Implementation of Tuple Space Machines

Nicholas John Carriero, Jr.

Research Report YALEU/DCS/RR-567
December 1987

Contents

List of Figures

Chapter 1

Introduction

THERE is little question that future computers will necessarily employ not only advances in circuit design but also multiple execution units to increase speed. The Butterfly [BBN85], Connection Machine [Hil85], Multimax [Enc86], NCUBE [NCU86], iPSC [iPS86], Balance [Seq85], Cray-XMP [Cra84], S/Net [Ahu83] and RP3 [PBG*85][1] are some examples of a wide variety of multiprocessor machines. They vary in almost every conceivable way: power of nodes, amount of memory, type of interconnect, shared or disjoint memories, MIMD or SIMD, commercial product or research effort, and so on. But they do share one characteristic. Exploiting their parallelism can be very difficult without a programming environment.

> ... just beyond the speed barrier lies another: the software barrier. ... We do not know how to program the new machines! (emphasis in the original) [Den86]

The foundation of a programming environment for these machines must be a programming language that handles the new elements needed by "parallel" architectures: communications, synchronization, and process creation and management. Numerous proposals have been made for such languages (or fragments): CSP [Hoa78], RPC [BN84], Concurrent Prolog [Sha86], Ada [Uni82], Multilisp [Hal85], PLITS [Fel79], and ParAlfl [Hud86] to name a few. Our work is based on Linda [Gel85,Gel82,CGL86]. Our preference for Linda over the proposals listed (and others) is not the theme of this work, although we will address this point briefly below.

Our primary concern is the next step in the development of a parallel programming language — its implementation. We will report in detail on implementations of Linda's communication model, *tuple space*, for two different architectures covering three different machines. We will present performance measurements that demonstrate real wall-clock speedups of real problems on real machines. These results in turn demonstrate that tuple space (the crucial portion of Linda from the point of view of programming multiprocessors) can be implemented efficiently enough to be a useful tool (a virtual tuple-space machine) for parallel programming of multiprocessor machines.[2]

We wish to make clear from the start that this work is not a full implementation of Linda. Linda's tuple space can be fairly easily dissected from the rest of the language proposal and transplanted into a variety of "host" languages. We are concerned with implementing tuple spaces that are supported by a particular host language (C) on two classes of hardware. Our results demonstrate that the resulting *Linda-C* is a useful programming language.

[1]Some of these are trademarks.

[2]The *Thesis*.

Note that the process of dissecting and transplanting is in itself interesting. As we will show, in rough terms "lindafying" a language requires a machine independent preprocessor and a machine dependent backend. A high level description of the various interfaces between the preprocessor, a host language, and a host machine would be the first step in automating the "lindafying" process.

After discussing tuple space in depth in Chapter 2, we will describe in Chapter 3 the compile-time system shared by the Encore and Sequent implementations. Next comes the run-time kernels which are machine specific. Chapter 4 will describe the first kernel — the S/Net's. Chapter 5 will present the run-time kernel for the Encore Multimax and the Sequent Balance.[3]

1.1 Why Linda?

Tuple space has a number of attributes that make it attractive for parallel programming. It can be thought of as a global, shared memory that is associative and that supports read, remove, and insert operations. These operations can be synchronous or asynchronous. Much more will be said about them when tuple space is discussed in detail in Chapter 2.

What are the benefits of these attributes? Briefly, because tuple space is a global, shared memory, the programmer is freed from topological considerations. Because tuple space is an associative memory, the programmer can use anonymous communication methods, i.e. no addresses are needed. Because tuple space has remove and insert rather than update in-place, it is easy to update atomically structures built in tuple space. Because some of the operations are synchronous, tuple space can be used to coordinate processes. Because there exists an operation that integrates process creation with tuple generation, spawning new processes is simple. Thus, tuple space neatly handles the new elements introduced by parallelism and it needs only six operations to do so.

The real question is not "Why Linda?", but "Can it be efficiently implemented?" The answer is yes.

[3]Actually the kernels for the Encore and the Sequent are not identical. For the most part the differences have been confined to one module or hidden inside of macros.

Chapter 2

Tuple Space

THE most important and perhaps the most distinguishing characteristic of the Linda proposal is its notion of tuple space (or what Gelernter calls *generative communication*). Linda's elegance is derived from the extreme simplicity of this model. This elegance in turn leads to a reduction of the (parallel) programmer's burden. This chapter presents the concept of tuple space in detail. We discuss the process of tuple matching and enumerate properties of tuple space that follow from its definition. Certain idiomatic uses which seem to occur frequently will be presented as well as some troublesome limitations that will be addressed by future work.

2.1 Tuples

For our purposes, it will suffice to think of a tuple as an ordered collection of typed data or place holders (a.k.a. "formals"). A "field" is one element of the tuple. For a given language and implementation there may be limits on the number and types of the fields. For example, our Linda-C/shared-memory system allows tuples of up to 16 fields with C types `long`, `char *` (null terminated string), `float`, `double`, and `LINDA_BLOCK` (a Linda defined structure that describes a contiguous block of data). For a language such as Lisp, atoms and s-expressions might be appropriate, while an object-oriented language might support an even more varied collection of types (or might introduce tuples as a new class of object or tuple space operations as new operations on existing classes).

There are six basic operations on tuple space:

eval
in
inp
out
rd
rdp

The operations are presented in three groups: those that generate tuples, those that synchronously access tuples and those that asynchronously access tuples.

2.1.1 Generative Operations (out and eval)

`out` and `eval` generate tuples: when an `out` or `eval` completes, a tuple has been created.

This stands in strong opposition to many communication models (CSP, Ada, Concurrent Prolog, etc.) where the interprocess message is exceedingly ethereal.

`eval` differs from `out` in the way its arguments are evaluated. The arguments for an `out` are processed in much the same way as the arguments for any subroutine: the `out` process's thread of control will execute code to evaluate each of the arguments. When all have been evaluated, the `out` is performed. On the other hand, the arguments to an `eval` are evaluated by a newly spawned process. For example:

```
out(foo(x))
```

evaluates `foo(x)`, then puts the result in a tuple in tuple space, while

```
eval(foo(x))
```

informs the system that a tuple needs to be created to hold the value of `foo(x)` and that a process needs to be created to evaluate `foo(x)`. The new process will eventually place the value of `foo(x)` into the tuple.

Thus `eval` acts as a forking operation. There are significant unanswered questions about `eval`. Some of the most important concern variable bindings in `foo()`. Can `foo()` have unbound variables? If it can, when will these variables be bound? At the time `eval` executes? At the time `foo()` executes? If `foo()` cannot have unbound variables, how do we enforce this? Other important questions arise over issues of dynamic process creation and management. Some of these problems are machine independent (binding issues) and others are machine dependent (process management). In the present work, `eval` is largely ignored. The compiler does include some support, but no run-time system exists for `eval`. Process forking is done using slight variations of the native operating-system fork mechanism.

2.1.2 Synchronous Extracting Operations and Matching (in and rd)

`in` and `rd` extract data from tuples placed in tuple space by an `out` or `eval`. The particular tuple from which data will be extracted is determined by tuple space's matching rules. If we refer to an `in` or `rd`'s arguments collectively as a template, then the rules for a match are:

1. A tuple and a template must have the same number of fields.
2. Corresponding fields of the tuple and template must be type consonant.
3. Corresponding data items must be equal.
4. There must be no corresponding formals.

For example:[1]

`in("foo")` will not match the tuple `("foo", "foo")` because of 1).

`in(1)` will not match the tuple `(1.0)` because of 2).

`in("foo")` will not match the tuple `("bar")` because of 3).

[1]Here and throughout assume that variables `i,j,k` have type `int`; `a,b,c` have type `float`; `r,s,t` have type `char *`; and `x,y,z` have type `LINDA_BLOCK`.
We use the notation "?" to designate a formal field (the use of formals is analogous to the use of arguments with C's `scanf()`, hence the `&` operator in these examples).

`in(? &i)` will not match `(? &j)` because of 4).

`in("foo", "foo")` will match `("foo", "foo")`.

`in(1)` will match `(1)`.

`in("data", ? &i)` will match `("data", 5)`.

When an in or rd executes, if no tuples in tuple space match, then the in or rd will block until an out or eval places a matching tuple in tuple space. If more than one tuple in tuple space can match, the first found will be the one that does match. The ordering implied by the term "first" is implementation dependent. A user should make no assumptions about the search order of tuples in tuple space. When an in finds a match, the matched tuple is removed from tuple space. When a rd finds a match, the matched tuple remains in tuple space.

Once the matching process has completed, data copying and tuple deletion may occur. If a template has a formal field that corresponds to a data field in the tuple (which the formal field of a template *must* do, see (4) above), and if the formal field has associated with it an address, then the data in the tuple is copied to that address. This process is analogous to parameter passing (by value) in subroutine calls. To illustrate:

`out("data", 3)` has been executed.

`in("data", ? &j)` will result in j = 3.

`in("data", ? int *)` will return, but no data will be copied.

"`? int *`" in the last example means that we do not care about the value of that field. Allowing a "don't care" value makes it easier to clean up tuples. It also completes the symmetry of in and out. We do not care about the value of a variable associated with a formal field in an out, so we may use this construction to avoid declaring such a variable. For example, a process might direct a request for service to a particular print server using

```
out("print request", "imagen", file_name);
```

or to any server with

```
out("print request", ? char *, file_name);
```

Without the "don't care" in the second field, we would have to create a dummy variable to provide type infomation:

```
char *foo;

out("print request", ? foo, file_name);
```

Upon execution, the statement

```
in("foo", ? &j, ? &j, ? &j)
```

will assign in arbitrary order the appropriate data fields of the matching tuple to j. Whatever was copied last will be the value of j upon completion of the in.

If a collection of ins and rds are waiting for the same tuple, then when that tuple is generated some arbitrary subset (possibly empty) of the rds and one in will be satisfied. No guarantees of fairness are made.

Figure 2.1 depicts the effects of these various operations on tuple space.

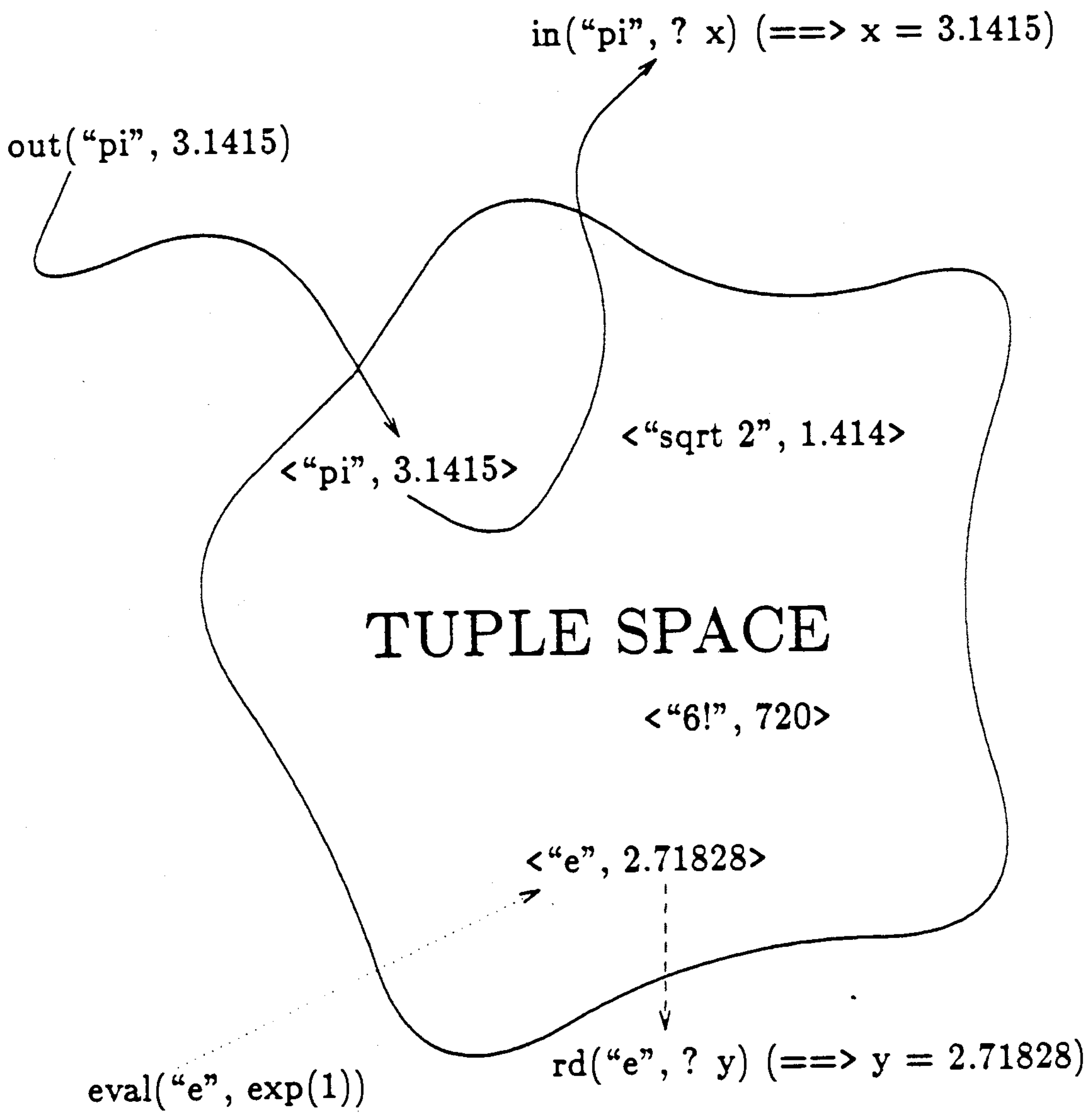

Figure 2.1: A snapshot of tuple space.

2.1.3 Asynchronous Extracting Operations (inp and rdp)

Blocking when no suitable tuple is present is not necessary. The operations inp and rdp are identical to in and rd except for their blocking behavior. inp and rdp never block; instead they return a value indicating success or failure.

2.2 Notes on the Fine Structure of Tuple Space

In the parlance of TEXthis and the next two sections constitute a "dangerous bend". These sections may be omitted by those reader's interested only in the details of the current implementation.

Clearly inp and rdp are more primitive than in and rd in that in(...) can be macro expanded to while(!inp(...)) and rd(...) can be expanded to while(!(rdp(...)).

There are a few somewhat subtle issues pertaining to the p operations. The most important of these is what exactly the failure status of an inp or rdp means. Intuitively a failure means that all of tuple space was searched at least once and that the sweep found no matching tuples. What is troublesome is that while the sweep is finishing up, a matching tuple could be sneaked into an already checked cubbyhole.

Is this a problem? If it is, should we lock tuple space during a sweep to prevent this? This would raise some very tricky implementation questions (remember our abstract semantics must somehow be implemented efficiently on a variety of architectures).

More precisely, intuition suggests that given two processes A and B:

```
A:
.
<arbitrary code, except that
 no "status" tuple is generated>
.
status = inp("foo");
out("status", status);
<no further tuple space operations>


B:
in("status", ? &ok);
if (!status)
  if (inp("foo")) printf("oops.");
```

and a correct implementation of inp (or rdp), then there is no execution sequence of just these two processes (regardless of whether A and B run on the same or different processors) that reaches the printf in B. The question remains whether the constraints implied by this are too weak, too strong, or just right.

Having noticed that in and rd can be built out of inp and rdp, we might be tempted to build rdp in terms of inp and out:

```
rdp:  if (inp) {
           out;
           return 1;
      }
      else {
            return 0;
        }
```

But this does not quite work. We want multiple executions of the same rdp, assuming that a satisfying tuple exists, to succeed no matter what the interleaving of their execution is. We cannot guarantee this with the code above.

2.3 Is 6 Enough? Too Many?

Given that in and rd can be written in terms of inp and rdp, why do in and rd still exist? On the other hand, arguments can be made in favor of other, more complicated operations (such as those necessary to manage an ordered queue of tuples). What is an appropriate set of basic operations?

Clearly the answer is in part a matter of judgement. However, there are some hidden issues concerning posterity and efficiency. Aside from aesthetic considerations, we must also consider the magnitude of the labor involved in putting next generation Linda systems on new architectures. The more we specify now, the more will have to be done in potentially hostile environments later. Balanced against this are efficiency concerns. It seems clear that in certain cases, in versus inp for example, efficiency penalties will be paid if we stubbornly stick to the primitives. in will allow the system to internalize the details of process blocking, while the inp formulation will not. This difference has important repercussions for process scheduling.

The path we have decided to take is to chose a set (in, inp, rd, rdp, out, and eval) we consider expressive enough. Any new operations which can be entirely expressed in terms of the operations in this set are then candidates for *kernel compilation*; i.e., reduction to low level systems activities that faithfully honor the Linda semantics, but that do not necessarily use the Linda operations.

An example of this is the increment operator written by Bjornson [Bjo] for the Encore system:

```
    in(foo, ? &bar);
    out(foo, bar + 1);
```

is replaced with

```
    incr(foo, ? &bar);
```

An in followed by an out would almost certainly require destroying and then rebuilding data structures that could have been left in place. While we have not seriously investigated the possibility, we think it is plausible to automate this process (via a so-called kernel compiler).

With the notion of kernel compiling in mind then, it falls to the Linda system builder on a new machine to:

1. Implement the basic set of six operations.
2. Write special code (or kernel compile) any additional operations that are desirable and that can benefit from special treatment on the system in question.
3. Provide a library written in Linda to implement any special operations wanted but not taken care of in 2).

What is novel about this? Certainly C and its libraries are very similar. The point here is that tuple space operations are amenable to this kind of process while other languages arguably are not. Because the Linda operations can be easily understood as operations on memory (like many of the operations in a programming language) one can more easily program with them than the queue-based interprocess communication schemes of CSP or Ada. We will see examples of this below.

2.4 Properties of Tuple Space

One might ask how tuple space differs from asynchronous buffered communication proposals (e.g. the iPSC system [iPS86]). While there is a superficial resemblance between a tuple and a buffered message, probing a little deeper reveals significant differences.

Consider first the issue of lifetime of the tuple (or message). In most buffered systems buffering is little more than life-support equipment meant to keep the message alive until it can be delivered. In tuple space, tuples can be consulted (via `rd`'s) or mature (if inserted via `eval`).[2]

Intent distinguishes the two even more clearly. Linda's generative communication model is intended to provide a shared, associative memory, not merely a messenger service. Just as communication degenerates to memory accesses with real shared-memory architectures, so too in tuple space communication becomes a matter of tuple space accesses (reading, removing, or inserting). Buffered communication systems have no aspirations to be general purpose memory, and provide no simple, flexible access to all buffered messages. This difference is quite noticeable in queue based systems when one attempts to solve a problem that requires even a slightly different buffer organization than the one supplied (see [Bar80]).

Its shared, associative memory semantics is the single most important property of tuple space. Clearly its content addressibility sets tuple space apart from most shared memory architectures. However, another property helps to distinguish tuple space from physically shared memory. Physically shared memory has a data granularity imposed by hardware. On the other hand, tuple space has the tuple as its unit of granularity, which means the granularity is determined by the user. The operations on tuple space act on whole collections of bytes that are semantically meaningful to the user. This provides a level of convenience beyond unadorned shared memory roughly comparable to the convenience introduced to a language by the addition of Pascal-like structures.[3]

We saw above that the shared memory provides a communication medium. The atomic nature of `in`, `inp`, `rd`, `rdp`, `out`, and `eval` form the basis for tuple space's synchronization properties. Communication and synchronization are *sine qua non* for parallel or distributed computing. Tuple space has an additional property that brings a degree of flexibility to communication. Communication in tuple space is anonymous. Once again, in analogy with a physical shared memory, when a process reads a "3" from memory location 113, it has no knowledge of what process stored the "3" in location 113. Nor did the process that wrote the "3" need to know which processes were going to read it. This is usually fine, as it does not matter. When it does matter, arrangements can be made to allow for that information as well. Similarly with tuple space. In contrast to most interprocess communication systems,

[2]Tuples can even become immortal — one could build long term, disk-backed tuple spaces that would subsume the role of files in a Linda-based operating system.

[3]This has its advantages for the implementor too. The "cost/byte" of operating on a shared memory is lower if the average operations affects a larger number of bytes than 4.

tuple space does not require the sender to know the receiver, or vice versa, merely that the two agree on the format of the data (analogous to agreeing on location 113). This anonymity leads to a decoupling of the execution threads of the processes, which in turn provides an extra degree of freedom for developing algorithms.

2.5 Some Idiomatic Uses of Tuple Space

In an effort to illustrate these concepts, and to prepare a context for discussion and examples to come, we present here some examples of tuple space usage. They have been used frequently enough to be considered idiomatic.

One of the simplest and yet most useful models of parallelism is the *master/worker* paradigm. In its simplest form, a master generates a number of independent tasks that can be carried out by any one of a number of workers (similar schemes for task management date as least as far back as Pluribus [KEM*78]). As an example, consider an application of this model to matrix multiplication. Each inner-product is an independent computation. The master may therefore generate a task for each inner product. A worker takes one of the tasks, computes the indicated inner product, updates the product matrix, and then loops; the process continues until all tasks are complete.

In general, any loops whose successive iterations are independent of one another is a candidate for this technique. In Linda-C a skeleton of a program implementing this is:

```
master() {
    for all tasks {
        /* Build task_structure
           representing this iteration. */
        .
        .
        .
        out("task", task_structure);
    }
    for all tasks {
        in("result", ? &task_id, ? &result_structure);
        /* Update total result using
           this result and task_id. */
        .
        .
        .
    }
}

worker() {
    while (inp("task", ? &task_structure)( {
        /* Execute task. */
        .
        .
        .
        out("result", task_id, result_structure);
    }
    }
```

There are numerous variations on this theme. One can vary the degree of task granularity. Workers can be allowed to generate new tasks. The requirement for independence of tasks can be waived by adding synchronization during task execution.

As the skeleton indicates, this model is easily supported by tuple space. Tuple space's support of anonymous communication makes the dynamics of such a program quite interesting. Because of decoupling, there is no logical difference between running with 1 or n workers: the algorithm need not change if the user wants to change the number of workers executing the program.[4] An additional advantage of this decoupling is automatic load balancing. Workers that execute faster or receive simpler tasks will inp tasks more often, while those that run slower or are given more difficult tasks will request fewer total tasks. As a result all workers will tend to be kept busy.

The previous example depended on an unordered collection of tuples (the *task bag*), but it is often the case that some ordering is desired. A queue can be used to impose such an ordering and is easy to provide:

```
init_queue(name)
char *name;
{
    out("queue head ptr", name, 0);
    out("queue tail ptr", name, 0);
}

add_to_tail(name, val)
char *name;
int  val;
{
    long ptr;

    in("queue tail ptr", name, ? &ptr);
    out("queue tail ptr", name, ptr+1);
    out("queue", name, ptr, val);
}

take_from_head(name)
char *name;
{
    long ptr;

    in("queue head ptr", name, ? &ptr);
    out("queue head ptr", name, ptr+1);
    in("queue", name, ptr, ? &val);

    return val;
}
```

[4]For this reason, this model is often referred to as the *replicated worker* model.

Note the use of the associative memory property. The queue here is much like that at a bakery [FLBB79]. Each process that wants to take from the queue is issued a number. It then uses content addressibility to wait for a tuple with that identifying number.

A slightly different queue mechanism, and a telling example of the power and simplicity of Linda, appears in Gelernter's solution to the readers/writers problem [CGL86]:

```
long my_slot;

init_queue(){
    out("r/w tail", 0);
    out("r/w head", 0);
    out("r/w reader count", 0);
}

ok_to_read(){
    incr("r/w tail", ? &my_slot); /* Get a slot at tail. */
    in("r/w head", my_slot);      /* Wait until at head. */
    incr("r/w reader count",      /* Bump reader count. */
        ? &count);
    out("r/w head", my_slot + 1); /* Advance queue. */
}

exit_read(){
    decr("r/w reader count",      /* Adjust reader count. */
        ? &count);
}

ok_to_write(){
    incr("r/w tail", ? &my_slot); /* Get a slot at tail. */
    in("r/w head", my_slot);      /* Wait until at head. */
    rd("r/w reader count", 0);    /* Wait for no readers. */
}

exit_write(){
    out("r/w head", my_slot + 1); /* Advance queue. */
}
```

See Section 2.1 for a definition of incr() (decr() is similar). The solution in Ada, given by Barnes [Bar80], is more complicated — and incorrect (as Barnes concedes).

Both the task bag and the queue are examples of what we refer to as *distributed data structures*, which are central to the idiomatic use of tuple space. Linda allows the construction of data structures that are accessible to all processes. The operations on tuple space make it possible for a user to change these structures atomically. One might say that algorithms + distributed data structures = parallel programs (cf. [Wir76]).

One final example involves the common problem of determining when a certain collection of tuples is empty. In the master/worker example, if some workers are forked before the master generates the first task, they will immediately exit because the loop inp will fail. Clearly this is not what was intended. It turns out that the use of the p operations to determine the non-existence of a tuple requires some extra synchronization (not too surprising, given that the p operations are asynchronous).

We could generate all the tasks before the workers are forked. This would ensure that a failed inp would mean no more tasks exist. But this would also introduce unnecessary serialization of the task generation and execution stages. A more general solution is:

```
task_producer() {
    while <tasks need generating> {
        /* Generate a task. */
        .
        .
        .
        out("task", task_structure);
        .
        .
        .
    }
    out("task generator finished");
}

task_consumer() {
    while (!rdp("task generator finished")) {
        while(inp("task", ? &task_structure)) {
            /* Do task. */
            out("result", ...);
        }
    }
}
```

The new synchronization element is the tuple ("task generator finished"). This solution does not require serialization of the generation phase and execution phase. It also requires less tuple space activity than the alternative of maintaining a count of tuples in tuple space. The latter requires at least two[5] extra operations on tuple space per task. It also requires that the generator have some a *priori* knowledge of the total number of tasks (or at least always run ahead of the consumers).

2.6 Problems

There are, to be sure, a number of things that tuple space could handle better. A few examples:

Protection: Currently tuple space is "flat". It is the user's responsibility to chose the format and content of tuples such that unintentional "aliasing" does not occur. This is a burden on the programmer and makes implementing protection somewhat difficult. This problem could be attacked by using some system routine that distributes unforgeable id's and adding a type to Linda that would support them.

[5]Maybe as many as four if the producer increments the count with every new task, rather than initializing the count to some known total value of tasks.

Duplication: Many algorithms would be made considerably simpler if tuple space supported some type of duplicate detection — that is, set semantics rather than multiset. A typical class of such problems is the synthesis of some new object from objects in tuple space. The new object should be added only if it does not already exist. Of course, one could create a critical section which could be used to perform an atomic "inp and out if not there", but something simpler would be nice.

Bulk Tuple Movements: It would also be useful to be able to operate on collections of tuples, e.g. change all tuples

```
("foo", *, *)
```

to

```
("bar", *, *)
```

In many such cases, routines could be written to do this using inp's and some synchronization, but again something simpler would be nice.

The technique of kernel compiling could be applied here, but these examples taken together seem to be raising an orthogonal issue. We sketch an approach to these, and other shortcomings, based on the concept of *multiple tuple spaces.*

Multiple tuple spaces are really just the realization of first-class status for tuple spaces. Rather than have one omni-present tuple space, Linda generalized to multiple tuple spaces will support operations that create and manipulate whole tuple spaces. Each tuple space will be characterized by a number of attributes. In this environment the problems mentioned above have natural solutions:

Protection: Just an attribute of a tuple space, much like many schemes for file protection.

Duplication: one of many possible additional attributes:

1. Duplicates or no duplicates.
2. Read only (after creation).
3. Insert only (by offspring processes).
4. Life-time of tuple space same as that of the creating process or life-time independent of the creating process.
5. ...

Bulk Tuple Movements: Reduced to operations on tuple spaces.

Much work remains to be done on the theory of multiple tuple spaces, and even an approximate implementation is some time away. We mention this concept to stress that the development of Linda is proceeding on many fronts and to caution the reader that to conclude that Linda cannot either easily, efficiently, or elegantly solve a problem, because we have not explicitly said it could, is risky.[6]

[6]No chip on our shoulder here, just the voice of hard experience.

Chapter 3

Compile-Time Support

THE implementation of a tuple space certainly requires a run-time system to manage communication, synchronization, and finding/matching. However, a smart compile-time system can make a significant contribution to the efficiency of the run-time system. In addition, it can provide support for a relatively pleasant user interface in the form of high-level syntax added to the host language. Finally, it can extract information from the source text to be used by run-time debugging tools.

Some of what the compile-time system does is clearly machine-independent: parsing, analysis of patterns of tuple activity, extraction of call text for debugging support. On the other hand, some is machine-dependent, in particular the use to which the results of the analysis is put depends upon the nature of the architecture. Thus a full Linda system has three pieces: the (largely) machine-independent (but host-language-dependent) parser/analyzer, the (largely) host-language-independent (but hardware-dependent) run-time system, and the interface between the two.

We will now present one approach to analyzing Linda code at compile time, and mention some alternatives. We will then describe an implementation of that approach.

3.1 What's Hard About Finding/Matching?

Two issues arise when attempting to find a tuple that matches an `in`[1]: where to look in the memory hierarchy of the the host machine, and, once a region of memory is selected, how to scan it for a match to the `in`. The first is important for architectures that have disjoint memories, since tuple space will be spread over the disjoint memories in some way. The first is also highly machine dependent. We will postpone further discussion of it until Chapter 4. The second is always important. It is less machine-dependent than the first and we will take it up now.

How does the system find a tuple that matches an `in` template? In the abstract, it searches through the extant tuples in some order until a match is found or tuple space is exhausted. Clearly, we can do better than an exhaustive search of tuple space. How much better depends to some extent on the amount of information we can deduce at compile-time about patterns of tuple referencing.

For example, we know from the matching rules that

```
    out(i, j);
```

[1]For the rest of this Chapter, statements about in will be understood to apply to all the extraction operations, while those about out will hold for eval as well.

will never match

```
    in(k);
```

since they have different numbers of fields. If we could somehow partition tuple space such that the tuples generated by the out are isolated, then that group of tuples will never have to be checked when the in is processed.

This example can be extended to include types and the actual/formal distinction (dubbed "polarity" by Leichter [Lei]). That is, we could classify tuples when generated by a number-of-fields/types/polarities signature so that ins would only have to search groups of tuples labeled with a signature compatible with the signature of the in. This signature classification *could* occur at run time. But all the necessary information is available at compile time, so it makes sense to pre-compute the signatures. A shift to compile time brings additional advantages. Consider the following:

```
    out("data for foo", j)
    out("data for bar", k)

    in("data for foo", ? &j)
    in("data for bar", ? &k)
```

(Assume that the rest of the program enforces the intended distinction.) The signature method will fail to separate the tuples produced by the two outs. Since it is likely that there will be heavy use of tuples with small numbers of fields, such "collisions" are cause for concern. Fortunately, such situations can be avoided by a combination of programming style and compile-time checking.

At compile time we can collect the call text of all the Linda operations in the program. These calls can then be partitioned into sets based on the signatures. Now, within sets we can detect situations such as the above by extending the pre-match (this is what the signature partitioning amounts to) one step further to comparing known fields (i.e. fields whose values are constants). In the above example all the operations would be initially placed in the same set, but the second phase of the pre-match would split this set into two sets:

```
    out("data for foo", j)
    in("data for foo", ? &j)
```

and

```
    out("data for bar", k)
    in("data for bar", ? &k)
```

based on the available constant data. We rely here on the programmers' use of a self-documenting style when constructing tuples. However, if they do not they are no worse off then with the original signature. Details are given below where it will also be shown that using this style has no run-time cost (that is, no penalty in memory or execution speed is paid for including constant information as in the above).

We can also extract information that will be useful for a number of other purposes: deciding how to organize tuples within a partition, what fields are needed for matching, where (in a disjoint memory architecture) tuples should be stored, etc. The remainder of this Chapter and Section 5.1 will give examples of the first two uses, while the latter is briefly discussed in the conclusions of Chapter 4.

3.2 A Compile-Time Analysis System

We will present details of the methods discussed above by describing an implementation for an Encore Multimax. Figure 3.1 gives a high-level view of lcc (Linda-C compiler) for the Encore. Input to lcc is user code with in and out statements. Output is a runnable image linked to the Linda run-time system.

The first step in the process is to run the source code through the C preprocessor (*cpp*) which will perform file inclusions and macro expansions. The Linda preprocessor (*comp*) then extracts all Linda calls from the source and puts them into a file for later use by the analyzer (*parseout*). The Linda preprocessor also produces a slightly modified version of the source with nulls inserted before and after each Linda call.

The analyzer partitions the extracted calls into sets based on signatures and constants. These sets are analyzed for patterns of field usage. The results are used to select a paradigm for managing the tuples in each set. Then, for each call, new in-line code is generated to replace the original call text. This new code will depend upon the handling paradigm appropriate for the set as well as the text of the original arguments for a particular call. The new code for all the calls is passed to a routine (*fixcalls*) that runs through the slightly modified text, transplanting the new in-line code for each Linda call. The final results are passed to the standard C compiler (*cc*) and then on to the linker (*ld*).

3.2.1 The Linda Preprocessor: The Parser

The Linda preprocessor is a modified version of the Portable C Compiler [Joh78]. The modifications are in the main small, but numerous. It is worth detailing them to give a sense of what would be required to produce a Linda-Pascal or a Linda-FORTRAN.

First the new keywords in and out were added. These are grammatically similar to a function_id in C. They differ from a normal C function_id in that they do not lexically nest (however, note that while nesting does not make sense for in and rd, it does for inp and rdp) and that they take an argument list that uses a slightly different syntax. They are alike in that they can occur in the same places as a C function_id.[2]

Grammar rules were added for the argument list of a Linda function. This was necessary to support syntax for indicating that a field was a formal. We also wanted to support anonymous fields — that is, formal fields that have no storage associated with them. These prove handy in certain situations, such as cleaning up tuple space. The rules are (using YACC [Joh75] notation):

```
    linda_list      : linda_arg %prec CM
                    | linda_list CM linda_arg
                    ;

    linda_arg       : expression
                    | QUEST linda_arg_prim
                    ;

    linda_arg_prim  : e
```

[2]Since the appearance of one of these keywords as a function_id introduces a new syntax for argument parsing, one should not take the address of one of these functions and expect dereferencing the resulting pointer to work.

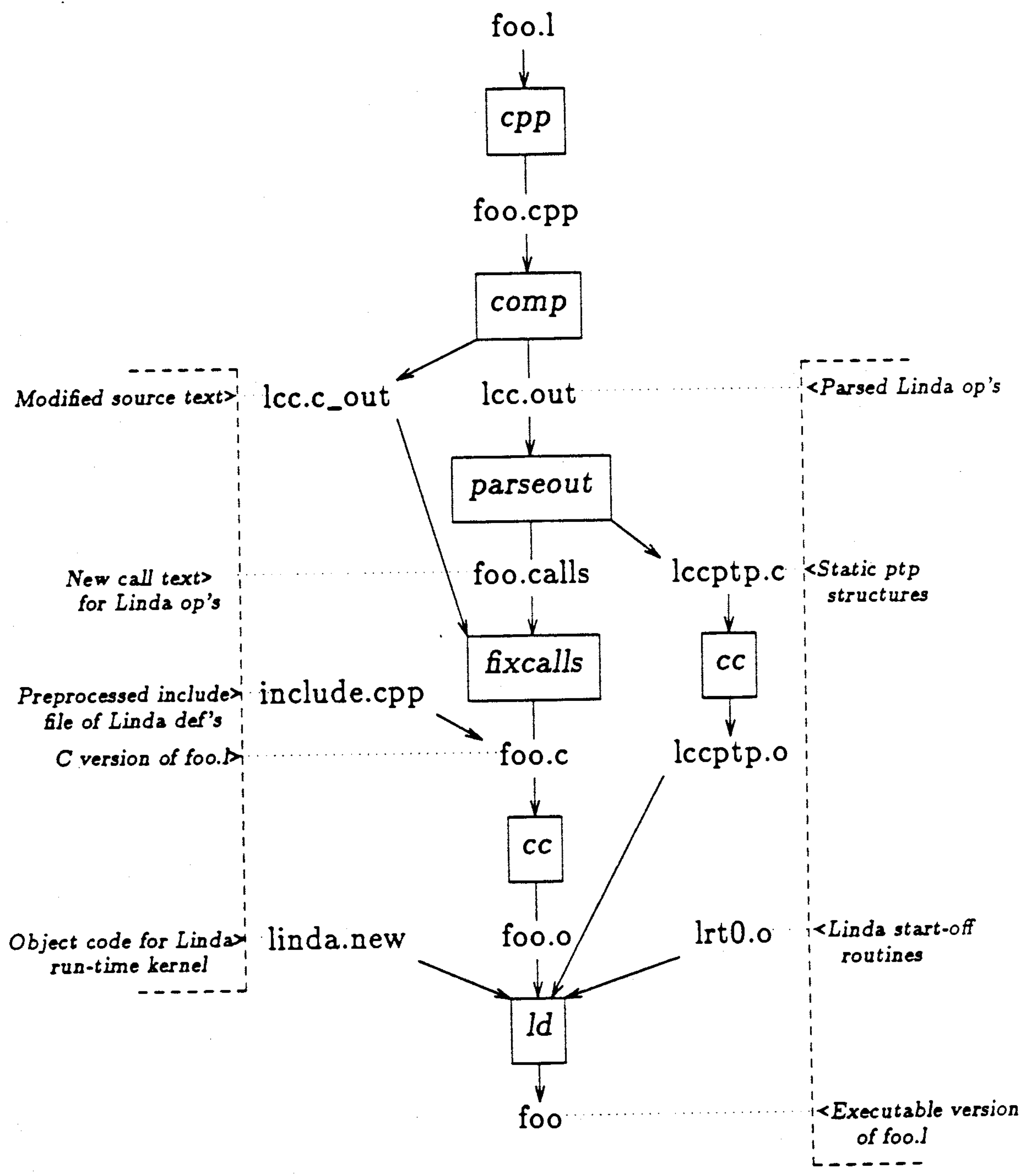

Figure 3.1: The structure of the MULTIMAX compile-time system.

```
                        | cast
                        ;
```

where CM is the "," token, QUEST is the "?" token, and cast is the production for specifying the type within a C cast operator.

Actions associated with the grammar rules for parsing a Linda keyword cause the text of Linda operations to be copied to a file as the text is being parsed. Actions associated with rules for parsing arguments of a Linda operation cause special tokens to be output as well. These special tokens separate the source text into individual arguments. When the Linda operation has been parsed, a list of the argument types is written to the file. Thus, after the processing of a call has finished, the file contains the text of the call parsed into arguments, then a list of typing information for arguments (including whether an argument is constant, and if so the value). For example the in in:

```
#include "linda.h"

lmain()
{
    long        i;
    float       f;
    LINDA_BLOCK b;

    in("testing", i, 1, f, 2.71828, ? &b);
}
```

produces the following parsed call and typing information:

```
::call #0
::start line 9
in(
.."testing",
.. i,
.. 1,
.. f,
.. 2.71828,
.. ? &b)
::finish line 9
::1 1 0 PTR strty
::0 0 2.71828000000000e+00 double
::0 1 0 float
::0 0 1 int
::0 1 0 int
::0 0 0 PTR char
::done
```

Note that the arguments' types are listed right-most first. The typing information consists of three fields and the type. The first two fields indicate whether the argument is actual or formal and constant or variable, respectively. The third field is the value if the argument is constant (a char *'s value, if constant, is just the text of the corresponding argument). An argument with type `strty` is assumed to be a `LINDA_BLOCK` — no other structure is valid as a field of a tuple.

Since the Portable C Compiler is being used as a preprocessor here, code emission is disabled.

When all the calls have been processed, the file contains all the information that we use to analyze the calls. There are a number of additional pieces of information that could be extracted. Some examples and the uses to which they would be put are given at the end of this chapter and at the end of Chapter 4.

Note that the format of the parser's output file constitutes an implicit interface between the parser and the analyzer. There is a similar interface that describes how to turn the new call code generated by the analyzer into an appropriate sequence of operations and system calls in the host language. This level of interfacing precludes the coupling of the Linda analyzer and the optimizer for the host language, which may be a loss. But this level of interaction is clearly simpler to maintain and easier to apply to other languages.

3.2.2 The Analyzer

After parsing, the file containing information about the Linda calls is processed by the analyzer, which builds a data structure for each call. These data structures will become elements in sets of calls. Earlier we spoke about tuples and templates: the former are generated by `out`, the latter are the arguments to an `in` against which tuples will be matched. Note that the sets of calls generated at compile time correspond to multisets of tuples and templates at run time (each call can be executed an arbitrary number of times).

At compile time, we have to introduce two new forms, "out-pattern" and "in-pattern". The distinction arises from the fact that at compile time the arguments have not been evaluated. What matters at compile time is not whether `i` and `j` have the same value (we simply do not know), but rather whether a given field is formal or actual and, if the latter, constant or variable. Clearly the matching rules for in-patterns and out-patterns will have to be different from those for tuples and templates. However, the intent is to capture as best we can the semantics of tuple/template matching:

1. Number of fields must match, as before.
2. Field types must match, as before.
3. Constants must match, as before.
4. Actuals are assumed to match.
5. Corresponding constants and actuals are assumed to match.
6. Corresponding actuals and formals always match, as before.
7. Corresponding formal fields never match, as before.

For example:

The out-pattern `("foo", j)` matches the in-pattern `("foo", i)` because of 4).

The out-pattern `("foo", "bar")` matches the in-pattern `("foo", s)` because of 5).

The out-pattern `("foo", "bar")` does not match the in-pattern `("foo", "baz")` because of 3).

The next section addresses the formal relation between tuple/template matching and out-pattern/in-pattern matching.

3.2.3 Analytical Properties of Tuple Space

We must develop a few formal properties of tuple space that are needed to justify the analysis procedure that we use to partition tuple space.

Theorem 1 *If a template/tuple pair match, then their corresponding in-pattern and out-pattern must match.*

Proof:
By contradiction. Assume there exists a template/tuple pair that match, but whose corresponding in-pattern and out-pattern do not match. If the template/tuple matched, then the standard matching rules were satisfied, implying rules 1, 2, 3, 6, and 7 are satisfied. This leaves the effects of rules 4 and 5 to consider. But neither rule 4 nor rule 5 can result in an in-pattern/out-pattern pair not matching and thus the contradiction. □

Let A be the set of all patterns:

Definition 1 *Given a pattern x, the* partition *P_x of A is the set containing x and all $y \in A$ such that y matches some element of P_x.*

Definition 2 *A* partitioning *of A is given by $P_0, P_1, \cdots, P_n$ where $P_i = P_x$, $x \in A - \bigcup_{0 \leq j < i} P_j$.*

That is, we can partition all the patterns by picking a pattern and using it to seed P_0, then picking a pattern not in P_0 and using it to seed P_1, and so on, until no patterns remain. By construction, it is clear that this results in a disjoint partitioning of the patterns.

Theorem 2 *No tuple with an out-pattern in one partition will satisfy a template with an in-pattern in a different partition.*

Proof:
By contradiction. Assume that there exists a tuple from one partition that matches a template from another partition. By Theorem 1, since this tuple and template match, their corresponding out-pattern and in-pattern must match. This implies that the patterns must be in the same partition, contradicting the disjoint partitioning of the patterns. □

Given these properties, if we partition on the basis of in-pattern/out-pattern matching, then we will have a disjoint partition on the basis of tuple/template matching. This leads to an obvious partitioning of tuple space that will potentially speed the matching process: only the tuples generated by a partition need be checked for a matching tuple.

Note that we do not claim to have found the maximally fine partition of tuple space. In particular, there is a problem with formals causing partitions to merge:

```
A: in("node", 1, ? &i);

B: out("node", 1, node_state);
```

and

```
C: in("node", 2, ? &j);
```

```
D: out("node", 2, node_state);
```

will yield reasonable results (i.e. separate partitions: A, B and C, D); adding

```
E: in("node", ? &node_id, 3)
```

results in the merger of the two partitions. We could support a finer partitioning, but it would likely come at the cost of the match process having to consult an unknown number of partitions.

Finally, we mentioned above that including a documenting label as the first field of every tuple resulted in no run-time penalty.

Theorem 3 *If there exists a field that is a constant for all patterns in the partition, then that field must have the same constant value in every pattern.*

Proof:

Follows immediately from rule 3 and the transitivity of equality. □

Thus we can simply ignore such a field.

3.2.4 Partitioning of Tuple Space

As a first step in implementing this partition scheme, we group calls on the basis of number of fields and types. Each of the resulting groups is in turn partitioned according to the above rules. This is done by comparing every out in a group with every in. If the patterns of the calls "always" or "sometimes" (use of rule 4 or 5) match, then each call is added to the other's set of matchables.

When all groups have been processed, the matchable sets of the calls in each group are labeled with set identifiers. A call is picked and labeled with an id and then the call's matchables are recursively labeled with the same id until no new calls are marked. Then the next unlabeled call is picked and similarly labeled. This process repeats until there are no unlabeled calls.

We could have stopped here and been satisfied with having imposed order on the chaos of tuple space. The run-time system would now have enough information to organize tuples generated by outs and to focus the efforts of ins on a subsection of that organization. However, much more can be built on top of this foundation.

3.2.5 Classification of Fields

By studying the pattern of field usage within a set, we can often deduce an appropriate paradigm for storing the tuples generated by the out-patterns of the set. That is, not only can we partition tuple space, but we may also be able to say something about the fine structure of the partitions.

Consider two sets:

```
A:      out("foo data", i);
        in("foo data", ? &j);

B:      out("vector bar", i, j);
        in("vector bar", k, ? &l);
```

When we examine the two fields of set A, we note that the first is constant data (and in fact can be ignored at run time), while the second is always actual on out and formal on in. It follows that neither field requires run-time matching. Having observed this at compile time, the preprocessor can request that the run-time system use some simple paradigm for managing the tuples generated by the out. A lifo queue will do nicely. When an out is executed, the tuple generated is placed on the queue. An in blocks if the queue is empty, otherwise it removes the queue's first element.

What about set B? Here, too, the first field is constant (this is the style of tuple usage we advocate), and the last field is used like the last field in A. However, the middle field does require matching. But note that the middle field is always actual on out and actual on in which means that we always have a key that we can use for searching. This suggests that a hash table be used to store tuples in this set, the value of the second field being used as the hash key.

In these two examples we have reduced the powerful, but potentially costly, semantics of tuple matching to nothing more then the relatively efficient manipulation of simple data structures. What is more, these examples were chosen for their complete banality — simple data passing and referencing an element of a vector. They are not exotic special cases. Even more dramatic reductions are possible:

```
out("sem");
in("sem");
```

corresponds to a P and V of a semaphore in Linda. This is a degenerate case of the first example. Here not even data copying is necessary. Although we have not gone quite so far in our current implementation, there is no reason why this could not be reduced at run time to a test-and-set operation, exactly what it would be if it were hand-coded. This is an extreme, but it is clear that substantial improvements in tuple handling can be realized on the basis of field usage analysis.

The analysis of field usage begins where the set analysis left off. Once sets have been identified, each set is scanned for field usage. Within a set, we first collect information (called the *out-status*) that summarizes how a given field is used by all the out-patterns in the set, and another piece of information (called the *in-status*) that summarizes how that field is used by all the in-patterns in the set. In both cases the information consists of a set of four boolean variables:

FORMAL: true if there exists a pattern in which the field is formal.

ACTUAL: true if there exists a pattern in which the field is actual.

CONSTANT: true if there exists a pattern in which the field is constant.

VARIABLE: true if there exists a pattern in which the field is variable.

An out-status and an in-status is generated for each field by scanning through all the patterns in the set. During this sweep through the set we also check to see if the set contains at least one in and one out. If this is not the case, then a warning is issued informing the user that there is an unmatched Linda operation. This is useful as such situations often arise from simple typographical errors.

Next, on a field by field basis the in-status and out-status are considered together and the following rules are applied (in the order they appear) to determine possible ways to handle each field:

1. If both are always constant (i.e. ACTUAL and CONSTANT are true, FORMAL and VARIABLE are false), then ignore this field (recall Theorem 3 above).
2. If both have FORMAL false, then this field might be used as a hash key.
3. If out-status has FORMAL false, the field might be used as a key for an ordered structure (e.g. a tree).
4. If the in-status has FORMAL true, then copying may be required.
5. If the in-status has ACTUAL true, then matching may be required.

The determination of what particular information to extract in this analysis was based largely on our (perceived) ability to exploit the information. Other information is available and may be used as this system evolves (see, for example, the discussion of copy-only fields at the conclusion of Chapter 4).

Now that we know what role each field *might* play in handling the set's tuples and templates, we have to decide between the possibilities. If we never find that matching may be necessary, then we know that this set can be represented by a queue. If we may have to match, then we consult the information about keys. If a hash key was found, we use hashing; if not, but if an ordering key was found, then we use a tree. (A more detailed discussion of these choices for ordering structures occurs in Section 5.1.) If no ordering key is available, we use a list to represent the set and check for matches by exhaustive search.

Note that the latter case requires that a formal must have been used in an out. The use of formals in outs is allowed, but is unusual. In fact, the code to implement the list/exhaustive-search paradigm has not been written because it has not been needed.

3.3 Some Open Problems

If more than one field is suitable for use as an ordering key, the leftmost one is chosen. This is an unsatisfactory choice. In general one does not want to throw away information: in particular some code would benefit from a more sophisticated treatment of multiple key fields.

Consider the two ins:

```
in("object", id, ? &state);
in("object", ? &id, STATE_3);
```

The first in is looking for an object with a particular `id`, while the second is looking for some object in `STATE_3`. Our implementation will use the `id` field as a search key. If it happens that the latter in-pattern occurs more frequently, then the user will either suffer a performance penalty or have to rearrange the order of the fields. If both patterns occur with about the same frequency, the only remedy is to restructure the patterns.

One possible approach in the case of multiple hash keys is to install tuples on multiple hash chains, but this strategy carries with it a certain amount of overhead. We have not yet found a good way to handle this case at run time. Thus, we do not know what we need from the analyzer in order to support the run-time system.

The analysis process will not handle fields that are "almost" constant, e.g. if all out calls in the set have a constant value for the field, but not necessarily the same constant. If the range of values for the field is small, it may be possible to use a paradigm that effectively segregates the tuples by the value of the almost constant field, and use a second key field to organize the tuples within a given subset. This case arises frequently, typically when tuples represent some data that alternates between a few different states, the state field being almost constant. This situation is in some sense a subset of the multiple-key-fields problem above.

As more experience is gained with using tuple space, other similar situations will arise. There will naturally be a blurring of approaches to the efficient handling of these: kernel compiling, improved analysis, subsumption by multiple tuple spaces (i.e. extensions to Linda itself), etc.

However, there are a few problems that are directly related to the current design of the analysis system. The first is the problem of "higher order" analysis.

```
out("matrix row", "A", 1, <data>)
in("matrix row", "A", i, ? &<data>)
```

will be handled well by the current system. On the other hand:

```
foo("matrix row", "A", 1, <data>);

bar("matrix row", "A", i, &<data>);
```

where foo is given by:

```
foo(row_or_col, mat_name, index, data_ptr)
char        *row_or_col;
char        *mat_name;
int         long;
LINDA_BLOCK *data_ptr;
{
    out(row_or_col, mat_name, index, data_ptr);
}
```

(and similarly for bar() which ins instead of outs) is not handled gracefully. To some extent, the more efficiently we handle multiple key fields, the less troublesome this will be. Right now the best approach to this is to rely on macro expansion — that is, replace the above function calls with macros.

Another major limitation is that our analysis is not designed to be incremental. In a system with many Linda modules,[3] changing one will require re-processing them all. This limitation arises largely from the desire to simplify the implementation. We could have defined a Linda object file that consisted of the original source with calls delimited and the text of the parsed calls. This would save some work, but everything would still have to be recompiled. More thought may allow more savings, but there are subtle issues. Consider the following lines from the files A and B:

```
A: out("cat", "mouse", j);

B: in("cat", "mouse", ? &i);
```

After analysis, these lines would result in code that simply ignored the first two fields. However, changing B:

[3]In fact, the implementation described below assumes that all tuple space operations occur in the same file.

```
B′: in("cat", s, ? &i);
```

would mean that A would now have to include code for processing the second field, almost certainly requiring us to recompile A. Probably the most straightforward compromise would be a system smart enough to know which files fall prey to effects like the above when a given file changes. We can then recompile just those files.

3.4 A Final Note on Tuple Space Analysis

We have stressed the symbiotic relation between the analysis and the run-time system. The latter needs the former to provide necessary information; the existence and activity of the former is justified and determined to some degree by the latter. In theory, the analysis should have a life of its own.

We have also described ways in which the analysis can be extended to support extensions of the run-time system. But we have not made an exhaustive survey.

In particular, almost no attention has been paid to gathering information useful for optimizing communication patterns and dynamic process management. Much could be obtained from analyzing sets in the context of an `eval` graph, i.e. a logical graph of the process structure. For example, consider a message-passing architecture. Assume process `foo()` forks a number of identical subprocesses, `bar()`. If some out-pattern in `bar()` only matches an in-pattern in `foo()`, then tuples generated by the out-pattern should probably be sent directly to `foo()`'s node. On the other hand, if an out-pattern matches an in-pattern in `bar()`, we may want to broadcast its tuples to all nodes executing `bar()`.

As this work is applied to Linda systems on the S/Net, hypercubes and other memory-disjoint machines, these issues will become more important, and the analysis will have to be extended to accommodate them.

Chapter 4

The S/Net Implementation

THE S/Net [Ahu83] is a bus-based multiprocessor machine developed at Bell Labs. It consists of a number of single board computers (sbc's) connected by an 80Mbit/sec bus. Each sbc has an MC68000 processor and between .5M and 1.5M of local memory. S/Net's have no shared memory, a fact that makes Linda implementation challenging. The bus supports reliable broadcast. A broadcast failure implies either a board is dead or a buffer is full (the latter condition is caught by hardware, which sends a negative acknowledgement to the broadcaster).

The resident operating system on the S/Net is Meglos [GK85]. It provides a UNIX-like environment for programs, manages the sbc's, and downloads programs.

The S/Nets are typically used as computation engines attached to a VAX or similar machine. Most of the dozen or so built to date were used as nroff/troff servers: a mainframe offloads cpu-intensive text formatting jobs onto an attached S/Net, one job per node. On the whole the designers of the S/Net soon realized that more general use of their machine would occur only after a suitable environment for parallel programming was provided. It was proposed that Linda might provide such an environment. We undertook to find out.

4.1 The S/Net Kernel

The fact that this was our first implementation influenced the design in important ways. Chief among them was the decision to make S/Net Linda-C a pure run-time system. Without previous experience, there was no body of knowledge to guide decisions as to what might be appropriate or important or useful to accomplish via a compile-time system.

Another consequence, one that follows from the run-time character of this system, is the emphasis on communication issues. Our desire to use the S/Net's reliable broadcast facility dictated many implementation choices, and in fact determined the physical structure of tuple space.

Numerous other implementation choices were made (and constraints imposed) during the development of this system. We will now describe in detail the S/Net kernel and, when appropriate, some of the roads not taken. After a discussion of performance results, we conclude this chapter with a sketch of a new implementation which will have a compile-time component.

When implementing a tuple space, two fundamental decisions have to be made: 1) How will the logical structure of tuple space be mapped to the physical memory of the machine? 2) What will be the structure of the Linda kernel software? In particular, how will it relate to the native operating system? These decisions are related, but we will discuss them independently.

4.1.1 The Representation of Tuple Space on the S/Net

Given that tuple space is logically similar to a shared memory, putting it on a machine that lacks shared memory presents a problem. We could put the tuple space on one node and then direct all tuple operations to that one node. This would create an obvious bottle-neck that would almost certainly be disastrous for performance. It seems clear that tuple space should be distributed over some subset (possibly all) of the nodes. Which subset will depend on the hardware involved, and so it will vary from architecture to architecture.

4.1.2 Issues in Distributing Tuple Space on the S/Net

In making the choice for the S/Net, we decided to take advantage of the machine's reliable broadcast. This facility makes it relatively easy to maintain copies of tuple space on every node in the machine. Any given global update to tuple space can be made with a constant number of bus accesses, rather than the $O(n)$ which might have been the case without broadcast. Several different processes can, in parallel, resolve **ins** by consulting their private copies of tuple space; only when the matching (which we thought had the potential for being the most time-consuming part of the system) is complete would some global protocol be needed to ensure atomicity of tuple space deletes. **rds**, which do not remove tuples, are very cheap; all the processing is done on the node of the requesting process. **outs** require global activity by all the nodes in the machine, but this activity requires far less time than matching. Earlier studies [Ahu] had suggested that it is unlikely that all this broadcasting would saturate the bus. The main drawback is profligate use of memory.

One alternative that we considered at length is the "inverse" kernel. In this scheme, tuple space is distributed over the machine by leaving tuples at the nodes where they were generated. **in** or **rd** consult the portion of tuple space on their node of origin. If no match is found, a request for a matching tuple is broadcast to all other nodes and a response is awaited. When a matching tuple is found it is sent directly (i.e. point-to-point rather than broadcast) to the requester. An inverse kernel would reduce memory consumption in two ways. First, since only **ins** are broadcast, templates, not tuples, are replicated and intuitively templates should be smaller than tuples. Secondly, there are likely to be far fewer templates than tuples. The number of templates is bounded by the number of processes, while the number of tuples is unbounded. Clearly this scheme solves the memory problem, but at what cost?

outs would be cheaper, as they would be purely local events. However, **ins** and **rds** would probably be more expensive. If an **in** or **rd** is matched by a tuple generated by an **out** that executed on the same node, then the entire process would be local and faster. If not, then the matching process becomes more complex. After consulting the portion of tuple space ~~on its node and failing to find a matching tuple, an **in** must broadcast a request~~ to all other nodes. This could double the time for a match, without considering any of the communication costs.

We must also ~~handle the problem of more than one node finding a matching tuple in~~ the case of a broadcast request. The **in** could accept delivery of only one of the matching tuples, but that would be wasting the potentially large effort expended by other nodes that found a successful match.

An alternative would be to accept all matching tuples. All but one of these would be placed in the in's node's local tuple space. The remaining tuple would be the one withdrawn. This process could be considered analogous to prefetching a cache line when a word reference misses. The assumption is that if this node wanted one tuple of this type, it will want others. This might often be true in the case of the replicated worker model. A worker that needed a task would collect an assortment of tasks from its neighbors and work away on them, bothering other nodes again only when this assortment had been completed. If the worker spends an abnormally long time on one of the tasks, then the rest of its collection will eventually be claimed by neighbors with no task descriptors in their local spaces.

On the other hand, this alternative leads to the clustering of information that was once dispersed, which is generally something we want to avoid in a distributed system. It is also unlikely that we want a rd to accept all matching tuples. A tuple that is consulted via rd is usually meant to be a piece of global information — that is, it is likely to be referred to by a number of nodes. Imagine what would happen if we handled a rd in the same way proposed for an in — a process on node A may rd something that happens to be stored on node B. In response, the tuple is moved from B to A. Then a process on D wants it, so it gets shipped from A to D. And then node C wants it and so on. Nothing will be gained by repeatedly yanking it out of one data structure and putting it into another. We would probably be better off leaving it on its node of origin.

This example suggests another interesting alternative: broadcast tuples sent in response to rd requests. Of course, then what happens if some process wants to in a tuple that has been broadcast because it has been rded? We will have to get rid of all the copies.

There are numerous refinements that could be made to this alternative scheme especially when one considers a number of heuristics that could be fueled by data from a compile-time system. Some tuples could be determined (see Section 3.4) to be "task bag"-like, others "point-to-point" in character, and still others "broadcast", giving rise to an amalgam that might claim a certain hybrid strength.

To some degree, greater experience and a more detailed understanding of patterns of tuple usage in a variety of programs would have allowed us to make a more authoritative decision between the two main schemes and appropriate refinements. On the whole it is clear that the inverse kernel would be considerably more complex to implement then the method we ultimately chose. Given that the benefits were not such as to convince us of an order of magnitude improvement in performance, we opted for simplicity.

4.1.3 Details of Tuple Space Structure

Once we decided on the manner in which tuple space was to be distributed over the S/Net's nodes, the next questions were what data structure would be appropriate for representing tuple space and how should this structure be manipulated?

The overriding consideration in choosing a data structure was support for the content addressability of tuple space. We have discussed in some detail the difficulties of finding and matching (Section 3.1) and an approach to coping with these difficulties (Section 3.2). The problem was handled differently here because there was no compile-time analysis to support a disjoint partitioning of tuple space at run time. In particular, with no a *priori* knowledge of the value of fields, the partitioning method had to be invariant with respect to the actual value of fields. The obvious method, partition on the basis of a type signature, was unlikely to produce a sufficiently fine-grained partitioning of tuples. For example, almost all matrix

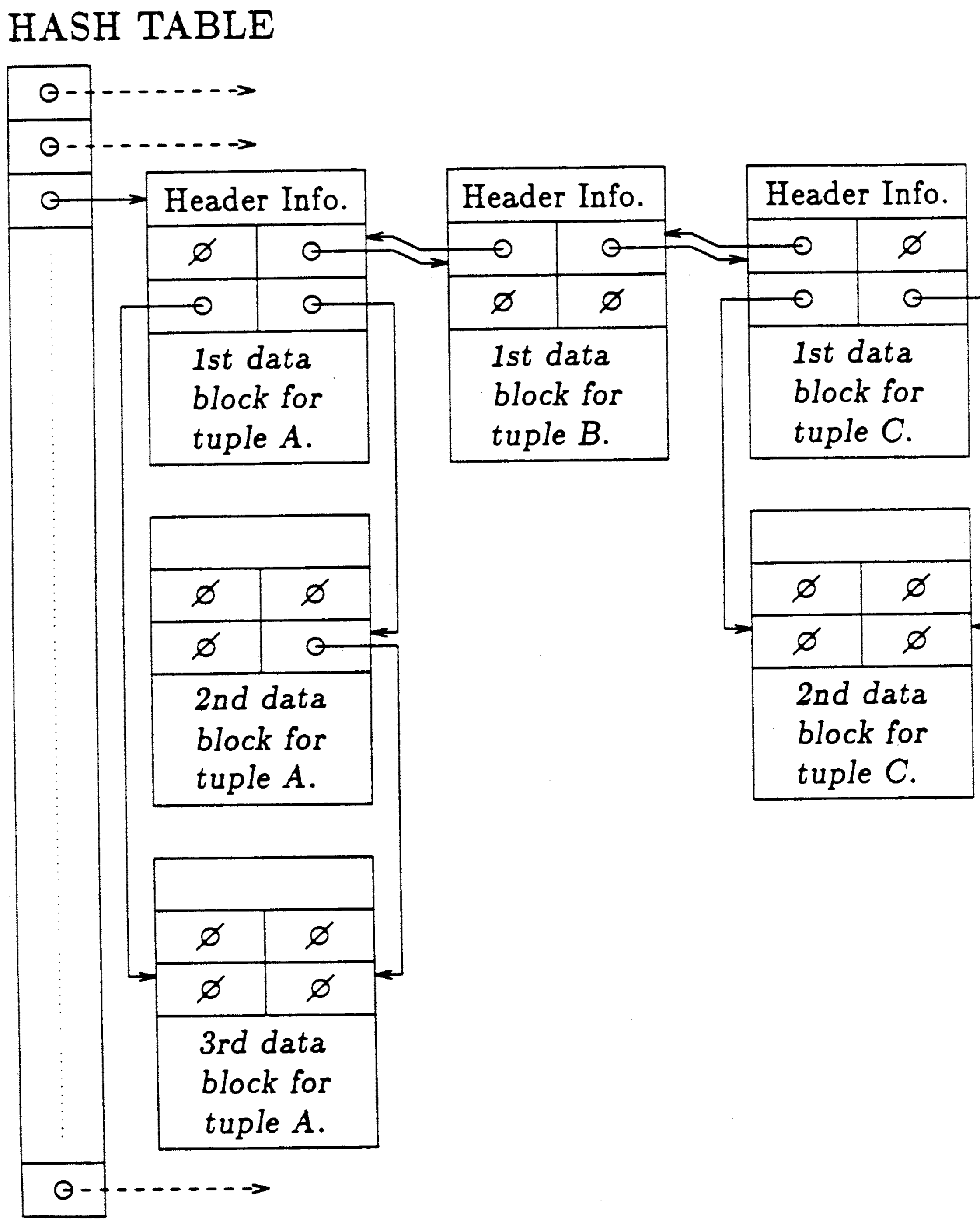
HASH TABLE
Header Info.
Header Info.
Header Info.
1st data block for tuple A.
1st data block for tuple B.
1st data block for tuple C.
2nd data block for tuple A.
2nd data block for tuple C.
3rd data block for tuple A.

Figure 4.1: The structure of the hash table representing tuple space.

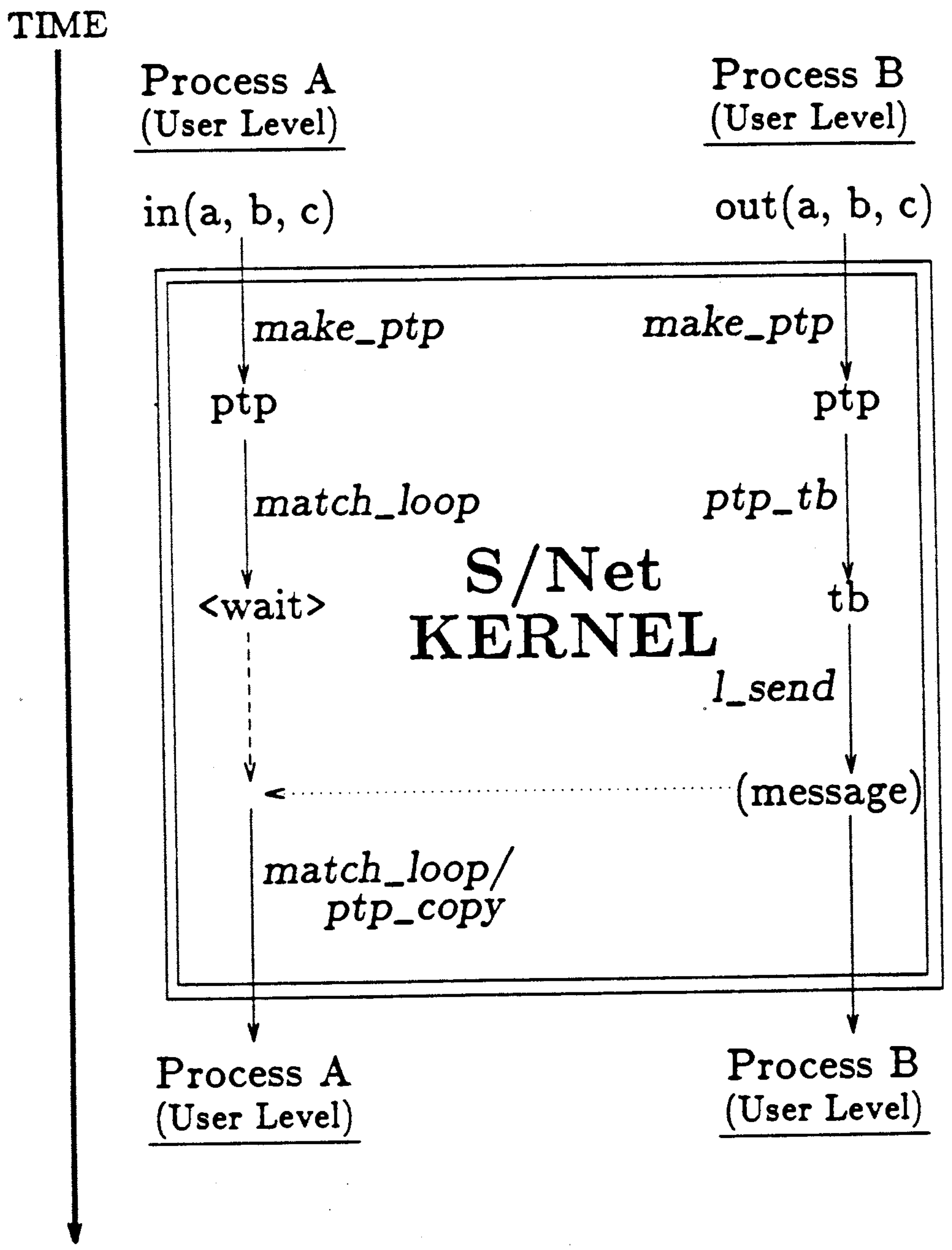

Figure 4.2: Kernel view of an out-in.

This last action causes the spinning `in` to break out of its loop. The `in` scans the appropriate hash chain, finds a match and initiates the delete protocol by broadcasting a request to delete the tuple. The request consists of the logical hash (to help kernels on other nodes find the tuple), a unique id (all tuples have an id, *tid*, which is generated from a 3-byte counter maintained by each node's kernel and a one byte node id), and a return address. The nodes in the network respond by deleting the tuple specified by the id, if they have it.

The node of origin of the tuple responds to a delete request with an acknowledgement that is sent only to the requesting node. If more than one request for the same tuple occurs simultaneously, the bus arbitration logic effectively resolves the dispute: the first node to access the bus will be the first to have its request delivered to the node of origin which in turn will acknowledge the request. All later requests will receive a negative acknowledgement.[6]

When the delete acknowledgement has been received, the final step is to copy the actual tuple field to the template formal. This is done by a routine (`ptp_copy()`) that makes use of the information gathered at match time as mentioned above. The copy complete, control returns to the user's program.

There is one other important issue which we should mention. A node's tuple space is manipulated both because of user level activity (e.g. `in`) and system level interrupts (caused by a message coming in over the S/Net's bus). It follows that measures must be taken to avoid inconsistent views of tuple space. We handled this by installing a lock bit on every tuple header block. Using critical sections implemented via interrupt masking, this lock bit is set whenever a tuple is being consulted for a match. For example, the kernel honors this lock and postpones action on delete requests for locked tuples by setting a delete bit in the header. The match code checks the delete bit before it unlocks the tuple and if set, atomically (via an interrupt mask) performs the delete.

4.2 S/Net Performance Results

We will present timings for a number of primitive operations and for two applications, matrix multiplication and LU decomposition. The applications' results will give some idea of total system performance.

Timings were taken on an 11 node S/Net. Each node was an MC68000 running at 10MHz with 1152K of on-board memory. All runs were made with no other users on the S/Net.

The timing routines are provided by Meglos. Separate tests indicated that these routines provide a time value that accurately reflects wall-clock elapsed time. Three loops were timed to obtain some basic measures of a node's performance. Each loop had the form:

```
for (i = 0; i < LIMIT; ++i) {
    <body>
}
```

One iteration took 5.19 μsecs for a null loop body, 7.37 μsecs for "`sum += i;`" as the

[6] A slightly simpler version of this protocol does away with the acknowledgement. After a node sends a request, it listens to the bus. If its request is the first that it receives, the node "won", otherwise some other node gets the tuple. We made some quick modifications to the kernel to try this. We were rewarded with a fairly modest decrease in execution time, so we did not pursue the altered protocol in this version of the kernel.

body, and 14.19 μsecs for "`foo();`" as the body. `i` and `sum` where declared `int`. The compiler did not place them in registers. `foo()` was the null function "`{;}`".

4.2.1 Single Process Timings

First we will measure the time taken by the simplest form of **out**. We will then determine the time for an **in** by measuring the time for a single process to **out** and then **in** a simple tuple. Subtracting the time for an **out** will yield the time for an **in**.

The following code was used to measure the time for an **out**:

```
main()
{
    register long i;

    l_init();

    systime(&start_time);
    for (i = 0; i < 2900; ++i) {
       out("s", "a");
    }
    systime(&finish_time);

    print_elasped_time(start_time, finish_time);
}
```

The S/Net kernel allocates space for 3000 normal blocks and 1000 large blocks. Since there was no consumer of the tuples generated by **out**, we needed to limit iterations to less than 3000. The loop overhead is small compared to the cost of the **out**; we did not adjust for it.

The average execution time over 5 trials was 2.973 secs, or 1.025 msecs per **out**. Note that this includes not only the time to format, packetize, and send, but also the time for the **out**'s node to receive and install the tuple in the node's copy of tuple space.

Now, by adding

```
in("s", "a");
```

to the above (and increasing the loop limit to 10000 — no need to worry about exhausting tuple space) we measured a time of 1.992 ms for an **out-in** pair on one node, or 967 μsecs per simple **in**. The `rd` version of the above test needs to execute **out** only once. Making this modification and rerunning yields 462 μsecs per simple `rd`.

We used the following program to measure the time for an **out** with a variable-size data component:

```
long buf[200];

main(argc, argv)
int  argc;
char **argv;
{
    register long i;
```

```
        l_init();

        /* Set block length. */
        buf[0] = atol(*++argv);
        systime(&start_time);
        for (i = 0; i < 900; ++i) {
           out("s b", "a", buf);
        }
        systime(&finish_time);

        print_elasped_time(start_time, finish_time);
    }
```

The limit of 900 in the loop avoids exhausting the supply of large blocks.

We ran the variable-size data code with a number of different block lengths (see Figure 4.3). The discontinuity between 20 and 30 longs can be accounted for by the cost of the additional block needed once the data space in the header block is exhausted.[7] For outing blocks of data, formatting and sending the header cost 1.180 msecs (slightly more than a simple out due to the extra field), with a marginal time per long of 36 μsecs. 36 μsecs may seem like a relatively long time, but consider that each long must be copied from user space to system space, sent, received, and copied into a tuple block. Allocating, initializing, sending and receiving an extra tuple block takes 708 μsecs.

Finally, we reran the out-in test with variable-size data (see Figure 4.4). Subtracting the corresponding out times, we see that ining costs 1.128 msecs for the header, 6 μsecs per each additional long, and 30 μsecs per each additional large block. The first of these times reflects the cost for matching, the delete protocol, overhead for calling ptp_copy(), and reclaiming the header block. The latter two times account for one long copy and a few pointer manipulations to reclaim the additional large tuple block.

These times characterize the basic operations, but we have ignored a few details. There will be more discontinuities at each large block boundary as length of data increases. Times will change as the number of fields change. These times also do not reflect costs that would occur when matching more complicated tuples and templates. Unfortunately, there are too many combinations of number of fields, field types, matching requirements, and tuple-space contents to test them in any complete way. The applications given below will give some idea of the performance in more complicated situations.

To give an idea of the impact a compile-time system would have, we rewrote the test programs for the simple operations. The new versions mimic the behavior of a system in which ptp formatting and packetizing of constant tuples is done at compile time.[8] However, on the S/Net packetizing and sending are separate steps, so it will make sense to have the compile-time system generate packetized tuples when the contents are known at compile

[7] A header block has a data area of 26 longs and a large block has a data area of 128 longs. Some of the data area is used to record information about the block, so the transition occurred between 22 and 23 rather than between 26 and 27.

[8] This differs slightly from what the current compile-time system does on the Multimax. It does format ptps at compile time, but packetizing is done at run time. However, in the case of Multimax, the processes of packetizing and "sending" the tuple into tuple space are one and the same, and thus it makes little sense to pre-allocate packetized tuples since they have to be copied into tuple space on every out — virtually the same amount of work as is done now, without compile-time allocation.

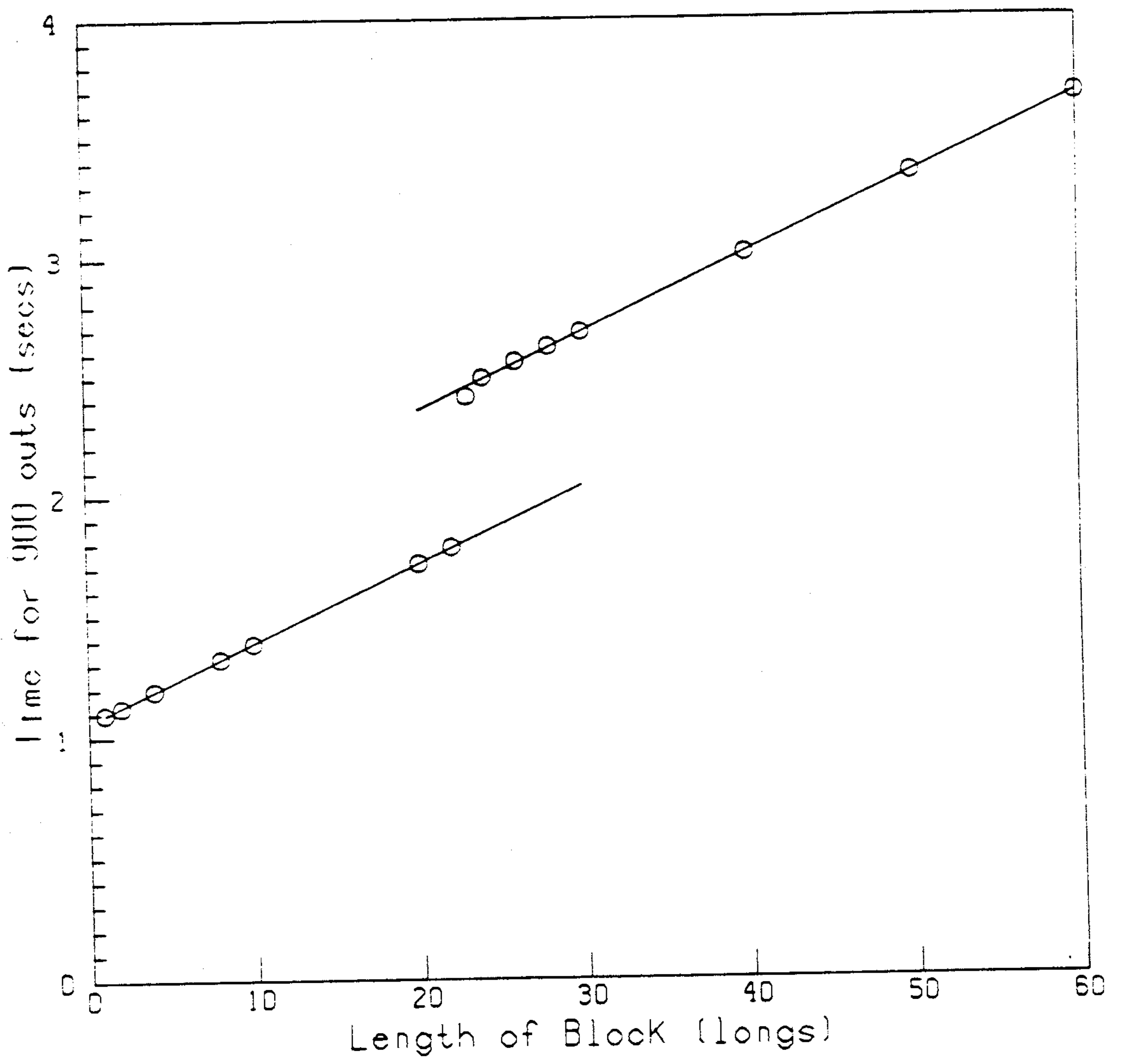

Figure 4.3: Execution time *vs.* block length for 900 executions of `out("s b", "a", block)`.

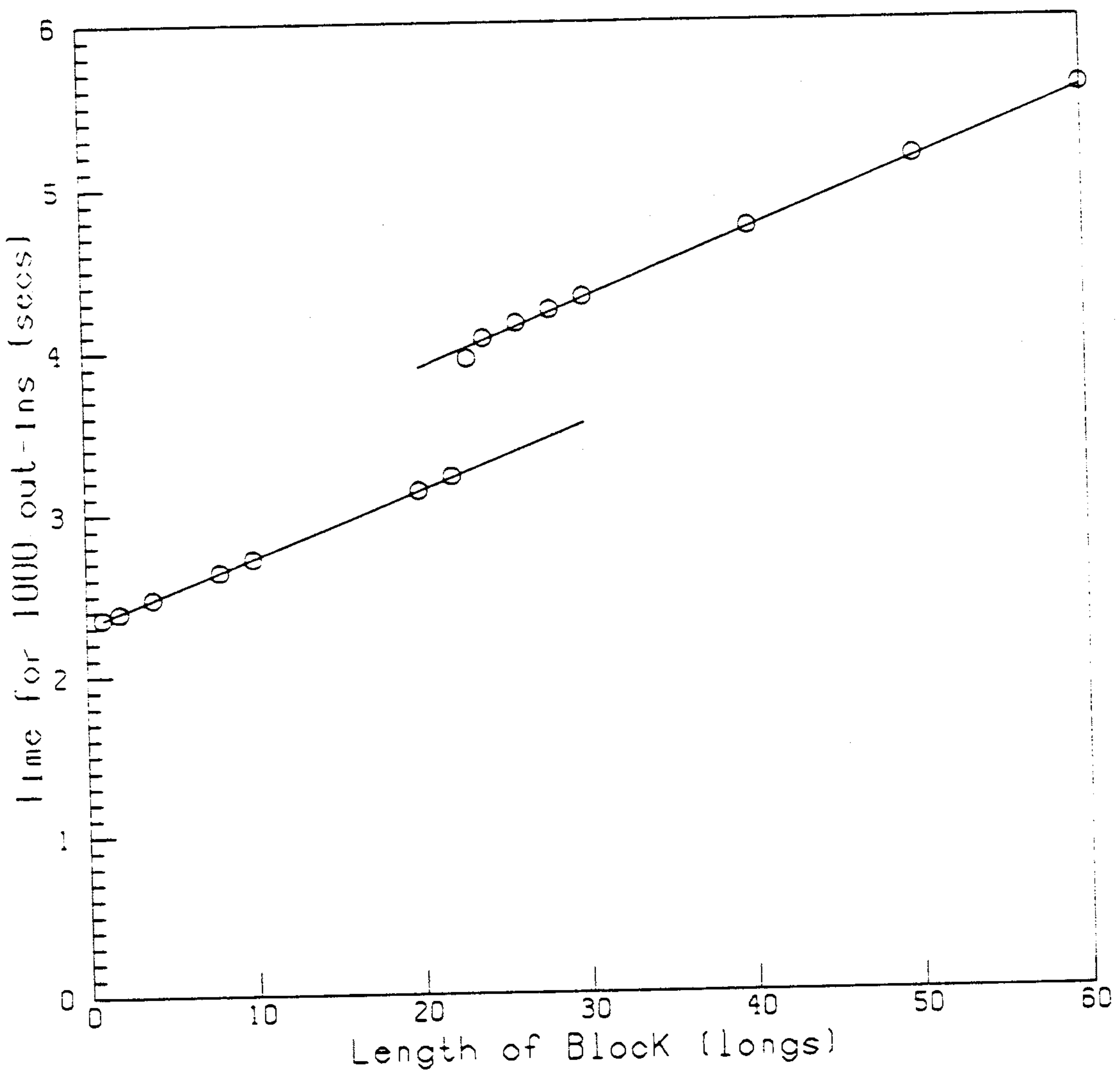

Figure 4.4: Execution time *vs*. block length for 1000 executions of out-in pairs with variable-size data.

time.[9] Each time such a tuple is outed, the address of the pre-packetized tuple will be passed to the send routine, avoiding the ptp_tb() routine.

The new versions of the timing programs made direct use of kernel routines for formatting, sending, and matching tuples. For example, the code to test out-in became:

```
main()
{
    register long i;
    char     *in_ptp, *out_ptp, *out_tb;

    l_init();

    systime(&start_time);
    out_ptp = linda_sys_out("s", "a");
    in_ptp = linda_sys_in("s", "a");
    out_tb = ptp_tb(out_ptp);
    for (i = 0; i < 10000; +ii)
       l_send(out_tb);
       match_loop(in_ptp, IN_SWITCH);
    }
    systime(&finish_time);

    print_elasped_time(start_time, finish_time);
}
```

linda_sys_out() returns the ptp that represents an out with the same arguments, similarly for linda_sys_in(). The results of these measurements were 593 μsecs for out, 753 μsecs for in and 248 μsecs for rd. Clearly, substantial savings can be realized by compile-time support.

The "optimized" times can be compared to the standard times to deduce the costs of various internal stages of the S/Net's kernel (for these examples, which use maximally simple tuples):

make_ptp(): 214 μsecs.

ptp_tb(): 218 μsecs.

match_loop(): 248 μsecs.

Delete protocol: 505 μsecs.[10]

Send (and install): 593 μsecs.

4.2.2 Multiple Process Timings

We now present results that characterize multiple process performance. The first measurement is the basic *transaction* time — from an out on one node to an in on another. We

[9]Better yet would be the integration of packetizing and sending as in the Multimax system.

[10]The delete protocol executing entirely on a single processor — i.e. in and out executed on the same node — does not send the point-to-point ok-to-delete acknowledgement.

have two measures of this. The first is a series of one-way transactions. The second, which measures round-trip time, simulates a series of RPC calls between two processes.

For the first, Process A executes:

```
main()
{
    register long i;

    l_init();

    systime(&start_time)
    for (i = 0; i < 10000; ++i) {
        out("s", "a");
    }
    in("s", "done");
    systime(&finish_time);

    print_elasped_time(start_time, finish_time);
}
```

Process B (which is started before Process A) executes:

```
main()
{
    register long i;

    l_init();

    for (i = 0; i < 10000; ++i) {
        in("s", "a");
    }
    out("s", "done");
}
```

The results were 15.41 secs total execution time, or 1.541 msecs per transaction. Since the out and in are executing on separate nodes, some of the processing of each operation is done concurrently, hence the lower transaction time here than in the single-process case (1.992 msecs).

The RPC experiment consisted of a pair of processes:

```
ping()
{
    register long i;

    l_init();

    systime(&start_time);
    for (i = 1; i < 10000; ++i) {
        out("s", "a");
        in("s", "b");
        }
        systime(&finish_time);

        print_elasped_time(start_time, finish_time);
    }

    pong()
    {
        register long i;

        l_init();

        for (i = 1; i < 10000; ++i) {
            in("s", "a");
            out("s", "b");
        }
    }
```

pong() can be though of as a remote procedure being called by ping(). The time per iteration was 3.998 msecs, which reduces to 1.999 msecs per transaction. There is tighter synchronization here between the two processes, serializing most of the work (but not quite all: one process will have begun in processing, while the other outs) and thus requiring more time per transaction. In particular, in the previous case the in could clean up (after executing the delete protocol) in parallel with the out formatting and packetizing the next tuple, while in this example this activity is serialized.[11]

As above, we were interested in the potential performance with the assistance of a compile-time system. So we "optimized" these routines using direct kernel calls. Performance improved: 1.336 msecs per transaction in the first case, 1.572 msecs per transaction in the RPC case.

In order to identify possible contention problems we ran multiple versions of the optimized ping-pong test simultaneously (the tuple contents were changed to avoid collisions between pairs of processes — the contention here is for the bus): see Figure 4.5. While the time per transaction increased (2.980 msecs average with four pairs of processes), so did the overall rate of transactions (a total transaction rate of 1342/sec with four pairs as opposed to 636/sec with one pair).

1 Pair	2 Pairs	3 Pairs	4 Pairs
31.43	40.17	48.81	59.47
	40.17	49.27	59.48
		49.46	59.70
			59.71

Figure 4.5: The effects of contention on the execution time (secs) of ping-pong as the number of pairs increases.

[11]The close agreement between the transaction time here and that for the simple, single-process out-in (1.992 msecs) is somewhat coincidental. Two factors distinguish these cases: the single-process version overlaps none of the in and out processing; the multiple-process version needs to send one more message, the delete acknowledgement. As it happens, these factors apparently offset one another.

A final measurement indicates the cost to a non-participant in a tuple transaction. We reran optimized ping-pong with an additional S/Net node looping for a fixed number of iterations, and compared the elasped time to what was required for the node to loop the same number of times with no Linda activity. The result was a difference of 11.96 secs, or 598 μsecs of interrupt activity per transaction.[12] Note that the previous tests would seem to indicate there was about 9 secs of interrupt activity per 20000 transactions, rather than 12 secs. This discrepancy is most likely due to a subtle point concerning context switches. The contention test involved processes that were all performing Linda activity; these processes were more likely than the looping process to be in the kernel. This would mean a bus interrupt was more likely to cause a context switch for the looping process.

4.2.3 Matrix Multiplication

Matrix multiplication was the first application we ran on the S/Net. It falls into a class of algorithms that are commonly referred to as "embarrassingly parallel" — meaning that if we cannot get a Linda-C matrix multiplication routine to run well, then we probably will not be able to parallelize anything well. Fortunately, the Linda-C version achieved good performance: execution times were modelled closely by $a/n + b$, where $a + b$ is close to the uniprocessor execution time (i.e. there is a small amount of overhead), b is small (i.e. the algorithm has a small sequential component), and n is the number of workers (our Linda-C version uses the master/worker model). We will refer to a as the *parallelizable* component and b as the *sequential* component of the execution time: the term in which a appears contributes inversely with the number of workers to the total execution time, while b's contribution is fixed.

We used the master/worker model as the basis of our Linda-C version (see Figure 4.6 and 4.7). The master dumps the rows and columns, as appropriate, of the matrices to be multiplied and issues the first task. (The two matrices used for the test were such that the first's rows were identical, as were the second's columns; hence the repeated outing of the same row and column.) The master then loops reading in rows of the product matrix. The workers are designed to loop continuously, accepting tasks for computing an entire row of a product matrix. The worker generates a new task to replace the one removed (up to a total number of tasks equal to the dimension of the product matrix).

We chose a fairly coarse granularity here; workers could, for example, compute one dot product. The coarser the granularity, the lower the ratio of communication to computation. The tradeoff is usually loss of available parallelism. For example, in the extreme case a task could compute the entire product matrix. If many (independent) matrix multiplications need to be done, then this extreme may be reasonable; otherwise workers are likely to be idle. It is fairly easy to adjust the level of parallelism using Linda-C, which is important since the tradeoff will vary from machine to machine. For example, the level chosen above works well on the S/Net, but a coarser level works better on the hypercube [Bjo].

We also chose to have the workers generate new tasks and to have the master withdraw the product rows in order. Neither is logically necessary. The decisions were made merely to demonstrate some properties of Linda-C and of the master/worker model. The former is an instance of multiple sources for tasks (the master need not and in some cases cannot generate all tasks), while the latter is an application of tuple space's associative-memory

[12]The test employs an unlikely amount of tuple-space activity — interrupt delays as a percentage of total time will be much less in more reasonable situations. Nonetheless we would like to minimize this time since every node must pay it.

```
/* MASTER */
float a[MAXDIM + 1], b[MAXDIM + 1], c[MAXDIM], right;

main(argc, argv)
int  argc;
char **argv;
{
    long dim, index, x, y;


    l_init();

    dim = atol(*++argv);
    generate_test(dim);

    systime(&start_time);

    /* out rows of A and columns of B. */
    for (index = 1; (index <= dim); ++index) {
        out("d s s b", index, "a", "row", a);
        out("d s s b", index, "b", "col", b);
    }

    /* out first task:
        ("dot", <row to compute>, dimension, array names).
    */
    out("s d d s s s", "dot", 1L, dim, "a", "b", "c");

    /* in result rows. */
    for (index = 1; index <= dim; ++index) {
        in("d s fb", index, "c", c);
    }
    systime(&finish_time);

    print_elasped_time(start_time, finish_time);
 }
```

Figure 4.6: Matrix multiply — master.

```
/* WORKER */
float        ra[200], ca[200], pa[200];

main()
{
    char  mat1[10], mat2[10], mat3[10];
    long  dim, length, row, x, y;
    float sum;

    l_init();

    while (1) {
        /* Get next task. */
        in("s fd fd fs fs fs", "dot", &row, &dim,
            mat1, mat2, mat3);

        /* Reached end of this array, get another task. */
        if (row > dim) continue;

        /* Generate new task. */
        out("s d d s s s", "dot", (row + 1), dim,
            mat1, mat2, mat3);

        rd("d s s fb", row, mat1, "row", ra);
        *(long *)pa = dim;
        for (x = 1; x <= dim; ++x) {
            rd("d s s fb", x, mat2, "col", ca);
            sum = 0.0;
            for (y = 1; (y <= dim); ++y) {
                sum += ra[y] * ca[y];
            }
            pa[x] = sum;
        }

        out("d s b", row, mat3, pa);
    }
}
```

Figure 4.7: Matrix multiply — worker.

behavior (as are the rds that read the proper matrix elements).

We ran the matrix code using single-precision floating-point matrices of dimension 50, 75, and 100. Eight of the S/Net nodes have SKY floating point processors, we used these eight nodes for the runs.

For each dimension we ran the code with from 1 to 7 workers (for a total of from 2 to 8 processes including the master), one worker to a processor. Given the master/worker model, this is trivial to do. We merely started more workers: no source code changes or executable rebuilding was necessary. Figures 4.8, 4.9 and 4.10 show the results. The solid descending curve is a plot of the fitted $a/n + b$ model and the ascending curve is speed-up relative to a sequential C program. The dashed line represents ideal speed-up.[13]

The uniprocessor times (t_C) were 12.38 secs, 41.59 secs and 98.37 secs, respectively. Thus in all cases the total overhead ($(a + b) - t_C$) was less than 31% of the uniprocessor time and the sequential component was less than 7%. The sequential component arises from two main sources: the time needed to out the rows and columns and the time every node spends in the Linda-C kernel handling interrupts. Both elements are independent of the number of workers. The parallelizable overhead ($a - t_C$) is accounted for largely by the time taken by the Linda operations exclusive of time spent in the kernel.

These figures indicate that, at least for this application, Linda-C works well. Two Linda workers ran faster than the sequential version and execution times continued to fall as more workers were added. In the best case, we measured a speed-up of about 5.5 for 7 workers. The data suggest that larger problems will have even better speed-up.

4.2.4 LU Decomposition

LU decomposition was our second application. The benchmark will be discussed in detail in Section 5.4.5. We mention here that the algorithm is not as "embarrassingly parallel" as matrix multiplication, because some global processing is done regularly throughout the course of the computation.

We ran the code on the nodes with floating point using matrices of dimension 50, 75, and 100. The results are summarized in Figures 4.11, 4.12, and 4.13. Because there is no preprocessor (yet) on the S/Net, the code was significantly (syntactically) messier than the original.[14] The original Multimax Linda-C version is given in Section 5.4.4, we will omit the S/Net version here. Note that these computations are double precision.

The master process does the global processing (selecting the pivot value and generating the vector of multipliers) and so is no longer a silent partner in the overall computation. As a result the line representing ideal speed-up should be a more complicated function. What we present as ideal speed-up assumes that the master contributes nothing to the computation.

Once again, though, we can see that Linda-C was able to produce speed-up in all cases (about 4.5 in the best case). As the problem size increases, the data indicates that Linda-C will do even better.

[13]We are cheating a little, since we are not counting the master process in these plots. The master process does very little here, and we could justify the ruse if we had multiprocessing per node. We could then swap in another worker over the master. This issue is more important in the next application.

[14]Our choice for a LINDA_BLOCK on the S/Net was terrible. Requiring that the length immediately precede the data makes outing subsections of arrays quite ugly. We changed this on the Multimax.

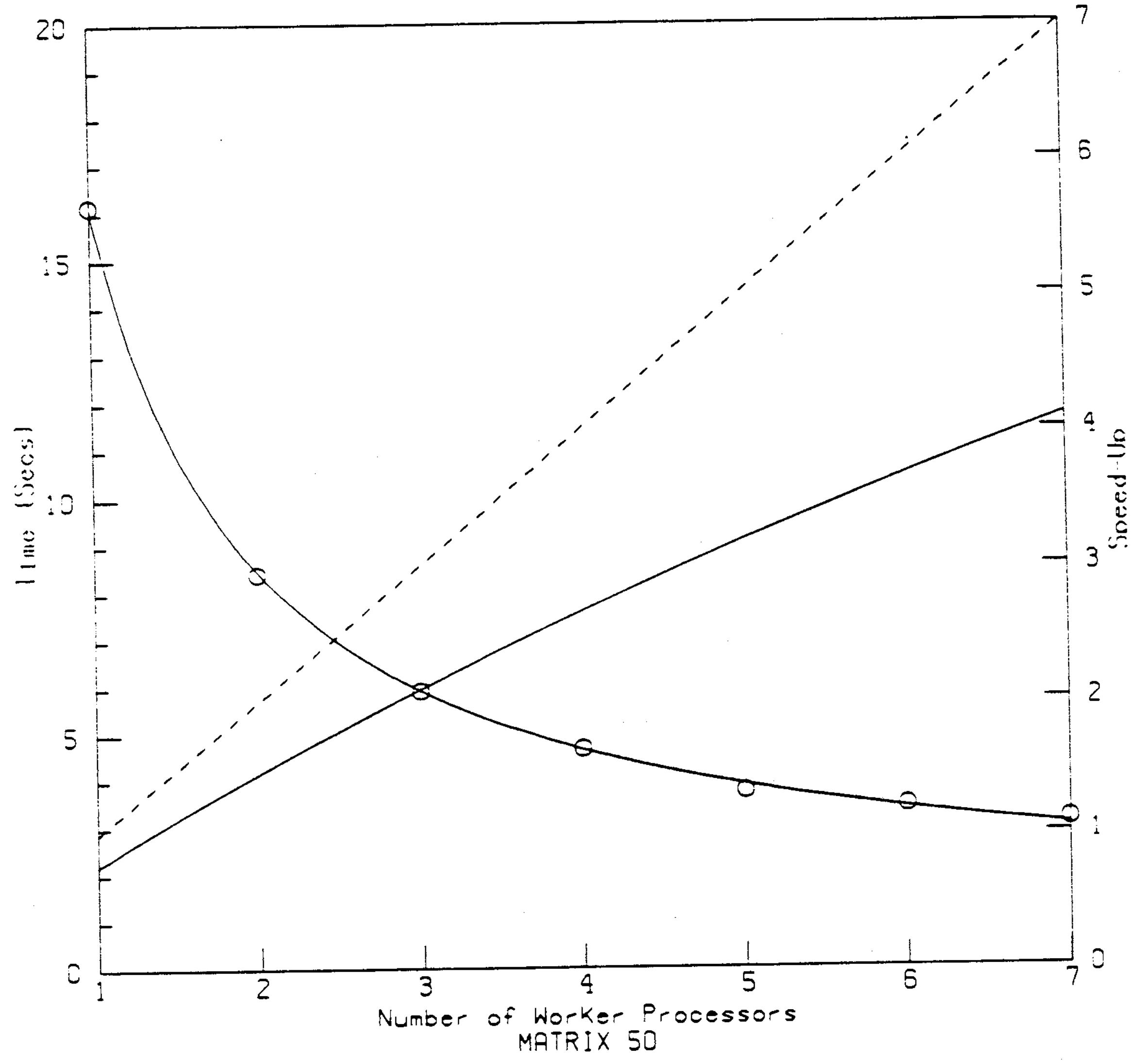

Figure 4.8: Execution time (o) *vs.* number of workers for a 50 × 50 matrix multiplication. Descending curve is an $a/n + b$ model ($a = 15.29$ secs; $b = .83$ secs). Ascending curve is speed-up relative to sequential C. Dashed line is ideal speed-up.

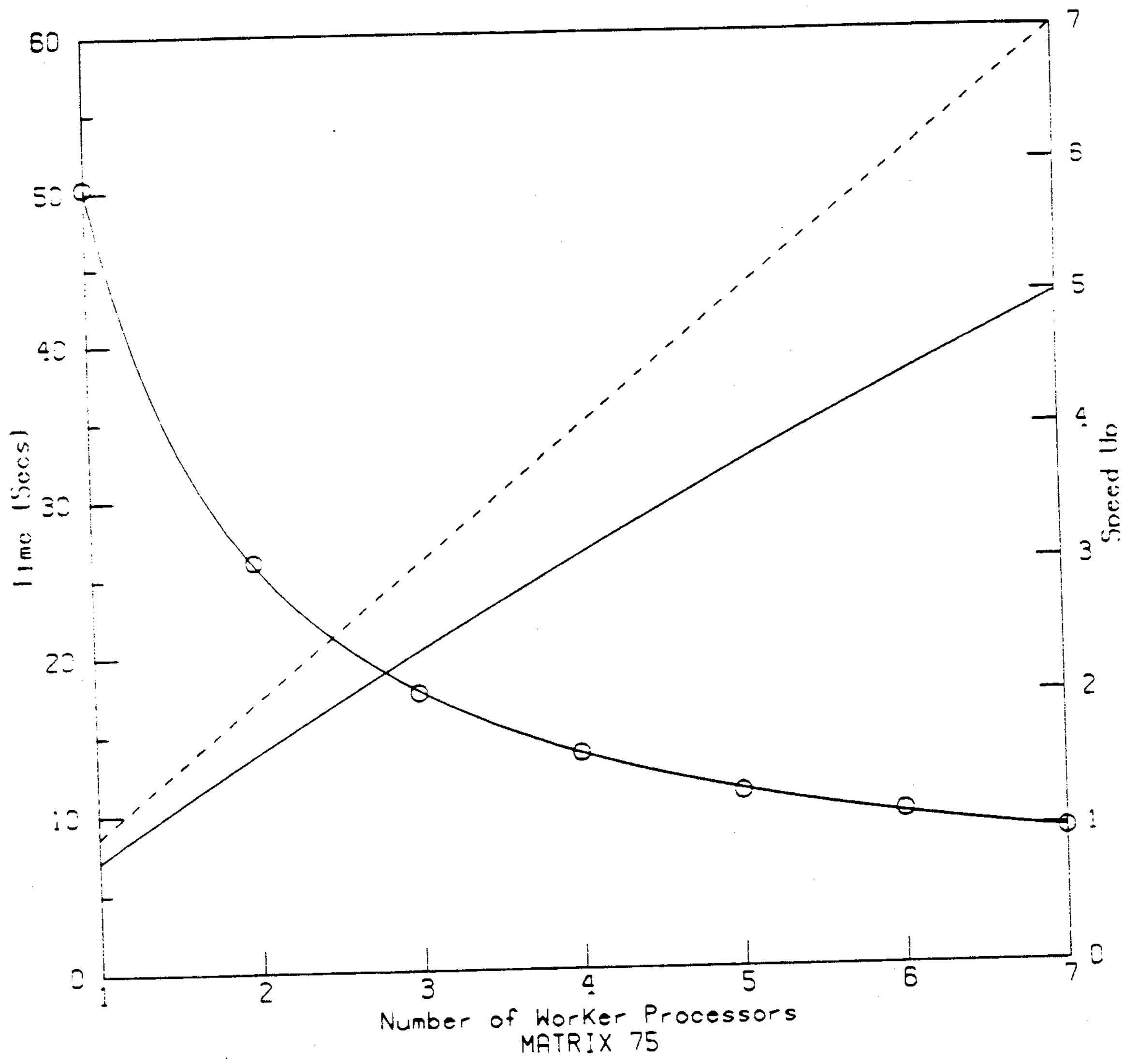

Figure 4.9: Execution time (o) *vs.* number of workers for a 75 × 75 matrix multiplication. Descending curve is an $a/n + b$ model ($a = 48.96$ secs; $b = 1.36$ secs). Ascending curve is speed-up relative to sequential C. Dashed line is ideal speed-up.

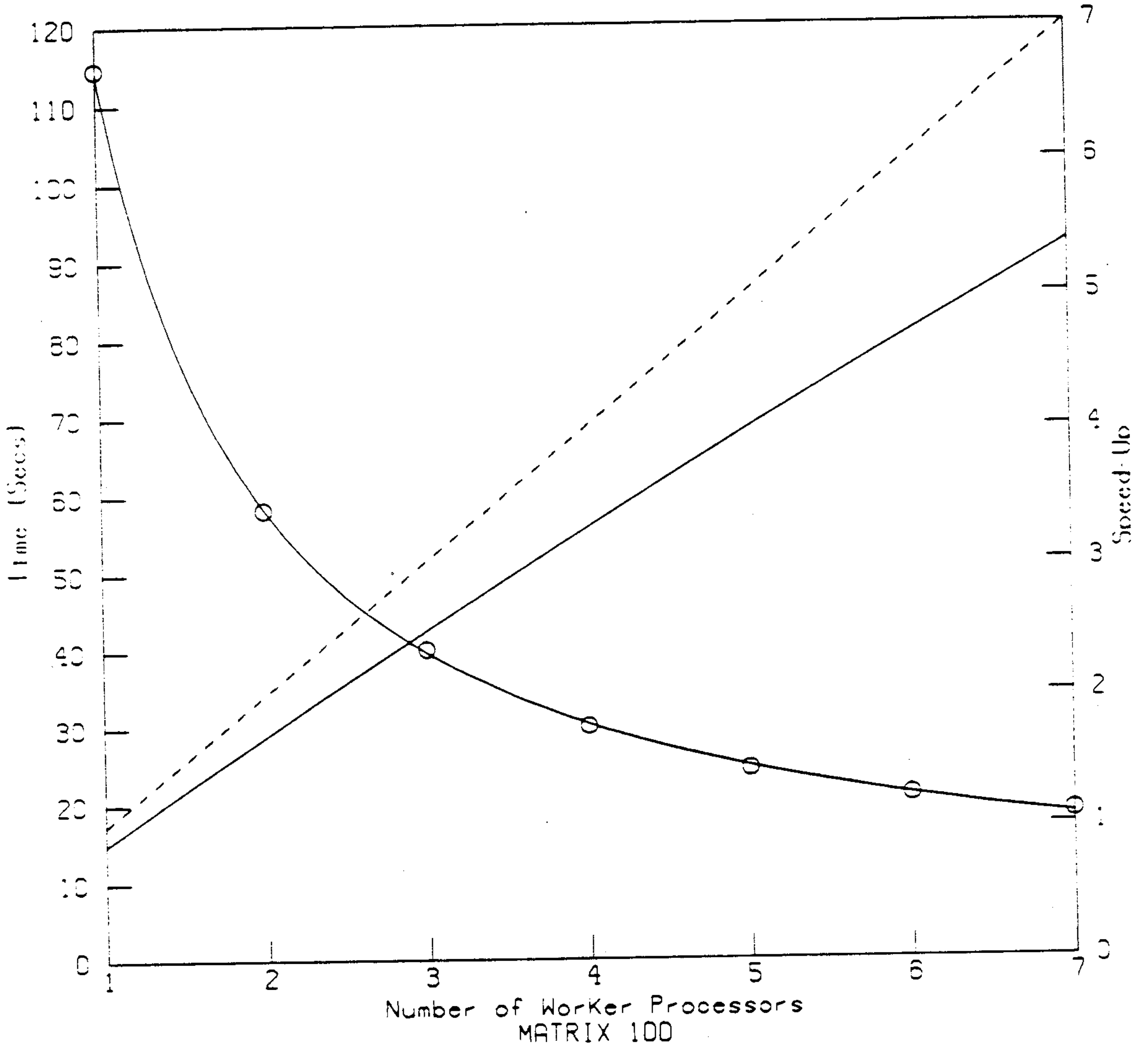

Figure 4.10: Execution time (o) *vs*. number of workers for a 100 × 100 matrix multiplication. Descending curve is an $a/n + b$ model ($a = 112.24$ secs; $b = 2.25$ secs). Ascending curve is speed-up relative to sequential C. Dashed line is ideal speed-up.

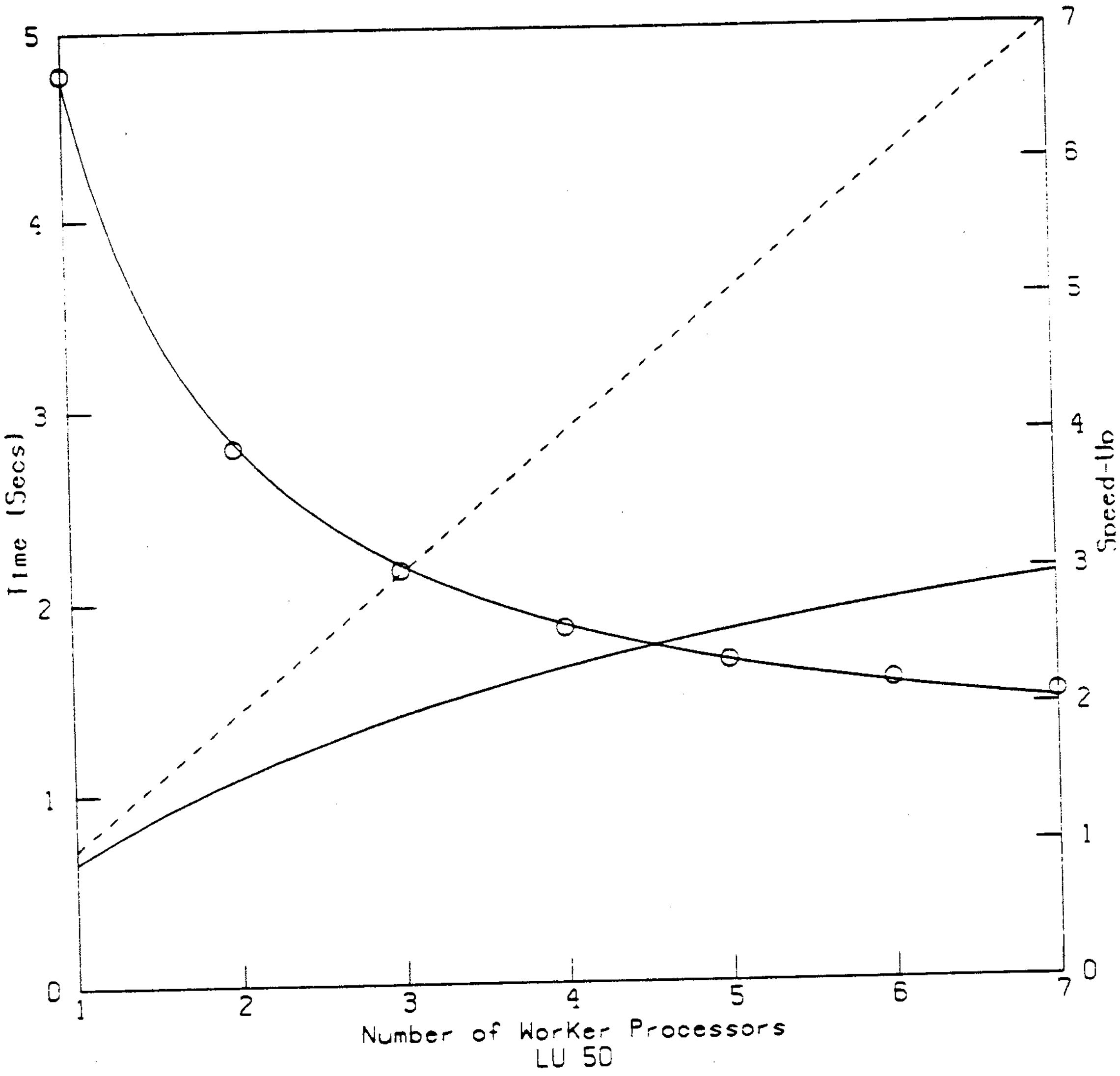

Figure 4.11: Execution time (o) *vs.* number of workers for the LU decomposition of a 50 × 50 matrix. Descending curve is an $a/n + b$ model (a = 3.85 secs; b = .91 secs). Ascending curve is speed-up relative to sequential C. Dashed line is ideal speed-up.

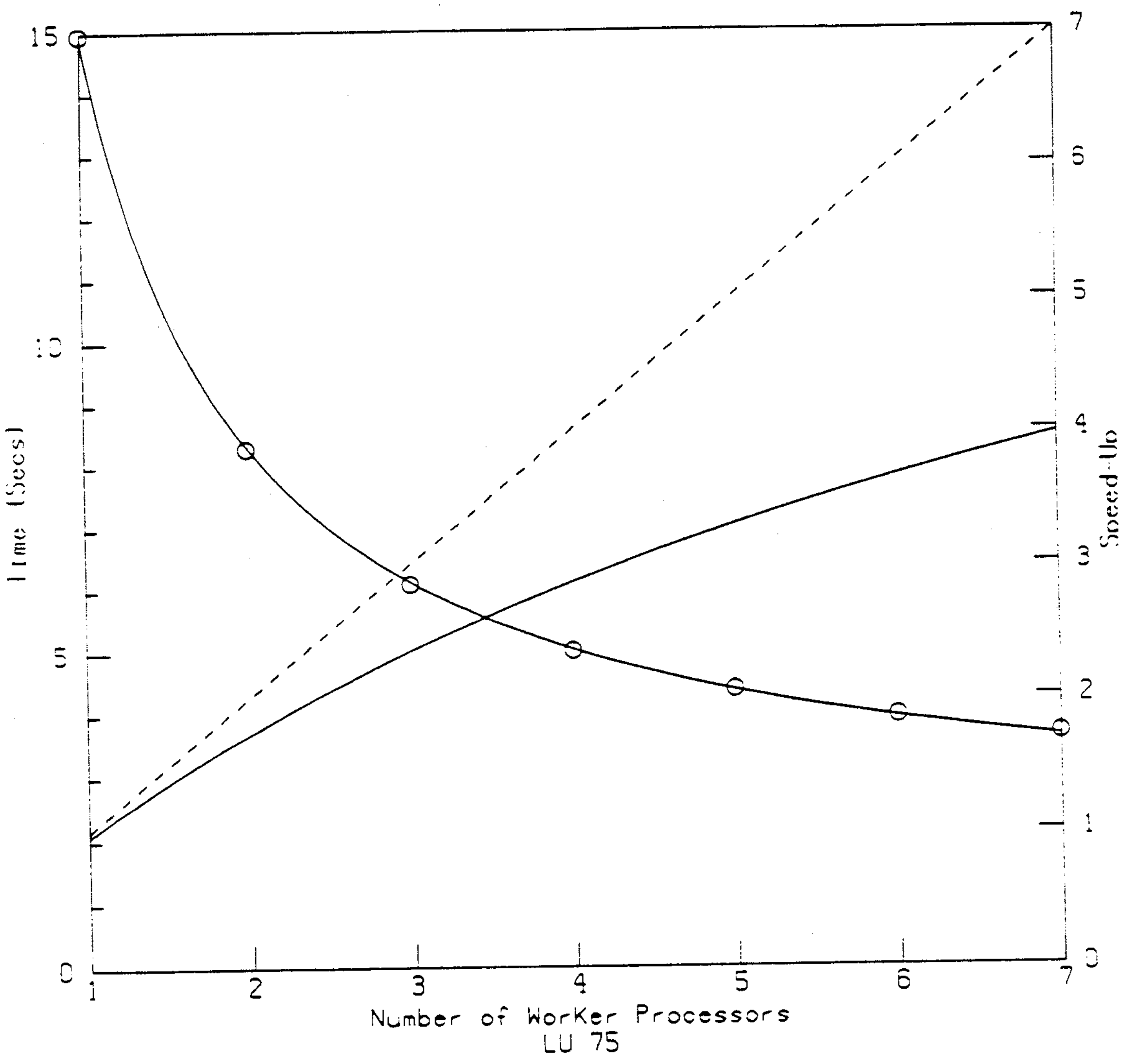

Figure 4.12: Execution time (○) *vs.* number of workers for the LU decomposition of a 75 × 75 matrix. Descending curve is an $a/n + b$ model ($a = 13.18$ secs; $b = 1.74$ secs). Ascending curve is speed-up relative to sequential C. Dashed line is ideal speed-up.

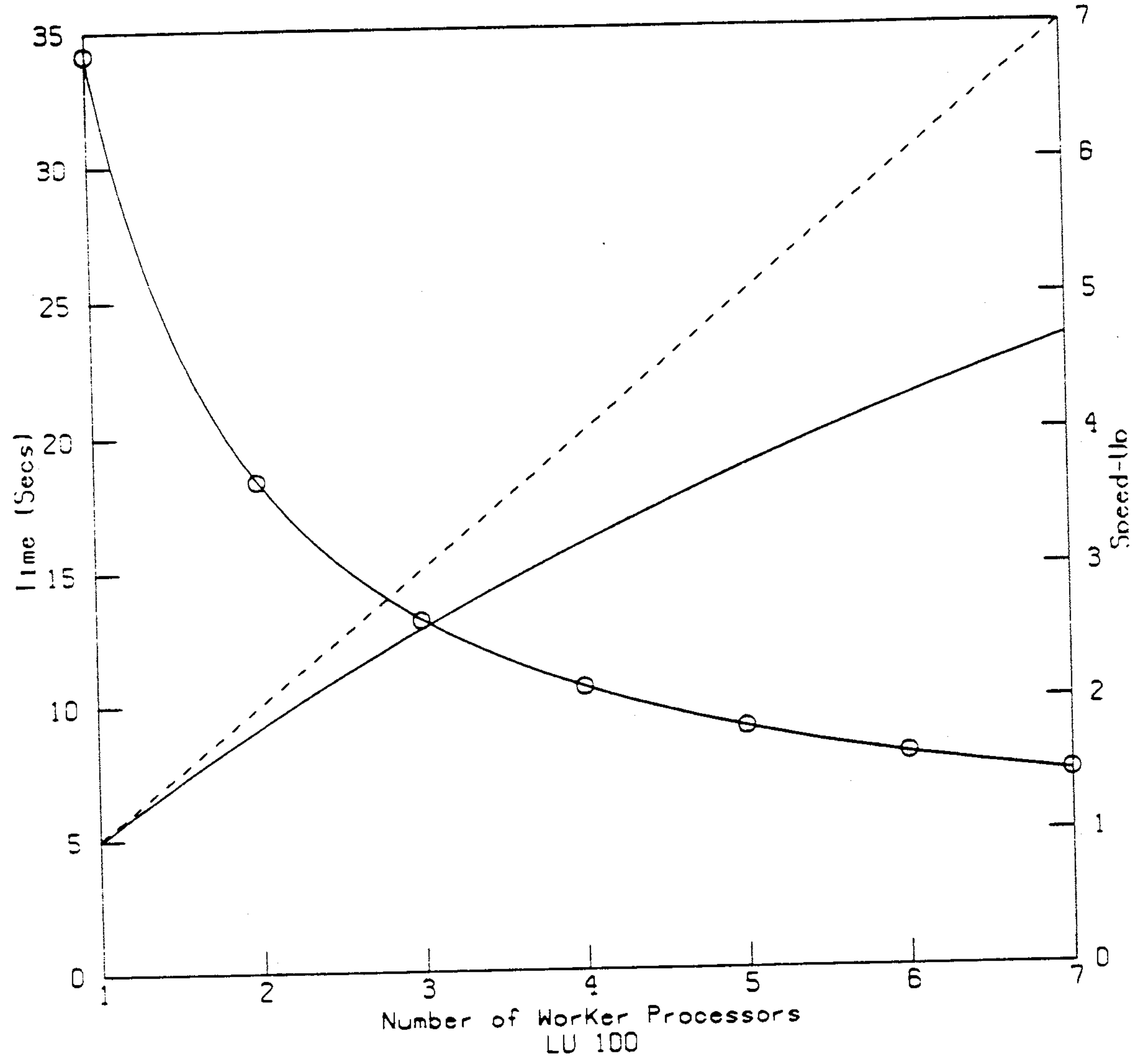

Figure 4.13: Execution time (o) *vs.* number of workers for the LU decomposition of a 100 × 100 matrix. Descending curve is an $a/n + b$ model ($a = 31.39$ secs; $b = 2.73$ secs). Ascending curve is speed-up relative to sequential C. Dashed line is ideal speed-up.

4.3 The S/Net Kernel: Some Conclusions

The work done on the S/Net demonstrates that "real" problems can be parallelized with a Linda system and yield good results. In itself, it has proved sufficient to cast serious doubt on the reservations held by some that, while tuple space was an elegant proposal, it would not be possible to implement it efficiently enough to make it a practical tool. However, given later development of a compile-time system along with developments at Bell on the hardware front, we believe that the days of this system are numbered. Indeed, considerable effort has been spent on the design of a second generation S/Net system.

What effect will the addition of a compile-time system have on the S/Net? It will offer the usual benefits of more pleasant syntax, error detection, improved matching performance, and debugging support. In addition, it will eliminate the run-time ptp formatting costs and lift or ease some restrictions: the first field need no longer be special and the number of fields will be increased. By extending the analysis, though, more can be accomplished.

For example, we hinted above that we might derive more about field usage than we do currently. One application of this new information allows us to avoid copying fields which contain large blocks of data when the tuples containing the fields are replicated on every node. In particular, we can determine which fields in a set are "copy only" and which are "sometimes matched". In the former category are those fields which are always actual on out and always formal on in, while in the latter are those fields that are sometimes formal on in. When a tuple is sent, large (i.e. `char *` or `LINDA_BLOCK`) copy-only fields would not be sent. They would be requested only after matching has succeeded — in fact, they could be sent as part of the delete acknowledgement protocol. At this point in the processing we know the matching template, so a large field can be copied directly off the S/Net bus into user space, avoiding an intermediate stop in the receiving node's tuple space. Given that most large fields are copy-only, a significant reduction in the memory overhead of the replicated tuple space scheme can be achieved. At the same time we reduce the load on the bus and the total number of cpu cycles spent on interrupts across all boards.

We have also mentioned in Section 3.4 the possibility of extending the analysis to issues of communications (e.g. classifying tuples as being "point-to-point" versus "broadcast"). Overall, it is clear that the addition of the existing analysis system will significantly improve performance, while extensions to the analysis system will help pave the way for future development of the S/Net implementation.

Chapter 5

Shared-Memory Implementations

WE have implemented Linda on two shared-memory machines, the Encore Multimax and the Sequent Balance. Linda's tuple space is clearly a natural fit to shared-memory machines, but it does not follow that implementing Linda on a shared-memory machine is trivial. One merely has the luxury of expending less effort on the communication details and more on support. As the communication times fall, the time taken by other portions of the run-time kernel become relatively more important.

Why does one need Linda on a shared-memory machine? Why put a shared-memory model on top of a physically shared memory? First, there are many details to attend to when directly using physically shared memory. Linda takes care of such things as shared-memory allocation and synchronization (i.e. atomic updating). It also offers a simple, yet powerful set of operations that are not native to any existing shared-memory machine and which provide content addressability. Linda operates on collections of data convenient for the user, not the machine. And Linda does all of this in a portable way.

Note too that Encore and Sequent did not initially conceive of their machines as engines for computing a task with multiple threads of control. Instead they saw parallelism at the level of user jobs. Thus they did not supply mature software for supporting parallel programming. Linda fills the gap on both systems.

The first step in developing this implementation was the porting of the S/Net kernel to the Multimax. This required that we remove explicit communication and place tuple space in shared memory. The next step was to design and implement the compile-time analysis system as described in Chapter 3. The final step was the complete revamping of the run-time system to exploit information made available by the analysis phase and our experience with previous implementations.

As in the previous chapter, we will describe the run-time system by first detailing the structure of tuple space and then follow a tuple through the system.

5.1 The Structure of Tuple Space in Shared Memory

The differences between the structure of tuple space here and on the S/Net are due in part to the hardware differences and in part to the influence of compile-time analysis.

The question of how to distribute tuple space on a shared-memory machine is moot: it is placed in a memory segment shared by all processes. This is the main hardware-induced difference. As it happens, tuple space has a much richer structure on the shared-memory systems — but this structure grows out of the compile-time analysis. It could be a feature of an S/Net implementation as well.

The hash table is still with us, but it is one of four different paradigms for storing tuples. All tuples belonging to hash-paradigm sets (or simply *hashed sets*) will be hashed into one common hash table. Having one table per set might seem more natural. However, we do not know *a priori* the number of elements that will be stored in any given hash table. If we make each table large, optimizing for performance over memory, we may waste space. If we make each table small, we conserve space at the cost of poor performance for those sets that have many elements. Using one large common hash table seemed like a reasonable alternative.

The actual field whose value is used for hashing is determined by the analysis and will vary from set to set. This information is conveyed to the run-time system via the ptp structure. Since we place all tuples generated by hashed sets into the same hash table, we have a problem analoguous to one on the S/Net — namely, we need a fast method for screening tuples on a hash-bucket chain so that we execute expensive full-matches on promising tuples only. On the S/Net we used the logical name and the type signature. After a moment's reflection it should be clear that the set identifier (see Section 3.2.4) is at least as good a distinguishing characteristic of a tuple or template, and we use it as a filter here.

There are three other paradigms for managing tuples: queue, tree, and list. Section 3.2.5 describes the patterns of tuple usage that would make these appropriate. Recall that once instances of `ins`, `outs`, etc., had been partioned into sets, each set was examined for fields whose values could be used as a search key.

If no key is needed, i.e. no match is needed, then we use the queue paradigm. On `out` tuples are pushed onto a lifo queue. An `in` pulls the first tuple off the queue if the queue is not empty; otherwise it blocks. A separate queue is maintained for each set that uses this method: to do otherwise would considerably complicate a very simple algorithm.

If there is a key that is always present for every `out` and `in` of the set, we use the hashing method described above. If the key is actual on every `out` but not on every `in`, then we use a binary tree to hold the set's tuples. The tree is ordered by a hash of the value of the key field. Using a hash of the value rather than the value itself avoids unbalanced trees caused by in-order tree insertions and simplifies handling variable size values (`char *` or `LINDA_BLOCK`). If an `in` has an actual in this field, then a normal $O(\log(n))$ tree search is used to find matches; otherwise we walk the tree ($O(n)$).

Note that using a private hash table to represent a tree set would have certain advantages over a tree representation: it is simpler to maintain, for reasonable size data sets it is probably faster [Knu73], and it is more "accessible" — an issue that will be addressed below. However, unlike with hashed sets, a single hash table shared by all tree sets would have problems. We know that a set in this class includes at least one `in` that has a formal in the key field. This `in` would require an exhaustive search of the hash table. If each set had its own table, the search would not be so bad: we only touch tuples that we would have to look at anyway. But with a shared table, we would potentially examine a large number of tuples that could not be of interest.

If no key is available, we use a list. On `out` a tuple is added to the list; on `in` the list is scanned for a match. If no match is found the `in` blocks. When the `in` next attempts to find a match, the entire list need not be rechecked. If we always add tuples to the head of the list, we can use some marking technique to avoid rescanning old portions of the list.[1]

[1]Such a technique is used in both kernels for `ins` blocked on a hash bin list.

Actually the code to implement this paradigm has never been written. The preprocessor does generate calls to run-time routines that are suppose to handle this paradigm, but these routines do not exist. As we mentioned in Chapter 3, sets using this paradigm arise from a style of programming (using formals in `outs`) that we have never used.

5.1.1 The Structure of Tuples

Building on our experience with the S/Net, we also changed the structure of tuples. The S/Net's tuple space represents tuples as a linear chain of blocks in which the tuple's data is written sequentially. This turned out to be rather messy. Random access to fields was impossible, extra effort was needed to keep from falling off the end of a block, managing two different sizes of blocks was cumbersome, and so on. We still needed a way to manage shared memory, and having the block allocation and freeing code at hand, we stuck with it. But by altering the structure of tuples, we were able to make tuple-storage management much cleaner.

The tuple header became quite similar to the ptp. It is a fixed size structure with space for various pieces of information such as a set id, a unique tuple id (*tid*) and descriptors for 16 fields. Each field descriptor has space to hold information about the field, a pointer to the formal in case of a formal and an actual value (`int`, `float` or `double`) or a pointer to a chain of blocks that holds the actual value (`char *` or `LINDA_BLOCK`) in the case of an actual. (That is, the header is the root of an n-ary tree ($0 \leq n \leq 16$), with each branch being a chain.) It is now simple to find any arbitrary field. Variable length data always begins at the start of a block, and no two fields have data in the same block (or chain, for that matter). This design considerably simplifies the manipulation of tuples. Finally, the S/Net's different sizes of blocks arose from attempts to improve communication performance over the bus. No such considerations hold here, so we eliminated the larger size. Thus an extra data block is exactly the same size as a header block, and both come from the same free list. To avoid a potential bottleneck, each process maintains a small pool of free blocks. When a process's pool is empty, it is refilled from a global reservoir.

5.1.2 Locking Up Tuple Space

The bus arbitration logic and interrupt masking were used to implement atomic operations on the S/Net; spin locks are the analogous primitives for the shared-memory systems. Both the Multimax and the Balance have hardware support for test-and-set operations that are used to implement spin locks.

Unfortunately, the locks are not equally easy to use. The Sequent Balance has only a limited number of locks and they are managed as a typical UNIX device. The Multimax supports locks on any byte in memory and the locks are managed the same way memory is. The Sequent's only apparent advantage is that there is no subroutine overhead in actual lock operations (they are carried out as a result of normal read or write operations by special hardware associated with the lock devices); the Multimax scheme requires function calls, but only because the test-and-set instructions are apparently not supported by the C compiler. Since locks on the Multimax occupy normal memory, normal cache maintenance allows programs to spin on locks in cache without soaking up bus cycles. This is not the case with the Balance,[2] and this disadvantage probably more than offsets the subroutine

[2]Sequent does suggest a software technique (*shadow locks*) to alleviate this problem — at the cost of greater code complexity and run-time overhead.

advantage.

We assumed the lowest common denominator and designed a shared-memory system to conserve locks. This introduces a tradeoff: on the one hand, try to use a small number of locks; on the other, the more locks, the more processes can simultaneously access a data structure in that the structure can be locked on a finer grain.

For example, take the hash-table paradigm. At the extremes, we could use one lock for the entire table, or we could use one lock per bin. The first case conserves locks, but at the cost of serializing access to the hash table. The latter uses a lot more locks but allows as many processes as there are bins to be touching the hash table. A middle course is possible in which we group bins and assign one lock per group; the coarser the groupings, the more likely that the system will suffer from contention.[3] Given that we opted for one large common hash table, we did use one lock per bin.

With an eye to improving parallel access to tuple space, we wrote a preliminary kernel (similar to the S/Net's) that implemented the readers/writers paradigm for accesses to each hash queue: multiple read accesses (e.g. match searches) or only one write access (e.g. a tuple insertion) was permitted at any given instant on any given queue. A number of contention measurements convinced us that this scheme was performing poorly. The overhead, and in particular the accessing cost for the larger number of locks needed to implement this scheme, was not offset by the benefits of increased access to tuple space. This conclusion may in part be the result of the relatively few nodes on the machines we use. Machines with more nodes will require us to devote more thought to this issue.

Contention can be especially serious with a structure like a tree. Some accesses to the tree require traversing the tree until a match is found. If there is only one lock on the tree, this potentially expensive operation can cause a bottleneck. In the worse case, the out process that will ultimately provide the matching tuple is waiting in the backlog. Considerations such as these make the private hash table representation even more attractive for sets that currently use trees. We will present data in Section 5.4.3 that suggests that private hash tables are in fact a better choice.

We use one lock per queue for the queue paradigm. At one lock per queue (and hence per set), the lock costs are significantly less than for the other paradigms.

5.2 The Execution of a Linda Program

To follow the action in the shared-memory system, we must start at compile time. Consider, once again, the case of two processes:

```
A:
    in("cat", ? &i)

B:
    out("cat", 3)
```

where we assume that A executes in before B does out.

[3]Using a simple model, one can see that what really matters is the total number of locked bins across all the hash tables, assuming that references to any bin in any hash table is equally likely. This observation leads to one more scheme: as the number of hash tables increases dynamically, multiplex a pool of locks over all the bins. A simple way to do this is to assign one lock to all bins whose indices are congruent modulo the size of the lock pool.

The ptp structure has been generated at compile time, not at run time as was the case for the S/Net. The preprocessor produced a file containing a ptp description of every tuple space operation. The code generated to replace a particular operation references the appropriate descriptor.

At run time the variable portions of the data structure (actual, non-constant data or the addresses of formals) are updated, and a pointer to the structure is passed to a routine which implements the in handler for the appropriate paradigm. If the only operations of interest are the ones listed above, the paradigm selected will be the queue paradigm and the code executed will be:

```
(_lp_0000.field0.formal.d = &i,
in_queue(&_lp_0000, 0, 1));
```

_lp_0000 is the name that was assigned to the ptp descriptor for this call by the preprocessor. in_queue() implements in and rd operations for the queue paradigm. The extra arguments to the call select between an in, rd, inp, or rdp.

in_queue() checks the queue holding the tuples for the set of which _lp_0000 is a member (the set id is stored in the structure). In this example the queue will be empty and the process will block.

We use a slightly more sophisticated approach to process blocking then we did on the S/Net. The in process queues itself on a list of processes waiting for a tuple belonging to the set, then spins on a lock it owns. An out frees queued processes by toggling their locks. (With a little more information, we might free only the first process if it was known to be waiting on an in or inp.)

Process A is now spinning. B executes:

```
(_lp_0001.field0.actual = 3,
out_queue(&_lp_0001));
```

where _lp_0001 is the name of the out's ptp descriptor. out_queue() copies the information in the ptp to a tuple header block and installs the tuple on the queue.[4] It then checks the waiting-process queue, at which time Process A is freed from its spin loop, while process B has completed the out and returns.

The intent of this queue-based blocking structure is to lessen contention for shared data structures. Since every tuple-ordering structure carries a lock, the safe way to access such structures is to acquire the lock, check the structure's status, and then release the lock. Serious performance degradation would result from processes which repeatedly grabbed and then released a stucture's lock. Given our method, each process spins on its own lock, thereby avoiding contention for a shared resource.

In fact, the need to do this is less clear in the case of the queue paradigm. A process could "peek" at the queue pointer, waiting for it to become non-zero. This is unsafe, but could probably be made to work. In general, however, "peeking" would be much harder with more elaborate structures like a hash table or a tree.

The queue of waiting processes was also a convenient place to put hints that can be used to speed matching in more complex cases. For example, when an in using the hash paradigm blocks, the hash value it is interested in is stored with information about the in process on the queue of processes waiting for the same hash bin. An out will free only

[4]Actually, we could optimize here by first checking the process queue associated with this tuple queue and, if there is a process waiting to in or inp, copying the data to it directly.

processes which are waiting for the bin and whose hash values match that of the out's tuple.[5] One additional advantage of the queue-based blocking method is that it provides a framework for a true Linda scheduler.

Continuing with the example, the newly freed process A now removes the tuple from the queue, calls a copy routine to update the formal, and then returns from the in.

Delete synchronization rests on the primitive spinlock operations. If more than one process is notified by an out that a tuple of mutual interest has been generated, the in process that first acquires the lock of the structure holding the tuple will get the tuple.

5.3 Debugging Features of The Shared Memory Systems

As we pointed out in Chapter 3, the compile-time system supports some error detection and debugging. We saw there how some suspect code (unmatched ins or outs) could be detected. Here we will say a few words about run-time debugging.

When a ptp structure (the _lp_XXXXs in the above) is built at compile time, the ASCII text of the call is included at the end of the structure. The Linda run-time system uses this added information to support debugging via a collection of macro calls:

```
LINDA_TRACE_ON(OFF)
LINDA_DD_ON(OFF)

LINDA_LIST_ON(OFF)
LINDA_ACTUALS_ON(OFF)
```

The first two macros turn on (off) tracing and deadlock detection. Tracing provides a log of tuple space operations. "Deadlock" is defined in a limited way here: it means that every process is blocked at an in or rd.[6]

The last two macros indicate what information should be provided to the user. The default is to list a source code line number and a process id. These macros indicate that the text of the call or the value of the actuals should also be included.

This relatively fine control over output format is intended to simplify the design of the interface between the run-time debugging system and a postmortem movie system which replays tuple space activity as recorded in debugging traces. movie displays a number of vertical strips, each with a dot, which are used to represent a process. Each time a process executes an operation, its dot is moved along its strip in proportion to the operation's position in the source text, and additional information about the call is displayed in an information window.

While movie currently plays back hand-coded toy scripts, a good deal of work remains to be done before it is fully interfaced to the run-time debugging system.

[5] There is a many-to-one mapping of hash values to hash bins. A 32 bit hash value is always generated even though only the low order 10 bits are used to identify a bin. Given this fact, it makes sense to compare the *entire* hash values of the template and tuple as a quick filter.

[6] While the current system does detect deadlock (in this sense), it also detects it in no sense whatsoever, i.e. it declares deadlock even when the code is not deadlocked. The debugging of the debugging code continues.

5.4 Encore Performance Results

In this section we will present Encore timings for the various test programs discussed in Section 4.2. While all the programs to be discussed will run on the Balance (and most have been run on the Balance) we will present timings only for the Multimax. The Balance performance is roughly similar.

The Multimax at Yale has 16 processors and 16Mbytes of memory. Each processor is a NS32032 with a floating-point coprocessor. Most of the timings were taken with no users on the machine. However, since the Multimax is set up as a time-sharing system running UMAX (a UNIX-like system), there are a number of background processes that periodically use cpu cycles and memory bandwidth. Performance is also affected by paging cost and by the tendency of the scheduler to migrate processes over a number of processors.

We used UMAX's `gettimeofday()` as a timer. The results it produced were similar to those obtained from a microsecond clock supported by a special library call in UMAX. These results tended to be more consistent than higher-level timers such as `csh`'s `time`. On the whole the timings were far less repeatable than those taken on the S/Net — probably due to the performance factors listed above. Three loops were timed to obtain some basic measures of a node's performance. Each loop had the form:

```
for (i = 0; i < LIMIT; ++i) {
    <body>
}
```

One iteration took 3.21 μsecs for a null loop body, 3.72 μsecs for "`sum += i;`" as the body, and 12.75 μsecs for "`foo();`" as the body. `i` and `sum` where declared `int`. The compiler placed these variables in registers. `foo()` was the null function "`{;}`".

5.4.1 Single Process Timings

As before, we begin with the time taken by the simplest form of out. The following code was used on the Multimax:

```
lmain()
{
    register long i;

    start_timer();
    for (i = 0; i < 4900; ++i) {
        out("a");
    }
    timer_split("Done.");
    print_times();
    in("a");
}
```

We ran these tests with a kernel that had room for 5000 tuples, so the loop limit must be less than 5000.

The entry point for Linda-C programs is called "`lmain()`". This was the most portable way to hide the initialization code. The Multimax and Balance disagree on `crt0`, the routine typically called on UNIX systems to start a C program. They agree that `main()` will be the default entry point, so we co-opted `main()` for our initialization routines. The user is required to start at `lmain()`.

Note the `in("a")` tacked onto the end of the program. Without it, the analysis routine would identify the **out** as having no matching **in**, and the simplest tuple-management paradigm (the queue method) would not be used.

The 4900 iterations took .60 secs (after adjusting for loop overhead), so a simple **out** executes in about 120 μsecs. The average simple out[7] executes 50 instructions:

16 Allocate and initialize block.

16 Manage queue's lock.

9 Add to queue and check for waiting processes.

7 Entry and exit from out routine.

2 Check for debug flag.

Keep in mind that this program uses the simplest possible tuple. More complicated tuples will incur additional cost for argument processing and for maintaining appropriate tuple storage structures.

By moving

```
in("a");
```

from the end of the program into the loop, we can measure the time taken by an `out-in` and then deduce the time needed for an in. Running with the loop limit set to 100,000, this program took 20.9 secs, or approximately 210 μsecs for an `out-in`. So an in takes about 90 μsecs. The corresponding time for a simple **rd** is 80 μsecs.

Actually the subtraction is not quite justified. The **in** insures that the process's block pool is never empty, so the refill routine is never called. A typical in executes 44 instructions:

16 Manage queue's lock.

9 Free block.

7 Entry and exit from **in** routine.

6 To remove block from queue.

2 Check for debug flag.

2 Check for **rd** flag.

2 Check for copy.

So it takes 94 instructions for an `out-in` transaction. Prorating the time accordingly, the **in** accounts for about 100 μsecs and the out 110 μsecs. The new out figure leads to a **rd** time of 90 μsecs.

We timed the following program to see how out time varied with the amount of data in a tuple.

[7]Recall that each process maintains a small pool of free blocks. Every so often an out will find the pool empty and have to call a routine to fill it. Such an out is not average. Thus the time given for an out reflects the "normal" processing of an out plus a portion of the time needed to refill the pool. A trace of the refill routine indicates that it uses about 7 instructions per block.

```
#include "linda.h"    /* Defines LINDA_BLOCK */

long a[200];

lmain(argc, argv)
int  argc;
char **argv;
{
    register long i;
    LINDA_BLOCK   A;

    A.data = a;
    A.size = atol(*++argv);
    start_timer();
    for (i = 0; i < 1600; ++i) {
        out("a", A);
    }
    timer_split("Done.");
    print_times();
    in("a", ? &A);
}
```

The loop limit of 1600 is roughly one third of 5000, the size of the tuple block pool. Each of the tuples in the above program will take up to three blocks: one for the header and up to two more for data. An extra block (which is the same size as the header block) holds 77 longs. We ran the test with blocks of up to 110 longs, so we needed to provide for two extra blocks per tuple.

Figure 5.1 shows a graph of execution time versus block length for this program. On the S/Net there was a discontinuity, as one would expect, when one tuple block's data space was exhausted and another had to be allocated. On this system, the discontinuity occurs between 70 and 80 longs. The figure indicates that outing the header and first extra block requires 290 μsecs and each long adds another 2.4 μsecs. An additional extra block adds another 110 μsecs. If we assume that the 110 μsec figure applies to the cost for the first extra block, then we conclude that the cost of the header block in this case is about 180 μsecs as opposed to 110 μsecs for a simple out. The difference between these figures is due mostly to the cost incurred in processing an extra field.

Extending this test to an out-in yields the data presented in Figure 5.2. Processing the header and first extra block now requires about 490 μsecs and the second extra block takes 190 μsecs; the cost per long is 5 μsecs. Taken together with previous results (and making the same assumption of equality of processing the first and second extra blocks) we see that in takes 120 μsecs for the header, 80 μsecs for an extra block and 2.6 μsecs per additional long.[8] Once again, the increase in time to process the header compared to a simple in is largely due to the extra field. The time per block for in is smaller than for out because in spends no time initializing the block. That the time per long is the same is no surprise: for the out this is the time needed to copy from the user's data space to tuple space, while for the in it is the time needed to copy in the opposite direction. We mentioned in Chapter 4

[8]The same sort of adjustment that we discussed earlier for the "pool" effect should be made. However, the adjustment made little difference before and will make relatively less difference here.

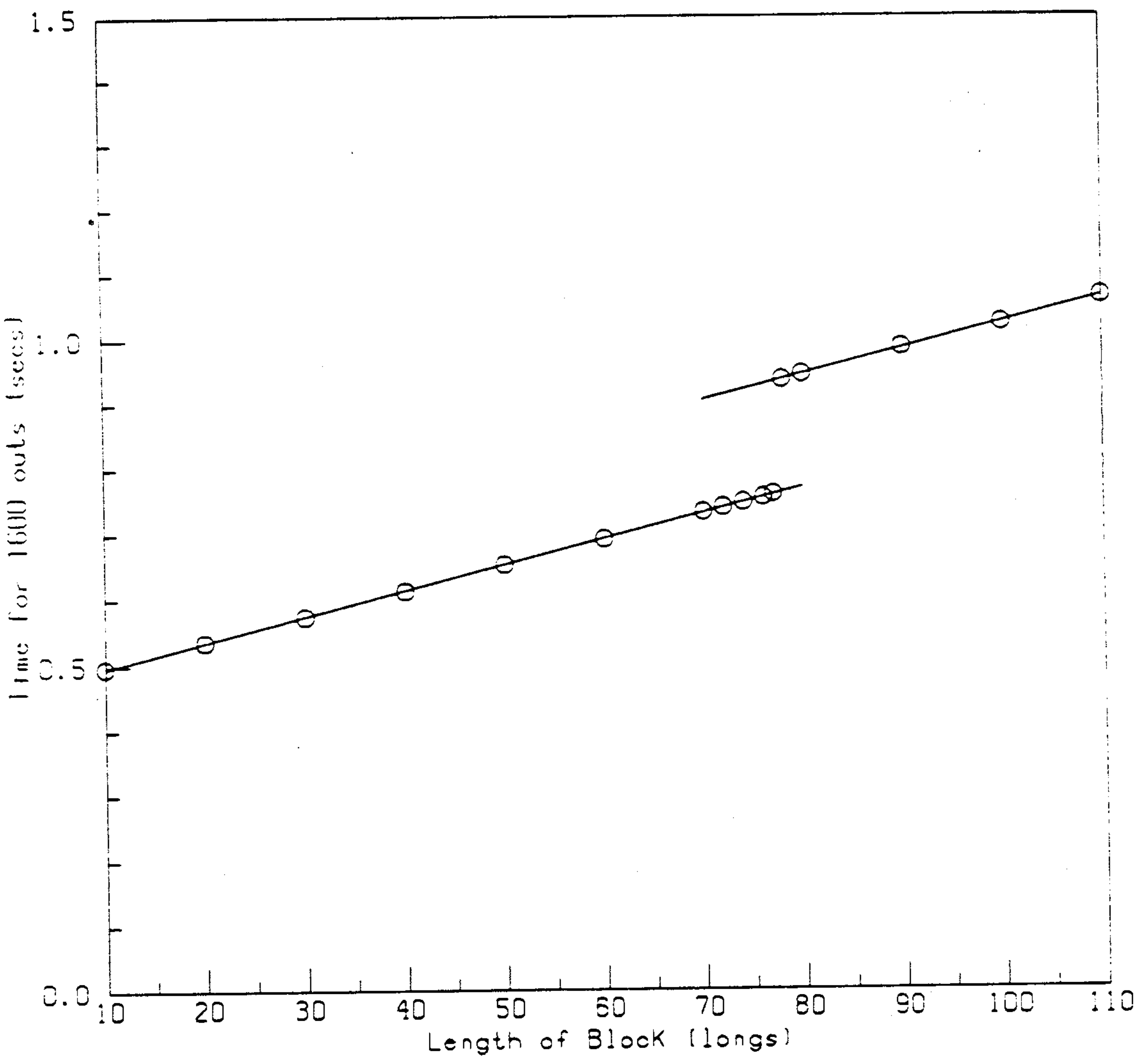

Figure 5.1: Execution time *vs*. block length for 1600 executions of out("a", block).

that there was a tradeoff between increased efficiency and increased internal fragmentation cost in selecting a block size — this tradeoff also occurs for our shared-memory system. This data suggests that the size we chose (i.e. the size of the header block) is probably about right. The asymptotic overhead cost per extra block is 190 μsecs (i.e., the cost for processing an extra block) while the cost of actually copying the data (the "real" work) is 385 μsecs (77 longs × 5 μsecs per long). This amounts to an overhead of about 33%. Lowering this to 10% would require a tuple size of about 340 longs which would likely result in very high internal fragmentation costs. (Our experience suggests the block sizes on the order of hundreds of bytes rather than thousands will be the most commonly used).

5.4.2 Multiple Process Timings

Now that we have an idea of execution times for the basic operations, we will move on to execution times when two or more processes are involved. We used a "toss-and-catch" program to measure one-way transactions (from **out** in one process to **in** in another):

```
lmain()
{
    register long i;

    eval(catch());
    start_timer();
    for (i = 0; i < 500000; ++i) {
        out("a");
    }
    in("done");
    timer_split("Done.");
    print_times();
}

catch()
{
    register long i;

    for(i = 0; i < 500000; ++i) {
        in("a");
    }
    out("done");
}
```

Execution averaged 71.9 secs, or 140 μsecs per transaction. Unlike the S/Net, there are no underlying kernel interrupts to worry about so almost all **out** and **in** processing is done concurrently. We would therefore expect the execution time to be about the same as the maximum of the time for **out** or **in**. The measured time is slightly greater than the maximum (110 μsecs for **out**). The increase may be due to lock contention: occasionally the **in** will have locked the queue (this program uses the queue paradigm for tuple management) when the **out** attempts to execute, causing a minor delay (see Section 5.2 for comments about "peeking" at queues and below for contention measurements). There is also some minor cost associated with inserting (by **in**) and removing (by **out**) a process blocked on an

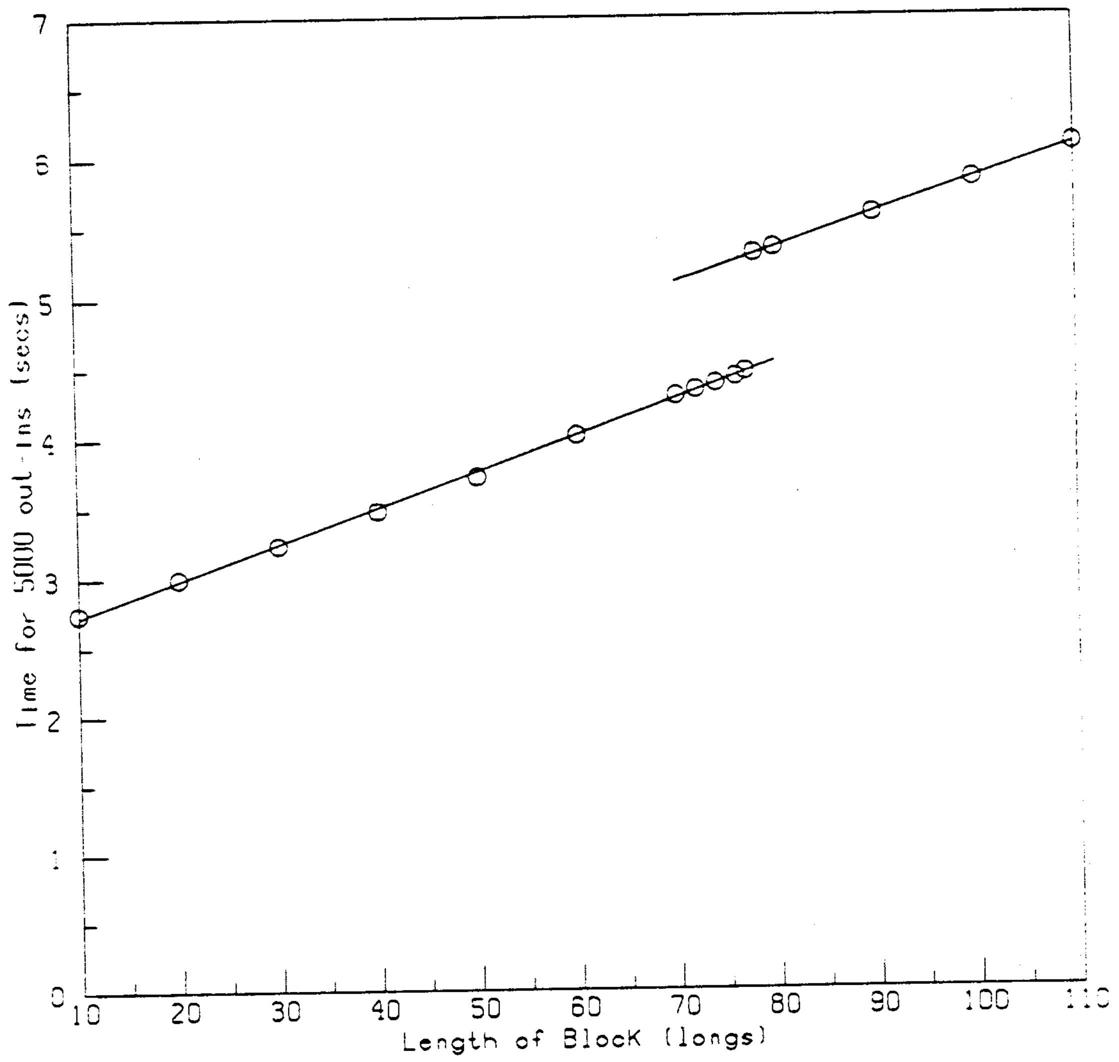

Figure 5.2: Execution time *vs.* block length for 5000 executions of out-in pairs with variable-size data.

in from a waiting list for the tuple queue. Note that the former is not a "serializing" cost (unlike the delay due to locking) and should be masked, as is the rest of the in processing, by the time needed to process the out.

The round trip time was obtained by timing the following program:

```
lmain()
{
    register long i;

    eval(pong());
    start_timer();
    for (i = 0; i < 100000; ++i) {
        out("a");
        in("b");
    }
    timer_split("Done.");
    print_times();
}

pong()
{
    register long i;

    for(i = 0; i < 100000; ++i) {
        in("a");
        out("b");
    }
}
```

This code completely serializes the transactions. After the first cycle, whenever an out executes, the corresponding in will be waiting — in particular, there will be no delays because of lock contention. We would expect that the above would result in a transaction time very close to the single process case. Actual execution time was 41.4 secs, giving a figure of about 210 μsecs per transaction, virtually the same as for a single process transaction.[9]

We modified this experiment in several ways to test for contention problems. First we ran multiple instances of the code concurrently. Each process pair used a different set of tuples. Since there are no global tuple space management costs in the Multimax system and tuple contents had been changed to avoid lock contentions, it is not surprising that from one to four pairs of processes ran with no significant difference in execution time. This result merely suggests that there are no "hidden" contention problems (for example, access to the tuple block pool).

To assess the magnitude of contention costs that we know will occur, we ran multiple process pairs with all pairs using the same two tuples. Figure 5.3 presents the results of some runs. The combination of the sequential nature of the programs and the non-deterministic effects of lock contention make detailed analysis of this data difficult. Overall it is clear that

[9]This close agreement (like a similar one in Chapter 4) is somewhat of a coincidence. Two minor factors apparently offset each other here: since the in is waiting, its subroutine entry and initialization costs are masked by out processing; on the other hand both the out and the in must execute a few more instructions to manipulate the waiting process queue.

while contention for locks is noticable, even in this "worse case" it is not crippling. Note that, as on the S/Net, the total transaction rate rises as we add pairs (a four-fold increase in transactions causes only a 70% or so increase in execution time).

1 Pair	2 Pairs	3 Pairs	4 Pairs
8.4	8.3	10.2	12.6
	9.1	10.5	13.1
		10.7	13.2
			14.5

Figure 5.3: The effects of contention for a queue lock on the execution time (secs) for 20,000 iterations of ping-pong as the number of process pairs increases.

5.4.3 Matrix Multiplication

We discussed two applications in Chapter 4, matrix multiplication and LU decomposition. Here we present results for the same applications on the Multimax.

The Multimax Linda-C version of the matrix multiplication program is algorithmically similar to the S/Net's. The versions differ chiefly because of the cleaner syntax supported by the preprocessor. This Multimax version is also a little less general (see Figures 5.4 and 5.5). The task does not specify identifying names for the matrices, and `dim` is "passed" to workers via the backdoor — it is initialized before the `fork`s implicit in the `eval`s are executed. There is no good reason for either of these changes.

As was the case for the S/Net, we ran the matrix code using single-precision floating-point matrices of dimension 50, 75, and 100. The Multimax's floating-point support is about three times faster than the S/Net for double precision computations and five times faster for single precision computations. The Linda operations run about five times faster on the Multimax than on the S/Net, thus we expect Linda overhead to be about the same for matrix multiply which uses single precision arithmetic. We see this in Figures 5.6, 5.7, and 5.8.[10] The solid descending curves are fitted $a/n + b$ curves and the open circles are measured times. Speed-up is given by the ascending curves, while the dotted lines represent ideal speed up. The times indicate that the Linda system permits nearly linear speed-up through 10 worker processes.[11] In all cases just two workers were needed to improve on uniprocessor time.

This matrix program has one major source of overhead not yet mentioned. Since the matrices are stored in tuple space, some time is spent copying data to and from tuple space. This is not logically necessary on a shared memory machine. We rewrote the program to place arrays in shared memory, and pass pointers via tuple space. The results are shown in Figure 5.8 plotted with diamonds: for a fairly large matrix, copying can account for about

[10] In these and later graphs, we have omitted the costs of `eval`ing the workers. We did this partially because we wanted to pretend we were in an environment where `eval`ed processes were much cheaper than a UNIX `fork`, and partially because including the cost would obscure the performance results as the number of processes increased. In the latter case, for a small problem, the decrease in time due to increasing the number of workers from 8 to 9 could easily be offset by the cost of the `fork`.

[11] We mentioned earlier that the Multimax timings are subject to fluctuations due to interference from the operating system. This problem grows worse as the number of processes in a computation increases. This probably accounts for the wiggles in the measured data in the region of 8 or more workers.

```
#include "linda.h"

long dim, workers;

lmain(argc, argv)
int  argc;
char **argv;
{
    float       col[256], result[256], row[256], true_result;
    long        index, row_index, col_index;
    LINDA_BLOCK COL, RESULT, ROW;

    workers = atol(*++argv);
    dim = atol(*++argv);

    start_timer();

    /* start workers */
    for (index = 0; index < workers; ++index) eval(worker());

    generate_test();

    timer_split("done setting up");
    /* out the rows and cols of the multiplicands.
       Note: the test case uses matrices with
       identical rows and cols. */
    for (index = 0; index < dim; ++index) {
        out("row", index, ROW);
        out("col", index, COL);
    }

    out("task", 0);

    /* in results. */
    RESULT.data = result;
    for (index = 0; index < dim; ++index) {
        in("prod", index, ? &RESULT);
    }
    timer_split("all done");
    print_times();
}
```

Figure 5.4: Matrix multiply — master.

```
worker()
{
    long        col_index, index, next_index, row_index;
    float       *cp, col[256], dot, result[256], row[256], *rp;
    LINDA_BLOCK COL, RESULT, ROW;

    /* Initialize LINDA_BLOCKS. */
    COL.data = col;
    RESULT.data = result;
    RESULT.size = dim;
    ROW.data = row;

    while(1) {
        /* Generate a new task, use -1 to signal end. */
        in("task", ? &row_index);
        if (row_index < 0) {
            out("task", -1);
            exit(0);
        }
        next_index = row_index + 1;
        if (next_index < dim)
            out("task", next_index);
        else
            out("task", -1);

        rd("row", row_index, ? &ROW);

        for (col_index = 0 ; col_index < dim; ++col_index) {
            rd("col", col_index, ? &COL);

            dot = 0.0;
            rp = row;
            cp = col;
            for (index = 0; index < dim; ++index, ++rp, ++cp) {
                dot += *rp * *cp;
            }
            result[col_index] = dot;
        }
        out("prod", row_index, RESULT);
    }
}
```

Figure 5.5: Matrix multiply — worker.

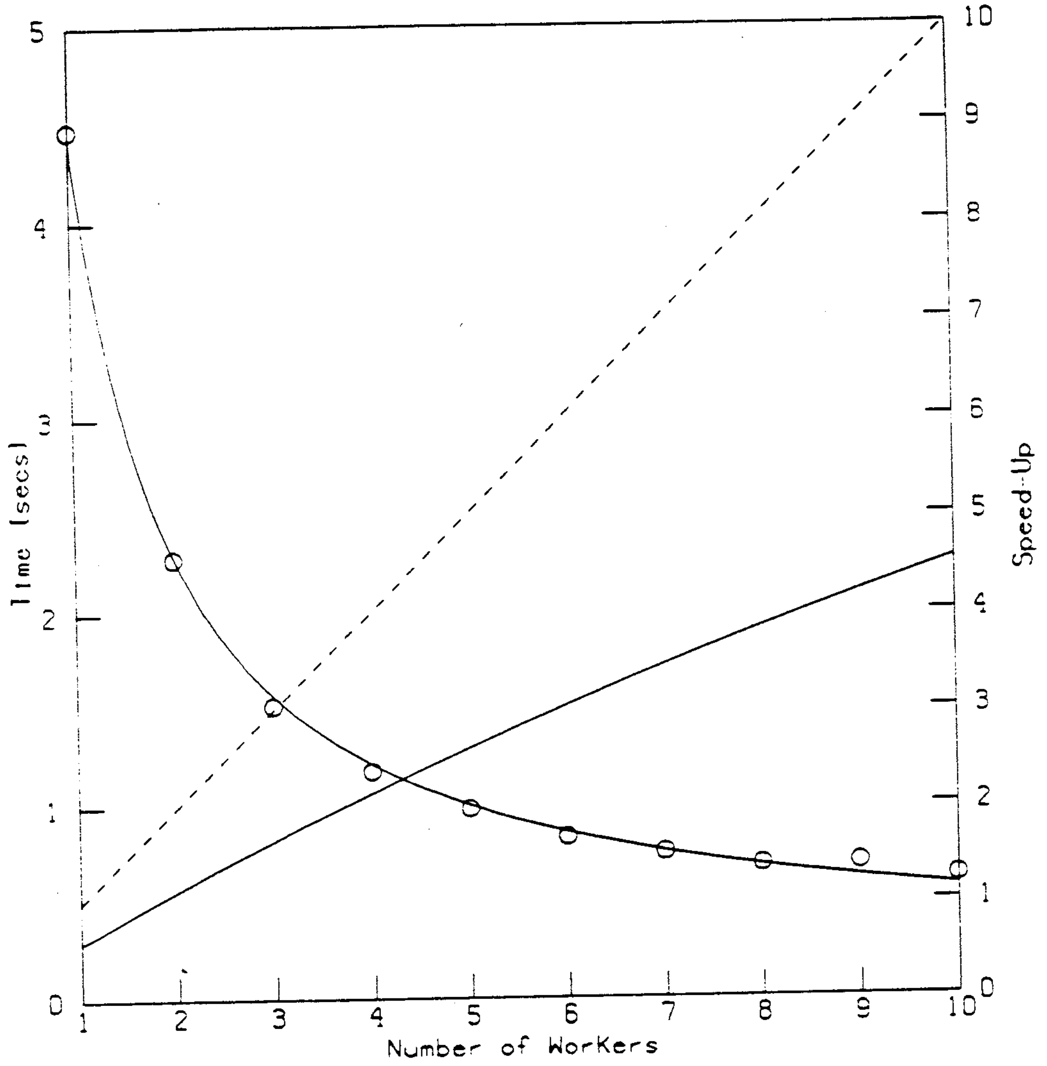

Figure 5.6: Execution time (∘) *vs.* number of workers for a 50 × 50 matrix multiplication. Descending curve is an $a/n + b$ model ($a = 4.30$ secs; $b = .143$ secs). Ascending curve is speed-up relative to sequential C. Dashed line is ideal speed-up.

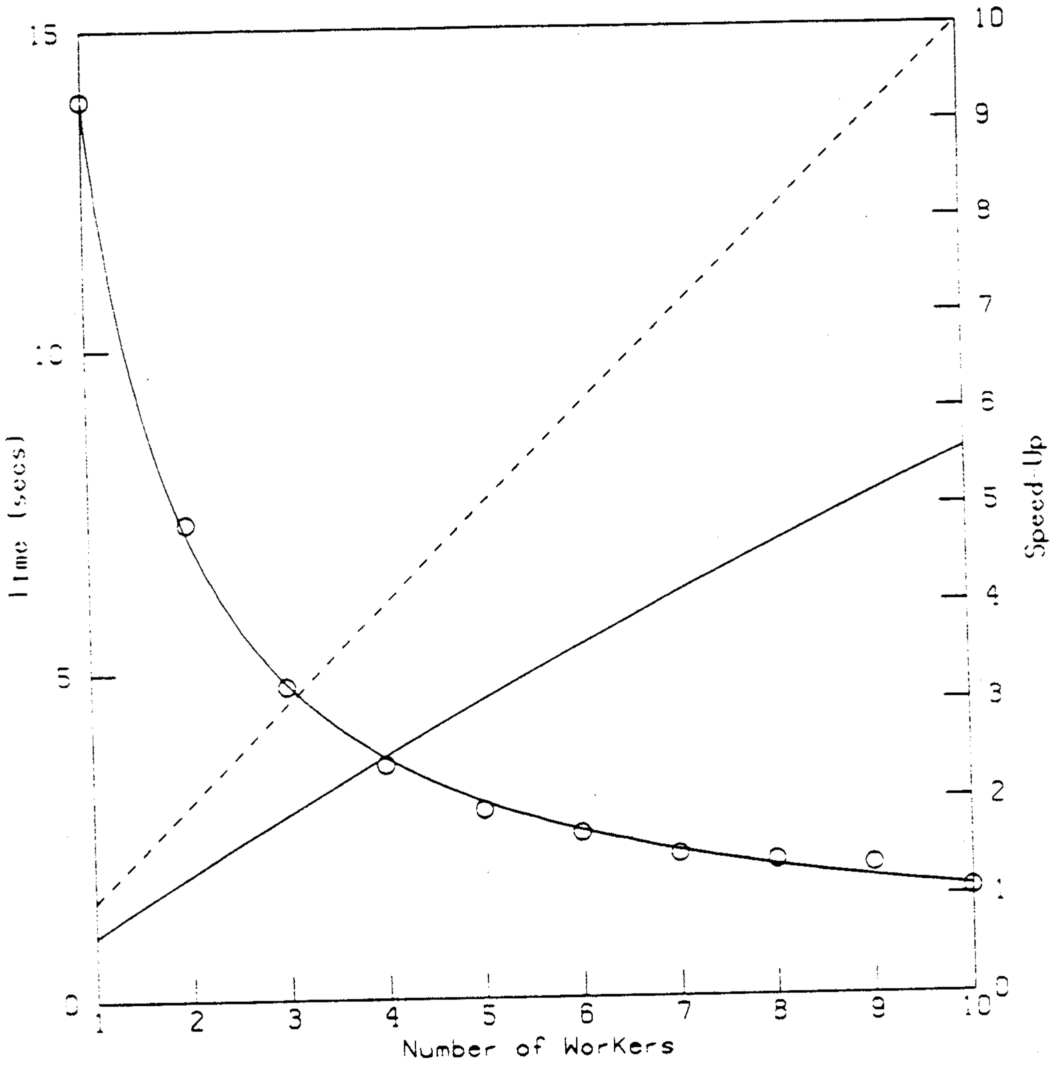

Figure 5.7: Execution time (o) *vs.* number of workers for a 75 × 75 matrix multiplication. Descending curve is an $a/n + b$ model (a = 13.7 secs; b = .267 secs). Ascending curve is speed-up relative to sequential C. Dashed line is ideal speed-up.

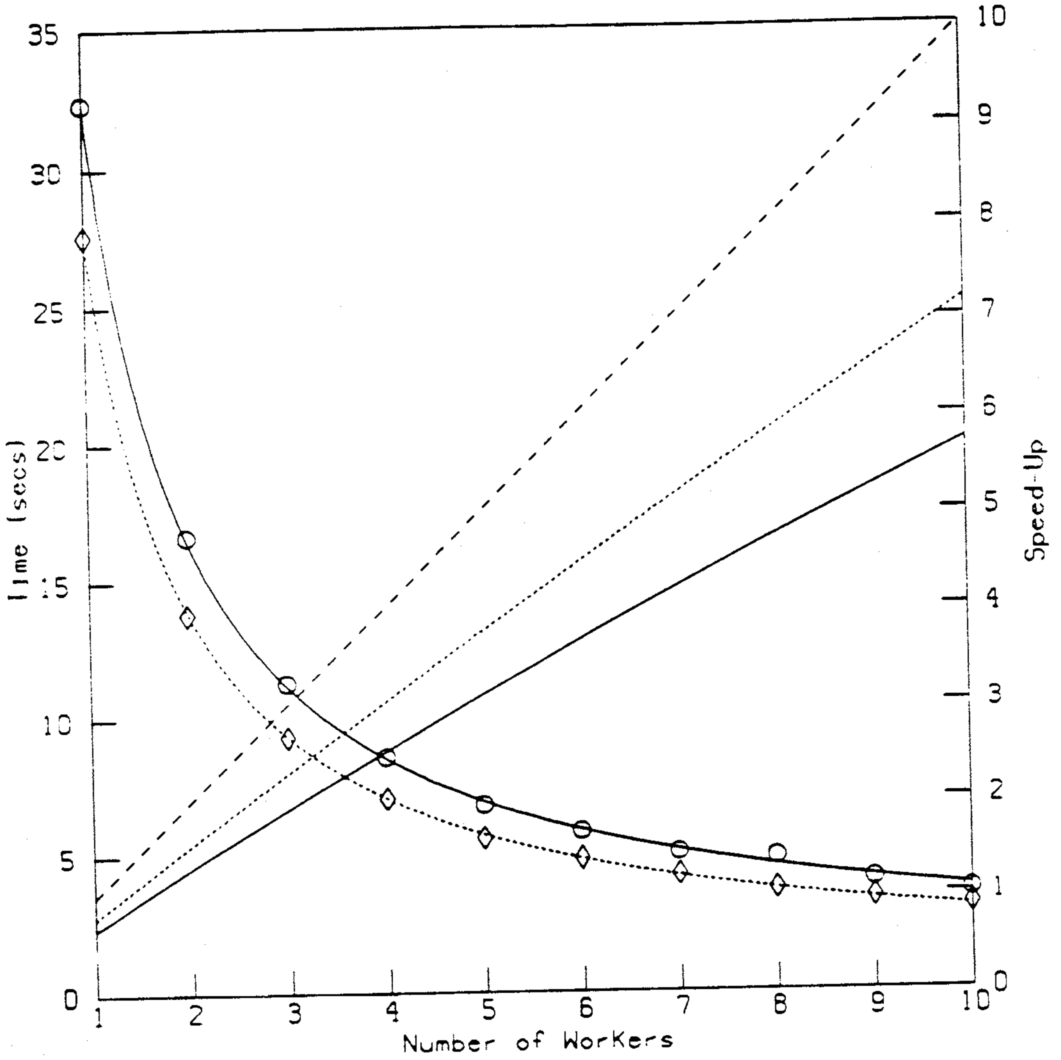

Figure 5.8: Execution time (∘) *vs.* number of workers for a 100 × 100 matrix multiplication. Descending curve is an $a/n + b$ model ($a =$ 31.9secs; $b =$ .553secs). Ascending curve is speed-up relative to sequential C. Dashed line is ideal speed-up. Execution times (⋄) are for the non-copying version.

half of the overhead. Of course, it is not really fair to mix direct use of physically shared memory with Linda: the results are machine-dependent and aesthetically displeasing. But in this case, there is a more reasonable alternative. Since, for $A \times B$, matrix B is needed in its entirety to compute any task, it makes sense to have every worker **rd** the columns of B just once, storing B in local memory for future use. Not only does this reduce copying in a machine-independent fashion, it also reduces the number of tuple operations. Making a copy of B for every worker will use more memory, and will probably make it desirable for a worker to concentrate on tasks from one matrix multiplication (as opposed to the S/Net version, in which workers can work equally well on a task from any ongoing multiplication). However, the important point is that Linda-C admits of a variety of approaches when efficiency issues like these develop.

In Section 5.1.2 we discussed possible problems with the tree paradigm. We used the matrix code as a test case by adding the following:

```
spoiler()
{
   int col_index, row_index;
   LINDA_BLOCK COL, ROW;

   rd("row", ? &row_index, ? &ROW);
   rd("col", ? &col_index, ? &COL);
}
```

`spoiler()` is never called. However, its presence caused the analysis to select the tree paradigm rather than the hash paradigm for handling the `row` and `col` tuples. The results are plotted for two problems sizes in Figures 5.9 and 5.10. The performance of the tree paradigm using a tree implementation is poor. We suggested an alternative implementation based on private hash tables, which we have implemented in an experimental kernel.[12] Data from this kernel is also shown in the graphs. The figures indicate that this method is quite promising and certainly better than the original implementation.

5.4.4 LU Decomposition

We produced a Linda-C benchmark for the Encore based on a LINPACK benchmark developed by Dongarra [Don87]. Dongarra's benchmark routine repeatedly solves systems of linear equations by the standard method: perform an LU factorization of the matrix, then perform a forward and backward solve. The $O(n^3)$ factor step dominates time costs, while the solve phase is $O(n^2)$. This being so (and considering also the lack of a promising strategy for parallelizing the backward solve at any but a very fine level of granularity), we wrote a Linda-C version of **dgefa()** (the factor routine) but not **dgesl()** (the solve routine). Note the "d" in "**dgefa**": these are double-precision calculations.

The resulting Linda code for **dgefa()** worked well. We tested two versions of the code (one copied data, one used pointers — similar to the matrix example above). Both exhibited close to linear speedup through ten workers. Both compared favorably with a "straight" C version (written by us) that made direct use of the Multimax's spinlocks for process synchronization.

[12]This was the only major change.

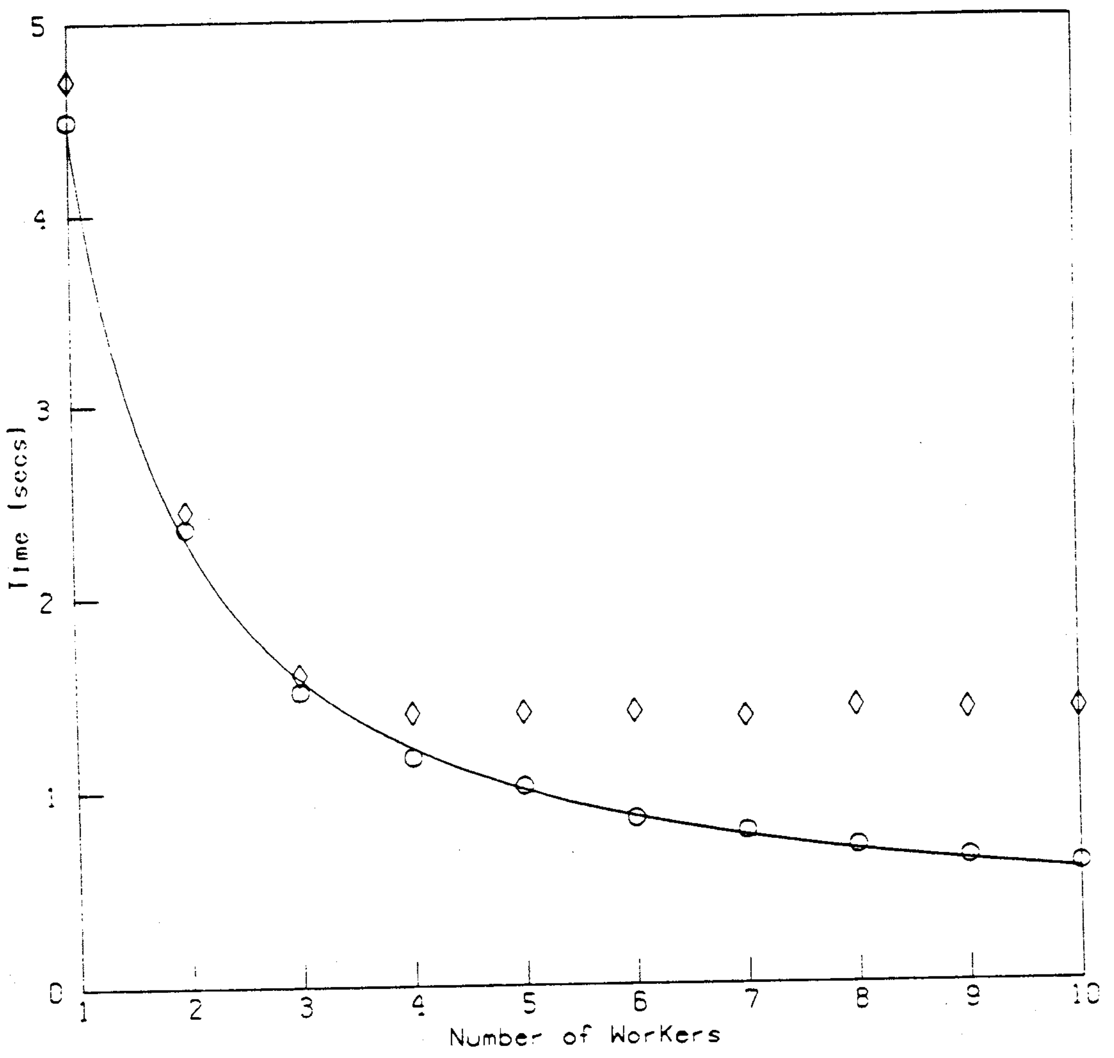

Figure 5.9: Execution time vs. number of workers for a 50 × 50 matrix multiplication — tree paradigm. Execution times (◇) are for the tree implementation, while (○) are for the private hash table implementation. The solid curve is the fitted curve for the queue paradigm data.

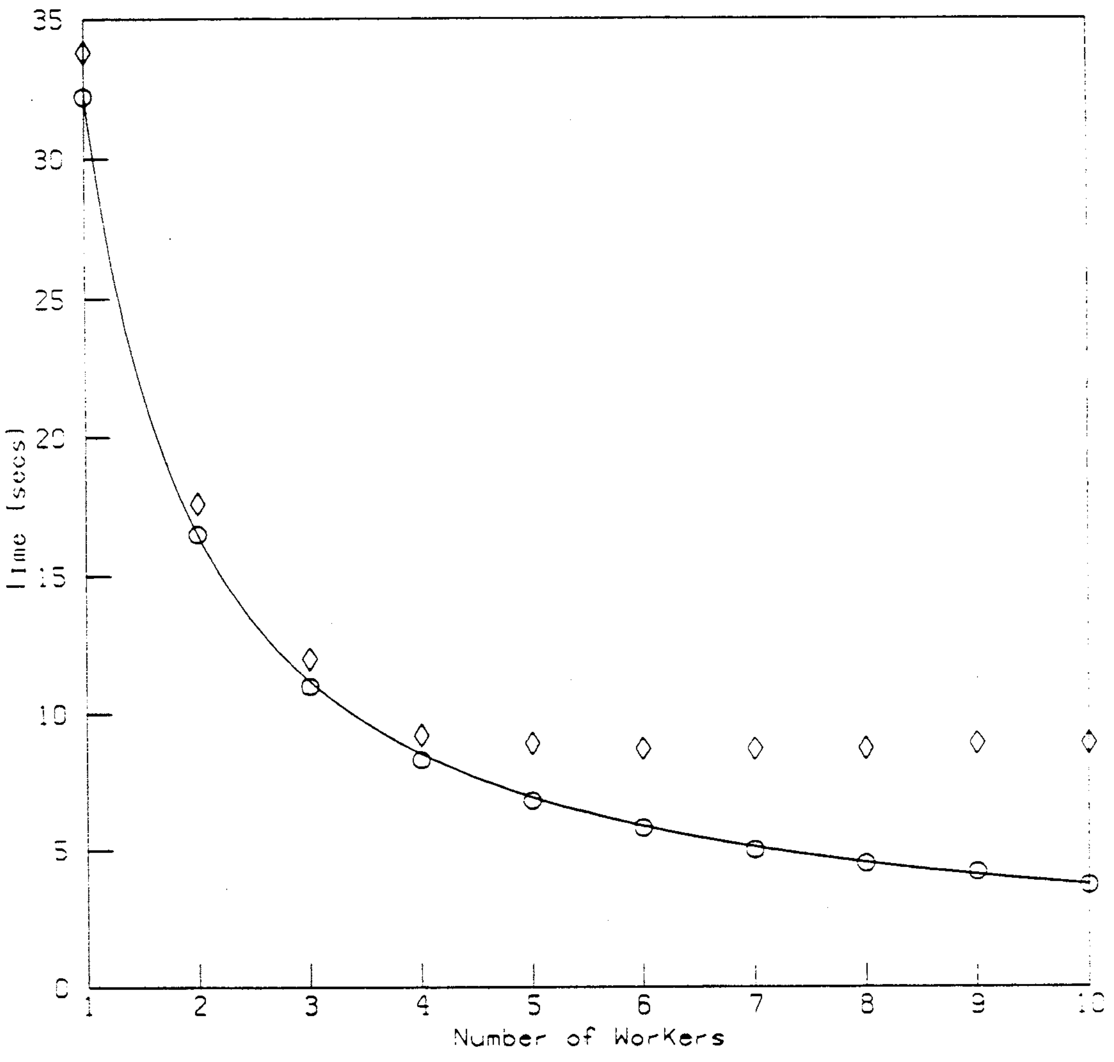

Figure 5.10: Execution time *vs.* number of workers for a 100 × 100 matrix multiplication — tree paradigm. Execution times (◇) are for the tree implementation, while (○) are for the private hash table implementation. The solid curve is the fitted curve for the queue paradigm data.

One goal of this test was to determine how much time was needed to convert sequential code to Linda-C. It took three man days to produce the first version of the Linda code. Two and one half man days were spent:

1. Locating a FORTRAN version of the benchmark.
2. Becoming familiar with the FORTRAN code.
3. Writing a straight, non-parallel C version.
4. Porting the C version to the Multimax.

The conversion from a non-parallel C version to Linda-C took one afternoon. All versions use the BLAS (*B*asic *L*inear *A*lgebra *S*ubroutines) provided in Dongarra's code (which is in FORTRAN).

We used the master/worker model for the conversion. To initialize the computation, the master process deals out matrix columns to the workers via tuple space. For a dimension n matrix, there will be $n-1$ stages in the computation. At each stage, the master consults a column of the matrix and calculates a vector. Each worker uses the vector to adjust the values of the worker's columns. Once all the adjustments are made, the next stage starts.[13] This computation is less "embarrassingly parallel" than matrix multiply because the workers depend on data which is generated by the master. This data can be generated only after some previous work has been completed by the workers.

The code for the Linda-C master is given in Figures 5.11 and 5.12 (this code has been modified to be consistent with the syntax used in this chapter — it was written before a change from `fml` to `?` to indicate a formal argument).

A driver routine was written in Linda-C. It initializes the Linda environment and then calls Dongarra's code (modified to run as a subroutine). Dongarra's code in turn calls `dgefac_()`.

The straight C version (listed in Appendix A) follows the outline of the Linda version. Synchronization is accomplished by a barrier at the end of every computation stage. An interlock permits the master to compute the next vector while the workers are finishing the current step. This is slightly less parallel than the Linda-C version: in the C version, no process can be one elimination step ahead of another, while in Linda-C processes can be up to n steps apart, where n is the number of workers. The Linda-C version therefore has the slight advantage of being better able to buffer variations in the length of time workers take to finish each step — the C version waits at every step for the slowest worker to finish. One could argue, on the other hand, that Linda has earned this advantage by making it simple to write the code this way.

Figures 5.13 and 5.14 give the results for two different-size problems. All times given are the times reported by Dongarra's code for the first call to `dgefac_()`. All runs (for a given dimension) were computationally identical. All had residuals better than 10^{-12}.

The Linda-C version with no copy runs about the same as the straight C version (slightly slower with small numbers of processors, slightly faster as the number of processors grow). What is the cost of writing in "pure" Linda-C? In this application, that cost consists largely of copying data into and out of tuple space. However, the Linda kernel makes this copying cost parallelizable wherever possible — i.e. locks are released before copying starts. With

[13]Actually, as we will see below, work on the next stage can begin as soon as the column the master will next consult has been modified.

```
#include "linda.h"

#define MAX    200
double  junk[MAX];
LB_EXT  JUNK = { MAX, junk, 0 };
int     num_workers;

dgefac_(a, lda, n, ipvt, info)
double *a;
int    *lda, *n, *ipvt, *info;
{
    int    j, k, l, length, nm1, one;
    double temp;
    double *l_ptr, *diag_ptr, *p_ptr, *m_ptr;
    LB_EXT COL, MULT, PIVOT;

    *info = -1;
    nm1 = *n - 1;
    one = 1;

    /* Start up workers.
       They were eval'ed in a start up routine -- this
       just wakes them up. */
    for (j = 0; j < num_workers; ++j) {
        out("LU worker data", num_workers, *n);
    }

    /* Dump columns of the matrix into tuple space. Multiply
       by 2 to reflect the fact a double is 2 longs. */
    COL.size = *n << 1;
    for (j = 1, COL.data = a + *lda; j < *n;
      ++j, COL.data = (double *)COL.data + *lda) {
        out("LU col", j, COL);
    }

    /* First column is first pivot. */
    COL.data = a;
    out("LU pivot col", COL);

    if (nm1 > 0) {
        /* Loop through the stages of the computation. */
        for (k = 0, PIVOT.data = a; k < nm1;
          ++k, PIVOT.data = (double *)PIVOT.data + *lda) {
            diag_ptr = (double *)PIVOT.data + k;

            /* Wait for worker to return next pivot column. */
            in("LU pivot col", ? &PIVOT);
```

Figure 5.11: LU — master.

```
            /* Find the max in the pivot column. */
            length = *n - k;
            l = idamax_(&length, diag_ptr, &one) - 1;
            l_ptr = diag_ptr + l;
            l += k;
            ipvt[k] = l + 1;

            /* If a zero pivot, no work this cycle -- signal
               workers via a zero length multiplier vector. */
            if ((*l_ptr) == 0.0) {
                *info = k;
                MULT.size = 0;
                out("LU multipliers", k, 1, MULT);
                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, &one);
            MULT.data = m_ptr;
            MULT.size = length << 1;

            /* Notify workers of new multipliers. */
            out("LU multipliers", k, 1, MULT);
        }
    }
    /* Get the last column. */
    in("LU pivot col", ? &PIVOT);
    ipvt[nm1] = *n;
    if (*(a + nm1 * *lda + nm1) == 0.0) {
        *info = nm1;
    }
    *info += 1;
}
```

Figure 5.11: Encore LU — master (cont'd).

```
lu_worker(id)
int    id;
{
    double a[MAX][MAX], (*c_ptr)[MAX];
    double (*first_col_ptr)[MAX], (*last_col_ptr)[MAX];
    int    dim, i, j, ji, k, l, length, nm1, num_workers, one;
    int    map[MAX];
    int    *map_ptr;
    double mult[MAX];
    double *l_ptr, *p_ptr, temp;
    LB_EXT COL, MULT;

    one = 1;

    /* Assume worker lives until killed by external agent. */
    while(1) {
        /* Get task data. */
        in("LU worker data", ? &num_workers, ? &dim);
        nm1 = dim - 1;

        /* Read in columns.  Stepping by num_workers shuffles
           columns among workers. map will map between worker's
           index and original index, 'a' will hold a worker's
           columns packed together. */
        for (i = id, j = 0; i < dim; i += num_workers, ++j) {
            /* skip 0th row (first pivot). */
            if (!i) {--j; continue;}
            map[j] = i;
            COL.data = a[j];
            in("LU col", i, ? &COL);
        }
        first_col_ptr = a[0];
        last_col_ptr = a[j];
        map[j] = -1;
        map_ptr = map;
```

Figure 5.12: LU — worker.

```
        MULT.data = mult;
        for (k = 0;k<nm1 && first_col_ptr<last_col_ptr;++k) {
            rd("LU multipliers", k, ? &l, ? &MULT);

            /* Loop through my columns doing daxpy's. */
            for (c_ptr = first_col_ptr;
             c_ptr < last_col_ptr; ++c_ptr) {
                /* If non-zero pivot, mult size is 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;
                    }
                    length = nm1 - k;
                    daxpy_(&length, &temp, mult,
                     &one, p_ptr+1, &one);
                }

                /* If my first column is next pivot,
                   send it back to master. */
                if (*map_ptr == k + 1) {
                    COL.data = c_ptr;
                    out("LU pivot col", COL);
                    ++first_col_ptr;
                    ++map_ptr;
                }
            }
        }
        /* 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, ? &JUNK);
            }
        }
    }
    exit(0);
}
```

Figure 5.12: LU — worker (cont'd).

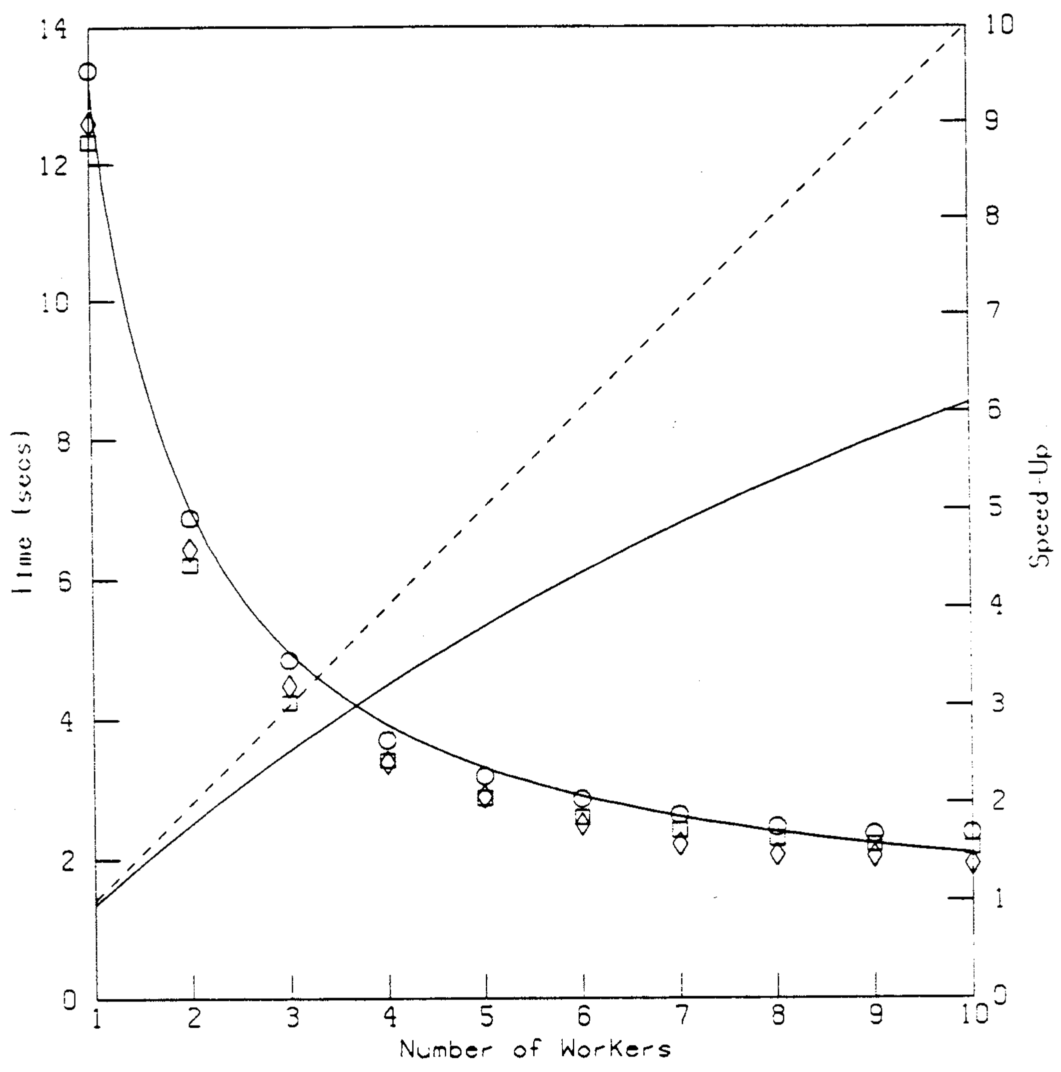

Figure 5.13: Execution time *vs.* number of workers for `dgefac_()` of a 100 × 100 matrix. Execution times (○) are the Linda-C version that copies data, (◇) mark times for the Linda-C version that does not copy data and (□) mark execution times for the C version. The descending solid curve is fitted to the data for the copying version ($a = 12.4$ secs; $b = .83$ secs). The ascending solid curve shows the speed-up of the copying version over a sequential version.

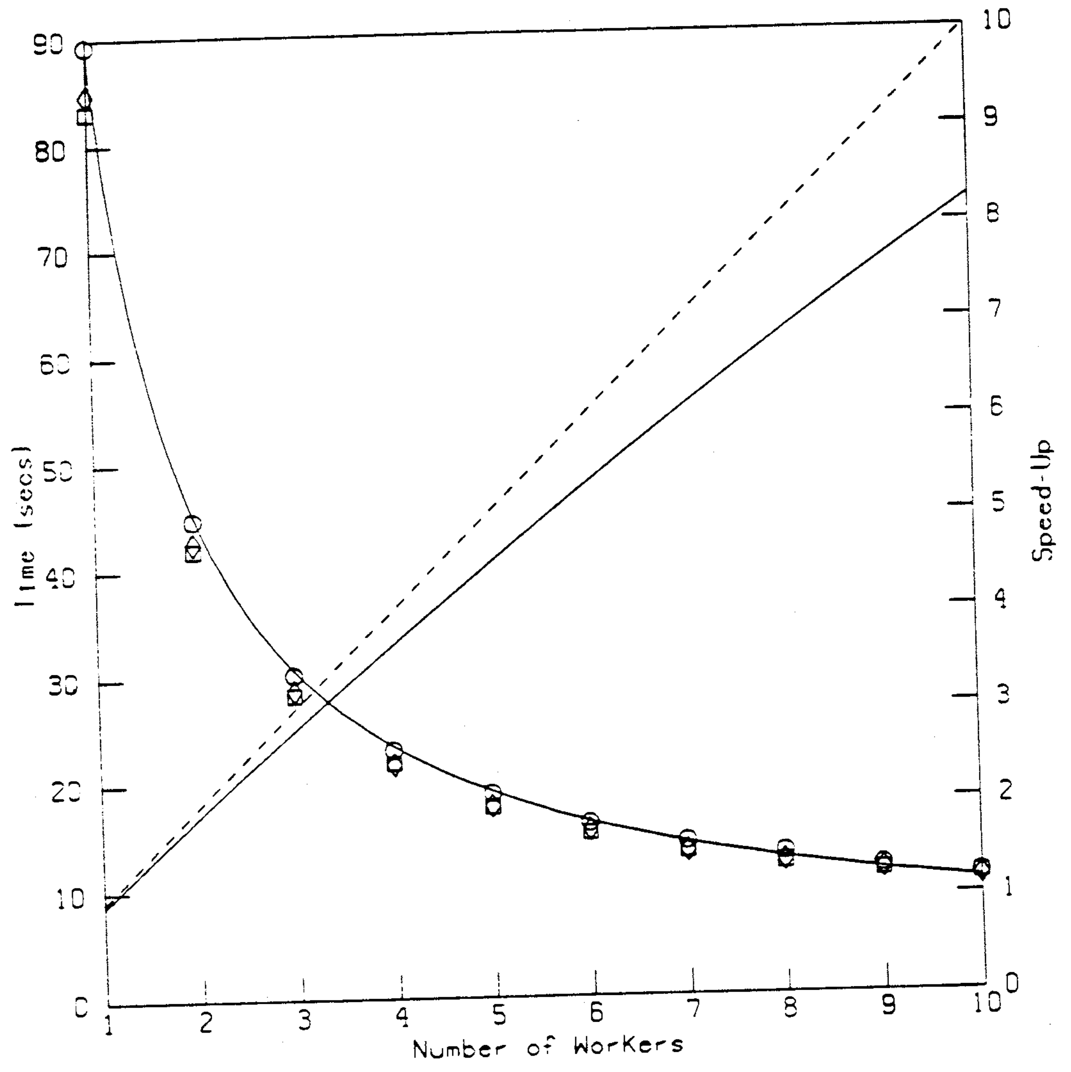

Figure 5.14: Execution time *vs.* number of workers for **dgefac_()** of a 190 × 190 matrix. Execution times (○) are the Linda-C version that copies data, (◇) mark times for the Linda-C version that does not copy data and (□) mark execution times for the C version. The descending solid curve is fitted to the data for the copying version ($a = 87.3$ secs; $b = 1.64$ secs). The ascending solid curve shows the speed-up of the copying version over a sequential version.

```
  if (argc != 3) {
    printf("Usage: <dim> <num>\n");
    exit(1);
  }
  dim = atol(*++argv);
  num_workers = atol(*++argv);
  aa = (double *)share(0, sizeof(double)*200*200);
  a = (double *)share(0, sizeof(double)*201*200);
  shared_ptr = (char *) share(0, 1000);
  globs = (struct shared_globs *)shared_ptr;
  shared_ptr += sizeof(struct shared_globs);
  globs->c_lock = 0;
  globs->run_lock = 1;
  globs->start_lock = 1;

  printf("calling ddr with dim %d, %d workers.\n", dim, num_workers);
  for (i = 0; i < num_workers; ++i) {
    if (!(workers[i] = fork())) lu_worker(i);
  }
  ddr_(&dim, aa, a);
  while (num_workers) {
    kill(workers[--num_workers], 9);
  }
}

dgefac_(a, lda, n, ipvt, info)
double  *a;
int     *lda, *n, *ipvt, *info;
{
  int     j, k, l, length, nm1, one;
  double  temp;
  double  *col_ptr, *l_ptr, *diag_ptr, *p_ptr, *m_ptr;

  *info = -1;
  nm1 = *n - 1;
  one = 1;
  globs->a = a;
  globs->lda = *lda;
  globs->n = *n;
  globs->start_lock = 0;
  globs->next_lock = 0;
  globs->counter = 0;
  globs->run_lock = -1;
  if (nm1 > 0) {
    for (k = 0, col_ptr = a; k < nm1; ++k, col_ptr = col_ptr + *lda) {
      while(globs->next_lock);
      globs->next_lock = 1;
      diag_ptr = col_ptr + k;
      length = *n - k;
      l = idamax_(&length, diag_ptr, &one) - 1;
      l_ptr = diag_ptr + l;
      l += k;
      ipvt[k] = l + 1;

      if ((*l_ptr) == 0.0) {
        *info = k;
        while(globs->counter);
        globs->mults = 0;
        globs->run_lock = 0;
        globs->counter = num_workers;
        while(globs->counter);
        globs->start_lock = 1;  /* VERY approximate... */
        continue;
      }

      if (l != k) {
        temp = *diag_ptr;
        *diag_ptr =  *l_ptr;
        *l_ptr = temp;
      }

      temp = -1.0 / *diag_ptr;
      length = nm1 - k;
      m_ptr = diag_ptr + 1;
      dscal_(&length, &temp, m_ptr, &one);
      while(globs->counter);
      globs->mults = m_ptr;
      globs->l = l;
      globs->counter = num_workers;
      globs->run_lock = k;
      globs->start_lock = 1;    /* VERY approximate... */
    }
  }
  ipvt[nm1] = *n;
  if (*(a + nm1 * *lda + nm1) == 0.0) {
    *info = nm1;
  }
  *info += 1;
}

int     map[MAX];

lu_worker(id)
int     id;
{
  int   dim, i, j, k, l, length, nm1, one;
  int   *map_ptr;
  double        *cols[MAX], *l_ptr, *p_ptr, temp;
  double        **c_ptr, **first_col_ptr, **last_col_ptr;
  double        *mult_ptr;

  one = 1;
  while(1) {
    while(globs->start_lock);
    dim = globe->n;
    nm1 = dim - 1;
    for (i = id, j = 0; i < dim; i += num_workers, ++j) {
      if (!i) {--j; continue;}  /* skip 0th row -- the first pivot. */
      map[j] = i;
      cols[j] = globs->a + i*globs->lda;
    }
    first_col_ptr = cols;
    last_col_ptr = &cols[j];
    map[j] = -1;
    map_ptr = map;
    for (k = 0; k < nm1; ++k) {
      while(globs->run_lock != k);
      mult_ptr = globs->mults;
      l = globs->l;
      if (mult_ptr == 0) {
        if(*map_ptr == k + 1) {
          ++first_col_ptr;
          ++map_ptr;
          globs->next_lock = 0;
        }
        spinlock(&globs->c_lock);
        --globs->counter;
        spinunlock(&globs->c_lock);
        continue;
      }
      for (c_ptr = first_col_ptr; c_ptr < last_col_ptr; ++c_ptr) {
        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;
        }
        length = nm1 - k;

        daxpy_(&length, &temp, mult_ptr, &one, p_ptr + 1, &one);
        if(*map_ptr == k + 1) {
          ++first_col_ptr;
          ++map_ptr;
          globs->next_lock = 0;
        }
      }
      spinlock(&globs->c_lock);
      --globs->counter;
      spinunlock(&globs->c_lock);
    }
  }
  exit(0);
}
```

Bibliography

[ACGK86] S. R. Ahuja, N. J. Carriero, D. H. Gelernter, and V. Krishnaswamy. Progress towards a Linda machine. In *Proceedings of IEEE International Conference on COMPUTER DESIGN 1986*, pages 97–101, IEEE Computer Society, October 1986.

[Ahu] S. R. Ahuja. Private communication.

[Ahu83] S. Ahuja. S/Net: A high-speed interconnect for multiple computers. *IEEE Selected Areas in Communication*, 751–756, November 1983.

[Bar80] J. G. P. Barnes. An overview of Ada. *Software Practice and Experience*, 10:851–887, 1980.

[Bjo] R. D. Bjornson. Private communication.

[BBN85] *Butterfly Parallel Processor Overview*. BBN Laboratories, 1985.

[BN84] A. D. Birrel and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computing Systems*, 2(1), February 1984.

[Car84] J. P. Carriero. *Descartes and the Autonomy of the Human Understanding*. PhD thesis, Harvard University, 1984.

[CGL86] N. J. Carriero, D. H. Gelernter, and J. Leichter. Distributed data structures in Linda. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 236–242, Association for Computing Machinery, January 1986.

[Che86a] M. C. Chen. A parallel language and its compilation to multiprocessor machines or VLSI. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 131–139, Association for Computing Machinery, January 1986.

[Che86b] M. C. Chen. *Placement and Interconnection of Systolic Processing Elements: A New LU-Decomposition Algorithm*. Research Report 498, Yale University, October 1986.

[Cra84] *The Cray X-MP Series of Computer Systems*. Cray Research, Inc., 1984.

[Den86] P. J. Denning. Parallel computing and its evolution. *Communications of the ACM*, 29(12):1163, December 1986.

[Don87] J. J. Dongarra. *Performance of Various Computers Using Standard Linear Equations in a FORTRAN Environment.* Technical Memorandum, Argonne National Laboratory, 1987.

[Ell85] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures. ACM Doctoral Dissertation Award Series*, MIT Press, 1985.

[Enc86] *Multimax Technical Summary.* Encore Computer Corporation, 1986.

[Fel79] J. A. Feldman. High level programming for distributed computing. *Communications of the ACM*, 22(6):353–368, June 1979.

[FLBB79] M. J. Fischer, N. C. Lynch, J. E. Burns, and A. Borodin. Resource allocation with immunity to limited process failure. In *20th Annual Symposium on Foundations of Computer Science*, pages 234–254, IEEE Computer Society, 1979.

[Gel82] D. H. Gelernter. *An Integrated Microcomputer Network for Experiments in Distributed Programming.* PhD thesis, State University of New York at Stony Brook, 1982.

[Gel85] D. H. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[GK85] R. D. Gaglianello and H. P. Katseff. Meglos: an operating system for a multiprocessor environment. In *Proceedings of the Fifth International Conference on Distributed Computing*, pages 35–42, May 1985.

[Hal85] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, October 1985.

[Hil85] W. D. Hillis. *The Connection Machine. The ACM Distinguished Dissertation Series*, MIT Press, 1985.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):667–677, August 1978.

[Hud86] P. Hudak. Para-functional programming. *Computer*, 19(8):60–70, August 1986.

[iPS86] *iPSC User's Guide.* intel Corporation, April 1986.

[Joh75] S. C. Johnson. *Yacc — Yet Another Compiler Compiler.* Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[Joh78] S. C. Johnson. A portable compiler: Theory and practice. In *Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 97–104, 1978.

[Jor86] H. F. Jordan. Structuring parallel algorithms in an MIMD, shared memory environment. *Parallel Computing*, 3:93–110, 1986.

[KEM*78] D. Katsuki, E. S. Elsam, W. F. Mann, E. S. Roberts, J. G. Robinson, F. S. Skowronski, and E. W. Wolf. Pluribus — An operational fault-tolerant multiprocessor. *Proceedings of the IEEE*, 66(10):1146–1159, October 1978.

[Knu73] D. E. Knuth. *The Art of Computer Programming.* Volume 3, Addison-Wesley, 1973.

[Kra87] D. Kranz. *Orbit: An Optimizing Compiler for SCHEME.* PhD thesis, Yale University, 1987.

[KSC*84] D. J. Kuck, A. H. Sameh, R. Cytron, A. V. Veidenbaum, C. D. Polychronopoulos, G. Lee, T. McDaniel, B. R. Leasure, C. Beckman, J. R. B. Davies, and C. P. Kruskal. The effects of program restructuring, algorithm change and architecture choice on program performance. In R. M. Keller, editor, *Proceedings of the 1984 International Conference on Parallel Processing,* pages 129–138, IEEE Computer Society, IEEE Computer Society Press, August 1984.

[LKK85] G. Lee, C. P. Kruskal, and D. J. Kuck. The effectiveness of automatic restructuring on nonnumerical programs. In D. Degroot, editor, *Proceedings of the 1985 International Conference on Parallel Processing,* pages 607–613, IEEE Computer Society, IEEE Computer Society Press, August 1985.

[Lei] J. Leichter. Private communication.

[Luc86] S. Lucco. A heuristic Linda kernel for hypercube multiprocessors. In *Proceedings of SIAM Conference on Hypercube Multiprocessors,* September 1986.

[NCU86] *NCUBE Handbook; Version 1.1.* NCUBE, 1986.

[PBG*85] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM research parallel processor prototype (RP3): Introduction and architecture. In D. Degroot, editor, *Proceedings of the 1985 International Conference on Parallel Processing,* pages 764–771, IEEE Computer Society, IEEE Computer Society Press, August 1985.

[Seq85] *Balance 8000 System Technical Summary.* Sequent Computer Systems, Inc., December 1985.

[Sha86] E. Shapiro. Concurrent Prolog: A progress report. *Computer,* 19(8):44–58, August 1986.

[Uni82] *Reference Manual for the Ada Programming Language.* United States Department of Defense, July 1982.

[Wir76] N. Wirth. *Algortihms + Data Structures = Programs.* Prentice-Hall, 1976.