The Implementation and Performance of Hypercube Linda

Robert Bjornson, Nicholas Carriero, and David Gelernter
Research Report YALEU/DCS/RR-690
March 1989

This work is supported by National Science Foundation SBIR grant ISI-8704025, by National Science Foundation CCR-8601920 and CCR-8657615, and by ONR N00014-86-K-0310.

The Implementation and Performance of Hypercube Linda

Robert Bjornson, Nicholas Carriero and David Gelernter

Yale University
Department of Computer Science
New Haven, Connecticut

March, 1989

Abstract. Is it possible to combine the programmability of a shared object memory with the hardware advantages of a scalable distributed-memory architecture? We argue that the answer is yes; we discuss the implementation and performance of a Linda system for the Intel iPSC/2 hypercube. Our results demonstrate that this system can be used to solve significant problems effectively. Good efficiencies require coarser task granularities than lower-level message passing systems can support, but these granularities are routinely achieved in a variety of problems that are by no means "embarassingly parallel", and the programs we discuss lean heavily on the expressive power of Linda's tuple space.

This work is supported by National Science Foundation SBIR grant ISI-8704025, by National Science Foundation CCR-8601920 and CCR-8657615, and by ONR N00014-86-K-0310.

1 Introduction

We have argued at length that a global object memory is a more natural, more powerful basis than message-passing for parallel programming [Gel85, CGL86, CG89]. Architects have demonstrated in the meantime that linking autonomous nodes into a local-memory-only network constitutes a relatively simple, inexpensive and scalable way to build a parallel computer. Message passing is the native idiom on such a machine; its hardware provides no support for a shared object memory. Is it possible to combine the programmability of a shared object memory with the hardware advantages of distributed memory?—to build, in other words, a shared object memory on top of distributed-memory hardware?

The answer is yes: we demonstrated this some time ago by implementing Linda on Bell Lab's experimental S/Net system [CG86]; recent work has also shown that Linda can be used to support coarse-grained parallel applications across local networks [WL88]. The significant number of Linda-on-Transputer implementation projects now underway[1] make it plain that a a variety of groups are convinced that Linda on distributed-memory machines is both useful and feasible. But we have not demonstrated that Linda is practical on scalable parallel architectures like the Intel iPSC/2 hypercube (the machine we focus on here). At 64 nodes our iPSC/2 is substantially larger than the shared-memory machines we have available; much larger hypercubes are possible, and it will be clear from the discussion that our hypercube implementation is also designed to be scalable. Unlike the earlier S/Net implementation, the hypercube system can't depend on reliable broadcast as an implementation tool; unlike the network system, a hypercube system can't be considered a success if it supports coarse-grained applications only.

We asked a generic question about object memories on hypercubes, and proposed one specific answer, in terms of Linda. Linda is a parallel programming model that centers on a shared, associative object memory. Is it a good exemplar of the "shared object memory" class? If we define an object memory as a memory that supports *add-object*, *read-object* and *remove-object* operations in such a way that they can be executed safely by concurrent processes, Linda is one of the only exemplars. It's important to note that so-called concurrent object-oriented languages tend to use either message-passing or remote-procedure-call for inter-process communication: the object-space (the collection of all objects in the program) is not the the communication medium, and it doesn't constitute a shared object memory in our sense. (The point is pursued in [CG89]; concurrent object-oriented systems based on Linda, e.g. [MatKa88], are an obvious exception.) Actors [Agh86] and Dally's CST [DC88] are two other systems that have an object-based component but don't meet the specific definition given above. With respect to Linda in particular, the arguments that support its value have been repeated too often to make them worth including here. In summarizing them briefly, Stephen Zenith of Inmos writes that "Linda is elegant and easy to use... [Linda supports] automatically scalable algorithms; ... the same model on different parallel architectures; a unified process and communication model; an integral persistent storage concept; Linda is easily added to existing sequential languages to create parallel variants of the original languages[2]". This paper is accordingly premissed on the contention that Linda is a valuable alternative to message passing.

[1] including commercial implementations (for example Cogent Research, Human Devices), a variant called "Brenda" for the Trollius operating system (by M. Braner of the Cornell Theory Center), and a number of independent academic research efforts.

[2] S.E. Zenith, "The simplicity of Linda", posted to the comp.parallel bulletin board on the Arpanet

Our goal in this paper is to describe an implementation of Linda on the iPSC/2 and to demonstrate its effectiveness. Because of the sophistication and "high-levelness" of its communication model, Linda *does* exact a performance penalty relative to plain message-passing on the iPSC/2. But—as per the performance penalty exacted by compilers as opposed to assemblers, or virtual as opposed to static memory systems—the existence *per se* of this penalty is irrevelant. Here is the important question: *is the penalty small enough and the potential gains large enough to make the system worth using despite the existence of lower-level alternatives?* We'll argue that the answer is yes.

1. *Nature of the penalty.* A basic user-level message-send on the iPSC/2 takes 390 μsec; a basic tuple-exchange in Linda (one process **outs** a singleton tuple to tuple space and a second **ins** it) requires 1.47 msec, reflecting the fact that, in the general case, one message transaction is required to add and two to remove a tuple from tuple space. Having made this comparison, the next thing to note is that it is largely meaningless: Linda operations are different from and more powerful than messages passing; see (2).

2. *Nature of the potential gain.* Linda's **out** operation can mimic the funtionality of a message-passing *send*, but it does more: it generates an object and adds it to a global associative memory. As a consequence Linda makes it easy to support a class of programs that are hard to think about and hard to implement in message-passing terms—programs that rely on shared data structures built out of global objects (so-called "distributed data structures" [CGL86]). These structures are important because, among other things, they make it easy to parcel work out dynamically and adaptively at runtime, thus saving the conceptual overhead of distributing shared data explicitly to participating nodes, or constructing a static list of task assignments, and in some cases yielding major efficiency gains as well. A global object memory has other advantages too: Linda makes it natural (for example) to support a form of communication that looks like message passing, but where the participating processes aren't known until runtime (we discuss an example); we've defended at length the proposition that a global memory, which supports an "uncoupled" programming style, is conceptually simpler than message-passing.

3. *The bottom line* We will discuss Linda programs that (*a*) solve "real" problems posed by computing users (not by the systems community), (*b*) rely heavily on the power of Linda-style communication, (*c*) do non-trivial communicating relative to computing, and (*d*) show good speedup through 64 nodes on the iPSC/2.

In section 2 we describe the implementation and in section 3, its performance.

2 Implementation

2.1 The current system

Our current iPSC system implements C-Linda—the Linda tuple space operators added to C, yielding a parallel dialect. The structure of the runtime system is unchanged when we move to other Linda language hosts: we have implemented a prototype Fortran-Linda, and other groups have implemented, or are investigating (as are we), a wide variety of other language hosts.

The system consists of the C-Linda precompiler [Car87] and the runtime library. The precompiler (in outline) assembles information about all Linda operations in the source; at linktime, analyzes all Linda operations, parcels them into classes according to style of tuple space access, and maps them onto appropriate calls to the runtime library. Linda's `in` and `rd` operations support general associative lookup, but our goal at runtime is to use a tuple space access routine that supplies *exactly as much matching generality as this particular tuple space access requires, and no more.* Each basic Linda operation is accordingly represented not by a single library routine but by a family of routines; the precompiler implements each Linda operation in the user's source by selecting the most appropriate member of the family.

To do so, it separates all Linda operations into disjoint sets, such that an `in` or `rd` in one set can only match `out`s and `eval`s in the same set, and *vice versa.* It then classifies each set based on its matching pattern—which fields are used as match keys and which as data fields, which fields are always constant, or actual, or potentially formal. This analysis determines which library routine is optimal for each class; it also determines the way in which the class is stored on the hypercube.

For present purposes, the details of the precompiler's classification scheme aren't important; the important point is that tuple classes fall into two categories. Every `in` or `rd` operation has been assigned to some class at linktime; if this class is a *first category* class, a search key exists that will allow the `in` or `rd` to select a matching tuple from the *potentially matching* tuples that will occupy the class at any given moment during program execution. If the class is a *second category* class, there is no guaranteed search key, and `in` or `rd` will necessitate (in the worst case) an exhaustive search through all tuples that happen to occupy the class.

Runtime tuple storage. Tuple storage is based on a distributed table implemented in terms of the precompiler's tuple classification. Each first-category class is implemented as a distributed hash table covering every node in the network. The hash function accepts as input both the class identifier and the search key. The most important criterion for the hash function is that the node

and the intra-node bucket be chosen independently. Both the set identifier and the search key are likely to be small positive integers, and so the hash function must discriminate well in this range.

Second-category classes are handled similarly, except that no search key is guaranteed to be present, and accordingly only the class identifier can be used as input to the hash function. Every second-category class is mapped in its entirety to some arbitrary node. Within that node, data stuctures appropriate to the particular class are chosen to optimize search times [Car87]. It is important that the hash function scatter classes across the machine. In general, every node serves both as a tuple-space server and as a computation host.

Tuple rendezvous—the assignment of a particular tuple (generated by `out` or `eval`) to a particular tuple request (an `in` or a `rd` statement)—generally takes place on a node where neither the tuple-generating nor the tuple-requesting process is local. Both `in` and `out`, then, require the packaging and transmission of a tuple. (In the case of `in` or `rd`, this "tuple" is simply the list of arguments to the `in` or `rd`, precisely as for `out` — we refer to such a "anti-tuple" as a template). On `out` or `eval`, a tuple is dispatched to the storage node dictated by the hash scheme. It may find a matching template waiting for it; if so, it proceeds onward to the matching template's home node. If not, it is installed in a local table. Likewise for `in` and `rd`: arriving templates will either match a waiting tuple or be installed in the local table to wait hopefully for the right tuple to come along.

An `out` or an `eval`, then, requires a message-send. `in` and `rd` each require a message-send and a message-receive, where message-send and -receive are the primitives provided by the native communication system.

A concerted effort was made to reduce the size of the data structure representing a tuple, to save memory (the iPSC/2 has no virtual memory) as well as to reduce communication costs. The information pertaining to a single tuple can be classified into three types: *class, operation* and *tuple* specific information. Only tuple-specific information is actually stored in the tuple; included are values for the tuple fields, the hash value, and a tuple id. All other information describing a particular tuple is identical for all tuples produced by a particular Linda statement in the source—this category covers, for example, number and types of tuple fields and descriptive information for runtime debugging and tracing. This information is stored once in a separate data structure.

Most fields can be stored directly in the tuple data structure (the header). Larger fields, for example strings or structures, are copied to dynamic storage and pointed-to by the header. This strategy makes it possible to send the header only at first. The receiving node determines the number and size of large fields from the header. It allocates storage and then receive these larger fields directly into the allocated storage area, thus minimizing copying requirements. The

disadvantage of storing large fields separately is that an extra message send is required for each large tuple field, which is expensive on the iPSC/2. For very large fields our strategy makes sense, but on balance shorter "long fields" should be treated differently, and will be in future versions.

When large fields aren't needed for tuple matching (most cases), they aren't sent to the rendezvous node. They remain at the **out**ing node, and they are eventually forwarded to the **in**ing node when requested to do so by the rendezvous node.

Implementation of eval. The Linda operation **eval** presents special problems on the iPSC/2. **eval** is identical to **out**, except that it adds an *unevaluated* tuple to tuple space and creates a new process to evaluate it (concurrently with the **eval**-executing process); once the tuple has been completely evaluated, it is indistinguishable from a tuple created using **out**. Usually one field in an **eval** is a procedure call; **eval** is Linda's method for creating new processes.

eval is implemented entirely in terms of **in** and **out**[3], using a very simple scheduling strategy that does not attempt to multiplex processes over a single node. Each node runs an evaluator loop that withdraws from tuple space (using **in**) the description of an **eval**-created tuple for evaluation, evaluates it to completion, uses **out** to install the result tuple, and repeats. The **eval** operation itself is implemented by **out**'ing "eval descriptors", which contain procedure pointers (the same image is loaded on every node) and the values of any variables mentioned explicitly within the **eval** statement. More complex scheduling stategies are possible and we will experiment with them in later versions, but the present simple scheme is well suited to existing applications.

2.2 Optimizations: bucket switching; updates in place

Tuple space is a general associative memory, but it's desirable to use the *least-general* matching routine that will do at runtime. Similarly, tuple space is a general object store, but it would be desirable to notice cases in which a less general (and therefore cheaper) implementation strategy would work as well. We will discuss two examples here. The first involves a newly-implemented optimization that hasn't yet been fully tested; the second hasn't yet been implemented. We briefly describe these optimizations anyway because they form an important and obvious extension of the system described above, and will be completed in the near future.

Bucket switching. Some tuples are similar to messages in the sense that they are intended not for reading, and not for consumption by *any* interested process, but for consumption by one process in particular. When some process

[3] i.e., the **eval**s in the user's source are translated by the precompiler to the appropriate **in** and **out** operations.

A generates a collection of tuples that meet this restriction—i.e., they are all intended for consumption by one other process B—we should be able to detect this pattern. Instead of shipping these tuples to a rendezvous node, they should be sent from A's node directly to B's.

We implement this scheme via a strategy called "bucket-switching": the partition of the distributed tuple table in which these particular tuples were stored is remapped from some arbitrary rendezvous node to B's node. To accomplish remapping at the appropriate times, the rendezvous nodes keep track of the tuples consumed by each process. When a rendezvous node notices that a particular type of tuple is consumed by only (or primary) one process, it designates that process's node as the new rendezvous point for that tuple, forwards any remaining tuples of that type, and maintains a forwarding pointer. As new tuples or templates of that type arrive at the old rendezvous node, they are forwarded, and the sending process is told to amend its hash function. Eventually all processes interested in this type will hash (and send) to the new rendezvous node.

A similar tuple rehashing scheme was was first implemented by Lucco [Luc86], who reported very significant performance improvements for point-to-point style communication—but in the context of a much less efficient implementation that our current one, and without support from the precompiler's tuple classification. Early results from tuple rehashing in our system are encouraging. As a preliminary test, we ran a program that does repeated barrier synchronizations, using a standard spanning tree algorithm and point-to-point communication. After only a few sends, tuple mapping had been reconfigured in such a way that point-to-point communication required only a single message send. With 16 nodes, performing 5000 barriers took 51.1 seconds; with rehashing, 37.5 seconds. This 25% improvement represents a highly preliminary and untuned version of the rehashing system. Further substantial improvement should be possible.

Updates in place. Another tuple-use pattern that lends itself to optimization is the `in(`t`); out(`t'`)` pattern: a tuple is removed from tuple space, updated, and immediately re-installed. Because the objects in tuple space are immutable, this pattern is required whenever existing tuples must be altered. The precompiler can (but doesn't yet) detect this pattern; when the necessary extensions are in place, it will be possible in many cases to implement this pattern by sending a "tuple update message" to the rendezvous node on which t is stored and performing the update in place and remotely, rather than carrying out the entire `in; out` sequence.

3 Performance

The raw performance of this Linda system is an elusive idea. We noted at the start that Linda operations are more costly than message sends and receives, and that they are also more powerful. To measure the cost of using Linda, we could take a Linda program, remove all Linda operations and substitute message exchange, and then compare the behavior of the two versions. But this is a poorly-defined experiment. A direct translation from Linda to message-passing would require us to reproduce the Linda kernel, or some other storage scheme for global shared objects, and then access the system by using messages instead of Linda calls. This experiment merely reproduces the Linda implementation in altered form, and tells us nothing. The *real* test would require our comparing the Linda program not to a Linda-style program using message passing, but to a "comparable" program written in idiomatic message-passing style. Superficially this calls to mind the procedure we follow in computing speedup figures: we measure speedup relative to a "comparable" *serial* program, in essence the program that served as our starting point when we built the parallel version. But this procedure doesn't carry over to a comparison between two parallel programs; starting with a serial program, there is no canonical message-passing parallelization, but rather (in most cases) a broad spectrum of possibilities. Choosing one arbitrarily and comparing its performance to the Linda version's is of doubtful value. It might be argued that the correct procedure is to start with some message passing program, translate it into Linda and compare these two. But on reflection, this experiment suffers from the same basic problem with which we started. Message-passing programs are not idiomatic Linda programs: they ignore precisely those aspects of Linda that constitute the whole point of the model.

It's important nonetheless to make the attempt to establish the practical granularity limit for Linda programs under this implementation. We used a real application (rather than a synthetic test program) for this purpose; it uses the Linda operations in an idiomatic way, but allows us to vary the granularity of the task-step conveniently.

3.1 Measuring acceptable task granularity

The program we used is a prime number finder, designed specifically to find all primes between 1 and some upper bound. The Linda program uses the master-worker model. A task consists of checking the integers between n and $n +$ *ChunkSize* for primality. Tasks are distributed dynamically: a worker grabs a task-descriptor using `in`, then immediately generates a new task-descriptor (i.e., "check the next chunk of integers after mine") for some other worker to pick up. It proceeds to check its own chunk, reports the new primes to the master using

`out`, and grabs another task. The master uses tuple content-addressing to pick up the results in order (in essence, the workers write a distributed (FIFO) stream of tuples, and the master reads it: this is a simple distributed data-structure technique discussed in [CGL86]). The master uses the incoming results to construct (using `out`) a table of primes in tuple-space (another simple distributed data structure, also dependent on tuple content-addressing). A worker consults the table (using `rd`) as necessary in extending its sieve. This program and its development is discussed in detail in [CG89b].

By varying *ChunkSize* we can use this program to test the effect of task granularity on program performance. The results must be approached with caution in the sense that they apply (obviously) to one program only. But the program is typical of existing Linda applications, insofar as (1) it distributes tasks dynamically via tuple space, (2) it uses tuple space to store global data objects and to return intermediate results as well, and (3) it is conservative in its use of tuple-space operations: for example, workers use `rd` to consult the global primes table, but as they do so they incrementally copy the table into a local data structure, so that each entry in the global table is consulted only once.

We ran the granularity test on 32 nodes of the iPSC/2, using 31 workers and one master process; the task was to find all primes between 1 and three million. The results are shown in figure 1. We tested chunk sizes down to an interval of 30. The computation involved in searching an interval for primes is dominated by divisions performed; averaging over the entire range of 1 through 3 million, an interval of 30 requires about 550 divisions at 3.7 μsec each, so the smallest task step we tested amounts to slightly more than 2 msec. At no point, even at the relatively fine granularity of less than 5 msec per task, does the Linda program fail to show speedup relative to a serial version running on a single node: absolute running time for the serial version is shown in the dotted line on the graph. Nonetheless some general characteristics are clear. Down to an interval of 400, performance is essentially constant; there is minor degradation down to an apparent knee at 300; speedup then falls off more rapidly but irregularly. So the minimal grain size for near optimal performance on this program is around 300, corresponding to an average task step in the region 20 of msec.

Figure 2 shows speedup for this program as we vary participating nodes upwards through 64. The chunk size here is 2000; smaller chunk sizes (within bounds of the discussion above) would have given roughly comparable performance. One processor of an Intel iPSC/2 hypercube requires about 421 seconds to run the sequential C program; one master and 63 workers running on all 64 nodes of our machine require just under 8 seconds, for an efficiency of about 85%.

3.2 Measured performance on a related series of applications

The important question in measuring this system's performance is the following: *can it be used to solve real problems efficiently*? (*Efficiency* should be measured in terms of absolute speedup of a Linda program relative to a conventional serial program running on a single node.) We discuss this question in the context of a problem that lends itself to an interesting range of solutions with varying communication requirements and task granularities.

The problem. Geneticists at Yale posed the following problem: when new DNA sequences are discovered, it is of interest to determine which previously-known sequences they resemble, where "resemblance" is a qualitative measure that can be approximated using string-matching-like algorithms. Comparing two long sequences can be computationally expensive (a current algorithm of choice is $O(mn)$, where m and n are the lengths of the two sequences); usually, such comparisons will be performed against part or all of a large database of sequences ("GenBank").

Parallel sequence-to-sequence comparison. Comparing two long sequences is time-consuming, and such comparisons can be parallelized. The algorithm requires computing a similarity matrix in such a way that all elements along each counter-diagonal can be computed simultaneously once the previous counter-diagonal has been filled in. We use the standard transformation from a computation on matrix elements to a computation on matrix sub-blocks to control granularity. We wrote a master-worker program: each of a collection of identical workers performs tasks until no tasks remain; each task is the computation of one block of the similarity matrix. Once the first sub-block (first set of columns) in the first band is computed, work can begin on the second band. (These task will have a staggered start, which means that this algorithm cannot achieve ideal speedup.) Linda's tuple-space operations come into play when (1) the initial tasks are distributed, (2) boundary values are communicated between workers responsible for adjacent strips, and (3) each worker returns its best value to the master at termination. In each case, the requisite data is dropped into tuple space via Linda's **out** operation and retrieved using **in**. This represents an extremely simple use of Linda. (But note that, although this form of communication looks like mere message passing—a band-k worker must send a series of "messages" to the band-$k+1$ worker—it isn't. Processes assume the band-j role *dynamically* as the computation proceeds; workers don't know *a priori* to whom they will be sending these "messages".)

Figure 3 shows performance on two different comparisons involving two sequences of 3389 and 6779 bases respectively. The dotted line shows ideal speedup, relative to running time for a serial version, for this algorithm (recall that the algorithm is incapable of perfect speedup because of start-up and shut-

down). Task granularity drops as the number of workers increases. Consider the longer comparison in figure 3: it shows linear speedup fairly close to ideal until we reach fifty workers, where performance starts to deteriorate (although countinuing to show large absolute speedups). For this run, the task step at fifty workers required computing about 3500 entries in the comparison matrix, which takes about 60 msec. At this task granularity, we would continue to show linear speedup through all 64 nodes on a comparison of two roughly 8000-base sequences. This is a fairly modest computation. (Currently the longest sequences in Genbank are approximately 70,000 bases).

The cited performance figures predate the bucket-switching optimization. Bucket-switching is designed to detect patterns exactly like the ones in this program; preliminary data indicate that bucket switching will in fact cause messages to be sent directly from band k's node to band $k+1$'s after the first few message transactions.

Hybrid comparison strategies. Although comparing two long sequences can be expensive, geneticists usually require not a single comparison, but an extensive series of comparisons in which the target is compared against all or a major portion of a large database. It's clear, then, that we should consider parallelizing not the sequence-to-sequence comparison, but the database search—that is, have our program perform many sequential comparisons simultaneously. Such an approach is attractive because, since it involves coarser-grained parallelism, we expect a highly efficient program to result.

This approach does work well ([CG88])—except when we need to search a database partition that includes very-long sequences. The sequence-to-sequence comparison discussed above handles long sequences efficiently, but isn't appropriate otherwise. We can combine the virtues of both programs in a new and different approch that works as follows: we define as a task the computation of of an $i \times j$ sub-block of the comparison matrix. When comparing a target whose length is $\leq i$ to a database sequence whose length is $\leq j$, the entire comparison matrix fits inside an $i \times j$ block, and therefore constitutes a single task by itself. When dealing with larger comparands, the comparison is partitioned into a series of sub-block computations, which may be pursued by many workers simultaneously.

We have tested two versions of this basic approach. In the first, once a worker has chosen the first sub-block in a band, it will proceed to compute in sequence all remaining sub-blocks in the same band. As soon as it completes the first sub-block, it generates a task-descriptor tuple specifying computation of the next band (the one immediately beneath this one) as an available task; as it completes each sub-block, it generates a tuple describing that sub-block's bottom row. The performance of this program is shown in figure 4, which involves a search of the Primate sub-section of Genbank (roughly 2 MBytes) against a median-length sequence (around 500 bases) using a block size of 40,000.

figure 1

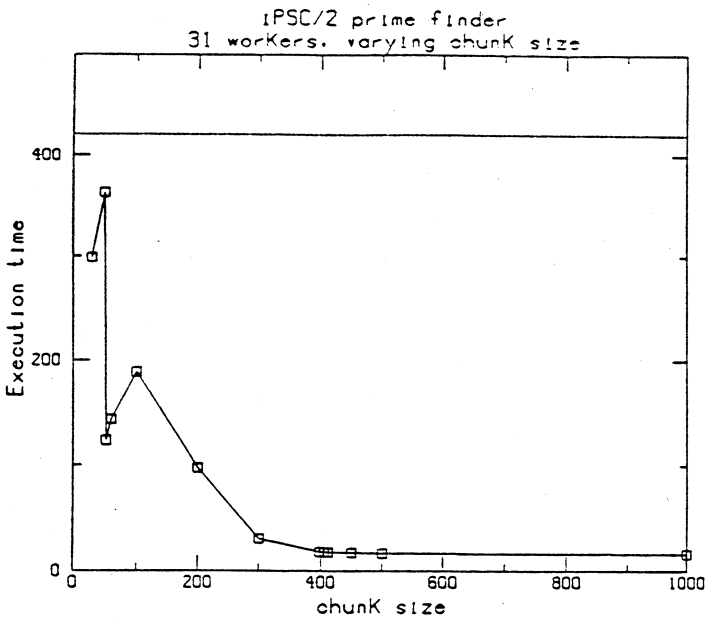
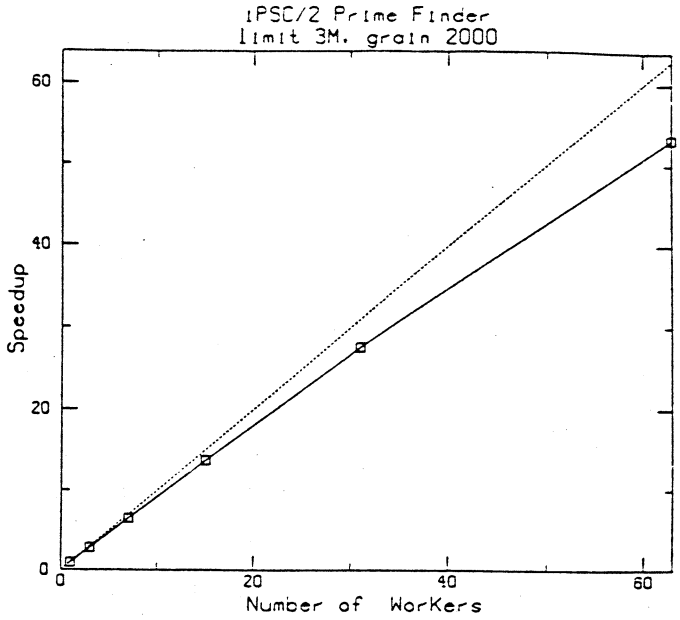


figure 2



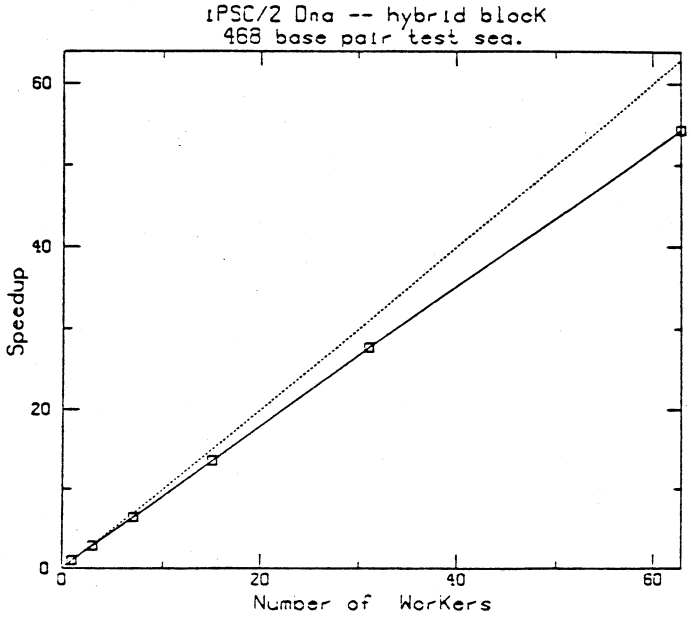


figure 4

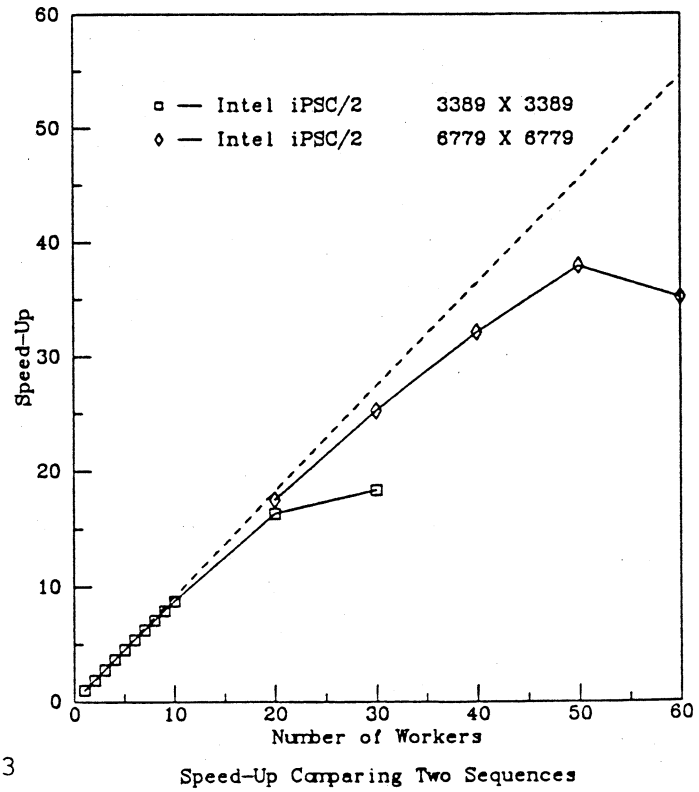


figure 3

Speed-Up Comparing Two Sequences