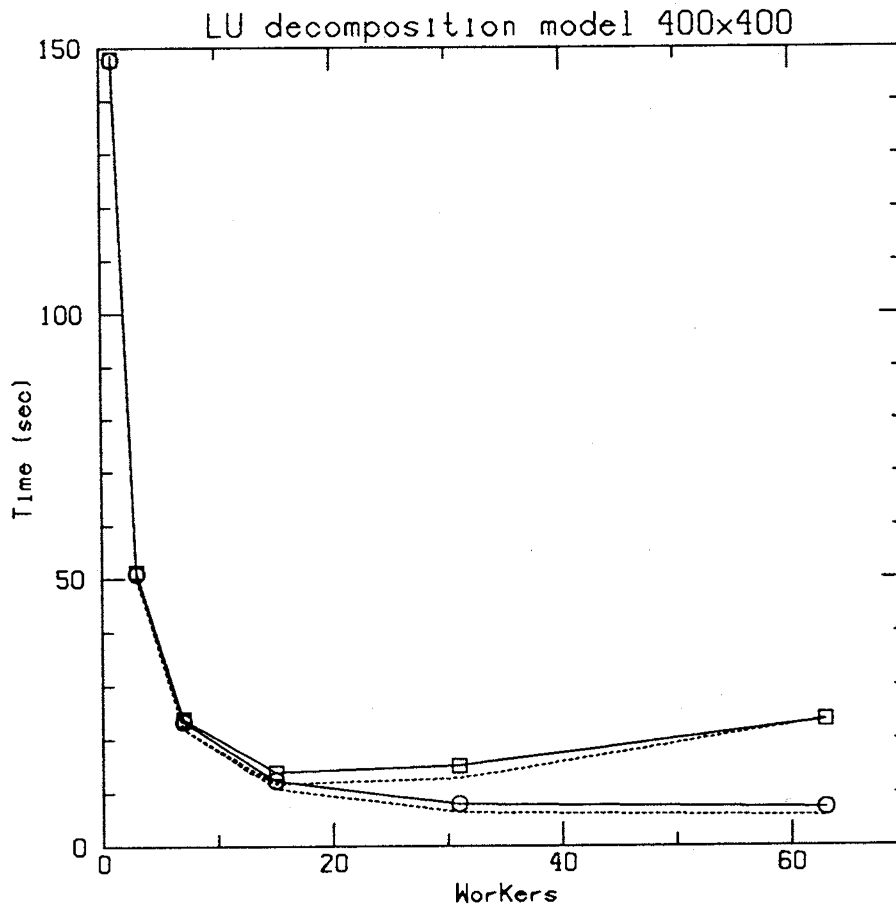

Figure 6.27: LU decomposition model, dimension 400x400. Tuple broadcast enabled ○ and disabled □ . Dashed lines indicate predicted times.

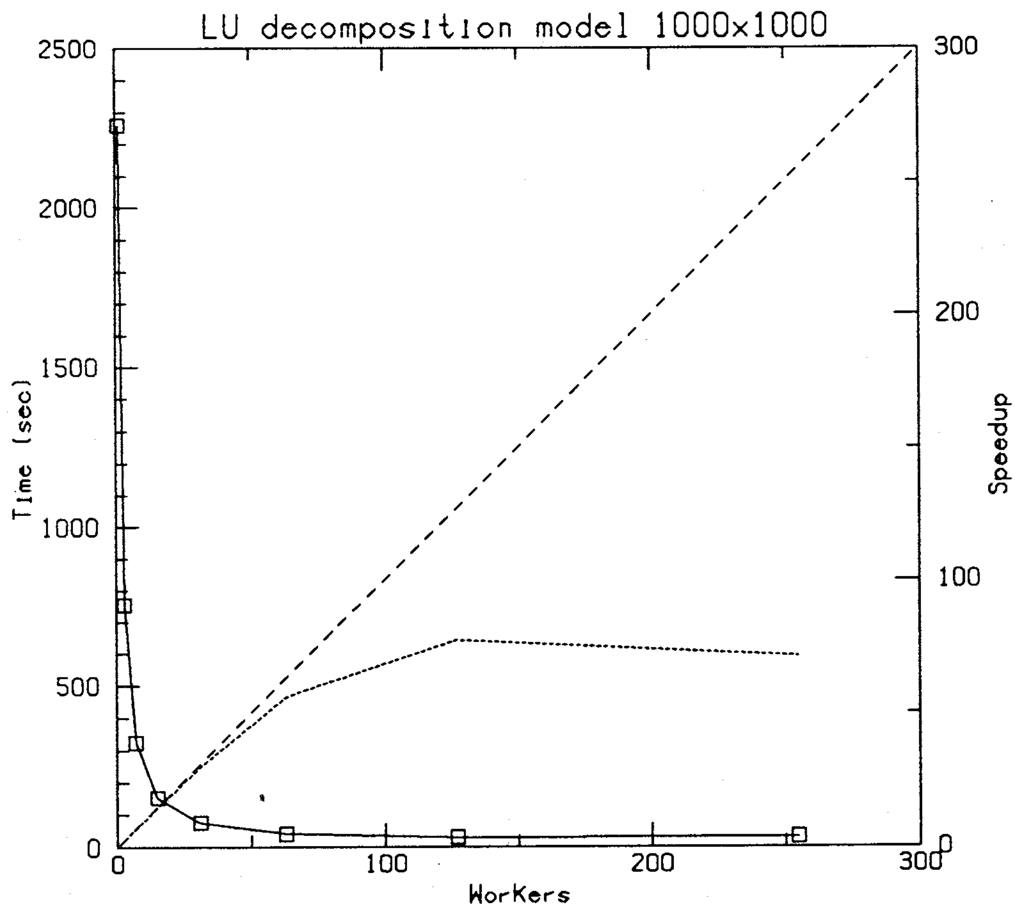

Figure 6.28: LU decomposition model, dimension 1000x1000. Tuple broadcast enabled. Predicted times only. Speedup is relative to the single worker time.

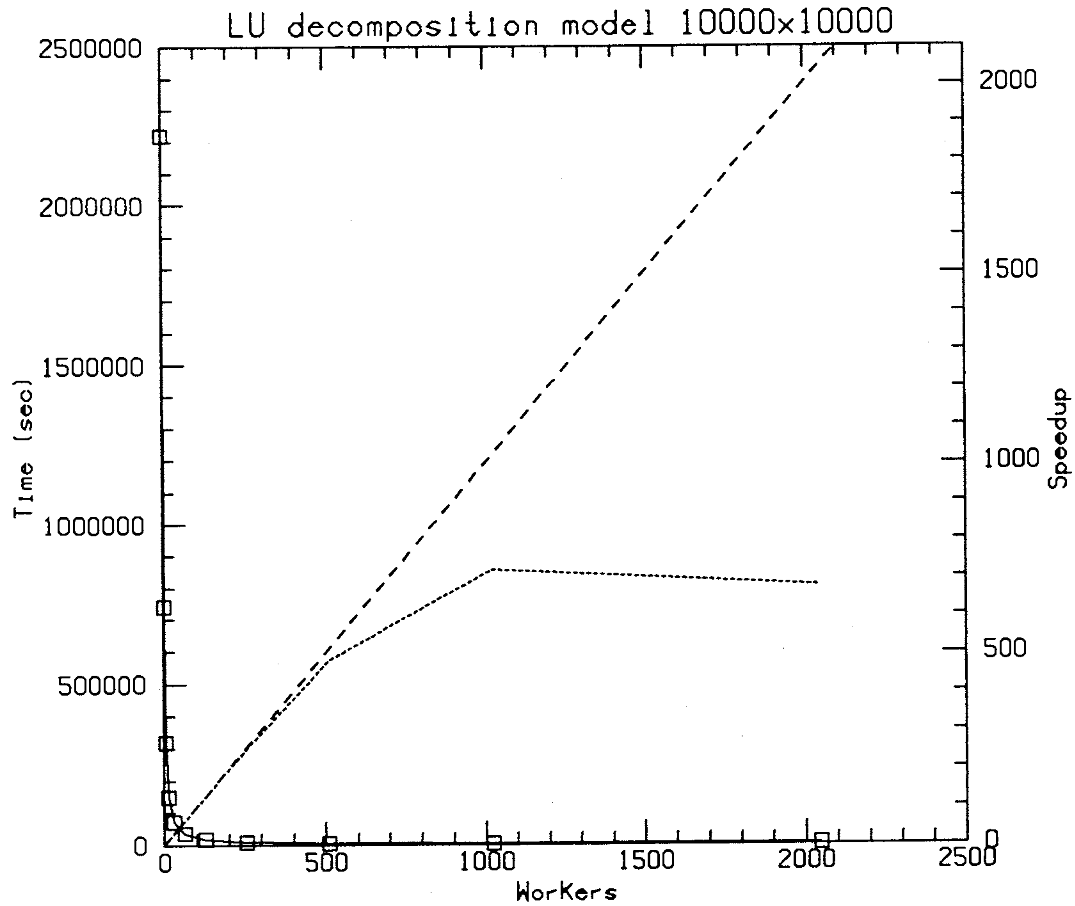

Figure 6.29: LU decomposition model, dimension 10000x10000. Tuple broadcast enabled. Predicted times only. Speedup is relative to the single worker time.

6.8 Erosion

This algorithm takes a mesh of altitudes, representing a terrain, and simulates rainfall and the resulting erosion, producing gullies, silt flats, and other natural terrain features. The sequential version was developed by Ken Musgrave as part of his work with fractal landscapes. The model used is too complex to explain here. For a more detailed explanation of the model, see [MKM89].

Basically, we start from an initial mesh representing the uneroded landscape. Each mesh point is represented by a 64 byte structure describing several values, including the amount of silt, talus, and water present, the bedrock level, and the current surface altitude. From here the simulation proceeds by computing a fluid flow, if any, from each point to each of six neighbors. For each downhill neighbor, a portion of the total fluid flow (water and silt) is apportioned, based on the gradients, the amount of water, and the carrying capacity of the water. The fluid is displaced to the neighbors, and the carrying capacity of the new water levels is recomputed, possibly causing silt to be eroded or deposited. Calculating each mesh point is fairly floating-point intensive, requiring over 120 floating point operations. A typical production run could erode a 1024×1024 mesh for hundreds or thousands of time steps, resulting in runs of hours or days on available uniprocessors.

We parallelized the algorithm by subdividing the mesh into rectangular subblocks, and assigned each block to a processor. For each time step, each processor:

1. Communicates its edge values to its (logical) neighbors.
2. Receives its neighbors' edge values.
3. Erodes its subblock.
4. Communicates contributions to its neighbors' edge values.
5. Receives neighbors' contributions to its edge values.

Figure 6.30 illustrates the communication pattern for a single block and timestep. In keeping with the spirit of portability, we made no attempt to map logical neighbors to nearest-neighbor positions on the hypercube. After the required number of time steps have been completed, each worker communicates its subblock to the master.

6.8.1 Granularity

Let $n \times n$ be the size of a sublock, i.e. $n = \frac{N}{\sqrt{P}}$ Each iteration requires each process to do two communications per neighbor; for interior blocks this is 12 communications totaling $256n + 128$ bytes. The computation involved in one timestep for an $n \times n$ block is $120n^2$. Thus, we have:

$$\frac{120n^2}{256n + 128} \approx 0.47n = \frac{0.47N}{\sqrt{P}} \text{ FP ops/byte}$$

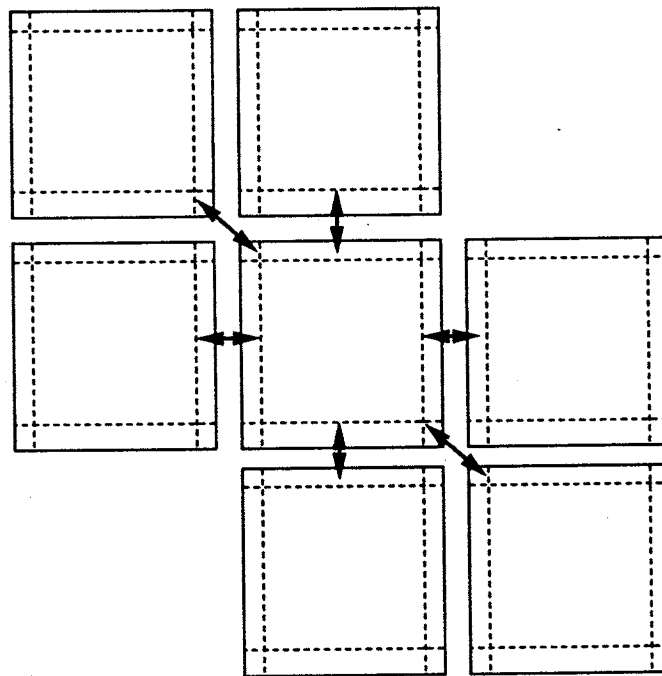

Figure 6.30: communication pattern for erosion simulation.

6.8.2 Performance

Figure 6.31 shows the performance of the program for a 200×200 mesh and 200 timesteps. The rehashing optimization showed the greatest performance improvement. This was due to the fixed communication pattern, which was the same for each iteration. This particular problem, running on 64 nodes, has a granularity of approximately 11.5 flops/byte

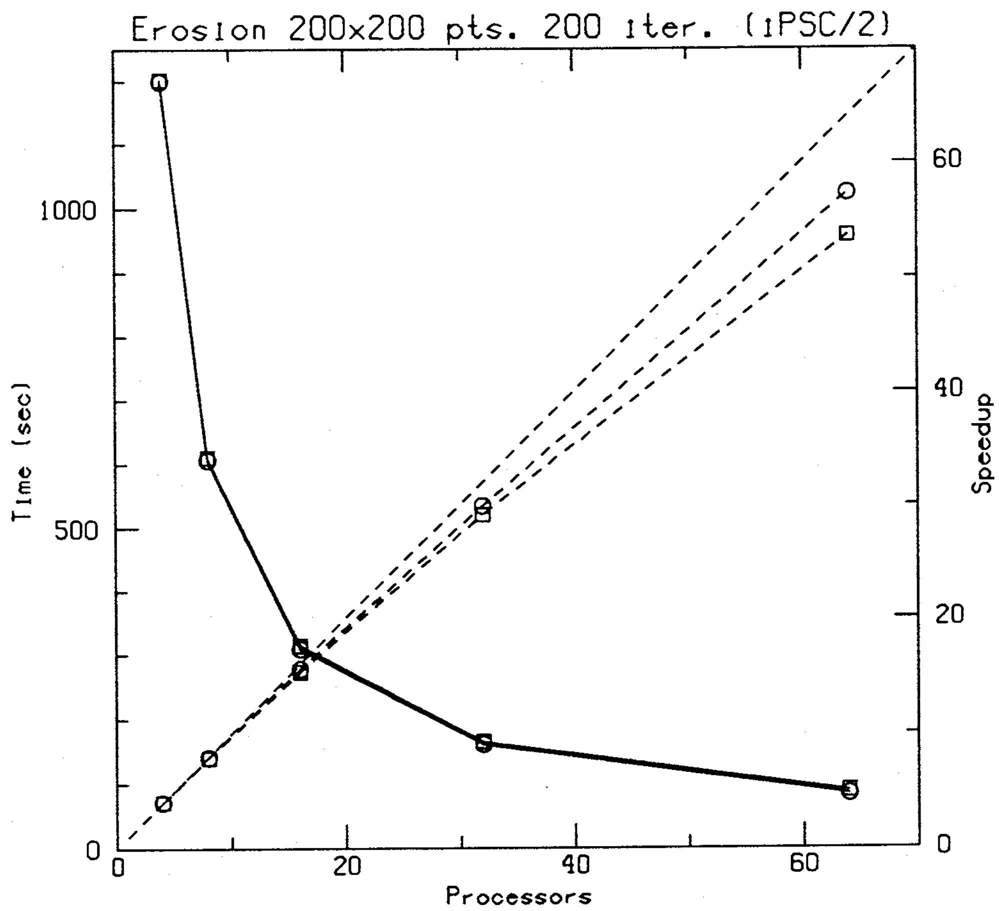

Figure 6.31: Erosion simulation on 200x200 grid, 200 iterations. ○ is optimized, □ not optimized. The problem would not run on less than 4 nodes due to memory limitations. Speedup is relative to the 4 processor case.

6.9 Intelligent Cardiovascular Monitor

The Intelligent Cardiovascular Monitor (ICM) is an expert system designed to monitor cardiac patients in an intensive care unit in real-time. The monitor is an application written on top of the Process Trellis Architecture [Fac90], which in turn is implemented in Linda. Both the ICM and the Process Trellis Architecture are far too complex to describe fully here; we will only attempt to provide a sense of the computation and communication structure.

The Process Trellis is a tool for building real-time expert systems. A programmer building an expert system defines a set of independently executing decision processes. Each process maintains a state, and has defined for it a set of inferior and a set of superior processes. When some process in its inferior set updates its state, the process attempts to update its own state. If its state changes, it sends the new state to the nodes in its superior set. The various inferior and superior relations form an acyclic hierarchy of the nodes. At the bottom of the hierarchy, the processes receive input from hardware monitors, while the top level processes generate higher level predictions typical of expert systems.

The Process Trellis was "Lindafied" by Michael Factor as part of his dissertation. It was initially run on shared memory implementations of Linda; he then ran the same code on our iPSC/2 implementation.

His parallelization strategy had multiple workers (one per node in the case of the iPSC/2) repeatedly sweep over the process trellis, looking for processes which have been activated, and executing them. In order to reduce communication of process states, he mapped the decision processes statically to workers, using several heuristics that attempted to minimize the inter-worker communication necessary[13]. Only inter-worker communication incurred Linda operations since decision processes mapped to the same worker shared the same address space.

We present performance results gathered by Factor for the Intelligent Cardiovascular Monitor. This Process Trellis application consists of about 100 decision processes. At the bottom of the hierarchy are processes that would normally accept input from hardware monitors; the top level processes form diagnoses and suggest treatments. No hardware inputs were actually used; the bottom level processes simulated real-time hardware input by reading data (recorded from actual intensive care patients) at the rate of one per second.

Figure 6.32 shows the performance of the program. Speedup is relative to the single worker Linda time, although this should be very close to a sequential time, since all of the communications are intra-node in the single worker case, so no Linda communication actually occurs (see the discussion of mapping, above). For technical reasons having to do with the ICM rather than Linda, the program could not be run on 64 nodes[14]. This problem is quite fine-grained for the iPSC/2. Most of the trellis processes execute for less than 10 milliseconds, during which they do perform several communications. Because

[13] Optimal partitioning is NP-complete[Fac90].

[14] The Process Trellis scheduler guarantees real-time performance. Rather than finding the fastest method of running a particular expert system, it attempts to find the least number of processors such that the trellis can be computed within a time limit. Given this vantage point, the process trellis scheduler failed to find a configuration that ran on anything between 32 and 64 nodes. This is probably due to the scheduler estimating that adding more nodes would not increase performance.

of the complexity of the code, it proved difficult to determine a closed-form estimate of the granularity. On 32 nodes, we observed a granularity of approximately 35 flops/byte, assuming that 1 second of wallclock runtime was equivalent to 0.5 million floating point operations.

Running optimized on 32 nodes, the program achieved a 12-fold speedup. The largest decision process is roughly 1/20th of the total work, so speedup greater than 20 is impossible. This program is an important example, in that it is a large and complex body of code written by someone unfamiliar with the internals of our implementation, for a machine capable of supporting much finer granularity than the iPSC/2. Even so, it shows good speedup on up to 16 processors, and continues to improve on 32.

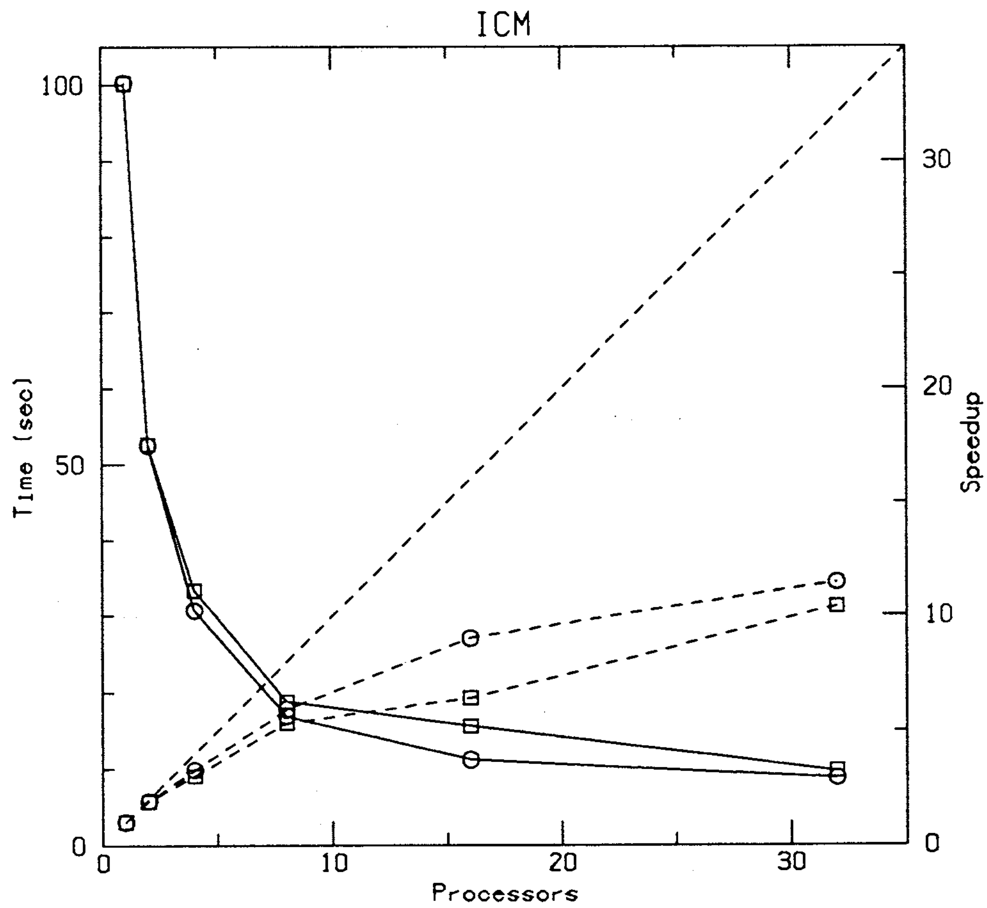

Figure 6.32: Intensive Care Monitor. ○ is optimized, □ not optimized.

6.10 Freewake

Freewake is a production computational fluid dynamics (CFD) code used by helicopter designers to simulate the wake produced by a helicopter rotor.[15] It is a discrete element simulation performed for many timesteps.

Each wake element is a point in the 3-space of blade, filament, and segment. The computation is a classical n-body problem; for each element, the influence of all other elements needs to be calculated. Within a particular timestep, all of these calculations can be done in parallel. The sequential code contains four nested loops, iterating over timesteps, blades, filaments, and segments, respectively. Each filament represents a wake stream trailing from a particular section of a blade; each filament is then broken up into segments. At the center of the nested loops, one element is updated. The complexity is $\mathcal{O}(n^2)$ in the number of wake elements for each time step: $\mathcal{O}(b^2 f^2 s^2 t)$.

The parallel code effectively performs the blade and filament loop iterations in parallel. A number of worker processes are created, normally one less than the number of processors available, since the master occupies one processor. Before each timestep, the master communicates the current values of all of the elements to each worker process and creates a number of tasks equal to the number of blades times filaments. These tasks are performed by the workers in some arbitrary order. Instead of simply waiting for the results to return, the master temporarily becomes a worker and computes tasks as well.

Each task corresponds to performing all iterations of the innermost (segment) loop of the original computation.[16] When finished with a task, a worker process sends the results back to the master process and looks for another task. When all tasks for the current timestep are completed, the entire process is repeated for the next timestep.

6.10.1 Granularity

Each time step requires $111N^2$ floating point operations, and the communication of $4(4P+3)N$ bytes. This gives a granularity of:

$$\frac{111N^2}{(16P+12)N} \approx \frac{7N}{P} \text{ FP ops/byte}$$

6.10.2 Performance

Figure 6.33 shows the performance of the code running with the rehashing and memory caching optimizations on a 2048 element problem. The problem size refers to the number of wake elements in the 3-space. All results presented were run for 5 time steps. Figure 6.34 shows the performance of an 8192 element problem on the 128 node Intel i860 hypercube at Oak Ridge National Laboratory.

[15]This code came from United Technologies Research Corp., and was "Lindafied" by Harry Dolan, a UTRC employee, initially for networks of workstations.

[16]This is the way we received the code. The segment loop could also be parallelized quite easily, which would increase the parallelism available in return for a small increase in overhead.

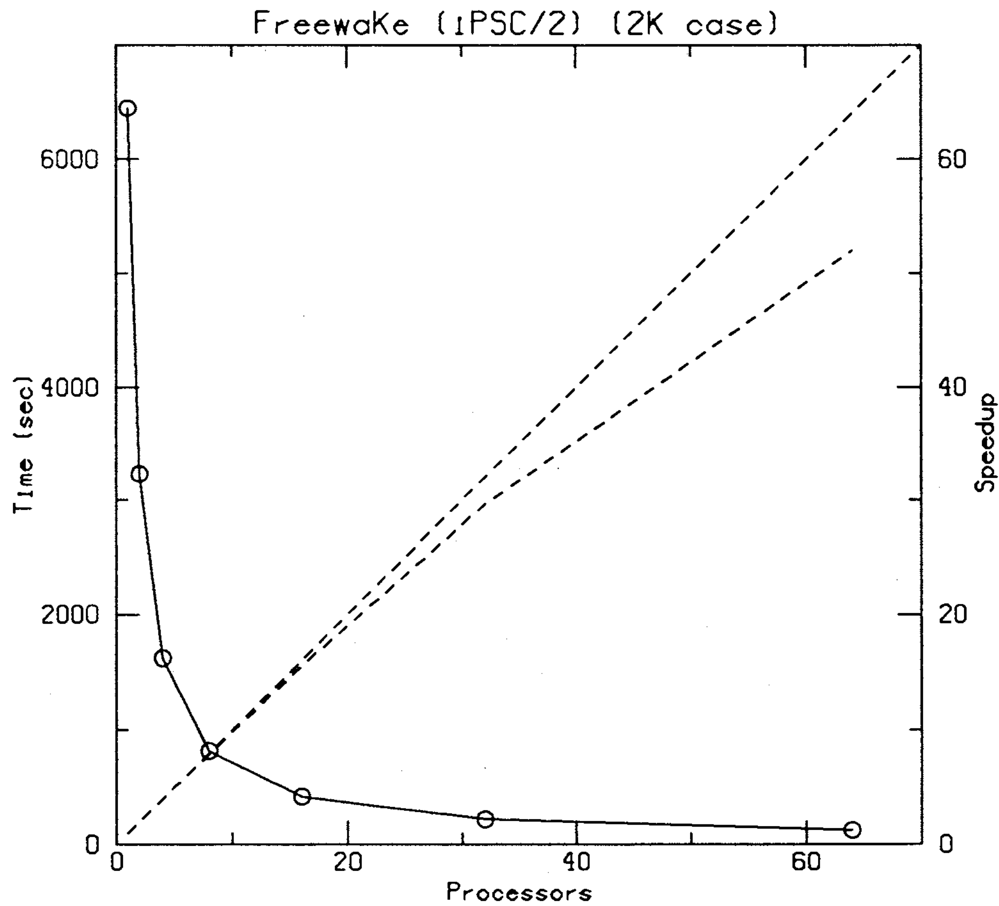

Figure 6.33: Freewake on iPSC/2. 2048 wake elements. Speedup is relative to the single processor Linda program.

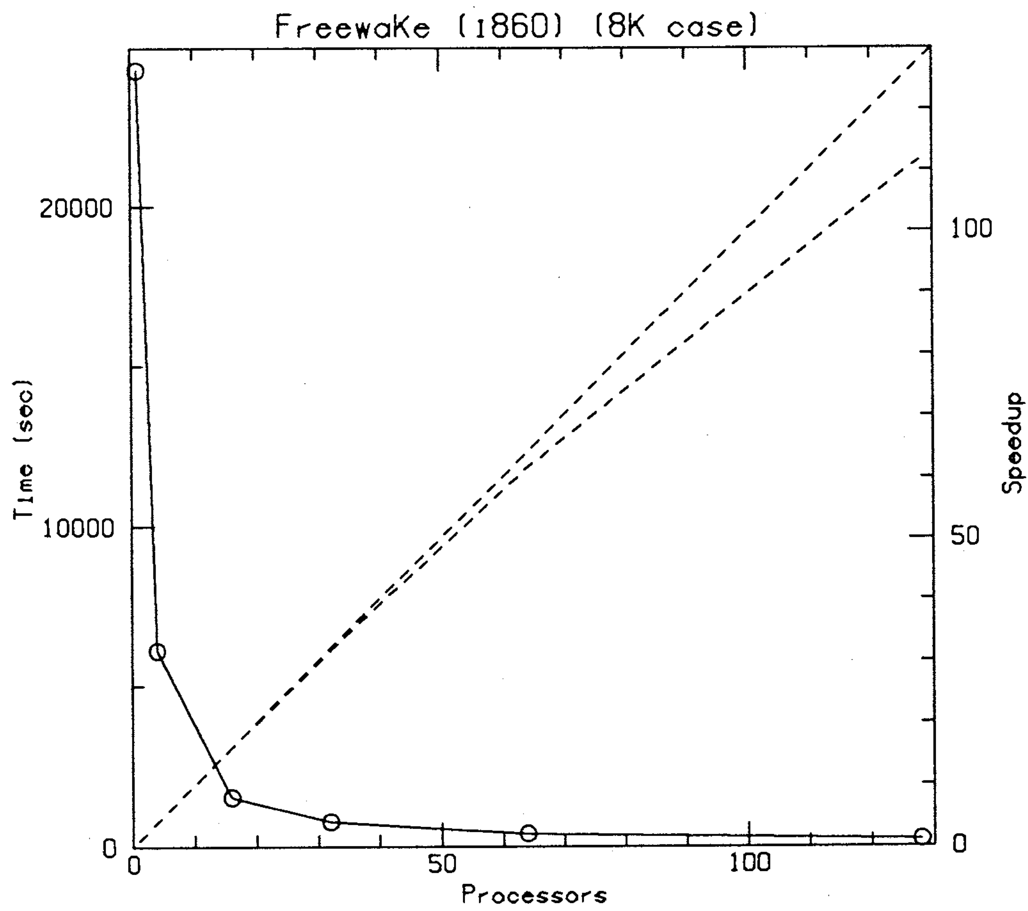

Figure 6.34: Freewake on Intel i860 hypercube. 8192 wake elements. Speedup is relative to the single processor Linda program.

6.11 Overview of application granularity and efficiency

In this section, we conclude the discussion of the applications with a rough measure of each program's granularity and how it relates to the efficiency achieved. By granularity, we mean the mean number of floating point operations performed for each byte of data communicated[17].

Table 6.2 presents a rough measure of granularity versus efficiency for seven of the applications. The formulas only take into consideration the total number of bytes communicated; communication startups are not considered. In some cases, particularly 2DFFT and DNA, this underestimates the communication cost by as much as 10-fold. When calculating the actual granularity of a particular problem, we estimated the additional effect of startups on granularity by counting the total number of ins and rds performed. Each was considered equivalent to an additional 1000 bytes transferred. Figure 6.35 plots the granularity against efficiency. We also plot the function $x/(x+1)$, showing the expected efficiency assuming that communicating one byte costs the same as one floating point operation.

Program	Gran.	Prob.	Processors 4		16		64	
			gran.	efficiency unopt/opt	gran.	efficiency unopt/opt	gran.	efficiency unopt/opt
DNA	$0.3x$	40x40	4.0	.81/.77	4.0	.76/.76	4.0	.70/.71
Block MM	$N/(4\sqrt{P})$	120x120	15	.94/.94	7.5	.75/.80	3.75	.39/.50
		240x240	30	1.07/1.08	15	.99/1.01	7.5	.77/.85
Clump MM	$a/2$	240x240	.439	.44/.75	.439	.23/.73	.439	.18/.63
		400x400	1.70	.81/.94	1.70	.72/.90	1.70	.52/.72
2DFFT	$5\log_2 N/8$	256x256	4.97	.95/.97	4.49	.84/.87	1.71	.35/.43
		1024x1024	6.24		6.08	.93/.99	4.22	.87/.91
LU	$N/12P$	100x100	2.08	.75/.73	0.52	.14/.24	0.13	
		400x400	8.33	.92/.94	2.08	.62/.71	0.52	.10/.28
Erosion	$0.47N/\sqrt{p})$	200x200	55	1.00/1.00	21.4	.95/.97	7.5	.84/.90
Freewake	$7N/P$	2048	3584	1.00/1.00	896	/.97	224	/.81

Table 6.2: Application Granularity and Efficiency. N is the problem size, P the number of processors. For DNA, x is the vertical block size. For clumped matrix multiply, a is the clumping factor for matrix A.

The particular problem size and parameters used for each entry in Table 6.2 follow:

DNA 468 base test sequence vs. DB ($\approx$ 730 sequences, totaling $\approx$ 163000 bases), 40x40 blocking. The actual granularity is 1.8 *integer* ops/byte, which was converted to FP ops to allow comparison with the others.

Block MM 120x120 and 240x240 matrices (single precision). 10 timesteps.

[17]The DNA code did not use floating point arithmetic; we assumed that 5 integer operations were equivalent to one floating point operation.

Clumped MM 240x240 matrix (single precision), A clumped by 1, B clumped by 8; 400x400 matrix (single precision), A clumped by 4, B clumped by 8.

2D-FFT 256x256 and 1024x1024 matrices (complex single precision).

LU 100x100 and 400x400 matrices (double precision).

Erosion 200x200 points, for 200 timesteps.

Freewake 2k problem (1 blade, 64 filaments, 32 segments). (single precision) 5 timesteps.

The granularities and efficiencies listed in the table cannot be directly compared to the granularity results from the BLP in Section 4.4.4, since the BLP is based on access to only one Tuple Space server. The BLP can be seen as a worst case; the applications perform considerably better to the extent that the compiler and kernel do their jobs in spreading out the load among the nodes or avoid communication altogether.

Using Figure 6.35, it is possible to gain a rough idea of the efficiency one can expect from a program running under DMM-Linda on the iPSC/2. Broadly speaking, applications coarser than 10 floating point operations per byte transferred should perform very well, with performance less predictable but degrading in the range 1-10, and poor below 1.

In comparison, what would the range of acceptable granularity be for native programs? Consider a program that sends 100 byte messages between compute phases. Sending 100 bytes takes roughly 415 μsec (see Figure A.1); if we consider, as we did for Table 6.2, that a communication startup is equivalent to 1000 additional bytes, we have 415/1100 or 0.38 μsec/byte. In order for the program to be 80% efficient, it should compute for x μsec between communications, where:

$$\frac{x}{x + 0.38} = 0.80$$

$$\text{so } x = 1.52\ \mu\text{sec}$$

1.52 μsec is roughly equivalent to 3 FP operations, as compared to 10 for Linda. Thus, the acceptable granularity range for message passing and Linda is close. Programs with a granularity greater than 10 FP ops/byte should function well in both cases; those with granularities finer than 3 may be too fine for either. The range 3-10 FP operations/byte is where we would most expect to see message passing outperform Linda; we consider this an acceptably narrow range.

6.12 Summary

In this chapter, we presented a number of Linda applications and discussed their optimized and unoptimized performance on the iPSC/2. The granularities ranged enormously, from hundreds to fractions of operations/byte communicated. Almost all of the programs showed acceptable performance (efficiency greater than 80%) when running with a granularity greater than 5 floating point operations per byte communicated; some performed well at even lower granularities.

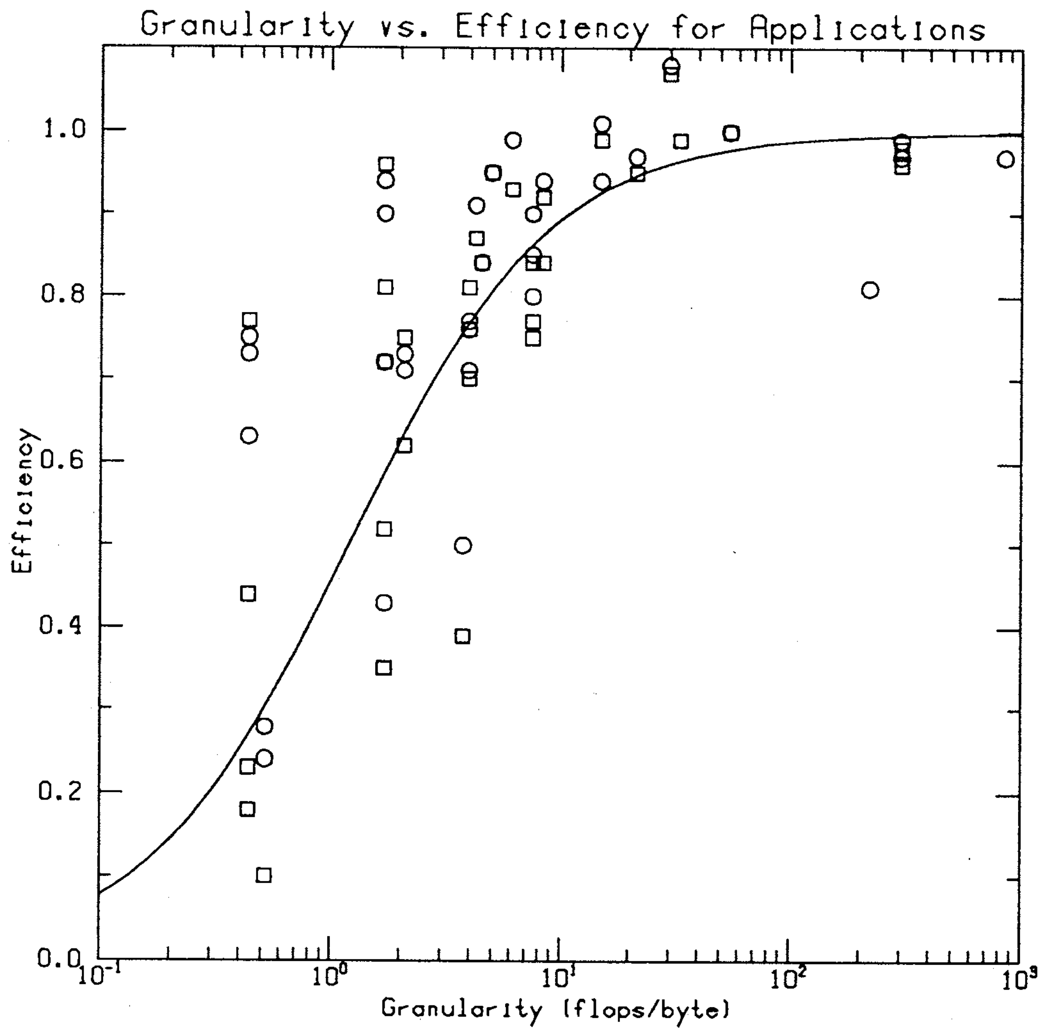

Figure 6.35: Efficiency vs. granularity for applications. ○ is optimized, □ unoptimized. Also shown is the function x/(x+1).

In two cases (LU and 2DFFT), we presented a detailed comparison of the performance achieved by the Linda version versus a hypercube code that was carefully written by an independent expert. These programs have a fairly small granularity; still, the Linda version came within 90% on large numbers of processors and modest sized problems.

Chapter 7

Massive Parallelism

7.1 Introduction

In the previous chapters we have explored the performance of Linda on an Intel iPSC/2 with 64 nodes. This was a reasonably large commercial machine, but considerably smaller than the new generation of massively parallel machines. Intel's new generation of machines based on the Touchstone should extend to 2048 processors [Lil90]. The nCUBE2 is expandable to 8196 nodes [nCU90]. Thinking Machine's CM5 [Ste91] is claimed to scale easily to 16000 32-bit processors, which would represent a teraflop machine. See [FF88] for a discussion of the issues involved in building a massively parallel multiprocessor.

How well would the Linda system described in this dissertation function on machines with many thousands of processors? There are several areas of concern, including communication latency, memory utilization, cpu contention, and link contention.

7.2 Increasing communication latency

The communication latency experienced by a message traveling the full diameter of the machine will no doubt increase somewhat. However, machine designs that are held to be scalable must, by definition, address this issue. Such a machine will have a diameter that increases by a lower order than the number of nodes (for instance, $\sqrt{n}$ for meshes, $\log_2 n$ for hypercubes.) In addition, modern routing technologies, such as "worm hole" routing [Nug88], are greatly reducing the per-hop latency relative to the startup cost for communication. Thus, latency will become less of a concern, relatively speaking. In so far as it does remain a concern, several of the optimizations address latency reduction: rehashing, tuple broadcast, and data caching. The effect of these optimizations on latency has been discussed in Chapter 5.

7.3 Memory utilization

It is becoming clearer that manufacturers of massively parallel DMM's are increasing the per-node memory along with the number of nodes. Both the newest generation Intel and

Ncube machines provide more memory than on previous models. Thinking Machine's new CM5 represents a radical departure from the CM2, with each node containing 32 MBytes of memory [Ste91]. Still, no machine ever has enough memory, and as the machine grows the implementation has to become increasingly wary of overloading particular nodes with more than a fair share of Tuple Space. This criteria, that memory usage needs to be held reasonably even across the nodes, has some interesting implications for several of the optimizations. These optimizations, originally designed to reduce latency, can instead be used to improve memory usage.

Rehashing can offload some of an overloaded node's TS to a less-loaded node. If a node begins to run low on memory, it can choose a set or sets (or particular key in the case of a hash set) and rehash those tuples to anther node. At present this must be done blindly, since no node has any knowledge of the state of other nodes. It would be useful to periodically piggyback this information on other requests, so that nodes could make more informed guesses. This piggybacking would be very inexpensive, since it would only require adding a few extra bytes to other messages.

Tuple broadcast will need to be enhanced to allow selective caching of copies of tuples, as well as efficient flushing of copies if memory runs short. Only the rendezvous node is required to retain a copy, all other nodes can decide whether or not to store copies based on their individual memory situation, allowing each node to trade off memory and communication costs individually.

The **random queue** optimization directly benefits balanced memory usage by spreading large queue sets among several nodes.

In its present configuration, **local data caching** can either ameliorate or aggravate the balanced usage of memory. For instance, if several nodes **out** tuples that are hashed to one node, this optimization is helpful, by storing the large elements on the **out**ing nodes rather than on the one rendezvous node. Conversely, if one node **out**s tuples that are hashed to several nodes (by whatever method), then local data caching interferes with good memory usage, since the elements will all be stored on the single **out**ing node.

The optimization needs to be improved by having it take the above facts into account. As we shall argue later in this chapter, on large machines, avoiding contention is generally more important than reducing latency. Accordingly, we would like to tune the optimization as follows, on large machines:

1. If a set will be stored on few nodes or just one node, use the optimization, but keep track of the total amount of local memory used to store such elements. When some threshold is exceeded, begin storing the elements on other, randomly chosen nodes rather than on the node of origin. This will increase the latency by one transmission of the element, but will reduce both memory and CPU contention and even out memory utilization.

2. If the set will be spread out evenly among the nodes of the machine, do not use the optimization; just send the data along with the tuple header to the rendezvous node.

Thus, hash sets would never use the optimization. Queue and hybrid sets usually would, storing the element either locally or on some random node, unless some alternate paradigm

was in use (such as the random queue method) that was judged to reduce contention (i.e. spread the tuples out) sufficiently to allow the elements to be sent along with the tuples. This is a generalization of the present optimization, recognizing that large elements may be stored separately from the tuple itself for reasons of memory utilization as well as the original motivation, latency reduction.

7.4 Node CPU contention

As the machine grows, it is possible that some nodes will become hot spots of tuple traffic, handling more than their fair share of transactions and eventually becoming swamped. This problem is closely related to memory utilization, since nodes that store many tuples will also tend to handle more TS requests.

To reduce CPU contention, much the same techniques can be used as were described for reducing memory usage imbalance. If a node notices that it is becoming overwhelmed with TS requests, it can rehash one or more sets to a less busy node, or it can force some queue or hybrid sets to an alternate representation, such as the random queue method. New large data items to be stored on random nodes under the local data caching optimization can be directed to other nodes.

7.5 Link contention

Thus far in this dissertation we have largely ignored link contention. Link contention can arise between two messages either because they are bound for the same destination, or because their paths happen to coincide. An important factor in link contention is path length; longer paths will tend to raise contention for links. We have already discussed the problem of CPU contention that arises when too many messages are bound for the same node; the same solutions will help reduce link contention. Reducing the number of messages that just happen to cross is more difficult, although arguably less important.

In its present form at least, Linda does not attempt to map processes to the particular machine topology. That the programmer is not required to think about topology is generally considered to be an important strength linguistically, since it makes programs much easier to write and more portable. However, it also means that the average message travels half the diameter of the machine. This compares to 1 hop for a carefully handcrafted nearest-neighbor program. For small- to medium-sized hypercubes, the difference between 1 and $\log n/2$ is insignificant. However, for much larger machines, particularly those with polynomial rather than logarithmic diameters (e.g. tori, with $\sqrt{n}$ diameters), the added path length may become important.

The problem of reducing message path lengths is essentially a problem of mapping `eval`ed processes to nodes in such a way as to place processes that communicate as close as possible together. Once this is done, many communications could be automatically reduced to nearest-neighbor communications via the rehashing optimization. The problem of intelligent mapping of evals to nodes is beyond the scope of this dissertation. Other researchers have developed techniques for analyzing programs to determine data dependencies [PW86]; these techniques should allow a compiler to make reasonable guesses

about which tuples a particular process was likely to produce and consume, and place the processes accordingly. Fortunately, no incorrectness will result from wrong guesses; only reduced efficiency.

Another way to reduce path length involves the randomized queue method. The choice of node to search for a tuple could be biased in terms of distance, choosing nearer nodes first.

7.6 Contention vs. Latency

It is our conviction that as DMM's grow, latency will become secondary to node and memory contention. This is because latency will grow at a slower order ($\sqrt{n}$ or $\log_2 n$), whereas contention can potentially experience $\mathcal{O}(n)$ increase, for instance if a node is holding data that many nodes need simultaneously. In fact, it will be possible and probably advisable to trade constant order increases in latency for decreases in contention, especially when contention problems are actually observed[1]. A good example of this is the proposed change to the local data caching optimization. Originally designed to reduce latency by allowing long elements to be sent only once rather than twice, we envision it used to store these elements on a random node, potentially decreasing the contention while actually increasing latency. For an `out` followed by `in`, this new use will require three short and two long messages, vs. three short and one long for the original optimization, and two long unoptimized.

In a similar vein, the randomized method always increases latency, but (potentially) brings a much greater benefit in terms of reduced contention.

7.7 Alternate hybrid and mess set representation

In order to address the problems of CPU, link, and memory contention as the machine grows, we must be able to split up the responsibility for highly active sets. The standard distributed hashing paradigm works well for hash sets, and the randomized queue method, `inout` collapse, and tuple broadcast partially address the problem for queue sets.[2] Still unaddressed are hybrid and mess sets, which are mapped to single nodes, which could become bottlenecks. First, we will show how hybrid and mess classifications can properly be considered two halves of a more general classification, termed *partial-key*. We will then present an alternate implementation strategy for the partial-key class that spreads the set across many nodes.

Recall the definition of a hybrid set: such a set has no non-constant element that is always actual (i.e. no total key, and thus not a hash set). Furthermore, at least one element

[1] Contention can be measured in several ways. The length of the incoming message queue gives a good measure of overall contention. Once contention is noticed, the particular parts of Tuple Space responsible can be located using an access count that is periodically cleared. This technique is loosely analogous to strategies used to determine page usage for virtual memory (they choose the least rather than the most used members).

[2] Left out are sets with small numbers of tuples that are primarily ined. Such sets are *logical* bottlenecks and are difficult to distribute effectively.

is always actual for **outs** and sometimes actual for **ins** (thus requiring matching, so not a queue set.) This element is called a partial key, since all tuples have it, but only some templates do.

Sets are classified as mess if they fail to meet the conditions for queue, hash, or hybrid. If one carefully analyses the remaining possibilities, the classification can be given a positive definition as well: mess sets are those with the following characteristics:

1. No non-constant element is always actual across all **ins** and **outs** (i.e. the set is not hash).
2. No non-constant element is always actual across all **outs** (i.e. the set is not hybrid).
3. At least one element is sometimes actual on **out** and sometimes actual on **in** (i.e. not queue).

Thus we see that the distinction between hybrid and mess is solely that hybrid requires that the partial field always be actual in tuples, whereas mess allows it to be formal as well. We can define the new class partial-key as the combination of hybrid and hash. The normal implementation of this new class is almost identical to that of hybrid. Recall that hybrid is implemented by a private hash table with n lists for tuples and templates with keys, plus one extra list for all unkeyed templates. We can implement partial-key sets by adding just one more list for unkeyed tuples. The only change to the processing done is to have keyed templates check this list in addition to the list of tuples in the hashed-to list. This implementation should greatly improve search times for mess operations with an actual in the partial key position.

We still have a problem if the set becomes a hot spot: how can we spread out the set among several nodes? The following is a proposal for distributing partial key sets.

- We create a private hash table as described above for each set on each node. Individually, these hash tables can be considerably smaller than the single private hash table usually used, since the set will be divided into many of them.
- Tuples and templates with keys are handled much like the hash paradigm: a hash function takes the key and returns a node and bucket within that node's private hash table for that set. The tuple or template is added to that bucket.
- Tuples and templates without keys may potentially need to examine every node for a match. For now, we assume that they are simply broadcast to all nodes, although "lazier" options are possible. Each node searches its private hash table for a match. If one is found, a copy of the tuple is returned to the requester, the original set aside, and the template deleted. The requester will eventually receive one or more replies to the request. The first will be used, and a positive acknowledgement sent to the node that held it. This ack informs the node that it can delete the tuple permanently. The other replies are simply discarded, and the nodes that sent them are given negative acknowledgements, causing them to reinstall the tuple in TS. This is a variation on the "negative broadcast" paradigm presented in Section 2.4.2, and is very similar to the method used by Arango and Berndt [AB89] in an early network implementation.

Instead of performing a true broadcast, a spanning tree with pruning could be used to reduce the number of messages required. When a node received such a request, it would first search locally for a match. Only if none were found would it forward the message to its children in the spanning tree.

7.8 Summary

Our design has only been tested on machines up to 128 nodes. In this chapter we anticipated some of the concerns for the design on much larger machines, including node and link contention, memory usage, and increasing latency. Several of our optimizationsshould also improved performance on very large machines by reducing contention, including rehashing and data caching, which were originally intended for latency, not contention reduction.

Chapter 8

Related Work

8.1 Introduction

In this chapter we will survey several other approaches for programming distributed memory multiprocessors, focusing on portable models. The models can be classified as:

- Virtual Shared Memory
- Object Oriented Languages
- Semi-automatic Data Parallelism
- Functional Languages
- Logical Languages
- Portable Message Passing
- Other Linda Projects

8.2 Virtual shared memory

It is generally accepted that programming shared-memory multiprocessors is easier than distributed-memory. This approach seeks to combine the advantages of each: the scalability of distributed memory, and the programmability of shared memory.

Shiva

Shiva [LS89] provides the programmer with a shared-memory environment essentially identical to that of shared memory machines by supporting a global shared virtual memory and a interprocess paging mechanism. The shared virtual memory is supported by allowing pages to migrate from processor to processor. Each page is assigned to a *manager* that coordinates its movement and a current *owner* that holds the canonical copy. At any given moment, a page is either readable or writable. If it is writable, only one copy may exist; multiple copies can exist of readable pages.

When a process page faults on a read request, it sends a request to the manager, which forwards it to the current owner. The owner sends the requester a copy, and the manager is notified of the new copy. A fault on a write request causes the current owner to transfer ownership to the requester, and the manager invalidates all other copies by sending messages to the appropriate nodes.

The initial strategy maps pages to managers arbitrarily via a hash function. The multi-stage protocol can be optimized by altering the mapping heuristically so that the owner and manager are the same node as much as possible.

In addition to communication via shared address space, Shiva provides direct message-passing as well, which should be more efficient than page mapping for certain communication patterns.

Shiva has been implemented on hypercubes and networks of workstations.

Munin

Munin [BCZ90] is another proposal simulating shared memory on a distributed memory multiprocessor. It differs from Shiva in that coherency is handled on the basis of user defined data objects, rather than by system defined pages. Several different coherency strategies are used, and the programmer must annotate data objects with the appropriate strategy for each object, although the system does not require the annotations to be correct. Finally, the coherency observed is *loose*; any state that is plausible is acceptable. This allows remote updates to be delayed considerably and combined with other updates, thereby reducing overhead.

Through studies of shared memory programs, Bennett *et al.* recognized eight separate access patterns to data objects:

Write-once Written only during initialization. These objects may be replicated on each node, subject to memory limitations.

Private Referenced locally only. No special runtime support is necessary.

Write-many Written frequently between synchronizations. The delayed update strategy usually allows each process to keep its changes local until a synchronization.

Result Similar to write-many, but the object is only read at the end of the computation. This allows updating to be delayed even more.

Synchronization Implements locks. These objects are maintained by distributed lock servers, one per node, that coordinate the passing of locks from process to process. If possible, locks are preemptively migrated to nodes likely to need them next.

Migratory Accessed by one process at a time. They are frequently associated with a lock, in which case the object and lock are migrated together, possibly preemptively.

Producer-consumer Produced by one process and consumed by a fixed set of others. These objects are preemptively moved to likely consumers as soon as they are produced.

Read-mostly Predominately read. These objects are replicated and updated in place via broadcast.

General read-write Read and written by multiple processes without falling into one of the other categories. These objects are handled via standard ownership cache coherency.

Bennett *et al.* claim that only a small percentage of communication patterns must be handled via general read-write. By classifying each data object by its access pattern and implementing it via a distribution and coherency mechanism tailored for that pattern, they hope to incur lower overheads than those incurred by a fully general strategy of the kind used by Shiva, at the cost of more involvement by the programmer.

Munin has been implemented on a network of workstations.

8.3 Object-oriented models

In object-oriented programming, communication and computation are subsumed by objects and their member functions. The model is inherently distributed, since it emphasizes distinct, independent objects communicating via messages rather than centralized control constructs. However, care must be taken to avoid excessive overhead.

Interwork II

Interwork II [Bai88] is a commercially available product from Block Island Technologies. It combines a lightweight process model with object-oriented programming and a global name space.

Each processor hosts up to hundreds of lightweight tasks. Each task creates and manipulates objects similar to those in C++. Every object is mapped to a *home node*, thus distributing the namespace across the machine. The home node is responsible for coordinating access to the object; however, objects can and do move from node to node as they are accessed. The node on which an object is currently located is its *resident node*. A task can "open" an object, which causes it to be located and moved to the node on which the task is running. No other task may access the object until the task "closes" it.

Moving an object is expensive; rather than opening an object a task can "apply" a member function and a set of parameters to an object. When applying, the parameters are transported to the object and the result is returned. No movement of the object itself is required. Applied member functions may not block, allowing them to be invoked without creating a new thread of control, further reducing costs.

The most general method of locating an object requires querying the home node, which always maintains a pointer to the current resident node for each of its objects. This is expensive; in order to reduce the overhead each node maintains two cache tables. One lists all objects currently held locally, while the other lists the most recently accessed remote objects. By making use of these tables, the load imposed on the home nodes is reduced, as is the time required to access an object.

Other object-oriented models for distributed memory multiprocessors include **Concurrent Smalltalk** [YT86], **Orca** [BT88], and **Emerald** [BHJ+86]. Emerald is unusual in that it supports task as well as object migration.

8.4 Semi-automatic data parallelism

Several proposals involve annotating sequential imperative languages (e.g. Fortran) with directives for mapping processes and data to processors.

DPL

In **DPL** [KZ89], the programmer starts with a Fortran77 code and adds annotations specifying a virtual process structure, i.e. a number of processes and their interconnection, for example rings or grids. The program is specified with respect to this virtual topology; the system is responsible for mapping it to the actual hardware.

Next, the programmer adds annotations that decompose the arrays that appear in the program. This is typically done by breaking each array into rectangular subblocks. The annotation also specifies a mapping between each subblock and a virtual process, which becomes the owner of that block. Only the owner is allowed to write to a block.

Finally the programmer can specify block images, essentially read-only copies of a block. By allowing other processes to cache copies locally, communication is reduced. However, the copies may not be written to.

The compiler takes the annotated Fortran and produces code containing explicit message-passing calls whenever a reference is made that cannot be determined to be local. This has the potential of producing very inefficient execution; a great deal of effort is made to optimize away as much communication as possible.

The process of determining the virtual topology and mapping the data to it can be difficult for the programmer. Some progress has been made towards automating this task [CHZ91].

Other examples of semi-automatic data parallelism include **Parti** [SCMB90] and **Kali** [KMVR90].

8.5 Functional languages

Functional languages are essentially variations of the Lambda calculus [Chu41]. They are attractive candidates for expressing parallelism because they are side-effect free. This means that there are no hidden data dependencies. Furthermore, functional programs are inherently parallel; because they are free of side-effects, each function invocation can evaluate all of its arguments and possibly the function body in parallel. The real problem is not discovering parallelism but controlling it so as to keep overhead to an acceptable level.

Alfalfa

Alfalfa [Gol88] is a parallel implementation of Alfl [Hud84]. Unlike some other parallel functional language proposals, e.g. Paralfl [HS86], Alfalfa does not rely on annotations to specify when to create new threads of control. It depends entirely on the compiler to determine reasonable units of computation, i.e. granularity.

Alfalfa compiles lambda expressions into *combinators*, which have the property that they contain no free variables. A combinator consists only of sequential code compiled for the target machine and input parameters, themselves combinators. These initial combinators are far too fine-grained to allow efficient execution on distributed memory machines.

The second step of the compilation aggregates the simple combinators into serial combinators of sufficient granularity. The cost of evaluating a combinator is usually difficult to determine exactly; instead a heuristic is used to estimate the cost. The quality of this estimation is important for performance, since serial combinators that are too fine will result in excessive overhead, while making them too coarse reduces the parallelism available.

Each node of the machine functions as a combinator machine, evaluating combinators (tasks) and in the process creating new tasks. In order to balance the load, these new tasks may need to be scheduled onto other nodes of the machine. Goldberg experimented with a number of oblivious and communicating diffusion schemes for distributing tasks.

Crystal

Crystal [Che86a] is a functional language intended for scientific computation. Crystal programs are written as functions and recurrence relations. The primary elements of Crystal programs are index domains and data fields.

Index domains are used to define the abstract "shape" of composite data objects. For instance,

```
dom D=[1..n]x[1..n]
```

defines a two-dimensional index domain of n^2 elements. Data fields are analogous to arrays in imperative languages, although elements may only be written once, since as a functional language Crystal is side-effect free. An example data field:

```
dfield a(i,j):D=i*n+j
```

The process of compiling Crystal programs to run efficiently on distributed memory machines consists of three distinct phases [Li91].

First, the abstract Crystal program is transformed to a shared memory representation. An analysis determines dependencies between data fields as a whole; based on this analysis, data fields are partitioned into closely dependent subblocks. Next, all data fields in a particular block are transformed to a common index domain. At this point, individual data elements are analyzed for dependencies. Finally, looping control structures are synthesized to provide the appropriate iteration through the index space. Putting the blocks together produces a program that could be executed on shared memory.

Next, the shared memory program is transformed into an abstract message passing version. The parallel computations specified in the loops of the shared memory program are partitioned and distributed onto processors of a virtual machine. Communication calls are inserted to move data as needed from processor to processor. Communication metrics are used to estimate the cost of a particular layout; optimization techniques are employed to reduce the overhead. An important part of the optimization process involves recognizing communication patterns such as broadcast and gather and invoking the appropriate specialized routine.

Finally, the virtual machine is mapped onto the actual target multiprocessor. This requires mapping abstract processes to nodes, as well as mapping the communication patterns to library calls.

Sisal [Gri90] is another functional language that has been implemented on distributed memory machines.

8.6 Logic Programming

Logic Programming languages, of which Prolog [CG81] is best known, express programs as a set of *clauses*, which may be read procedurally or declaratively. For instance, the following clauses:

```
A:-B,C,D
A:-E,F
```

can be interpreted as "to do A, do either B, C, and D, or do E and F". Alternatively, we can view it as "A is true, if either B, C, and D are true, or E and F are true. Logic languages are a fascinating subject; we refer the interested reader to the work cited above and to [Sha87] for more information.

Prolog programs express two distinct forms of parallelism. First, several different clauses may be evaluated separately. This is called OR-parallelism, since only one of them must succeed. Secondly, each subgoal (in the above example, B, C, D, E and F are subgoals) can be executed in parallel, although data dependencies may limit the extent of parallelism. This is called AND-parallelism, since all of the subgoals must succeed for the clause to succeed.

Chare Parallel Prolog

Kalé *et al.* [KRS89] have implemented a version of parallel Prolog on the Intel iPSC/2, using their Chare Kernel [KS88] as the underlying communication mechanism. They support both AND- and OR-parallelism, although AND-parallelism is limited to fully independent subgoals.

Chares are small to medium grain, specialized processes that represent subparts of the computation. Chares are spawned by parent chares and execute for a small amount of time, possibly creating new subchares. Parent chares receive return values from subchares when they complete; aside from this, chares execute independently.

Each node of the system executes a chare interpreter that computes the pool of chares available. In order to provide load balancing, before being instantiated, chares may migrate freely from the node on which they were created. Kalé *et al.* have experimented with a number of allocation schemes, including randomized and gradient diffusion.

The Parallel Prolog interpreter turns subtasks from AND- and OR-parallelism into chares via a process called *closing*. Closing removes any references in the child task to unbound variables in the parent, freeing the child task to move to other nodes. Once the child completes, the values bound to the closed references are back-unified with the parent.

Other distributed memory implementations of Logic Languages include **H-Prolog** [LRRW89], **Parlog** [Fos88], and **Concurrent Prolog** [Sha87][TSS87].

8.7 Portable message passing

There are a number of proposals for portable message passing systems. Such systems retain most of the characteristics of typical native message-passing libraries, but provide a portable interface to allow moving codes from platform to platform. Examples include

PVM [GS91], **Express** [Par90], **Zipcode** [SS91], and **PICL** [GHPW90][1]. Some of the proposals extend functionality beyond that provided by native libraries; for instance, Express provides extensive parallel I/O facilities and graphical user interfaces.

8.8 Other Linda projects

Lucco [Luc86] implemented Linda on an early generation hypercube multiprocessor. His design included a heuristic for mapping tuples to their destination, similar to our tuple rehashing scheme.

Leichter [Lei89] performed the first network implementation of Linda on a network of DecStations.

Pinakis [Pin92] has implemented a distributed Linda Tuple Space system on a network of Unix workstations. The design is similar to ours in that Tuple Space is split into disjoint pieces and distributed to all of the processors. A fundamental distinction of his design is the use of Type Servers that are responsible for the dynamic mapping of tuple type signatures to individual nodes. When making a Tuple Space request, client processes first locate the appropriate Tuple Space server via a request to their local Type Server. Other interesting features of the design include encoding tuples into a canonical data representation, allowing hosts with different data formats to communicate via tuples, and the use of light-weight threads for both user and system processes.

Arango and Berndt [AB89] also designed and implemented a Linda system for networks of Unix workstations. Their design used a negative broadcast scheme (see Section 2.4.2). The User Datagram Packet protocol of the TCP/IP family was used to broadcast requests for tuples; replies were sent reliably via TCP connection.

[1]PVM and PICL are available via `netlib@ornl.gov`.

Chapter 9

Conclusions

In the course of this dissertation, we have tried to show that Linda can be efficient on distributed memory multiprocessors. We discussed a general strategy for distributing Tuple Space among the processors as well as several automatic optimizations that improve upon the performance of that strategy in a number of important cases. We reported on a specific implementation on Intel's iPSC/2, illustrating the performance of small, synthetic programs that illuminated specific behaviours as well as several real applications that exercised the entire system.

Clearly, the code clarity and portability of Linda is worth some performance penalty. The question is, how much? In the most general case, transferring data from one node to another using our Linda kernel will require three communications, as opposed to one if message passing is used. However, Linda communication is more powerful than message passing, since the producer and consumer need not exist at the same time, nor know each other's identity. In order to communicate via only one message, message passing requires that the exact destination (node and process) of the data be known at the time it is sent. Many Linda communication patterns cannot be translated to only one message, for example, a bag of tasks that workers access independently and unpredictably. This communication pattern can be expressed in Linda via a single `out` and `in`. A message passing equivalent will require more than one message per task, and may well result in poor load balancing if tasks are predistributed, or bottlenecks if a single task server is used. Using message passing, programmers desiring the flexibility natural to Linda will find themselves reinventing the Linda wheel, so to speak.

On the other hand, when such flexibility is not called for, it is important that Linda's overhead be as close to that of message passing as possible. Several of the optimizations are concerned with this aim. For those programs that are natural candidates for message passing, the corresponding Linda programs generally have very low overhead, because such programs usually respond very well to the rehashing optimization. Thus, a true comparison of Linda versus message passing is not three messages versus one. Generally, for both models, powerful, decoupled communication styles require two or three messages, while more straightforward directed communications will usually require one message in either case. Similarly, message passing programs that make use of broadcast usually have a natural Linda equivalent using `rd` and tuple broadcast.

Clearly, there are programs so fine-grained that they will not perform well on Distributed Memory Multiprocessors, whether programmed in Linda, message passing, or anything else. Similarly, there are programs so coarse that they will show good speedup regardless of how they are programmed. The real question is whether there is a significant range of granularity for which message passing is still efficient, but Linda is not. We have shown that range to be quite small on the iPSC/2.

Linda works. I can write a parallel program from scratch, or parallelize an existing code using tuple scope on my own workstation, then move it to a powerful compute engine, whether a collection of workstations or a dedicated supercomputer.

Appendix A

An overview of the iPSC/2

A.1 Introduction

This chapter provides an introduction to the Intel iPSC/2, the machine on which we implemented DMM-Linda. We will discuss hardware aspects of the machine, which includes the processors and the hypercube interconnect. Next, we will present the software environment provided, particularly the message-passing interface. Finally, we will present performance figures for both computation and communication.

The iPSC/2's characteristics have been elaborated by other researchers. See [Bra88], [SS90], and [Nug88] for more detailed descriptions than we present here.

A.2 Hardware

Fundamentally, the iPSC/2 is a *distributed memory* multiprocessor. Each processor has its own local memory, with no common memory shared among the nodes of the machine. It is a *multiple-instruction-stream/multiple-data-stream* (MIMD) computer; each processor issues its own instructions and works its own data, in contrast to single-instruction-stream/multiple-data-stream (SIMD) computers, which issue global instructions that all processors execute in lockstep. It is *loosely coupled*; access to local data is orders of magnitude faster than accessing non-local data.

Each node of the iPSC/2 contains a CPU, a Floating Point Unit (FPU) and a communication coprocessor. The CPU is an Intel 80386, running at 16MHz and rated at 4 million instructions per second (MIPS) [Clo88]. Three floating point options are available on the iPSC/2. The Intel 80387 is the companion numerical processor for the 80386, and performs at 0.16 MFLOPS[1] More performance is available by replacing the 80387 with the Weitek 1167, which provides 0.34 MFLOPS. Finally, a vector board option is available, which provides up to 2.5 MFLOPS [Clo88]. The communication coprocessor is responsible for handling most of the node's communication load, freeing the CPU from most of the costs imposed by message traffic. Each node may contain up to 16 Mbytes of memory. In addition, each node has 64 Kbytes of Cache Ram.

[1]The floating point speeds are for Linpack on a 1000x1000 matrix, see [Clo88].

An iPSC/2 may contain up to 128 nodes connected in a *binary-connected-hypercube*. Each node in the hypercube is directly connected to $\log_2 n$ neighbors, those nodes whose addresses differ from its own by a single flipped bit. Each pair of neighboring nodes is connected via two unidirectional channels, allowing simultaneous bidirectional communication.

Each node has eight independent routers, one for each incoming channel. Each router can connect its incoming channel to any of the outgoing channels, forming a crossbar switch at each node. The routers form a circuit-switched network from source to destination that remains open during the duration of a message.

A message is sent by first setting up the circuit. An initial packet is sent from the source to the destination, setting up the message path along the way. The route is chosen deterministically by correcting the bits that differ between the source and destination addresses, from least to most significant. If, at any point along the way, an outgoing channel is busy, the packet is buffered until the channel is free. When the initial packet arrives at the destination, an acknowledgement is returned along the nodes in reverse, setting up the status path. Once the acknowledgement is received at the source, the source begins streaming the entire message over the established circuit. Since the circuit is dedicated to this particular message, no intermediate buffering is necessary. As the end of the message passes each intermediate node, the connections are released, freeing them for use by other messages. Messages under 100 bytes use a simplified protocol, whereby the message is included in the initial packet, and no circuit construction is necessary.

Each node can concurrently send and receive a message, as well as forward as many additional messages as it has free links. Because of the speed of the routers, multi-hop messages experience delays only slightly higher than single-hop delays, assuming no contention.

A.3 Software

The iPSC/2 runs a simple operating system kernel called NX on each node. NX provides process control and buffered, queued message passing, as well as some of the functionality of Unix-like runtime libraries. Multiple user processes may exist on each node; each process is identified by its `<processor,process id>` pair.

The message passing routines include synchronous calls `csend()` and `crecv()`, which block until the operation is complete. `csend()` directs a message to a particular node and process id. In addition, the message is given a type identifier, allowing it to be distinguished from other messages. `crecv()` asks for a message of a given type; the call will block until a suitable message arrives. A message arriving for which no receive is posted is buffered in system space until requested; this incurs an additional copy that is avoided if the receive occurs first.

Asynchronous message passing routines `isend()` and `irecv()` return immediately with a message identifier. The message identifier can be polled to determine if the operation has completed. An interrupt version of the asynchronous receive, called `hrecv()`, allows a receive to be posted with an interrupt handler; when an appropriate message arrives, the handler is called.

A.4 Performance

The configuration of the iPSC/2 we used is presented in Table A.1.

nodes	64
memory/node	4 Mbytes
FPU	Weitex 1167
CPU	4.0 MIPS
FPU single prec.	0.72 MFLOPS
FPU double prec.	0.30 MFLOPS

Table A.1: Characteristics of the iPSC/2 used for our Linda implementation

The single precision MFLOP rate is a dot product measured by us. The double precision figure is for a **daxpy** measured by Close [Clo88].

Figure A.1 shows message latencies for messages of various lengths. Times for both `crecv()` and `irecv()` are shown. These times were gathered by passing a message around a ring of 16 processors, which is large enough to guarantee that the receive operation will precede the corresponding send. As mentioned above, receives following sends are slightly more expensive, since an additional copy is necessary. Each process in the ring simply sent to the next numbered processor; the ring was not mapped to be nearest neighbor. Each hop adds about 10 μsec, see Figure A.2[2].

[2]These figures are for a single 4 byte message sent in a repeated loop between two processes, using csend and crecv.

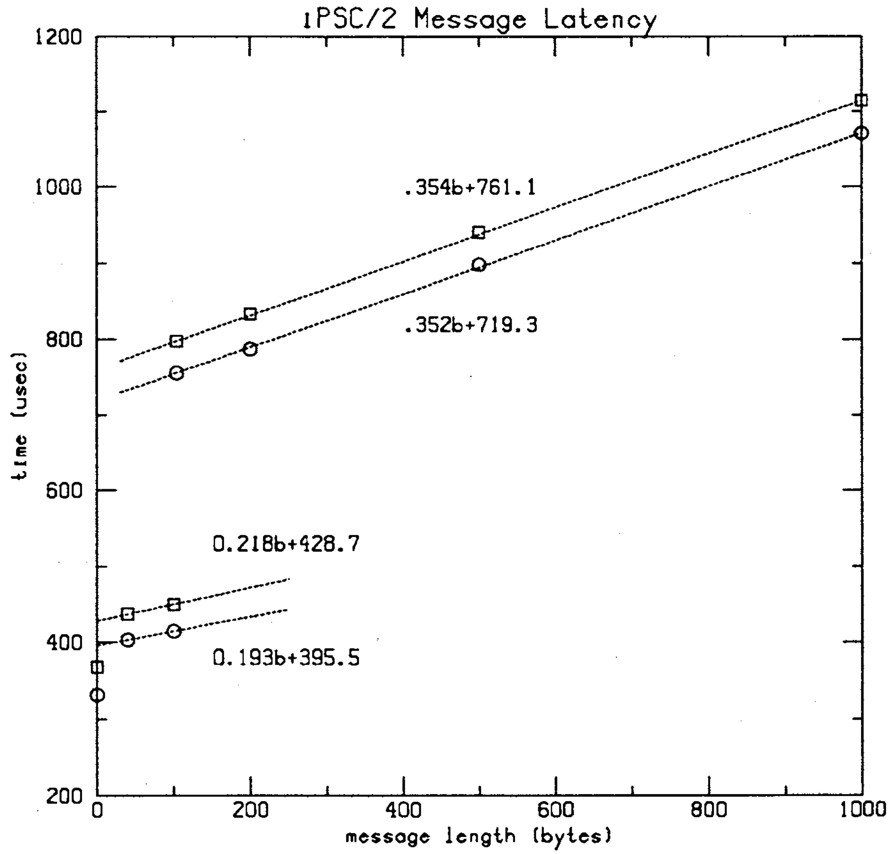

Figure A.1: Message latency for various message sizes on the iPSC/2, using **crecv()** ○ and **irecv()** □ .

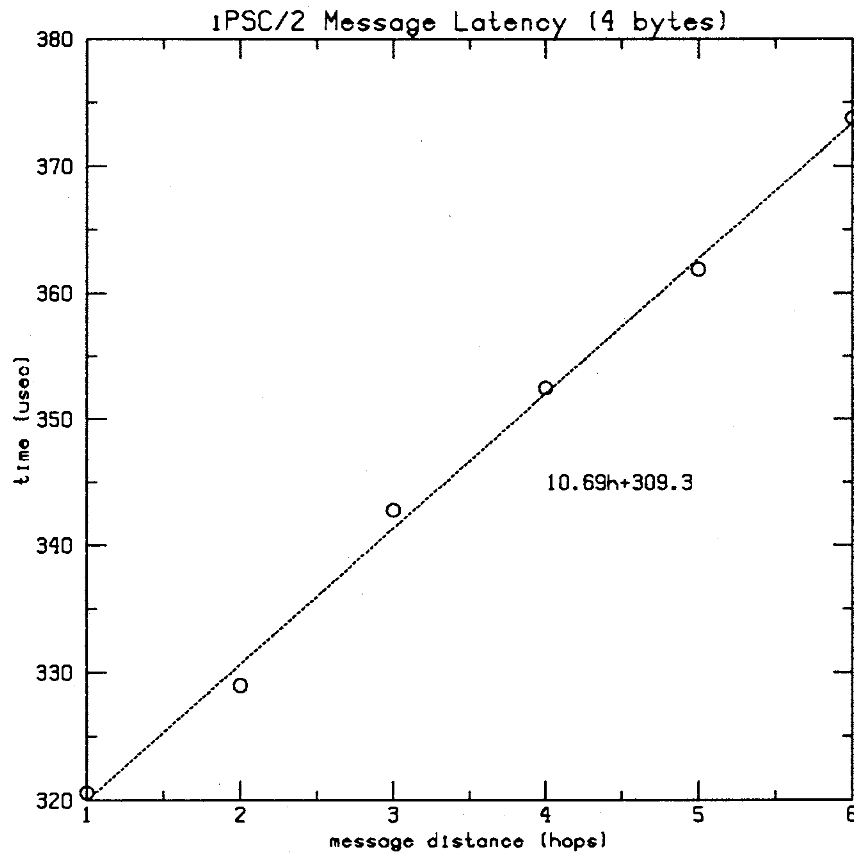

Figure A.2: Message latency for various path lengths on the iPSC/2, for a 4 byte message using `crecv()` and `csend()`

Appendix B

Application Source Code

B.1 DNA

dna_master.cl

```
#include        "dna.h"

  /* Sequence storage. Space (but not values) shared with gotoh_worker.cl. */
char    dbs[MAX_LENGTH + HEADER_LENGTH];
char    target[MAX_LENGTH];

extern void     close_db();
extern int      compare(), get_seq(), get_target(), atoi();
extern void     open_db();
int         z = 17;                                                        10

real_main(argc, argv)
int         argc;
char        **argv;
{
  char          *start, *end;
  MAX_TYPE          max, real_max;
  int           bf_h, bf_w, d_length, height, i, j, len, lower;
  int           num_workers, real_max_id, t_length;
  int           task_id, tasks, upper, width;                              20

  start_timer();

  if (argc != 8) {
    printf("Usage: %s <target> <db> <# workers> <h> <w> <l> <u>\n",
           argv[0]);
    exit();
  }

  num_workers = atoi(argv[3]);                                             30
  height = atoi(argv[4]);
  width = atoi(argv[5]);
  lower = atoi(argv[6]);
  upper = atoi(argv[7]);
  printf("%s: %d %d %d %d\n",
         argv[0], num_workers, height, width, lower, upper);

  /* Read target sequence from disk. */
  t_length = get_target(argv[1], target);
  out("target", target:t_length);                                          40
  out("block data", height, width);
  open_db(argv[2]);
```

dna_master.cl *(continued)*

```
    /* Start workers. */
    for (i = 0; i < num_workers; ++i) eval("worker", compare());

    timer_split("done setting up.");

    task_id = tasks = 0;
    bf_w = (t_length + width - 1)/width;

    while (d_length = get_seq(dbs)) {
      end = (dbs+HEADER_LENGTH+d_length);                                  10
      bf_h = (d_length + height - 1)/height;
      out("task", task_id, bf_h, bf_w, 0);
      for (i=0, start=dbs+HEADER_LENGTH; i<bf_h; ++i, start+=height) {
        len = MIN(height, end-start);
        out("segment", task_id^i, task_id, i, start:len);
      }
      ++task_id;
      if (++tasks >= upper) {
        printf("working %d\n", task_id);
        do in("done"); while (--tasks > lower);                            20
      }
    }

    while (tasks--) in("done");
    /* Send poison tasks. */
    for (i = 0; i < num_workers; ++i) {
      out("task", 0, 0, 0, 0);
    }
    printf("sent poison");
    close_db();                                                            30

    /* Gather maxes local to a given worker and compute global max. */
    real_max.max = 0;
    for (i = 0; i < num_workers; ++i) {
      in("result", ? task_id, ? max);
      printf("got a result\n");
      if (max.max > real_max.max) {
        real_max = max;
        real_max_id = task_id;
      }                                                                    40
    }
    timer_split("done.");

    printf("id %d: max = %d (%d, %d).\n",
           real_max_id, real_max.max, real_max.max_i+1, real_max.max_j+1);
    print_times();
  }
```

dna_worker.cl

```
#include "dna.h"

#define F(a, b, c) (a ^ (c << 3))

/* Sequence storage. The "extern" recycles space—it is not a backdoor copy. */
extern char     dbs[MAX_LENGTH + HEADER_LENGTH];
extern char     target[MAX_LENGTH];

/* Work space for a vertical slice of the similarity matrix. */
static ENTRY_TYPE                col_0[MAX_DIM + 2];                          10
static ENTRY_TYPE                col_1[MAX_DIM + 2];
static ENTRY_TYPE                *cols[2] = { col_0, col_1 };
static ENTRY_TYPE                junk[MAX_DIM + 2];
static ENTRY_TYPE                top_edge[MAX_DIM];

extern  z;
/* worker code. */
compare()
{
  SIDE_TYPE     left_side, top_side;                                           20
  MAX_TYPE        max, real_max;
  int           bf_h, bf_w, d_length, height, i, id, len, next_x;
  int           next_y, num_workers, real_width, real_height;
  int           start, t_length, width, x, y;
  int           real_max_id = -1;

  left_side.seq_start = dbs;
  top_side.seq_start = target;
  /* Read target sequence from tuple space. */
  rd("target", ? target:t_length);                                             30
  rd("block data", ? height, ? width);
  real_max.max = 0;
  while(1) {
    in("task",  ? id, ? bf_h, ? bf_w, ? y);
    if (!bf_h) {
      out("result", real_max_id, real_max);
      return 0;
    }
    in("segment", id^y, id, y, ? dbs:d_length);
    real_height = MIN(d_length, height);                                       40
    left_side.seq_end = dbs + real_height;
    left_side.seq_start = dbs - y * height;
```

dna_worker.cl *(continued)*

```
for (x = 0, start = 0; x < bf_w; ++x, start += width){

  real_width = MIN(t_length-start, width);
  top_side.seg_start = target + start;
  top_side.seg_end = target + start + real_width;
  if (!y) {
    /* "Top" top edge—Zero out top_edge buffer. */
    for (i = 0; i <= real_width + 1; ++i) top_edge[i] = zero_entry;
  }
  else {                                                              10
    /* "Inner" top edge-get from TS */
    in("top edge", F(id, x, y), id, x, y, ? top_edge:len);
  }
  if (!x) {
    /* Left edge—Zero out column buffer. */
    for (i = 0; i <= real_height+1; ++i)
      cols[0][i] = cols[1][i] = zero_entry;
  }
  max.max = 0;
  similarity(&top_side, &left_side, cols, top_edge, &max);           20
  if (max.max > real_max.max) {
    real_max = max;
    real_max_id = id;
  }
  next_x = x + 1;
  next_y = y + 1;

  if (next_y == bf_h)
    { if (next_x == bf_w) out("done"); }
  else {                                                              30
    /* put out next top edge */
    out("top edge", F(id, x, next_y), id, x, next_y,
      top_edge:real_width+2);
    if (x == 0) {
      /* start next row */
      out("task", id, bf_h, bf_w, next_y);
}}}}}
```

similarity.c

```
/* Similarity parameters. */
#define alpha          4       /* Indel penalty. */
#define beta           1       /* Extension penalty. */

/* Weight table for matching up genetic symbols "actg...".
   The set of 16 symbols has been mapped in the preprocessing
   of the data to the 16 characters beginning with ' '.
   Masking the characters with 0xF yields indices for this table
   in the range 0-15. */
static short     match_weights[16][16] = {
   0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
  -1,  3, -1,  2, -1,  2, -1,  1, -1,  2, -1,  1, -1,  1, -1,  0,
  -1, -1,  3,  2, -1, -1,  2,  1, -1, -1,  2,  1, -1, -1,  1,  0,
  -1,  2,  2,  2, -1,  1,  1,  1, -1,  1,  1,  1, -1,  0,  0,  0,
  -1, -1, -1, -1,  3,  2,  2,  1, -1, -1, -1, -1,  2,  1,  1,  0,
  -1,  2, -1,  1,  2,  2,  1,  1, -1,  1, -1,  1,  1,  1,  0,  0,
  -1, -1,  2,  1,  2,  1,  2,  1, -1, -1,  1,  0,  1,  0,  1,  0,
  -1,  1,  1,  1,  1,  1,  1,  1, -1,  0,  0,  0,  0,  0,  0,  0,
  -1, -1, -1, -1, -1, -1, -1, -1,  3,  2,  2,  1,  2,  1,  1,  0,
  -1,  2, -1,  1, -1,  1, -1,  0,  2,  2,  1,  1,  1,  1,  0,  0,
  -1, -1,  2,  1, -1, -1,  1,  0,  2,  1,  2,  1,  1,  0,  1,  0,
  -1,  1,  1,  1, -1,  1,  0,  0,  1,  1,  1,  1,  0,  0,  0,  0,
  -1, -1, -1, -1,  2,  1,  1,  0,  2,  1,  1,  0,  2,  1,  1,  0,
  -1,  1, -1,  0,  1,  1,  0,  0,  1,  1,  0,  0,  1,  1,  0,  0,
  -1, -1,  1,  0,  1,  0,  1,  0,  1,  0,  1,  0,  1,  0,  1,  0,
  -1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0
  };

#include "dna.h"

similarity(top_side, left_side, cols, top_edge, max)
     SIDE_TYPE *top_side, *left_side;
     ENTRY_TYPE *cols[2], *top_edge;
     MAX_TYPE *max;
{
  register char          *ap, *bp;
  register ENTRY_TYPE    *cp, *dp;
  register int           d;
  register short         *mwr;
  register int           p, q, t;
  ENTRY_TYPE             *temp;
  /* Compute similarity matrix for sub-block. */
  for (bp = top_side->seg_start; bp < top_side->seg_end; ++bp) {
    /* Rotate column buffers. */
    temp = cols[0];
    cols[0] = cols[1];
    cols[1] = temp;
    cols[0][0] = top_edge ? top_edge[bp - top_side->seg_start] : zero_entry;
```

similarity.c *(continued)*

```
/* Compute p,q,d values for current column.
   Note we need to refer to two entries in each column,
   hence the "++"s in the body of the loop. */
mwr = match_weights[*bp & 0xF];
for (ap = left_side->seg_start, cp = cols[0], dp = cols[1];
     ap < left_side->seg_end; ++ap) {
  d = dp->d + mwr[*ap & 0xF];
  if (d < 0) d = 0;
  ++dp;

  p = dp->d - alpha;
  t = dp->p - beta;
  if (p < t) p = t;

  q = cp->d - alpha;
  t = cp->q - beta;
  if (q < t) q = t;

  /* Maximize d. */
  if (p > d) d = p;
  if (q > d) d = q;

  ++cp;
  cp->d = d;
  cp->p = p;
  cp->q = q;

  /* Remember overall max. */
  if (d > max->max) {
    max->max = d;
    max->max_i = ap - left_side->seg_start;
    max->max_j = bp - top_side->seg_start;
  } }
 if (top_edge) top_edge[bp - top_side->seg_start] = *cp;
} }
```

seq_io.c

```
/* I/O support routines. */

/* Sys V/BSD difference. */
#ifdef CUBE
#include        <fcntl.h>
#else
#include        <sys/file.h>
#endif

#include <stdio.h>                                                  10
#include "dna.h"

extern int      open();
extern int      read();

/* Read target sequence. */
int get_target(target_file, target_buf)
    char        *target_file;
    char        *target_buf;
{                                                                   20
  int    t_fd;
  int    t_length;

  t_fd = open(target_file, O_RDONLY);
  if (t_fd < 0) {
    fprintf(stderr, "%s (%d): cannot read target file %s.\n",
            __FILE__, __LINE__, target_file);
    exit(1);
  }
  t_length = read(t_fd, target_buf, MAX_LENGTH);                    30
  close(t_fd);
  return t_length;
}
```

seq_io.c *(continued)*

```
/* Functions to manage a database file.

   This code is more complicated than you might expect. In order
   to reduce the number of i/o operations, the file format
   is such that one read gets one sequence and the length of
   the next sequence:

   [id0 len1][id len_next_seq seq]*[idn 0 seqn]

   The id and length fields constitue the header. Note that          10
   the first sequence is a null sequence. It's header provides
   the length of the first real sequence. The user supplied
   buffer must be big enough to hold seq data and the header.*/

static int     db_fd;
static int     len_next_seq;

extern int     atoi();

int open_db(db_file)                                                  20
    char       *db_file;
{
  char  header[HEADER_LENGTH+1];

  header[HEADER_LENGTH] = 0;
  /* Open database file. */
  db_fd = open(db_file, O_RDONLY);
  if (db_fd < 0) {
    fprintf(stderr, "%s (%d): cannot read data base file %s.\n",
            __FILE__, __LINE__, db_file);                            30
    exit(1);
  }

  /* Read header of initial null sequence.
     Something's wrong if a whole header cannot be read. */
  if (read(db_fd, header, HEADER_LENGTH) < HEADER_LENGTH) {
    fprintf(stderr, "%s (%d): corrupt header.\n", __FILE__, __LINE__);
    exit(1);
  }
  len_next_seq = atoi(header + LENGTH_OFFSET);                        40
}
```

seq_io.c *(continued)*

```
int close_db()
{
  close(db_fd);
}

int get_seq(buf)
     char          *buf;
{
  int                seq_len;
                                                                        10
  if (!len_next_seq) return 0;

  /* Read sequence. */
  seq_len = len_next_seq;
  len_next_seq += HEADER_LENGTH;
  if (len_next_seq != read(db_fd, buf, len_next_seq)) {
    fprintf(stderr, "%s (%d): corrupt sequence.\n", __FILE__, __LINE__);
    exit(1);
  }
  len_next_seq = atoi(buf + LENGTH_OFFSET);                             20

  return seq_len;
}
```

B.2 Block Matrix Multiply

block_mat.cl

```
/* Linda Block Matrix Multiply
 *
 * This program multiplies two dense matrices. The matrices are
 * generated on the fly. Each process receives one subblock of each
 * matrix.
 *
 * This program does dense matrix multiplication. The total number of
 * processes involved in the computation must be a perfect square (n*n).
 * The master builds two reasonable matrices, a and b, of dimension d. n
 * must divide d. Both a and b are broken up into n*n identical sized
 * blocks and passed out to the processes, such that process i,j gets
 * Ai,j+i and Bi+j,j. Each process multiplies its two submatrices,
 * adding the results to an accumulator matrix. Next, each process
 * passes its A block to the right and its B block down, and the process
 * repeats. After d steps, the accumulator holds Ci,j, one block of the
 * resultant matrix. This block is passed back to the master.
 *
 * In the program, all large data structures are allocated dynamically.
 * Only the master needs to allocate memory for the complete matrices A
 * and B.
 *
 * The program is invoked as follows:
 *
 * block_mat <dim> <blocking> [reps]
 *
 * <blocking> is the number of subblocks along each dimension.
 * Note that the master becomes a worker after initialization: i.e. only
 * (<blocking>**2)-1 evals are performed. Remember that workers must be a
 * square. <dim> is the dimension of A and B. <blocking> must divide it.
 * [reps] is an optional argument; it specifies the number of repetitions, i.e.
 * rep=4 yields the result of ((((A*B)*B)*B)*B). [reps] defaults to 1.
 */

#include <stdio.h>
#define halt exit

#define ABS(x) ((x)>0?(x):(-x))

#define F(i,j) ((i)*blocking+(j))
#define EPSILON .00001
#define ABASE 5.00
#define BBASE 1.05

#define START_THRESHOLD 6
```

block_mat.cl(continued)

```
#define PACK(array, yb, xb) {\
  int ii = 0, xx, yy; \
  for (yy=0; yy<block_dim; ++yy) { \
    for (xx=0; xx<block_dim; ++xx) { \
      mess[ii++] = array[yy][xx]; }}}

#define UNPACK(array, yb, xb) {\
  int ii = 0, xx, yy; \
  for (yy=0; yy<block_dim; ++yy) { \
    for (xx=0; xx<block_dim; ++xx) { \                          10
      array[yy][xx] = mess[ii++]; }}}

#define MAX 256
typedef float ELEM;
ELEM a[MAX][MAX];
ELEM b[MAX][MAX];
char *malloc();

real_main(argc, argv)
int argc;                                                       20
char *argv[];
{
  int blocking, workers, block_dim, dim, len;
  int x, y, xb, yb, i, rlen, num_evals;
  ELEM *mess;
  ELEM correct, correct2;
  int worker();
  int reps, j;
  if (argc != 4) {
    fprintf(stderr, "usage: %s <dim> <reps> <blocking>\n", argv[0]);   30
    exit(1);
  }

  dim = atoi(argv[1]);
  if (dim > MAX) {
    fprintf(stderr, "maximum dimension %d\n", MAX);
    exit(1);
  }

  reps = atoi(argv[2]);                                         40
  blocking = atoi(argv[3]);
  workers = blocking * blocking;
  block_dim = dim/blocking;

  if(!(mess = (ELEM *) malloc(block_dim*block_dim*sizeof(ELEM)))) {
    fprintf(stderr, "malloc failed in master\n");
    halt();
  }
  printf("%s: dim %d reps %d blocking %d workers %d\n", argv[0], dim, reps, blocking, workers);
                                                                50
  out("global data", block_dim, blocking, workers, reps);
```

block_mat.cl(continued)

```
num_evals = workers - 1;

start_timer();

/* initialize arrays */

for (y=0; y<dim; ++y)
  for (x=0; x<dim; ++x) {
    a[y][x] = x * ABASE;
    b[y][x] = y * BBASE;                                        10
  }

correct = 0.0;
for (i=0; i<dim; ++i) correct += ABASE*i * BBASE*i;
for (j=0; j<reps-1; ++j) {
  correct2 = 0.0;
  for (i=0; i<dim; ++i) correct2 += correct * BBASE * i;
  correct = correct2;
}
timer_split("done initializing");                               20

/* eval workers */
for (i=1; i<=num_evals; ++i)
  eval("worker", worker(i));

timer_split("evaled workers");

/* out arrays */
len = block_dim * block_dim;
for (yb=0; yb<blocking; ++yb)                                   30
  for (xb=0; xb<blocking; ++xb) {
    PACK(a, yb, xb);
    out("init a block", F(yb, xb), mess:len);
  }
for (yb=0; yb<blocking; ++yb)
  for (xb=0; xb<blocking; ++xb) {
    PACK(b, yb, xb);
    out("init b block", F(yb, xb), mess:len);
  }
worker(0);                                                      40
timer_split("done multiplying");

/* get results */
for (yb=0; yb<blocking; ++yb)
  for (xb=0; xb<blocking; ++xb) {
    in("result", F(yb, xb), ?mess:rlen);
    UNPACK(a, yb, xb);
  }
```

block_mat.cl(continued)

```
    timer_split("all done");

    /* get workers */
    for (i=1; i<workers; ++i) in("worker", ? int);

    /* check results */
    for (y=0; y<dim; ++y)
      for (x=0; x<dim; ++x)
        if (ABS(a[y][x]-correct) > EPSILON)
          printf("value %d %d = %f, correct = %f\n", y, x, a[y][x], correct);          10
    print_times();
}

worker(my_id)
int my_id;
{
    int iter, ddim, x, y, my_x, my_y, dimsq, blocking, workers;
    ELEM *alinear, *blinear, *clinear;
    ELEM *cp;
    register ELEM dot;                                                                20
    register int i, dim;
    register ELEM *ap, *bp;
    int act_my_y, act_my_x, divisor, reps;

    /* get global data */

    rd("global data", ?ddim, ?blocking, ?workers, ?reps);

    dim = ddim;
    dimsq = dim*dim;                                                                  30
    if(!(alinear = (ELEM *) malloc(dimsq*sizeof(ELEM)))) {
      fprintf(stderr, "malloc failed in worker\n");
      halt();
    }
    if(!(blinear = (ELEM *) malloc(dimsq*sizeof(ELEM)))) {
      fprintf(stderr, "malloc failed in worker\n");
      halt();
    }
    if(!(clinear = (ELEM *) malloc(dimsq*sizeof(ELEM)))) {
      fprintf(stderr, "malloc failed in worker\n");                                   40
      halt();
    }

    /* determine my id */
    my_y = my_id / blocking;
    my_x = my_id % blocking;
```

block_mat.cl*(continued)*

```
/* get my data */
in("init a block", F(my_y, (my_x + my_y) % blocking), ?alinear:);
out("token");
in("init b block", F((my_x + my_y) % blocking, my_x), ?blinear:);
out("token");

if (my_id == 0) {
  out("go");
  timer_split("outed matrices");
}                                                                    10
rd("go");

/* start looping */
while (reps--) {
  for (iter=0; iter<blocking; ++iter) {
    /* do my multiply */
    for (y=0; y<dim; ++y) {
      for (x=0; x<dim; ++x) {
        cp = clinear + (y*dim+x);
        dot = 0.0;                                                   20
        for (i=dim, ap=alinear+y*dim, bp = blinear+x;
             i;
             --i, ++ap, bp += dim){
          dot += *ap * *bp; /* c[y][x] = a[y][i] * b[i][x]; */
        }
        *cp = dot;
      }
    }
    if (iter == blocking-1) break; /* don't need the last send */
                                                                     30
    /* pass a to the right */
    out("a block", F(my_y, (my_x+1)%blocking), alinear:dimsq);
    in("a block", F(my_y, my_x), ?alinear:);

    /* pass b down */
    out("b block", F((my_y+1)%blocking, my_x), blinear:dimsq);
    in("b block", F(my_y, my_x), ?blinear:);
  }
  /* copy c to a */
  for (i=0; i<dimsq; ++i) alinear[i] = clinear[i];                   40
}
/* send_results */
out("result", F(my_y, my_x), clinear:dimsq);
}
```

B.3 Clumped Matrix Multiply

matclump.cl

```
/* Linda Matrix Multiply (Clumping Version)
 *
 * This program performs dense matrix multiplication C=AxB using a
 * master/worker approach.
 * First, the master outs the rows of A and B. Next, the workers are
 * evaled and a task tuple initialized to 0 is outed.
 *
 * Each worker gets a task from the task tuple, and performs the
 * task described. In order to increase granularity, both A and B can
 * be "clumped", in which case multiple rows are put into a single tuple.      10
 * A single task consists of computing one clump of C, which requires
 * reading the appropriate clump of A and all of B.
 *
 * This code also contains the mechanism for the inout optimization.
 * Normally the increment function (jump, in this case) would be produced
 * automatically by the compiler.
 */

#ifdef INOUT
#include "../ikernel/lstructs.h"                                              20
#endif

#define _l_delay(n) usleep((n)*1000)

typedef float    ELEM;
char     *malloc();

#define MIN(x, y) ((x) < (y) ? (x) : (y))

int gaclump;                                                                  30
int gdim;

/* This is the function used for the INOUT collapse optimization.
 * It increments the row index by gaclump
 */
#ifdef INOUT
int jump(ptp_ptr)
PTP_PTR ptp_ptr;
{
  ptp_ptr->field[0].ii = MIN(ptp_ptr->field[0].ii+gaclump, gdim);             40
}
#endif
```

matclump.cl *(continued)*

```
real_main(argc, argv)
     int          argc;
     char         *argv[];
{
  long            dim, aclump, bclump, workers;

  if (argc != 5) {
    printf("Usage: %s <workers> <dim> <aclump> <bclump>\n", *argv);
    lexit(1);
  }

  workers = atol(argv[1]);
  dim = atol(argv[2]);
  aclump = atol(argv[3]);
  bclump = atol(argv[4]);

  printf("%s: workers %d dim %d aclump %d bclump %d\n",
         argv[0], workers, dim, aclump, bclump);
  master(dim, aclump, bclump, workers);
}

master(dim, aclump, bclump, workers)
     long         dim, aclump, bclump, workers;
{
  long            index, i, j;
  long            true_result, row_index, col_index;
  ELEM            *row, *col, *res, *rp, *cp;
  int             worker();
  int             cnt, total_cnt=0;

  row = (ELEM *) malloc(sizeof(ELEM) * dim * aclump);
  col = (ELEM *) malloc(sizeof(ELEM) * dim * bclump);
  res = (ELEM *) malloc(sizeof(ELEM) * dim * aclump);

  rp = row;
  cp = col;
  for (i = dim * aclump; i-- ;) *rp++ = (ELEM) 3;
  for (i = dim * bclump; i-- ;) *cp++ = (ELEM) 5;

  start_timer();

  for (i = 0; i < dim; i += aclump) {
    out(i, "row", row:dim*aclump);
  }
  for (i = 0; i < dim; i += bclump) {
    out(i, "col", col:dim*bclump);
  }
```

matclump.cl *(continued)*

```
  for (i=0; i<workers; ++i)
    eval("worker", worker(dim, aclump, bclump));

  out("task", 0L);
  timer_split("init done");

  true_result = (ELEM) 15 * dim;
  for (index = 0; index < dim; index += aclump) {
    in("res", ? row_index, ? cnt, ? res:);
  }                                                          10

  timer_split("finished matrix!!!");
  print_times();
}

worker(dim, aclump, bclump)
    long        dim, aclump, bclump;
{
  long          j, r, c, row_index, col_index, row_inv;
  register int        i;                                     20
  int                 cnt;
  register ELEM       dot;
  register ELEM       *rp, *cp;
  ELEM          *row, *col, *res, *res_ptr;

  gaclump = aclump;
  gdim = dim;
  row = (ELEM *) malloc(sizeof(ELEM) * dim * aclump);
  col = (ELEM *) malloc(sizeof(ELEM) * dim * bclump);
  res = (ELEM *) malloc(sizeof(ELEM) * dim * aclump);        30

  while (1) {
    cnt = 0;

#ifdef INOUT
    inout("task", ?row_index) jump;
#endif
    if (row_index == dim) return;
#ifndef INOUT
    in("task", ? row_index);                                 40

    if (row_index < dim)
      out("task", row_index + aclump);
    else {
      out("task", dim);
      return;
    }
#endif
```

matclump.cl *(continued)*

```
        rd(row_index, "row", ? row:);

        for (col_index = 0; col_index < dim; col_index += bclump) {
          rd(col_index, "col", ? col:);

          for (r = 0; r < aclump; ++r) {
            res_ptr = res + (r * dim + col_index);
            for (c = 0; c < bclump; ++c){
              dot = (ELEM) 0;
              rp = row + (r * dim);                                          10
              cp = col + (c * dim);
              for (i = dim; i > 0; --i, ++rp, ++cp)
                dot += *rp * *cp;
              *res_ptr++ = dot;
            }
          }
        }
        out("res", row_index, cnt, res:dim*aclump);
      }
    }                                                                        20
```

B.4 LU Decomposition

lu.cl

```
/* Linda LU Decomposition
 *
 * This program performs the first part of LU decomposition, factoring the
 * matrix. The code is based on Dongarra's LU benchmark, and Lindafied
 * by Nick Carriero.
 *
 * The program generates a reasonable test matrix, starts up the workers,
 * then calls dgefac_. Each worker gets a subset of the matrix, and
 * performs eliminations on its subset.
 *                                                                       10
 * Invocation:
 *
 * lu <dim> <workers>
 */

typedef struct linda_block {
  int size;
  double *data;
} LINDA_BLOCK;
                                                                         20
#define MAX    1200

double        junk[MAX];
double        mat[MAX][MAX];
long          pvec[MAX];
long          num_workers;

real_main(argc, argv)
     int      argc;
     char     **argv;                                                    30
{
  long  dim;
  long  i, workers[16];
  long  status;
  int   lu_worker();
  int   mclock(), t1, t2;
  char *flags;

  if (argc != 3) {
    printf("Usage: <dim> <workers>\n");                                  40
    exit(1);
  }

  dim = atol(*++argv);
  num_workers = atol(*++argv);
```

lu.cl *(continued)*

```
  for (i = 0; i < num_workers; ++i) {
    eval("worker", lu_worker(i));
  }
  start_timer();
  mat_gen(mat, (long) MAX, dim);

  timer_split("done setting up");

  dgefac_(mat, (long) MAX, dim, pvec, &status);
                                                                    10
  timer_split("done.");
  print_times();

  print_mat(mat, (long) MAX, dim);

}

dgefac_(a, lda, n, ipvt, info)
     double  *a;
     long     lda, n, *ipvt, *info;                                 20
{
  long     j, k, l, length, nm1, one;
  double   temp;
  double   *l_ptr, *diag_ptr, *p_ptr, *m_ptr;
  LINDA_BLOCK COL, MULT, PIVOT;

  *info = -1;
  nm1 = n - 1;
  one = 1;
                                                                    30
  /* Start up workers. */
  out("LU worker data", num_workers, n);

  /* Dump columns of the matrix into tuple space. */

  COL.size = n;
  for (j = 1, COL.data = a + lda;
       j < n; ++j, COL.data = (double *)COL.data + lda) {
    out("LU col", j, COL.data:COL.size);
  }                                                                 40

  /* First column is first pivot. */
  COL.data = a;
  out("LU pivot col", COL.data:COL.size);
```

lu.cl*(continued)*

```
if (nm1 > 0) {
  for (k = 0, PIVOT.data = a; k < nm1; ++k,
       PIVOT.data = (double *)PIVOT.data + lda) {
    diag_ptr = (double *)PIVOT.data + k;
    if (!(k % 10)) printf("doing %d\n", k);
    /* Wait for a worker to return next pivot column. */
    in("LU pivot col", ? PIVOT.data:PIVOT.size);
    /* Find the max in the pivot column. */
    length = n - k;
    l = idamax_(length, diag_ptr);
    l_ptr = diag_ptr + l;
    l += k;
    ipvt[k] = l + 1;

    /* If a zero pivot, no work to be done this cycle —
       signal workers via a zero length multiplier vector. */
    if ((*l_ptr) == 0.0) {
      *info = k;
      MULT.size = 0;
      out("LU multipliers", k, l,
          MULT.data:MULT.size);
      continue;
    }

    /* Pivot if necessary. */
    if (l != k) {
      temp = *diag_ptr;
      *diag_ptr =  *l_ptr;
      *l_ptr = temp;
    }

    /* Compute multipliers. */
    temp = -1.0 / *diag_ptr;
    length = nm1 - k;
    m_ptr = diag_ptr + 1;
    dscal_(length, temp, m_ptr);
    MULT.data = m_ptr;
    MULT.size = length;

    /* Notify workers of new multipliers. */
    out("LU multipliers", k, l, MULT.data:MULT.size);
  }
}
```

lu.cl*(continued)*

```
        /* Get the last column. */
        in("LU pivot col", ? PIVOT.data:PIVOT.size);
        ipvt[nm1] = n;
        if (*(a + nm1 * lda + nm1) == 0.0) {
          *info = nm1;
        }
        *info += 1;
      }
      lu_worker(id)
          long        id;                                                          10
      {
        double      (*c_ptr)[MAX], (*first_col_ptr)[MAX],
        double      (*last_col_ptr)[MAX];
        long        dim, i, j, ji, k, l, length, nm1, num_workers, one;
        long        map[MAX];
        long        *map_ptr;
        double      mult[MAX];
        double      *l_ptr, *p_ptr, temp;
        LINDA_BLOCK COL, MULT;
                                                                                   20
        one = 1;

        /* Get task data. */
        rd("LU worker data", ? num_workers, ? dim);

        nm1 = dim - 1;

        /* Read in columns — note stepping by num_workers shuffles the
           columns among the workers. map will map between the worker's
           index and the original index, a will hold a worker's columns packed    30
           together. */
        for (i = id, j = 0; i < dim; i += num_workers, ++j) {
          if (!i) {--j; continue;} /* skip 0th row — the first pivot. */
          map[j] = i;
          COL.data = mat[j];
          in("LU col", i, ? COL.data:COL.size);
        }
        first_col_ptr = mat[0];
        last_col_ptr = mat[j];
        map[j] = -1;                                                               40
        map_ptr = map;

        MULT.data = mult;
        for (k = 0; k < nm1 && first_col_ptr < last_col_ptr; ++k) {
          rd("LU multipliers", k, ? l, ? MULT.data:MULT.size);
```

lu.cl*(continued)*

```
          /* Loop through my columns doing daxpy's. */
          for (c_ptr = first_col_ptr; c_ptr < last_col_ptr; ++c_ptr) {
```

```
        /* If non-zero pivot then mult size will be non-zero. */
        if (MULT.size) {
          l_ptr = (double *)c_ptr + l;
          p_ptr = (double *)c_ptr + k;
          temp = *l_ptr;
          if (l != k) {
            *l_ptr = *p_ptr;
            *p_ptr = temp;                                          10
          }
          length = nm1 - k;
          daxpy_(length, temp, mult, p_ptr + 1);
        }
        /* If my first column is the next pivot, send it back. */
        if(*map_ptr == k + 1) {
          COL.data = c_ptr;
          out("LU pivot col", COL.data:COL.size);
          ++first_col_ptr;
          ++map_ptr;                                                20
        }
      }
    }
    /* Last worker cleans up the mults.
       — note last pivot column produces no mults. */
    if (*--map_ptr == nm1) {
      for (i = 0; i < nm1; ++i)
        in("LU multipliers", i, ? ji, ? double *:);
    }
  }                                                                 30
```

Appendix C

Benchmarks

This appendix presents the complete data from five of the applications presented more briefly in Chapter 6. Each application was run using at least two different, application specific parameters. Each parameter combination was run using a variety of optimization combinations, shown in the following list:

0. All optimizations off.
1. Memory caching only.
2. In/out collapse only.
3. Local data caching only.
4. Random queue only.
5. Tuple broadcast only.
6. Rehashing only.
7. All optimizations.
8. All optimizations except local data caching.

Two different timings methods were used. Total time refers to total elapsed wallclock time from the time the user's code begins executing. Net time is also wallclock time, but excludes the time spent initializing the problem, in one case (matclump, see page 119) also excluding the time to out some initial tuples.

The graphs are very similar to those in Chapter 6. Each graph presents a runtime curve for each optimization level, although in many cases only two distinct curves are visible. Only two of the speedup curves appear, indicating fully optimized and unoptimized. The (○) and (□) label the runtime and speedup curves of the fully optimized and unoptimized cases, respectively. For each optimization, the key indicates which of these two curves it was closest to at the right (maximum processors) end. Speedups are either relative to the sequential program or to the one optimized processor/worker Linda program, as indicated for each figure.

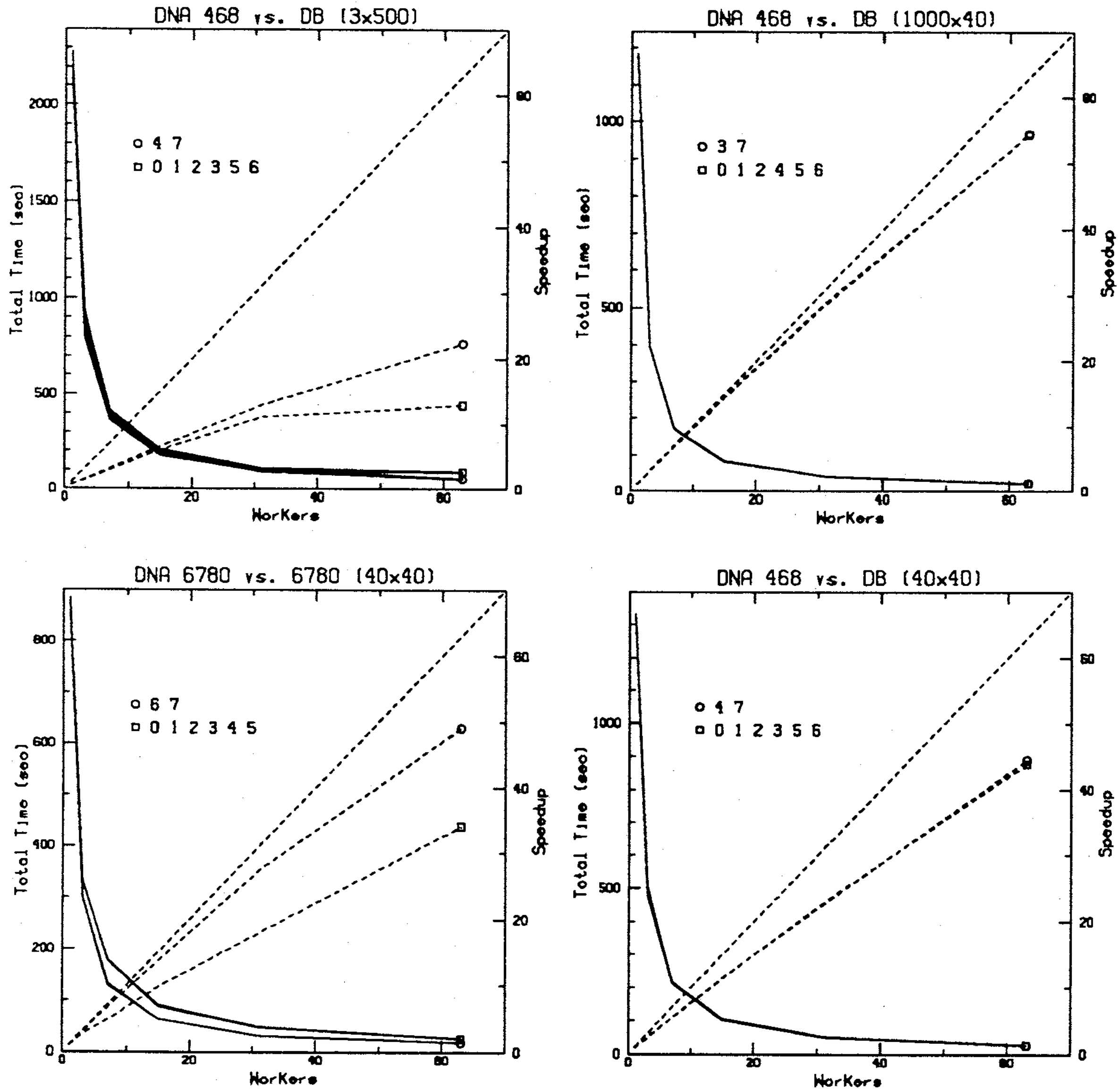

Figure C.1: Hybrid DNA Database Search. 730 sequences, totaling 163000 bases, compared against 468 base test sequence. Blocking sizes shown in parentheses. Total time of computation, including setup. Speedup is relative to the one worker Linda program. The label numbers refer to different optimization levels (see page 205). For discussion of the program, see Section 6.3.

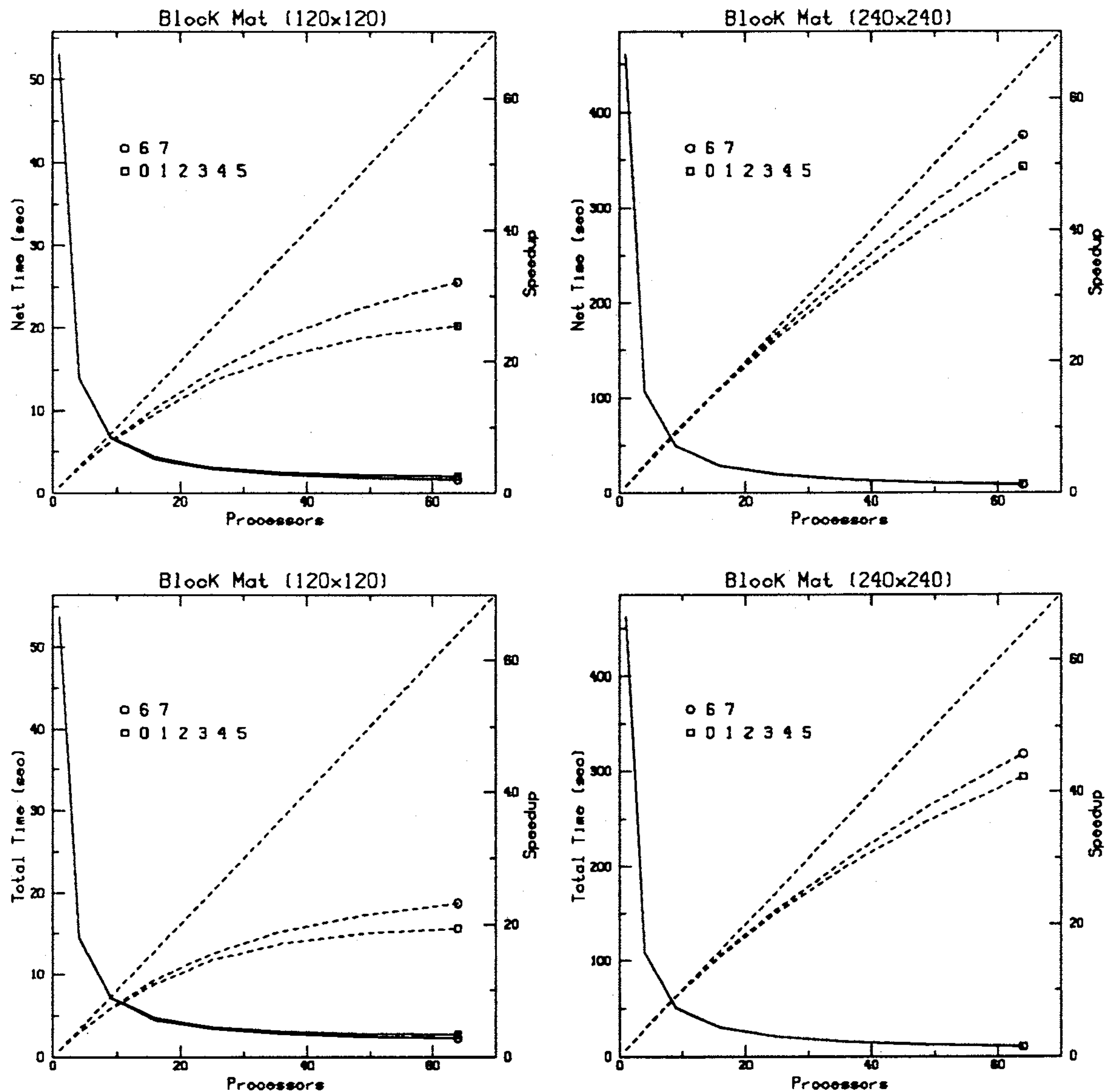

Figure C.2: Block Matrix Multiply. 10 repetitions. Total time (including setup) and net time (excluding setup) of computation. Speedup is relative to the sequential program. The label numbers refer to different optimization levels (see page 205). For discussion of the program, see Section 6.4.

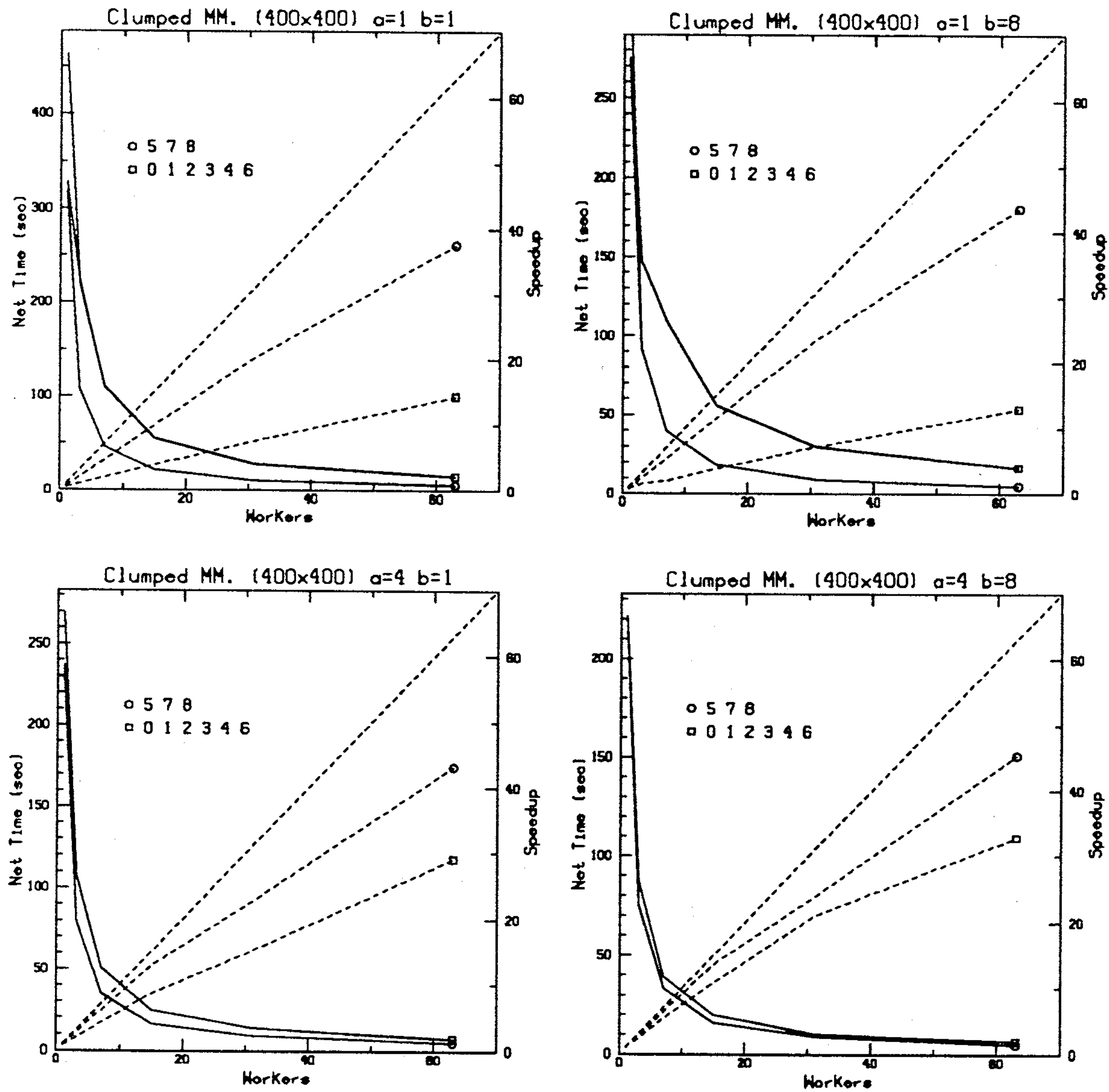

Figure C.3: Clumped Matrix Multiply (400x400). Net time of computation, excluding setup. a and b refer to the matrix clumping factors. Speedup is relative to the sequential program. The label numbers refer to different optimization levels (see page 205). For discussion of the program, see Section 6.5.

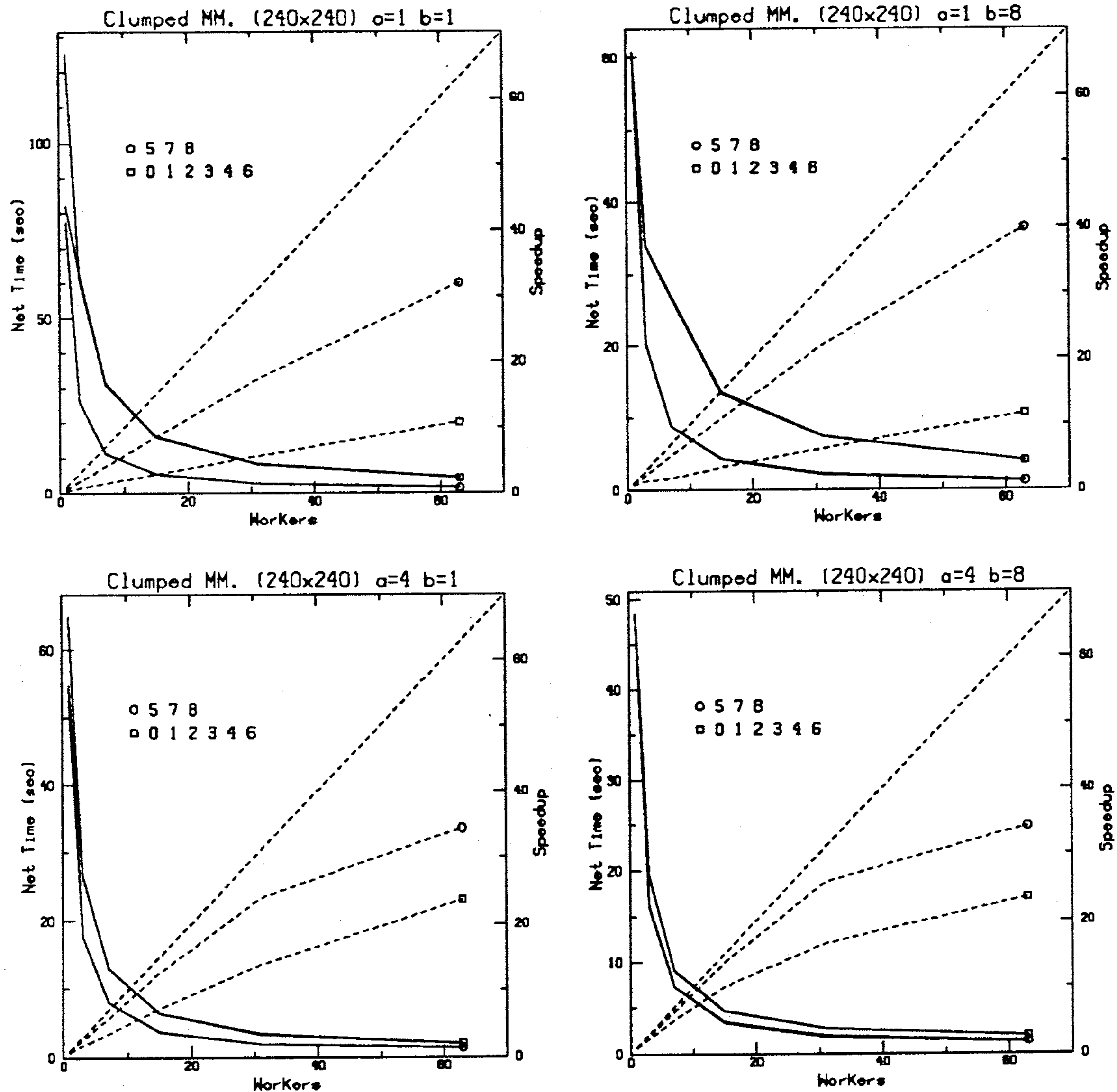

Figure C.4: Clumped Matrix Multiply (240x240). Net time of computation, excluding setup. a and b refer to the matrix clumping factors. Speedup is relative to the sequential program. The label numbers refer to different optimization levels (see page 205). For discussion of the program, see Section 6.5.

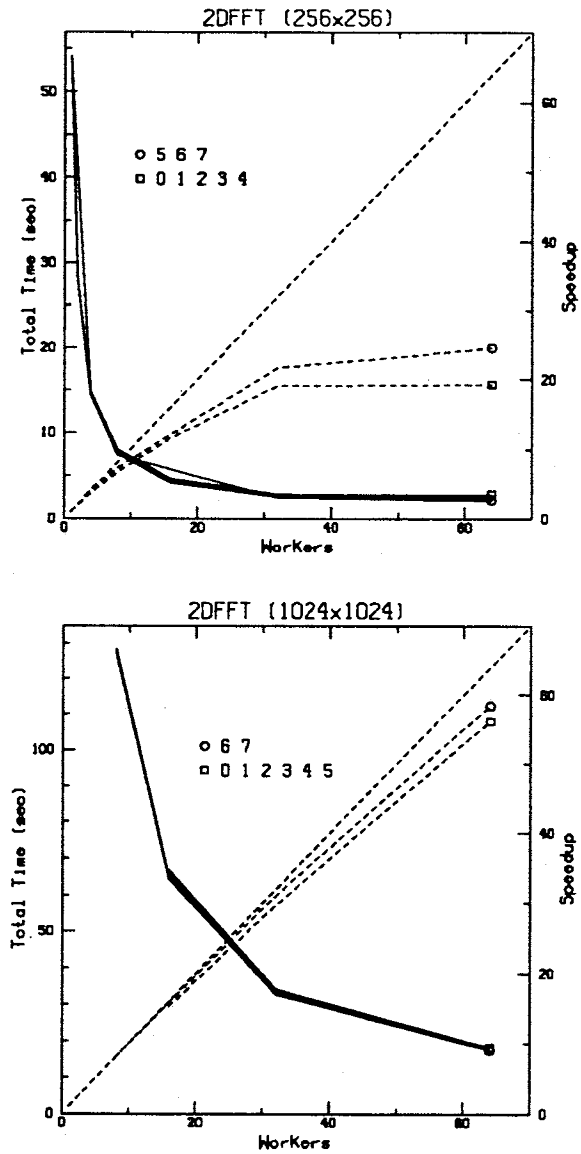

Figure C.5: 2D-FFT. Total time of computation, including setup, for two problem sizes. The 1024x1024 problem would not run on less than 8 nodes due to memory limitations, and for this case, speedup is relative to the eight processor case. For the 256x256 case, speedup is relative to the the 1 processor Linda program. The label numbers refer to different optimization levels (see page 205). For discussion of the program, see Section 6.6.

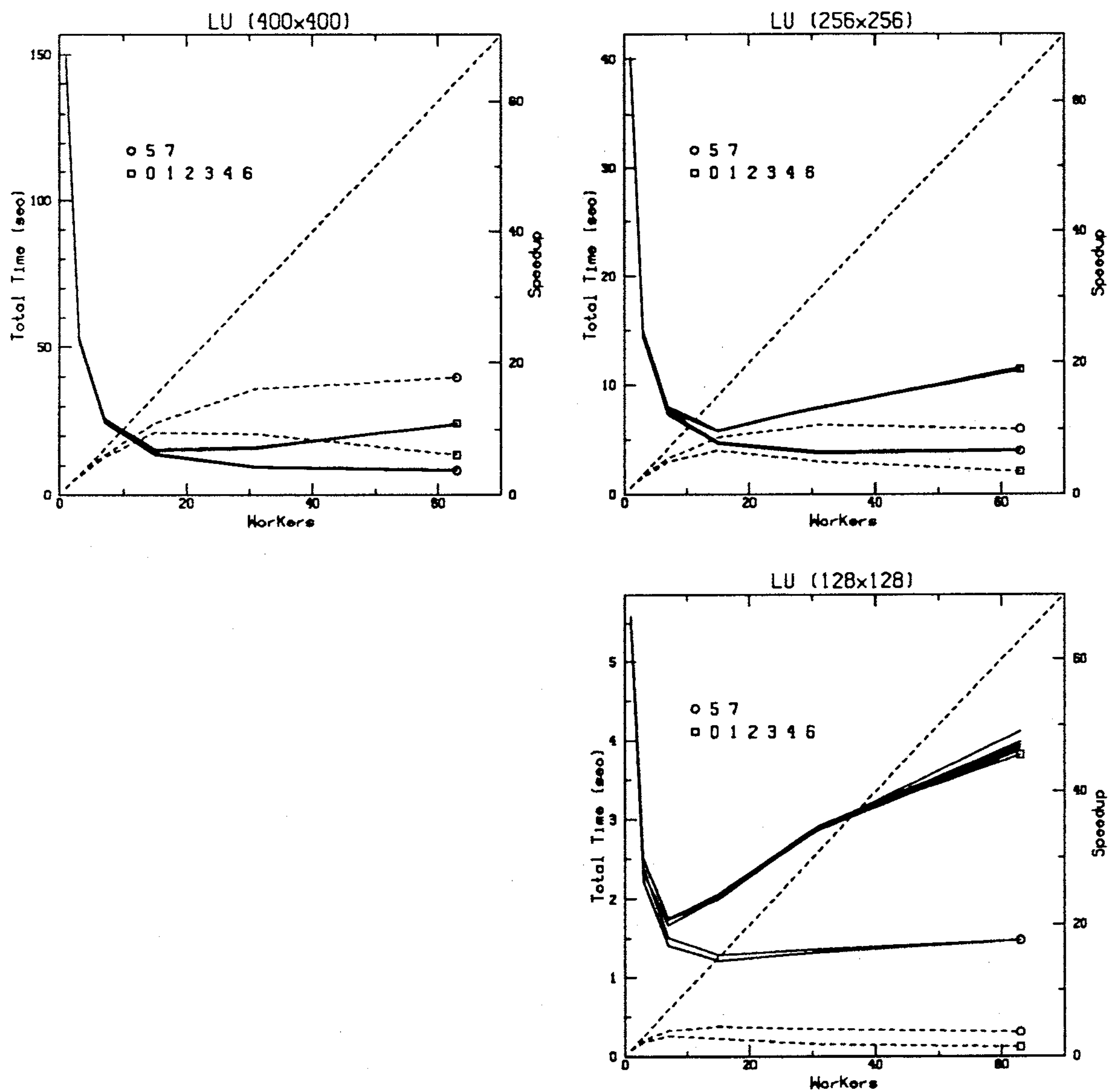

Figure C.6: LU decomposition. Total time of computation, including setup. Speedup is relative to the one worker Linda program. The label numbers refer to different optimization levels (see page 205). For discussion of the program, see Section 6.7.

Appendix D

Implementation of Eval

The Linda operation **eval** presents special problems on distributed memory multiprocessors, including the iPSC/2[1]. Recall that **eval** is much like **out** except that the tuple is added unevaluated to Tuple Space, where independent threads of control evaluate each element of the tuple in parallel. Once the tuple has been completely evaluated, it is indistinguishable from a tuple created with **out**. Often, the tuple created is of secondary importance; usually, at least one of the elements in an **eval** is a procedure call.[2] **Eval** is the canonical Linda method of creating new processes. **Eval** provides a portable method of spawning processes much in the same way that **in**, **out**, and **rd** provide a portable method of communication.

An important aspect of the semantics of **eval** concerns the scope or context of the new process. A new process created by an **eval** begins with a clean slate, with only the parameters passed to it and the values of any of its variables that were initialized by the compiler[3]. In particular, unbound variables in any procedures invoked in an **eval** will not inherit values from the parent process. This is quite different from the semantics given procedures created by **fork()** in Unix, for example, where the entire state of the parent process is inherited by the child. Although we have used fork to implement **eval** on shared memory machines, the method described below does not depend on any particular system call, and more interestingly, is built entirely on **in** and **out**. This is particularly attractive, since it means that implementing **eval** on *any* machine is only as difficult as implementing the other three operations.

In essence, implementing **eval** consists of a series of transformations of **eval** into **ins** and **outs**. Imagine the following operation, written in pseudo-C:

[1] The work presented in this chapter was done jointly with Nicholas Carriero.

[2] Most implementations only evaluate function calls in parallel. All other **eval** elements are evaluated inline, which is much more efficient while still semantically correct.

[3] Using **DATA** statements in Fortran, or initializers in C, for instance.

```
int i, foo(), bar()
float f
eval("test", i, foo(i), bar(i,f))
```

First we create one tuple which will be used to collect all of the elements that will eventually form the tuple produced by the **eval**. We go ahead and fill in non-function call elements with their values, but leave a place for values coming from function calls.

```
values[0] = i
count = 2
out("eval_values", t_id, e_id, values, count)
```

`t_id` is a unique value describing this particular instance of the **eval**. `E_id` designates the textual **eval**. (`E_id` is the same each time this **eval** is invoked, whereas `t_id` is unique for every execution of an **eval**). **Values** is an array of union fields, each corresponding to one field of the **eval**. `Count` is the number of fields remaining to be computed. (Note that the constant `"test"` will have been removed by the compiler, since it is not needed at runtime.) Then, for each function call, we `out` a tuple that describes that task:

```
out("eval_task", t_id, pos, f_id, args)
```

`E_id` and `t_id` are the same **eval** identifiers described above. `Pos` describes the task's position within the **eval**. `F_id` identifies the function to call, and **args** is an array holding the arguments to the function.

Meanwhile, a number of processes are serving as *eval servers*. They are executing the code found in Figure D.1, which is custom-built for each program by the Linda compiler. Each function call that appears in an **eval** is listed, along with the appropriate number of arguments.

Although it may be preferable on certain machines to use idiomatic methods to support **eval** such as **fork()** on shared memory, the method outlined above is simple and general, at the very least allowing Linda implementors to get **eval** working quickly on a new machine.

Readers interested in the fine details will have several questions about the method. We answer some of these questions below:

What about the types? C's unions are used to allow for varied types in a number of the identifiers used above, including values, args, and val. The present system only allows for C's standard scalar types. Thus, all of the identifiers found in an **eval** must be either double, float, char, int, long, or short. The problem with larger types is that we need to be able to `in` them, without knowing how large they are. One solution to this problem is to extend the functionality of formals, by allowing null valued formal pointers:

```
loop forever {
  in("eval_task", ? t_id, ? pos, ? f_id, ? args)
  switch (f_id) {

    ...

    case foo: {
      val = foo(args[0])
      in("eval_values", t_id, ? e_id, ? values, ? count)
      count = count-1
      values[pos] = val
      out("eval_values", t_id, e_id, values, count)
    }

    case bar: {
      val = bar(args[0], args[1])
      in("eval_values", t_id, ? e_id, ? values, ? count)
      count = count-1
      values[pos] = val
      out("eval_values", t_id, e_id, values, count)
    }

    ...

  } /* switch */

  if (count == 0) /* eval finished */

  switch (e_id) {
    ...
    eval_1: {
      in("eval values", t_id, ? values, ? count)
      /* this out corresponds to the original eval */
      out("test", values[0], values[1], values[2])
    }
  } /* switch */
  ...
```

Figure D.1: Eval Server

```
char *ptr=0
in(?ptr:len)
```

The Linda kernel would allocate the appropriate storage for the arriving message. The user would be responsible for freeing the space.

How many evals can compute simultaneously? Each eval server can latch onto only one eval task at a time, since allowing one process to multiplex tasks would allow tasks to effect one another's state. However, so long as we can create new eval servers, and assuming the presence of a preemptive scheduler, we can handle any number of eval tasks at the same time. Unfortunately, the Intel iPSC/2's operating system discourages creating many processes per node[4] so the iPSC/2 implementation restricts the `eval` servers to one per node.

How is the unique task identifier generated? We use Linda! During system initialization, a task id tuple is produced with the value zero. Whenever an eval needs a unique value, it ins and outs the tuple, incrementing the value, and using the old value. If this tuple ever becomes a bottleneck, we can allow a process to reserve a number of unique values with each access to the task id tuple.

Where do the eval servers come from? It depends on the machine architecture. On the iPSC/2, we create them at the beginning, one per node. Although the present system recycles eval servers, a more correct solution is to clone a new server from a virgin copy each time one is needed. Carriero [Car91] has built shared memory Linda systems that use cloned servers. Although the overhead is higher, this guarantees that the `eval` sees the correct initial state. On machines that support interprocessor process creation, we can be more parsimonious, creating eval servers as we need them.

[4]Its successor, the i860, forbids multiple processes all together.

Appendix E

Degenerate hybrid sets

As previously discussed, tuple sets that are classified as hybrid may need to be exhaustively searched when confronted with a template without a key. In many cases, though, the hybrid set degenerates into a queue. We can recognize this characteristic during compilation and put the knowledge to good advantage at runtime.

For example, consider the following three Linda statements, which form a classic hybrid set:

```
out("test", i)
in("test", i)
in("test", ?i)
```

The first `in` is very efficient, since we have a key and can go directly to the correct bucket in the private hash table. The second is more problematic, since it has no key. The current strategy will search each bucket for a match until one is found. Notice, though, that the first tuple found will always suffice, since there are no other elements to be matched, a property easily discovered by the compiler. In such cases, in addition to entering it in a particular bucket, we can link together all of the tuples in the set into a single queue. When encountering the second sort of `in`, we can use this list to immediately find a matching tuple.

To clarify which sets can benefit from this strategy, consider the following hybrid set:

```
out("test", i, j)
in("test", i, ?j)
in("test", ?i, j)
```

This set does not have the desired property, since whichever of the two fields is chosen for the partial key, the other will need to be matched. This set might benefit from multi-key hashing; see [Seg91].

Appendix F

Performance Data for LU and 2DFFT

This appendix presents the raw data used in Figures 6.18 and 6.23.

Problem	Nodes	Native	Linda	Ratio
64x64	16	.112	.156	.71814
128x128	16	.394	.420	.938
256x256	16	1.524	1.56	.977
512x512	16	6.378	6.46	.987
1024x1024	16	27.13	27.26	.995
64x64	32	.088	.180	.489
128x128	32	.260	.328	.793
256x256	32	.840	.908	.925
512x512	32	3.26	3.364	.969
1024x1024	32	13.63	13.79	.988
64x64	64	.113	.476	.237
128x128	64	.181	.366	.495
256x256	64	.552	.684	.807
512x512	64	1.77	1.923	.920
1024x1024	64	6.94	7.164	.969

Table F.1: Raw Data for 2DFFT from Figure 6.18. Single precision complex. Times are in seconds.

Problem	Nodes	Native	Linda	Ratio
128x128	16	.923	1.310	.700
256x256	16	4.266	4.884	.873
512x512	16	27.94	29.96	.933
768x768	16	90.03	95.368	.944
1024x1024	16	207.5	205.007	1.01
128x128	32	.818	1.297	.631
256x256	32	2.917	3.691	.790
512x512	32	15.55	17.33	.897
768x768	32	47.57	49.75	.956
1024x1024	32	108.3	110.62	.979
128x128	64	.695	1.388	.501
256x256	64	2.048	3.727	.550
512x512	64	9.130	12.285	.743
768x768	64	25.56	29.83	.857
1024x1024	64	56.95	62.262	.915

Table F.2: Raw Data for LU Decomposition from Figure 6.23. Double precision. Times are in seconds.

Appendix G

Glossary

Actual A tuple field with a value.

Address The node, and bucket within that node, in which a particular pair is stored.

Anonymous A formal field that throws away the matching value. Anonymous fields are designated by using a type name instead of a variable.

Classification The category assigned to a tuple set by the Linda compiler. This determines the storage paradigm that is used.

Coordination Language A coordination language is concerned solely with coordination between processes

Critical Path These are actions that contribute to the total execution time.

Element One field of a tuple.

Formal A tuple field without a value; a place holder that may receive a value during matching. Formal fields are designated by "?".

Key A distinguish tuple field that can be used to restrict the search space, similar to a key in a data base manager.

In-set The set of nodes to which the template is set when an `in` or `rd` is performed.

Long element An element larger that simple scalar types. In C-Linda, either a structure or an array.

Match The act of comparing a tuple and a template.

Out-set The set of nodes to which the tuple is set when an `out` or `eval` is performed.

Pair A particular set/key combination. This is the unit on which hashing is performed.

Partition See set.

Rendezvous node The node on which a given pair is stored is that pair's rendezvous node. Tuples and templates of that pair will rendezvous at that node.

Set A partition of the Linda operations in a program, such that an operation in a particular set can only interact with other operations in the same set.

Template A pattern produced by `in` and `rd`, and compared against tuples. It has the same structure as a tuple.

Tuple The fundamental Linda object. A tuple consists of a collection of ordered, typed fields.

Tuple Space Abstract shared memory where tuples reside.

Bibliography

[AB89] M. Arango and D. J. Berndt. TSNet: A Linda implementation for networks of unix-based computers. Research Report YALEU/DCS/RR-739, Department of Computer Science, Yale University, August 1989.

[Ahu83] S. Ahuja. S/Net: A high-speed interconnect for multiple computers. *IEEE Selected Areas in Communication*, pages 751–756, Noc 1983.

[Bai88] W. L. Bain. A global object name space for the Intel hypercube. In *The Third Conference on Hypercube Concurrent Computers and Applications*, pages 570–574. ACM press, 1988.

[BBLS91] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The nas parallel benchmarks. Research Report RNR-91-01 Revision 2, Numerical Aerodynamic Simulation (NAS) Systems Division, NASA Ames Research Center, August 1991.

[BCGL88] R. D. Bjornson, N. J. Carriero, D. H. Gelernter, and J. S. Leichter. Linda, the portable parallel. Research Report YALEU/DCS/RR-520, Department of Computer Science, Yale University, January 1988.

[BCZ90] J. K. Bennett, J. B. Carter, and W. Zwaenpoel. Munin: Distributed shared memory based on type specific memory coherence. In *Proceedings of the Second ACM SIGPLAN Symposium on Principals and Practice of Parallel Programming*, volume 25, pages 168–176. SIGPLAN Notices, ACM, Mar 1990.

[BHJ+86] A. Black, N. Hutchinson, E. Jul, , and H. Levy. Object structure in the Emerald system. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications*, volume 21, pages 78–86. SIGPLAN Notices, ACM, Nov 1986.

[Boz84] G. Bozman. The software lookaside buffer reduces search overhead with linked lists. *CACM*, 27:222–227, March 1984.

[Bra88] D. K. Bradley. First and second generation hypercube performance. Research Report 1455, Department of Computer Science, University of Illinois at Urbana-Champaign, September 1988.

[BST89] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[BT88] H. E. Bal and A. S. Tanenbaum. Distributed programming with shared data. In *The IEEE CS 1988 INternational Conference on Computer Languages*, pages 82–91. IEEE press, 1988.

[Car87] N. J. Carriero. *Implementing Tuple Space Machines.* PhD thesis, Yale University, New Haven, Connecticut, 1987. Department of Computer Science.

[Car91] N. J. Carriero. Personal communication, 1991.

[CG81] K. L. Clark and S. Gregory. A relational language for parallel programming. In *Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture*, pages 171–178. ACM, 1981.

[CG89] N. J. Carriero and D. H. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[CG90] N. Carriero and D. Gelernter. *How to Write Parallel Programs, A First Course.* MIT Press, 1990.

[Che86a] M. C. Chen. A parallel language and its compilation to multiprocessor machines. In *Proceedings of the 13th Annual Symposium on Principles of Programming Languages*, 1986.

[Che86b] M. C. Chen. Very-high-level parallel programming in Crystal. Research Report YALEU/DCS/RR-506, Department of Computer Science, Yale University, December 1986.

[Chu41] A. Church. *The Calculi of Lambda-Conversion.* Princeton University Press, Princeton, NJ, 1941.

[CHZ91] B. M. Chapman, H. Herbeck, and H. P. Zima. Automatic support for data distribution. In *Proceedings of the Sixth Distributed Memory Computing Conference*, pages 51–58. IEEE Press, 1991.

[Clo88] P. Close. The iPSC/2 node architecture. In *The Third Conference on Hypercube Concurrent Computers and Applications*, pages 43–50. ACM press, 1988.

[CW79] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18, 1979.

[DNS81] E. Dekel, D. Nassimi, and S. Sahni. Parallel matrix and graph algorithms. *SIAM Journal of Computation*, 10:657–673, 1981.

[Don87] J. J. Dongarra. Performance of various computers using standard linear equations in a Fortran environment. Technical memorandum, Argonne National Laboratory, 1987.

[Fac90] M. Factor. *The Process Trellis Architecture for Parallel, Real-Time Monitors.* PhD thesis, Department of Computer Science, Yale University, New Haven, Connecticut, 1990.

[FF88] A. H. Frey and G. C. Fox. Problems and approaches for a teraflop processor. In *The Third Conference on Hypercube Concurrent Computers and Applications*, pages 21–25. ACM press, 1988.

[Fos88] I. Foster. Parallel implementation of PARLOG. In *Proceedings of the International Conference on Parallel Processing*, pages 9–16. Penn State University Press, 1988.

[Gel85] D. H. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[GHPW90] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. PICL a portable instrumented communication library. Research Report ORNL/TM-11130, Oak Ridge National Laboratory, July 1990.

[Gol88] B. F. Goldberg. *Multiprocessor Execution of Functional Programs.* PhD thesis, Department of Computer Science, Yale University, New Haven, Connecticut, 1988.

[Got82] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162:705–708, 1982.

[Gri90] D. H. Grit. A distributed memory implementation of SISAL. In *The Fifth Distributed Memory Computing Conference*, pages 1131–1136. IEEE Society Press, 1990.

[GS91] G. A. Geist and V. S. Sunderam. The PVM system: Supercomputer level concurrent computation on a heterogeneous network of workstations. In *Proceedings of the Sixth Distributed Memory Computing Conference*, pages 258–261. IEEE Press, 1991.

[Gus92] J. Gustafson. Personal communication, 1992.

[GV73] G. H. Golub and C. F. Van Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, MD, 1973.

[HGD90] M. T. Heath, G. A. Geist, and J. B. Drake. Early experience with the Intel iPSC/860 at Oak Ridge National Laboratory. Research Report ORNL/TM-11655, Oak Ridge National Laboratory, Sept 1990.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *CACM*, 21:667–677, August 1978.

[HS86] P. Hudak and L. Smith. Para-functional programming: A paradigm for programming multiprocessor systems. In *Proceedings of the Twelfth Symposium on Principals of Programming Languages*, pages 243–254. ACM, Jan 1986.

[HS87] S. P. Harbison and G. L. Steele Jr. *C: A Reference Manual.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1987.

[Hud84] P. Hudak. ALFL reference manual and programmer's guide. Research Report YALEU/DCS/RR-322, Department of Computer Science, Yale University, October 1984.

[Int84] Intel Scientific Computers, Englewood Cliffs, NJ. *Occam Programming Manual*, 1984.

[Int89] Intel Scientific Computers, Beaverton, OR. *iPSC/2 Programmer's Reference Manual*, 1989.

[Int90] Intel Scientific Computers, Beaverton, OR. *iPSC/2 and iPSC/860 User's Guide*, 1990.

[Joh87] S. L. Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *Journal of Parallel and Distributed Computing*, 4:133–172, 1987.

[KMVR90] C. Koebel, P. Mehrotra, and J Van Rosendale. Supporting shared data structures on distributed memory architectures. In *Proceedings of the Second ACM SIGPLAN Symposium on Principals and Practice of Parallel Programming*, volume 25, pages 177–186. SIGPLAN Notices, ACM, Mar 1990.

[Knu73] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 1973.

[Kri91] V. Krishnaswamy. *The Linda Machine — A Language Based Architecture for Parallel Computing.* PhD thesis, Department of Computer Science, Yale University, New Haven, Connecticut, 1991. In progress.

[KRS89] L. V. Kalé, B. Ramkumar, and W. Shu. Parallel Prolog on the Intel iPSC/2. In *The Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, pages 529–536. Golden Gate Enterprises, 1989.

[KS88] L. V. Kalé and W. Shu. The Chare-kernel language for parallel programming: A perspective. Research Report UIUCDCS-R-88-1451, Department of Computer Science, University of Illinois at Urbana-Champaign, August 1988.

[KZ89] K. Kennedy and H. P. Zima. Virtual shared memory for distributed memory machines. In *The Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, pages 361–366. Golden Gate Enterprises, 1989.

[Lam86] L. Lamport. *LaTeX: A Document Preparation System.* Addison-Wesley Publishing Company, Reading, Massachussetts, 1986.

[Lei89] J. S. Leichter. *Shared Tuple Memories, Shared Memories, Busses and LAN's – Linda Implementations Across the Spectrum of Connectivity.* PhD thesis, Yale University, New Haven, Connecticut, 1989. Department of Computer Science.

[Li91] J. Li. *Compiling Crystal for Distributed-Memory Machines.* PhD thesis, Department of Computer Science, Yale University, New Haven, Connecticut, 1991.

[Lil90] S. L. Lillevik. Touchstone program overview. In *The Fifth Distributed Memory Computing Conference*, pages 647–657. IEEE Society Press, 1990.

[LRRW89] R. Lee, M. Roth, T. Runge, and A. Witkowski. A Prolog implementation for hypercube multiprocessors. In *The Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, pages 537–540. Golden Gate Enterprises, 1989.

[LS89] K. Li and R. Schaefer. Shared virtual memory for a hypercube multiprocessor. In *The Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, pages 371–378. Golden Gate Enterprises, 1989.

[Luc86] S. Lucco. A heuristic Linda kernel for hypercube multiprocessors. In *Proceedings of SIAM Conference on Hypercube Multiprocessors*, September 1986.

[MKM89] F. K. Musgrave, C. E. Kolb, and R. S. Mace. The synthesis and rendering of eroded fractal terrains. *Computer Graphics*, 23(3):41–50, August 1989.

[nCU90] nCUBE Corp., Beaverton, OR. *nCUBE2 Programmer's Guide, r2.0*, 1990.

[Nel81] B. J. Nelson. Remote procedure call. Research Report CMU/CS/81-119, Department of Computer Science, Carnegie-Mellon University, May 1981.

[Nug] S. F. Nugent. *The iPSC/2 Direct-Connect Communications Technology.* Intel Scientific Computers.

[Nug88] S. F. Nugent. The iPSC/2 direct-connect communications technology. In *The Third Conference on Hypercube Concurrent Computers and Applications*, pages 51–60. ACM press, 1988.

[OA85] R. R. Oldehoeft and S. J. Allan. Adaptive exact-fit storage management. *CACM*, 28:506–511, May 1985.

[Par90] ParaSoft Corporation, Pasadena, CA. *Express C Reference Guide Version 3.0*, 1990.

[PFTV88] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, Cambridge, MA, 1988.

[Pin92] J. Pinakis. The design and implementation of a distributed linda tuple space, 1992.

[PS91] R. Peskin and E. Segall. Linda strategies for scientific computing environments. Research report, Rutgers University Department of Computer Science, May 1991.

[PW86] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *CACM*, 29:1184–1200, Dec 1986.

[Ran89] A. Ranade. *Fluent Parallel Computation.* PhD thesis, Department of Computer Science, Yale University, New Haven, Connecticut, 1989.

[RHH85] Jr. R. H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, October 1985.

[SCMB90] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(2), 1990.

[Seg91] E. Segall. Multiple search keys for Linda. In *Sixth Distributed Memory Computing Conference.* IEEE Computer Society Press, 1991, April 1991.

[Sha87] E. Shapiro, editor. *Concurrent Prolog: Collected Papers.* MIT Press, Cambridge, MA, 1987.

[SS90] S. R. Seidel and T. E. Schmiermund. Refining the communication model for the Intel iPSC/2. In *The Fifth Distributed Memory Computing Conference*, pages 1334–1342. IEEE Society Press, 1990.

[SS91] A. Skjellum and C. H. Still. Zipcode and the reactive kernel for the Caltech Intel Delta prototype and nCUBE/2. In *Proceedings of the Sixth Distributed Memory Computing Conference*, pages 26–33. IEEE Press, 1991.

[Ste91] G. L. Steele Jr. Personal communication, 1991.

[TSS87] S. Taylor, S. Safra, and E. Shapiro. A parallel implementation of Flat Concurrent Prolog. *International Journal of Parallel Programming*, 15(3):245–275, 1987.

[Wan90] N. Wang. Further optimization of the C-Linda compiler: The in-out collapse. Unpublished Master's Project, Department of Computer Science, Yale University, August 1990.

[YT86] Y. Yokote and M Tokoro. The design and implementation of Concurrent Smalltalk. In *Object Oriented Programming Systems, Languages and Applications*, pages 331–340. SIGPLAN Notices, Nov 1986.