Parallelization of General Linkage Analysis Problems

Sandhya Dwarkadas *
Department of Computer Science
Rice University
Houston

Alejandro A. Schäffer †
Department of Computer Science
Rice University
Houston

Robert W. Cottingham Jr. ‡
Department of Cell Biology
Baylor College of Medicine
Houston

Alan L. Cox §
Department of Computer Science
Rice University
Houston

Peter Keleher ¶
Department of Computer Science
Rice University
Houston

Willy Zwaenepoel ‖
Department of Computer Science
Rice University
Houston

Address for correspondence: Robert W. Cottingham Jr. Department of Cell Biology, Baylor College of Medicine, One Baylor Plaza, Houston, TX 77030.

*sandhya@cs.rice.edu
†schaffer@cs.rice.edu
‡bwc@bcm.tmc.edu
§alc@cs.rice.edu
¶pete@cs.rice.edu
‖willy@cs.rice.edu

Abstract

We describe a parallel implementation of a genetic linkage analysis program that achieves good speed improvement, even for analyses on a single pedigree and with a single starting recombination fraction vector. Our parallel implementation has been run on three different platforms: an Ethernet network of workstations, a higher-bandwidth Asynchronous Transfer Mode (ATM) network of workstations, and a shared-memory multiprocessor. The same program, written in a shared memory programming style, is used on all platforms. On the workstation networks, the hardware does not provide shared memory, so the program executes on a distributed shared memory system that implements shared memory in software. These three platforms represent different points on the price/performance scale. Ethernet networks are cheap and omnipresent, ATM networks are an emerging technology that offers higher bandwidth, and shared-memory multiprocessors offer the best performance because communication is implemented entirely by hardware. On 8 processors and for the longer runs, we achieve speedups between 3.5 and 5 on the Ethernet network and between 4.8 and 6 on the ATM network. On the shared-memory multiprocessor, we achieve speedups in the 5.5 to 6.5 range for all runs.

1 Introduction

Genetic linkage analysis is a statistical technique that uses family pedigree information to map human genes and locate disease genes in the human genome. Several computer packages have been written for linkage computations and most published linkage studies use one of these programs [20, 21, 16, 15, 17, 11, 10, 13, 14]. As the ability to collect large family pedigrees with more informative genes has improved, the magnitude of linkage computations that geneticists want to run has increased. It is not unusual for these runs to take hours or days, and many of the cases that geneticists would like to analyze are practically intractable on current computers. We see two complementary approaches that should be pursued to speed up linkage computations: better algorithms and parallel computers.

In this paper we report on a parallel implementation of programs from the LINKAGE [16, 15, 17] package, which is a very popular general purpose set of linkage analysis programs. This paper complements research conducted by two of us together with R. M. Idury in which we significantly sped up the *sequential* algorithms in LINKAGE [2].

We focus on parallelizing the computation for a single recombination fraction vector and for a small number of pedigrees. This approach distinguishes our work from a previous parallel implementation of LINKMAP from the LINKAGE package [19], in which likelihood computations on different pedigrees and with different recombination fraction vectors are distributed on different processors. This distribution is not appropriate for the most CPU-intensive of the LINKAGE programs, called ILINK, partly because ILINK has only one starting recombination fraction vector. Furthermore, in many disease-location applications the input contains only a small number of pedigrees. There has also been a similar parallel implementation of the MENDEL [13, 14] program [5]. In her Master's thesis, Vaughan parallelizes LINKMAP for a single recombination fraction vector, but her work focuses on load balancing issues in a heterogeneous computing environment and in the presence of other workloads [23].

Unlike both previous parallel LINKAGE implementations, we start from the faster sequential LINKAGE algorithms [2] instead of the algorithms that had previously been distributed in the LINKAGE package. It is important to investigate whether the new LINKAGE algorithms are also amenable to parallel implementation, so that the advantages of better algorithms and parallel computers can be combined.

Our parallel implementation is written in a shared-memory programming style. We evaluated our implementation on two different architectures: a shared-memory multiprocessor and a network of workstations. On the network of workstations we used TreadMarks, an experimental *distributed shared memory* system under development at Rice University [7]. Distributed shared memory is a software runtime system that enables processes executing on different workstations to share memory, even though the hardware connecting the workstations only provides message passing. We experimented with two different technologies for connecting the workstations: a standard Ethernet and an ATM (Asynchronous Transfer Mode) network, which is rapidly gaining popularity because of its increased bandwidth.

We seek to address three interdisciplinary questions:

1. Is it possible to parallelize linkage computations that have as input only a small number of pedigrees and a small number of initial parameter vectors?

2. Can the new sequential algorithms [2] be effectively parallelized?

3. Can reasonable speedup be achieved for long linkage computations on a network of workstations, which is much cheaper and more commonly available than a shared-memory multiprocessor?

This paper is organized as follows. In Section 2, we explain the basics of linkage analysis and the LINKAGE programs. In Section 3 we give a short outline of the sequential algorithm for computing the likelihood. In Section 4, we describe our new parallel algorithm. In Section 5, we describe the parallel computing systems on which we tried our implementation. In Section 6, we report the performance of our implementation on some sample data. We conclude with a discussion section.

2 Summary of LINKAGE

The fundamental goal in linkage analysis is to compute the probability that a recombination occurs between two genes G_1 and G_2. The closer the genes are, the smaller the probability will be. A variety of theories connect this probability to the actual distance between the two genes on the chromosome. Two genes are said to be *linked* if the recombination probability between them is less than .5. The recombination probability is denoted by θ. A thorough treatment of genetic linkage analysis is given in Ott's monograph [22]. We review a few particulars, especially concerning the LINKAGE programs, that are relevant to our parallel implementation.

The LINKAGE package contains four related programs LODSCORE, ILINK, LINKMAP, and MLINK; we shall discuss the first three. The improved sequential algorithms in [2] are applicable to all the programs.

The LODSCORE program searches for a maximum likelihood estimate $\hat{\theta}$ of the recombination probability. The likelihood is computed with respect to the input pedigree(s). Given a set of loci, LODSCORE estimates θ for each pair of loci, but LODSCORE does not analyze more than two loci simultaneously.

The notion of recombination can be generalized to more than two loci. Suppose $G_1, G_2, \ldots, G_k$ are multiple loci occurring in that order. Then we can define a vector $(\theta_1, \theta_2, \ldots, \theta_{k-1})$, where θ_i is the recombination fraction between loci G_i and G_{i+1}. The ILINK program searches for a maximum likelihood estimate of the multilocus θ vector. Both LODSCORE and ILINK start from a single initial estimate of the recombination fraction, and use an iterative procedure called GEMINI [9] to find the maximum likelihood estimate $\hat{\theta}$. Like most iterative procedures, GEMINI can only guarantee to find a local optimum and not a global optimum.

In contrast, LINKMAP takes multiple values of the θ vector and computes the likelihood for each one. The computation of the likelihood for each θ and for each pedigree are essentially independent except for some shared input/output. The parallel implementation of LINKMAP [19] takes advantage of this observation and distributes likelihood computations for separate pedigrees and θ vectors on different processors. The main challenge is to balance the load among the different processors, so that each processor is working most of the time.

Since LODSCORE and ILINK start with only one θ vector it is not straightforward to do subcomputations for different vectors on different processors. In applications where the goal is to locate a disease gene it is our experience that the number of different pedigrees tends to be small and most of the computation time is spent on just one or two pedigrees. Therefore, we need a parallelization strategy that distributes the likelihood computation for a single pedigree and a single value of the θ vector.

We focus on ILINK because that is the program where the runs tend to be longest and thus where parallel speedup is most needed, but our techniques are applicable to the other programs as well. Almost all the code we modified is shared by all the LINKAGE programs.

3 Review of Sequential Likelihood Algorithm

The basic structure of the likelihood computation as done in LINKAGE is outlined in the section on Numerical and Computerized Methods in [22]. The following summary describes LINKAGE 5.1 [16] and its faster version [2].

Given a fixed value of θ, the outer loop of the likelihood evaluation iterates over all the pedigrees calculating the likelihood for each one. Within a pedigree, the program visits each nuclear family and updates the probabilities of each genotype for each individual. Associated with each individual is an array `genarray` indexed by genotype numbers. The entry `genarray[`j`]` initially stores the probability that the individual has the phenotype associated with genotype j given the genotype j (normally this will be 1 or 0, except in cases of variable penetrance). There may be several possible phenotypes if the individual's phenotype is incomplete in the input. After traversing the part of the pedigree including the individual, `genarray[`j`]` stores the probability that the individual has genotype j and its associated phenotype, conditioned on the recombination fraction and on the genotypes of relatives already visited in the traversal.

Each update of a nuclear family updates the probabilities for either one parent conditioned on the spouse and all children, or updates one child conditioned on both parents and all the other siblings. In both of these update situations the algorithm starts with a double nested loop that iterates over the genotypes of the two parents, one loop per parent. One of the improvements made in [2] is that in the case where there is only one child, the bulk of the computation can be transformed into two disjoint loops, one on each parent, instead of a double nested loop. A separate `gene` array is used to accumulate the contributions from each pair of parental genotypes. The only arithmetic operations done in accumulating `gene` are additions and multiplications of non-negative numbers, and the contributions for each pair of parental genotypes are *added* together. At the end of the loop, the new value of `genarray`$[i]$ is set to the old value multiplied by `gene`$[i]$ for all genotypes i. This is, in effect, an application of Bayes' Theorem that converts the original unconditioned value of `genarray`$[i]$ into a value that is conditioned on the part of the pedigree that has already been visited.

It will help to think of the double loop iteration space as a square S whose side length is the number of genotypes. The point (i, j) in the square corresponds to the first parent having genotype i and the second having genotype j.

There are two biological facts about the `genarray`s that are relevant both to the improved sequential algorithms and to our parallelization strategy. First, the `genarray`s tend to be sparse because most of the possible genotypes can be ruled out based on the observed phenotype data. One way in which sparsity is used in the improved sequential algorithms is to precompute which rows and columns of S correspond to possible genotypes of each parent, leaving a much smaller iteration rectangle R. Ignoring procedure boundaries, we can think of the loops in the sequential likelihood calculation as follows. More indentation indicates deeper nesting.

```
For each pedigree
    For each nuclear family
        For double loop over rows and columns of R
            Do updates to genarray
```

As a result, most of the computation time is spent on probability updates for individuals whose `genarray`s are not sparse. Such individuals are referred to as *unknowns* because we do not know their phenotype at some of the loci being studied.

The second useful biological fact is that the genotypes can be partitioned into equivalence classes by a relation we call the *isozygote relation* [2]. Two genotypes are *isozygotes* if at *each* locus they have the same allele(s). Isozygotes differ in the placement of the alleles on each haplotype; i.e., one isozygote could have A_1 on the first haplotype and A_2 on the second, while another has A_2 on the first haplotype and A_1 on the second. Among other things, the computations for different isozygous genotypes are very similar, and some parts of the computation can be performed once for all genotypes in the same isozygote class.

4 Parallel Algorithm

In this section we describe our strategy for parallelizing ILINK. The same strategy can be applied to LODSCORE and LINKMAP. The main theme is that some understanding of the underlying biology, in particular, the ideas of sparsity and similar patterns of heterozygosity, are essential to designing a good strategy.

Recall that ILINK takes only one starting θ vector and ILINK may have only one input pedigree. Although two of the data sets we use in the next section have more than one pedigree, the computation time is dominated by one or two pedigrees. Therefore, we cannot parallelize by doing different likelihood calculations on different machines, but must parallelize within the calculation of the likelihood at a specific θ and a specific pedigree.

The genotype probability updates for different individuals are naturally sequential because the updated probabilities for the i^{th} individual are dependent on the updates for all the previous individuals visited. Therefore, we want to parallelize each individual's probability update.

We mention as an aside that the strict sequential nature of the updates is specific to the probability update algorithms used in LINKAGE, but only partially inherent in the original update algorithm of Elston and Stewart [4]. Elston and Stewart proposed an update order that was strictly bottom-up, which would allow some updates to be done in parallel. Because of space limitations and other practical implementation concerns, LINKAGE uses an update order with the invariant that the nuclear families whose updates have been completed always form a contiguous subtree of the pedigree. If we want to keep this order, then we cannot update the probabilities in nonadjacent parts of the pedigree in parallel.

The algorithm is parallelized by splitting up the iteration space over the rectangle R among the available processors. The single `gene` array used in the sequential algorithm is replaced by a number of `gene` arrays, each one local to a particular processor. Each processor then accumulates in its local `gene` array its contributions to the updated `genarray`. When a processor finishes computing its contributions, it waits until all the processors have completed their work. Then one processor obtains the contributions to `gene` from each processor, sums them together, and uses the resulting value of `gene` to update `genarray` in the same way as in the sequential algorithm. Since the contributions for each pair of parental genotypes are simply *added* together, they can be accumulated *locally* by each processor and summed together at the end. By using a local array to compute the contributions and summing them at the end, we avoid communication and synchronization at each update.

In order to achieve good speedups, R needs to be partitioned in a way that balances the load among the different processors. Different points in R may require different amounts of computation. For simplicity, suppose that we decide which processor gets the point (G_1, G_2) in R based only on the first parent's genotype, leading to the following loop structure:

```
For each pedigree
    For each nuclear family
        Split up rows of R into p sets
        For each processor
            Do updates to gene for assigned rows
        Synchronize processors to sum update together
```

We use a fact relating the computation time to the underlying biology in order to distribute the points in R among the processors with a good load balance. If G_1 and G_1' have the same pattern of heterozygosity and G_2 and G_2' have the same pattern of heterozygosity, then the sequence of arithmetic operations for the update at (G_1, G_2) is similar to that for (G_1', G_2'). Therefore, we distribute the genotypes among the processors, so that for each heterozygosity pattern, the possible genotypes with that pattern are distributed evenly among the processors. For reasons unrelated to our parallel implementation, the genotypes are already ordered so that all the genotypes with the same heterozygosity pattern are consecutive.

Suppose that $H_1, H_2, \ldots$ are the possible genotypes of the first parent. To balance the load we assign the genotypes to processors in a round-robin or striped fashion: H_1 goes to processor 1, H_2 to processor 2,..., H_p to processor p, H_{p+1} to processor 1, H_{p+2} to processor 2, ..., H_{2p} to processor p and so on. If the number of possible genotypes is large, then most of the consecutive sets of p items will have the same heterozygosity pattern, resulting in good load balancing. When there is a double loop, we do a striped assignment for the parent corresponding to the outer loop and the rows in R; within each row, all the genotype pairs in that row corresponding to different columns get assigned to the same processor. When we have two separate loops in the one-child case, we get two one-dimensional iteration spaces, and we do a striped assignment for each parent separately.

There are a few points in the computation where all the processors must synchronize and share their data. One is at the distribution of points in R or the one-dimensional spaces to all the processors, and another occurs just before we sum the contributions to `gene` from each processor. In the case where we are updating a parent based on its spouse and children, there is one more synchronization point needed so that an intermediate table can be propagated to all processors. This table stores for each haplotype, the probability that the second parent (the one in the inner loop) passes that haplotype on to a child. This table was introduced in [2] to speed up the sequential computation.

We applied the idea of sparsity one more time to further improve performance. Recall from the section discussing the sequential algorithm that most of the running time in the likelihood calculation is spent on those nuclear families where at least one parent's `genarray` is not sparse. This means that R will have a large area. We found that when R is sufficiently small, it is actually detrimental to perform the updates in parallel because of the overhead involved in data distribution and synchronization. Therefore, for the runs on a network of workstations we defined a threshold for the size of R; if R is smaller than the threshold, we do the update for that nuclear family using only one processor. For the experiments we report later, we set the threshold to be the sum of the two sides of $R < 100$ for the one-child case, or the product of the two sides of $R < 3000$ for the many-child case. We did not experiment extensively with different thresholds. There were a small number of nuclear families where the size of R was at or near the threshold. In almost all cases the size of R is much smaller or much larger than the threshold. Thus minor variations in the threshold do not result in noticeable changes in performance. When running on a shared-memory multiprocessor, the cost of synchronization is minimal, and hence the threshold was set to 0.

5 Methods

We evaluated parallel ILINK on two different types of parallel computers: a network multicomputer and a shared-memory multiprocessor. A network multicomputer is simply a cluster of ordinary workstations connected by a general-purpose local area network, such as ATM, Ethernet, or FDDI (which stands for Fiber Distributed Data Interface and is a 100 Megabit/s local area network in which the stations are connected in the form of a ring). In contrast, a shared-memory multiprocessor is a single machine containing several processors that are connected by a specially-designed bus or dedicated network.

These two types of parallel computers present different tradeoffs between cost and performance. On one hand, network multicomputers are cheaper. In fact, in many laboratories, the required hardware for a network multicomputer is already present. On the other hand, shared-memory multiprocessors are faster, because they implement communication and synchronization entirely by hardware. On workstations, a large software overhead is associated with sending and receiving messages over the network. For parallel computations where the individual processors communicate with each other frequently, shared-memory multiprocessors typically achieve better performance. The advent of faster general-purpose networks is, however, narrowing the performance gap between workstation networks and shared-memory multiprocessors.

Besides differences in cost and performance, shared-memory multiprocessors and workstations also typically present different communications interfaces. In a network multicomputer, processors communicate by passing messages with `send` and `receive` operations. A shared-memory multiprocessor supports communication by reading and writing shared memory. Fundamentally, neither mechanism is more powerful than the other. Either mechanism can be used to simulate the other through software. However, most sequential programs, including ILINK, are more easily parallelized in terms of shared memory. To use message passing, the programmer must write additional code to copy data into and out of message buffers and perform `send` and `receive` operations.

Motivated by the difficulty of writing message passing programs, we have developed a software *distributed shared memory* (DSM) system for network multicomputers called TreadMarks [7]. In essence, TreadMarks provides a shared memory abstraction to the programmer, and implements this abstraction efficiently using the underlying message passing system [8, 3]. Thus the programmer writes the program as if it were intended for a shared-memory multiprocessor, but the TreadMarks system enables the program to run on a network multicomputer.

At the present time, TreadMarks is still under development. We expect it to be ready for distribution some time in 1994. TreadMarks will be made available at low cost to universities and nonprofit institutions. At that time, we intend to distribute the parallel LINKAGE code that runs on top of TreadMarks.

The network multicomputer used to perform our evaluation of parallel ILINK consists of 8 DECstation-5000/240 workstations, each with 16 Mbytes of memory, running the Ultrix version 4.3 operating system. All of the workstations are connected to an Ethernet and a high-speed ATM network. TreadMarks can utilize either the Ethernet or the ATM network. The interface for Ethernet is a standard component of the workstation. The interface for ATM is a Fore Systems TCA-100 network adapter card supporting communication at 100 Megabit/s.

The shared-memory multiprocessor used to perform our evaluation of parallel ILINK is a Silicon Graphics Iris 4D/480 with 128 Mbytes of memory running the IRIX Release 4.0.1 System V operating system. This machine has 8 processors that communicate via a dedicated bus.

An important aspect of our evaluation is that the DECstation-5000/240 and the SGI Iris 4D/480 use the same type of processor running at the same speed. In addition, we used the same compiler, gcc 2.3.3 with -O flag for optimization, on both machines. The only significant difference between the two parallel computers is the method for implementing shared memory: dedicated hardware versus software on message-passing hardware.

6 Results

We present speedups for parallel LINKAGE with several input data sets. Uniprocessor execution times are given as well so that execution time differences may be inferred. We use two different network types - the commonly available Ethernet networks and the emerging ATM networks. The performance obtained on a network of workstations is then compared to the performance on a shared-memory machine with identical processor power.

We use two disease data sets from [2] and a new data set:

- RP01: data on a large family, UCLA-RP01, with autosomal dominant retinitis pigmentosa (RP1) from the laboratory of Dr. Stephen P. Daiger at the University of Texas Health Science Center at Houston. This pedigree has 7 generations with 189 individuals containing 2 marriage loops [1]. There are 86 individuals that are unknown at some loci. As shown in [1], this pedigree had to be split into three pieces because computation on the whole family together was prohibitively long. RP01-3 denotes the analysis with the family split in three pieces.

- BAD: data on a portion of the Old Order Amish pedigree 110 (OOA 110), with bipolar affective disorder (BAD) from the laboratory of Drs. David R. Cox and Richard M. Myers at the University of California at San Francisco. This pedigree spans 5 generations with 96 individuals and contains 1 marriage loop [18]. Data is available for three loci, the disease locus (number 1) and two others. 15 individuals are unknown at locus 2 and those same 15 plus 3 more are unknown at locus 3.

- CLP: Data on 12 families with autosomal dominant nonsyndromic cleft lip and palate (CLP) from the laboratory of Dr. Jacqueline T. Hecht at the University of Texas Health Science Center at Houston. Diagrams of the families are shown in [6]. Subsequent to that paper data was collected on 9 more individuals augmenting 4 of the families. The families include 110 individuals in all. We list for each family, the identifying number given in [6], the number of individuals, the number of generations, and the number of individuals that are unknown at at least one of the loci we used: [(100,9,2,1) (300,4,2,0) (500,6,2,0) (600,7,3,0) (700,9,3,1) (800,7,3,1) (900,8,3,1) (1000,17,3,3) (1100,22,3,12) (1200,11,2,2) (1400,6,2,3) (1500,4,2,0)]. The computation time is dominated by pedigrees 1000 and 1100 because of the larger size and the unknowns, although pedigree 1200 has a marriage loop.

The loci chosen for the RP01-3 data set have an allele product of $2 \times 6 \times 9$. Those for the BAD and CLP data sets have allele products of $2 \times 4 \times 4$, and $2 \times 4 \times 4 \times 4$, respectively. In all cases, the 2-allele locus is the disease locus, and these runs represent real runs one might want to execute in locating the disease gene.

In addition, we also compare the program on different sets of loci from the same pedigree set. In particular, we use three different sets of loci from the RP01-3 data set, with allele products $2 \times 6 \times 6$, $3 \times 5 \times 2 \times 3$, and $2 \times 6 \times 9$. This comparison shows how the running time changes as the allele product changes, but other factors stay the same. This is motivated by a common usage pattern for the LINKAGE programs. Once a set of pedigree information is collected, it is common for geneticists to do many linkage analysis runs on it, changing the set of loci each time.

The speedup figures are based on one-processor execution times for the faster version of ILINK used in the tests for [2], but run on a DECstation-5000. Table 1 presents the uniprocessor execution times on the DECstation-5000 workstations. All execution times are reported in seconds. In all the speedup graphs (Figures 1 to 6) the horizontal axis represents the number of processors and the vertical axis

RP01-3	RP01-3	RP01-3	BAD	CLP
2x6x6	3x5x2x3	2x6x9	2x4x4	2x4x4x4
901	10005	4805	848	6388

Table 1: Uniprocessor Execution Times in Seconds on the DECstation-5000/240

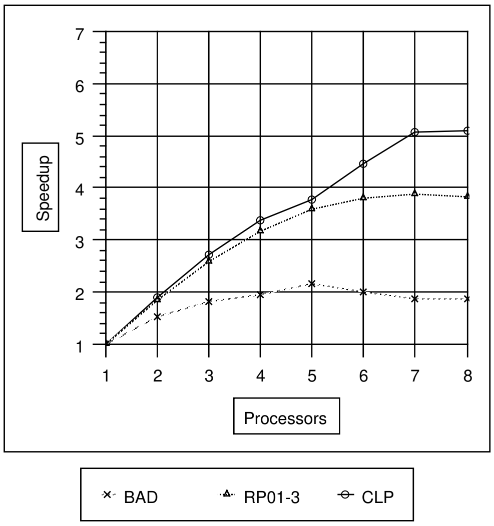

Figure 1: Speedup on an Ethernet Network - Different pedigrees

represents the speedup. The parallel version described here (with the code reorganization for load balancing and extra TreadMarks code) ran in approximately the same time on one processor as the sequential version of the code. The difference in running time was always a few seconds compared to the thousands of seconds of total execution time. Thus the use of TreadMarks code does *not* slow down the execution for one processor.

Figure 1 shows speedups for a run from each of the three data sets described on an Ethernet network. In Figure 2 we plot the speedups obtained on an Ethernet network using the three different sets of loci from the RP01-3 data set.

Figures 3 and 4 present speedups using an ATM network in place of the Ethernet network with the same runs as in the previous figures. While the performance of the program on an Ethernet network is reasonable, better speedups are obtained with the ATM network. The faster network removes part of the communication bottleneck, closing the gap in performance between different data sets and loci.

To determine the difference between the performance obtained using a network of workstations and the performance that is possible on a hardware shared-memory system, we present results for the same program running on an SGI shared-memory multiprocessor. Figures 5 and 6 show that the speedups achieved are slightly better

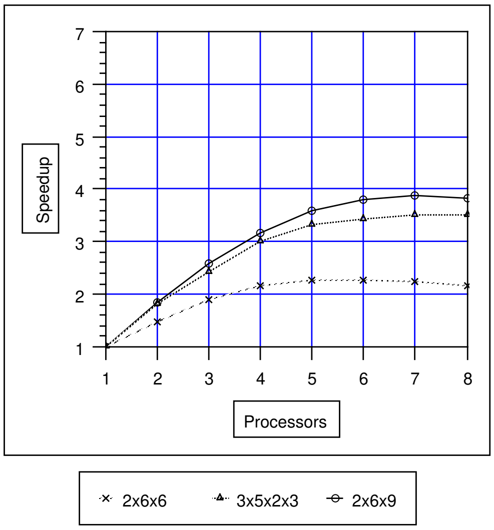

Figure 2: Speedup on an Ethernet Network - RP01-3 pedigree, different loci

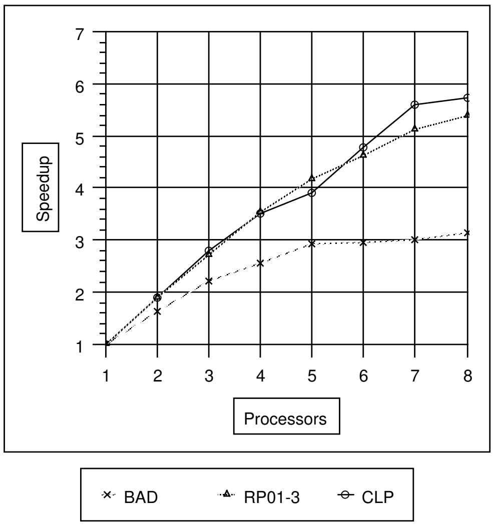

Figure 3: Speedup on an ATM Network - Different pedigrees

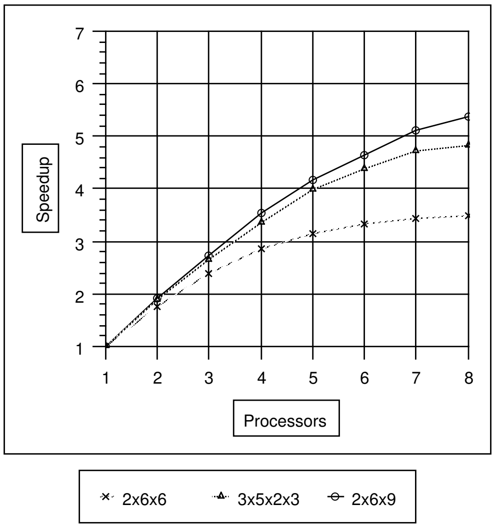

Figure 4: Speedup on an ATM Network - RP01-3 pedigree, different loci

RP01-3 2x6x6	RP01-3 3x5x2x3	RP01-3 2x6x9	BAD 2x4x4	CLP 2x4x4x4
904	9585	4808	936	6208

Table 2: Uniprocessor Execution Times in Seconds on an SGI Multiprocessor

than those obtained using the ATM network for the larger problems. On the smaller problems, the SGI machine does much better. The SGI uniprocessor execution times are presented in Table 2.

Two factors contribute to the less than perfect speedup observed in the experiments: load imbalance and — on TreadMarks — communication overhead.

Perfect load balancing cannot be achieved because of imperfect knowledge of the combined genotypes possible for the two parents. While our load balancing strategy takes advantage of the sparsity of each parent's `genarray`, it may be the case that a pair of genotypes (i, j) is not simultaneously possible, although i is possible for the first parent and j is possible for the second parent. An alternative strategy would be to determine the possible combinations on the master processor before distributing the work. The increase in sequential computation would, however, outweigh the benefits of better load balancing. Load imbalance as a result of the unequal assignment of possible genotype pairs is present to some degree in all the data sets.

The problem of deciding whether a pair of parental genotypes (i, j) is compatible with the children is different from the problem of genotype elimination as addressed in [12] or in the `unknown` preprocessor program that is part of LINKAGE. The

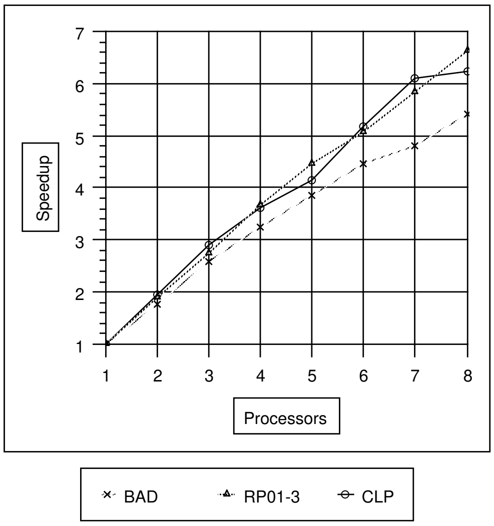

Figure 5: Speedup on an SGI Multiprocessor - Different pedigrees

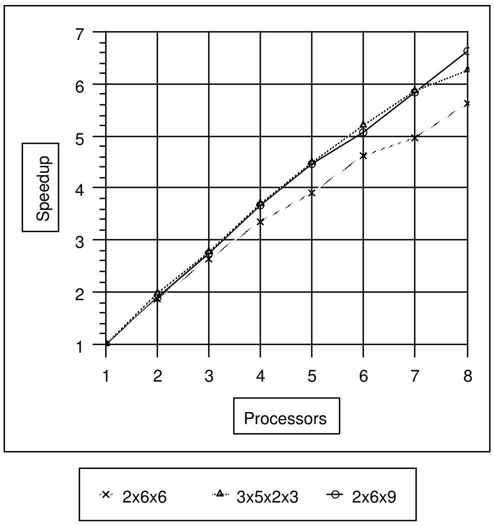

Figure 6: Speedup on an SGI Multiprocessor - RP01-3 pedigree, different loci

distinction can best be illustrated with a trivial one-locus, two-allele, one-child example. Let the parents be p and q and the child be r. Suppose that for each of p and q the possible genotypes are $\{1|2, 2|2\}$ and that r's genotype is known to be $1|2$. From this information it can be inferred that it is not simultaneously possible that *both* p and q have genotype $2|2$, but it is possible that either one has genotype $2|2$, while the other has genotype $1|2$. Such a situation is detected within the likelihood calculation itself and not by any genotype elimination algorithm. Genotype elimination algorithms only eliminate genotypes that an individual cannot have; they do not eliminate combinations of genotypes that collections of individuals cannot have simultaneously. It would require too much storage to precompute the set of possible genotype combinations for all nuclear families, even for moderate-size 3-locus problems.

Communication overhead adds to the decline in speedup on TreadMarks. The effect of communication overhead on the speedup depends on the input pedigree and the loci for which the θ vector is being estimated. As the number of possible alleles increases, the length of the computation increases with little change in the amount of data communicated, resulting in improved speedups. The run with 4 loci takes much longer because it takes many more likelihood estimates to converge, but the time per likelihood estimate is comparable to the large 3-locus run.

In addition, the presence of loops in the pedigreee can further increase the communication rate. One example is BAD. Much of BAD's complexity comes from the presence of the loop in the pedigree rather than a high allele product. The way that LINKAGE handles loops in the pedigree is that the input format designates one individual to be the *loop breaker*. For each evaluation of the likelihood estimate, LINKAGE does a separate traversal of the pedigree for each genotype that the loop breaker individual may have. Thus, one evaluation of the likelihood estimate may include many pedigree traversals. In each traversal each nuclear family update is parallelized as before, resulting in many small pieces of work and large communication overhead. For more details on how LINKAGE handles loops see pages 170–171 of [22].

As discussed in the section describing the parallel algorithm, when the size of the reduced iteration rectangle R drops below a threshold, the TreadMarks version performs the computation sequentially to avoid high communication overheads. Although this method reduces communication, it leads to further load imbalance in the TreadMarks version. This contributes to our algorithm's poor performance on BAD. This explanation is supported by the imbalance in synchronization wait times between the master processor that performs the sequential computation and all the other processors (a ratio of 1 to 6.5 on average in wait times at 8 processors on the ATM network). However, performing all the computation in parallel only results in worse speedup because of the small amount of computation relative to the communication overhead.

For each of the runs and data sets on the ATM network, Table 3 provides the

# Procs		RP01-3 2x6x6	RP01-3 3x5x2x3	RP01-3 2x6x9	BAD 2x4x4	CLP 2x4x4x4
2	msgs/sec	74	31	26	101	12
	%waittime	5.7	4.3	3.1	10.0	2.2
4	msgs/sec	477	225	189	533	80
	%waittime	14.1	10.7	7.5	20.4	8.7
8	msgs/sec	2019	1099	1042	1800	449
	%waittime	25.7	26.1	17.1	33.3	20.0

Table 3: Overhead Statistics on an ATM Network

average number of messages per second (`msgs/sec`), and the percentage of total running time that each processor on average spends waiting at synchronization points (`%waittime`). The number of messages per second and the percentage wait time increase with increasing number of processors, explaining the decreasing slope of the speedup curves.

The data in Table 3 also allows us to derive a quantitative estimate of the relative contributions of load imbalance and communication overhead to the decline in speedup on both the SGI and the ATM network. The formula

$$\text{speedup} = 8 \times (1 - \%\text{waittime}/100)$$

estimates the speedup if load imbalance were the only limiting factor and communication overhead were negligible. Focusing on the results with 8 processors, Table 4 shows that this predicted speedup matches very well with the observed speedup on the SGI for all data and input sets. In order to estimate the effect of communication overhead on the speedup, we assume that the reduction in speedup is linear in terms of the number of messages per second, or

$$\text{speedup} = 8 \times (1 - \%\text{waittime}/100) - \text{F} \times \text{msgs/sec}$$

where F is determined by a least squares fit. Again, the match between the predicted speedups and the speedups observed on the ATM is remarkable (see Table 5). These derivations, although approximate, confirm our basic conclusions: Speedup on the SGI is limited by load imbalance, while speedup on the ATM network is limited by a combination of load imbalance and communication overhead.

The experiments show that our parallel algorithm does a reasonable job of balancing the load between processors, and can achieve good speedups on runs that have a large computation-to-communication ratio. While some speedup can be obtained using the Ethernet, performance closer to that of a shared-memory multiprocessor is possible using an ATM network on large runs. For small runs, such as BAD, it is not clear that a parallel implementation of any sort is of much benefit.

ATM networks are gaining popularity because they are suitable for use in both high-performance local-area and wide-area networks. Our experiments show that

	RP01-3 2x6x6	RP01-3 3x5x2x3	RP01-3 2x6x9	BAD 2x4x4	CLP 2x4x4x4
Estimated speedup	5.94	5.91	6.63	5.34	6.40
Observed speedup	5.61	6.26	6.64	5.41	6.23

Table 4: Estimated and Observed Speedup on SGI: 8 Processors

	RP01-3 2x6x6	RP01-3 3x5x2x3	RP01-3 2x6x9	BAD 2x4x4	CLP 2x4x4x4
Estimated speedup	3.47	4.56	5.35	3.14	5.85
Observed speedup	3.49	4.83	5.38	3.15	5.73

Table 5: Estimated and Observed Speedup on ATM Network: 8 Processors

for linkage analysis, the performance of a shared-memory multiprocessor can be obtained at a fraction of the cost, without compromising the convenient shared memory abstraction presented to the programmer.

To be a little more quantitative about price/performance, in the Fall of 1992, the cost ratio was approximately 1:2:3 comparing 8 DECstations connected by Ethernet, 8 DECstations connected by ATM, and the SGI 4D/480, respectively, with prices falling for the ATM network and the shared-memory multiprocessor. Averaged over all data and input sets, the speedups achieved on 8 processors are 3.3 for the Ethernet, 4.5 for the ATM network, and 6.0 for the SGI. Although such comparisons need to be taken with a grain of salt, the Ethernet currently offers the best price/performance ratio, while the SGI offers the best performance.

7 Discussion

The structure of general pedigree linkage computations using the likelihood method does not lend itself very well to vector processing or fine grain parallel processing. A coarse-grain parallel machine, where a large memory is provided with each processor, is more suitable for programs such as LINKAGE. We have shown that using TreadMarks, a new distributed shared memory system, has resulted in significant performance improvement on all types of genetic linkage analysis problems. This includes problems involving a small number of pedigrees. Large single pedigrees frequently are the basis of disease studies and require some of the longest computation times. Performance results for this type of problem are specifically covered in the previous section.

Genetic linkage analysis is computationally intensive. With the recent growth in the number and informativeness of genetic markers, computation times have increased dramatically. Our research into ways to reduce computation time previously led to improvements in the *sequential* algorithms. Here, we demonstrate a general purpose method for processing the LINKAGE programs in *parallel* on a network of workstations that further reduces computation time as the number of available processors grows.

We presented a parallelization strategy that works even for single pedigrees and single starting vectors. Our strategy makes good use of the underlying biological theory and focuses on getting good speedup for long runs. Nevertheless, we are exploring some modifications of our strategy in the hopes of further improving parallel performance. We give three examples.

When ILINK estimates the partial derivatives of the likelihood function in multilocus analysis, a separate likelihood evaluation is done for each dimension of the θ vector. These evaluations could be done in parallel.

The pedigree traversal and nuclear family updates are very similar for each choice of θ. Therefore, it might make sense to measure the time of a given distribution of work among the processors at one function evaluation and use the timing results to better distribute the work on the next function evaluation.

The final idea in our parallelization strategy was to do a threshold test on the size of the iteration space R. Based on the result of the test we either used all processors or only one processor to do that update. One might consider a variety of options for how many processors to use (e.g., 2, 3, 4, or more) depending on the size of R.

As shown above, the parallel implementation provides a reasonable speedup for the number of available processors. Speedups improve as the size of the computation involved increases. Because the two methods of speed improvement, algorithmic and now parallel processing are implemented in completely independent ways, the speedups compound.

As TreadMarks becomes a mature software package we intend to organize large numbers of workstations on our local network for processing long linkage analyses. Because most workstations have a large amount of free cycles available, especially at night and on weekends, we would make use of these for linkage problems. If our local efforts are successful, we may wish to expand the network more broadly for very large problems and organize a linkage analysis consortium over Internet.

This effort confirmed our experience that a synthesis of the biology and computer science knowledge relevant to the problem is necessary to make linkage analysis software run much faster. We concur with the authors of [19] that to parallelize the LINKAGE programs effectively, the programmer must put *explicit* parallel instructions in the programs *using knowledge of the underlying genetic application domain.* Automatic parallelization tools and blind reliance on massive hardware installations are no substitute for human reasoning about the genetics and the algorithms.

Acknowledgments

We thank Dr. Stephen P. Daiger, Dr. Lori A. Sadler, Dr. David R. Cox, Dr. Richard M. Myers, Dr. Susan H. Blanton and Dr. Jacqueline T. Hecht for contributing the disease family data for our experiments. Development of the RP data was supported by grants from the National Retinitis Pigmentosa Foundation and the George Gund Foundation. The Amish family data was developed with the support of a grant from the National Institutes of Health. Development of the CLP data was supported by grants from the National Institutes of Health and Shriners Hospital. We thank Dr. Michael Scott at the University of Rochester for providing us access to their Silicon Graphics multiprocessor. The machine was purchased with funds from a National Science Foundation grant. We thank Honghui Lu for her work in getting the linkage programs running on the SGI. This work was supported by grants from the National Science Foundation, the Texas Advanced Technology Program, the Human Genome Program of the National Institutes of Health, and the W. M. Keck Foundation.

References

[1] S. H. Blanton, J. R. Heckenlively, A. W. Cottingham, J. Friedman, L. A. Sadler, M. Wagner, L. H. Friedman, and S. P. Daiger. Linkage mapping of autosomal dominant retinitis pigmentosa (RP1) to the pericentric region of human chromosome 8. *Genomics*, 11:857–869, 1991.

[2] R. W. Cottingham Jr., R. M. Idury, and A. A. Schäffer. Faster sequential genetic linkage computations. *American Journal of Human Genetics*, 53:252–263, 1993.

[3] S. Dwarkadas, P. Keleher, A.L. Cox, and W. Zwaenepoel. Evaluation of release consistent software distributed shared memory on emerging network technology. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 144–155, May 1993.

[4] R. C. Elston and J. Stewart. A general model for the analysis of pedigree data. *Human Heredity*, 21:523–542, 1971.

[5] T. M. Goradia, K. Lange, P. L. Miller, and P. M. Nadkarni. Fast computation of genetic likelihoods on human pedigree data. *Human Heredity*, 42:42–62, 1992.

[6] J. T. Hecht, Y. Wang, B. Connor, and S. H. Blantonand S. P. Daiger. Nonsyndromic cleft lip and palate: No evidence of linkage to hla or factor 13a. *American Journal of Human Genetics*, 52:1230–1233, 1993.

[7] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. To appear in *Proceedings of the 1994 Winter USENIX Conference*, 1994.

[8] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[9] J. M. Lalouel. GEMINI - a computer program for optimization of general nonlinear functions. Technical Report 14, University of Utah, Department of Medical Biophysics and Computing, Salt Lake City, Utah, 1979.

[10] E. S. Lander and P. Green. Construction of multilocus genetic linkage maps in humans. *Proc. Nat. Acad. Sci. USA*, 84:2363–2367, 1987.

[11] E. S. Lander, P. Green, J. Abrahamson, A. Barlow, M. J. Daly, S. E. Lincoln, and L. Newburg. MAPMAKER: An interactive computer package for constructing primary genetic linkage maps of experimental and natural populations. *Genomics*, 1:174–81, 1987.

[12] K. Lange and T. M. Goradia. An algorithm for automatic genotype elimination. *American Journal of Human Genetics*, 40:250–256, 1987.

[13] K. Lange, D. Weeks, and M. Boehnke. Programs for pedigree analysis: MENDEL, FISHER, and dGene. *Genetic Epidemiology*, 5:471–473, 1988.

[14] K. Lange and D. E. Weeks. Efficient computation of lod scores – genotype elimination, genotype redefinition, and hybrid maximum likelihood algorithms. *Annals of Human Genetics*, 53:67–83, 1989.

[15] G. M. Lathrop and J. M. Lalouel. Easy calculations of lod scores and genetic risks on small computers. *American Journal of Human Genetics*, 36:460–465, 1984.

[16] G. M. Lathrop, J. M. Lalouel, C. Julier, and J. Ott. Strategies for multilocus linkage analysis in humans. *Proc. Natl. Acad. Sci. USA*, 81:3443–3446, June 1984.

[17] G. M. Lathrop, J. M. Lalouel, C. Julier, and J. Ott. Multilocus linkage analysis in humans: detection of linkage and estimation of recombination. *American Journal of Human Genetics*, 37:482–498, 1985.

[18] A. Law, C. W. Richard III, R. W. Cottingham Jr ., G. M. Lathrop, D. R. Cox, and R. M. Myers. Genetic linkage analysis of bipolar affective disorder in an old order amish pedigree. *Human Genetics*, 88:562–568, 1992.

[19] P. L. Miller, P. Nadkarni, J. E. Gelernter, N. Carriero, A. J. Pakstis, and K. K. Kidd. Parallelizing genetic linkage analysis: A case study for applying parallel computation in molecular biology. *Computers and Biomedical Research*, 24:234–248, 1991.

[20] J. Ott. Estimation of the recombination fraction in human pedigrees– efficient computation of the likelihood for human linkage studies. *American Journal of Human Genetics*, 26:588–597, 1974.

[21] J. Ott. A computer program for linkage analysis of general human pedigrees. *American Journal of Human Genetics*, 28:528–29, 1976.

[22] J. Ott. *Analysis of Human Genetic Linkage*. The Johns Hopkins University Press, Baltimore and London, 1991. Revised edition.

[23] M.S. Vaughan. A distributed approach to human genetic linkage analysis. M.S. Thesis, Department of Computer Science, Duke University, 1991.