Faster Sequential Genetic Linkage Computations

Robert W. Cottingham Jr. [*]
Department of Cell Biology
Baylor College of Medicine
Houston

Ramana M. Idury [†]
Department of Computer Science
Rice University
Houston

Alejandro A. Schäffer [‡]
Department of Computer Science
Rice University
Houston

September 30, 1992

Revised February 9, 1993

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

[*] bwc@bcm.tmc.edu
[†] idury@cs.rice.edu
[‡] schaffer@cs.rice.edu

Abstract

Linkage analysis using maximum likelihood estimation is a powerful tool for locating genes. As available data sets have grown, the computation required for analysis has grown exponentially, and become a significant impediment. Others have previously shown that parallel computation is applicable to linkage analysis and can yield order of magnitude improvements in speed. In this paper, we demonstrate that algorithmic modifications can also yield order of magnitude improvements, and sometimes much more.

Using the software package LINKAGE, we describe a variety of algorithmic improvements we have implemented, demonstrating how these techniques are applied, and their power. Experiments show that these improvements speed up the programs by an order of magnitude on problems of moderate and large size. All improvements were made only in the combinatorial part of the code, without resorting to parallel computers. These improvements synthesize biological principles with computer science techniques to effectively restructure the time-consuming computations in genetic linkage analysis.

Introduction

With the advent of index maps of the human genome (NCHGR article 1991), maximum likelihood linkage analysis is becoming a powerful tool for rapidly locating disease genes. However, as effort has been expended to collect more data, the computational effort has increased exponentially and thereby significantly increased the time to getting a result. Factors which contribute to increased computational effort include increases in the number of genetic markers being considered simultaneously, the number of alleles per marker, the number of unknown individuals in a pedigree, pedigree size, and the degree of inbreeding. Of course, each of these factors provides additional information and is therefore of interest. As the genetic map is further improved and automated genotyping becomes commonly available, computational effort will further increase to the point of becoming the bottleneck in obtaining results given the current tools.

LINKAGE (Lathrop et al. 1984; Lathrop and Lalouel 1984; Lathrop et al. 1986) is one of the most popular collections of genetic linkage analysis programs, especially in the study of human disease genes, and is representative of current maximum likelihood linkage analysis packages based on the Elston-Stewart algorithm (Elston and Stewart 1971). Therefore, we chose to demonstrate our concepts by modifying the programs in this package.

Given the goal of significantly increasing the speed of the programs, one method would be to implement the programs on a parallel computer system. However, before doing so it is good practice to first improve the sequential speed of any algorithm since failure to do so will multiply the inefficiencies in the parallel environment.

We describe and validate a collection of improvements to the sequential implementation of the central, time-consuming, probability calculations in LINKAGE that speed up the programs substantially. We begin in the next section by describing the aspects of LINKAGE that are relevant to our improvements. The Methods section gives a brief description of the software and hardware tools used. Then we describe the biological and computer science basis of our algorithmic improvements. Finally the experimental results are given, showing how much the algorithmic modifications improved the speed of the LINKAGE programs, on some sample data sets. And we conclude with a short discussion section.

Summary of the LINKAGE package

The computationally intensive programs in LINKAGE (LODSCORE, ILINK, MLINK, LINKMAP) are structured to solve a variety of genetic linkage problems. The algorithms we have improved are used in all four programs. To simplify our description, we focus almost exclusively on LODSCORE, which does two-point analyses, but later we also present timing data on our new version of ILINK. The improvements are more important in ILINK, which does multipoint analysis, because the running time of multipoint analyses can be prohibitive. However, they are easier to explain in the two-point context of LODSCORE.

The primary input to LODSCORE is a list of family pedigrees and phenotype data. To whatever extent it is known, the phenotype of each individual is listed for a number of loci. In any given run, a set of loci is specified and LODSCORE computes an estimate of the recombination fraction θ for each pair of loci in the set. For simplicity, we assume in this description that only one pair of loci is specified. LODSCORE can analyze both autosomal and sex-linked markers and there are various allowable input formats, such as binary factors and numbered alleles. Again to simplify our discussion, suppose that the two loci are autosomal and that the genetic data is encoded by numbered alleles.

If we had perfect information, then at each locus i, where there are n_i alleles, each individual would be encoded with an allele number for each of the two chromosome strands with phase known. For example, if the first locus has 5 alleles and the second locus has 8 , we could have an encoding that looks like:

$$\begin{array}{cc} 2 & 4 \\ | & | \\ 3 & 7 \end{array}$$

where 2 and 4 are allele numbers for locus 1, while 3 and 7 are for locus 2. The values 2 and 3 come from the same chromosome strand, as do 4 and 7.

In practice, perfect information is not known. Typically, we have genotype information but do not know the phase (i.e., we don't know which alleles are on which chromosome). We may have only phenotype information (i.e., various genotypes are possible, but some are excluded), or we may have no information at all. Any uncertainty in genotypes or phase is accounted for by assigning nonzero (conditional) probabilities to several genotypes that any individual might have.

The program combines the two values on a chromosome into a joint haplotype number and then combines the two joint haplotypes into a joint genotype. In the example above, there would be $5 \times 8 = 40$ possible joint haplotypes and $40 \times 41/2 = 820$ possible joint genotypes (joint genotypes that differ only by flipping the role of two strands are not distinguishable).

The basic structure of the computation is outlined in the section on Numerical and Computerized Methods in Ott's book (Ott 1991). The programs have an outer loop that iteratively updates the estimate of θ using Lalouel's GEMINI optimization package (Lalouel 1979) to search for an optimal value $\hat{\theta}$ that maximizes the likelihood. Inside the main loop, LODSCORE traverses the pedigrees updating the probabilities of each joint genotype for each individual. Associated with each individual is an array **`genarray`** indexed by the joint genotype numbers. The entry **`genarray[`**j**`]`** initially stores the probability that the individual has the phenotype associated with genotype j given the joint genotype j (normally this will be 1 or 0, except in cases of variable penetrance). 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 genotypes of the individual's relatives and the recombination fraction. For example, to update the probability **genarray[j]** for a child, we multiply the old **genarray[j]** value by the probability that the child inherits genotype j conditioned on the probabilities of parents and siblings. Using the notation of (Lathrop and Lalouel 1988), this can be expressed as:

$$P(X, G \mid Y, \theta) \ = \ P(X \mid G) \ \ P(G \mid Y, \theta)$$

where X is the joint phenotype of the child, G is the joint genotype and Y represents the collective joint phenotypes of the relatives of X that have already been traversed. A similar expression can be written for updating a parent in terms of its children. The probabilities are generally very small, so they are scaled up by a multiplicative factor to avoid numerical floating point errors.

Elston and Stewart (Elston and Stewart 1971) devised a clever bottom-up strategy for traversing certain simple pedigrees to rapidly compute the conditional probabilities. This bottom-up strategy was later generalized by Ott, Lange, and Elston among others (Ott 1974; Lange and Elston 1975) to more general pedigrees. As explained on pages 170–171 of (Ott 1991), the LINKAGE programs implement a variation of the strategy described in (Lange and Elston 1975) where the traversal algorithm can either traverse in an upward direction or in a downward direction. The freedom to go in either direction allows the program to *peel* a contiguous subtree of pedigree consecutively.

Depending on the direction of the traversal taken to reach a particular individual, one of two fundamental routines is used to update the values in **genarray**. In one direction, **segdown** updates the probabilities for a child based on probabilities for the parents and other children. In the other direction, **segup** updates the probabilities for parents based on the probabilities for their children. Both routines use an important auxiliary routine **segfun** in the inner loop of the computation when there are multiple children. There are several variants of **`segdown`**, **`segup`**, and **`segfun`** that are used in different situations depending on autosomal/sex-linked, input format, or special places in the pedigree, but most of our improvements are applicable equally well to those routines.

In the original **`segup`** and **`segdown`** most of the computation is done inside two outer nested loops that each iterate over all joint genotypes. Each loop corresponds to the genotype assigned to one parent in the family.

To achieve our improvements, we completely rewrote **`segup`** and **`segdown`**, and added some new auxiliary routines, which replace **`segfun`**. As useful and intricate as our changes are, it is also important to summarize what has not changed. The GEMINI code and its use has not changed — the iterations and the sequence of estimates to $\hat{\theta}$ is the same. The higher level routines that decide in what order to visit the pedigrees and individuals have not changed. The new programs are input and output identical to the original.

Methods

Our improvements were carried out starting with the PASCAL source code for LINKAGE version 5.1a. We began by converting the PASCAL programs to C using the translation program **`p2c`**, available as a UNIX utility (UNIX is a trademark of AT&T). It has been reported several times on various computer bulletin boards that the LINKAGE programs could be translated from PASCAL to C in this way, and that the resulting C versions work properly.

There are two reasons for converting from PASCAL to C. The first reason is that there are a wide variety of good software development tools available with UNIX for C programs, while PASCAL is minimally supported. The second reason is that the best C compilers available to us produce much better assembly code than the best PASCAL compilers. Some small experiments we did suggest that unmodified C code obtained directly from p2c runs about 25% faster than the original PASCAL, despite the fact that **`p2c`** is quite conservative in its translation. Our speedup results compare only the unmodified C programs with the modified C programs, and *do not* incorporate the additional speedup obtained by switching from PASCAL to C.

In addition to standard editors and debuggers, we used two UNIX tools to understand the behavior of the LINKAGE programs. The first tool is a function-level profiler called **`gprof`**. When a C program is compiled with the appropriate **`gprof`** flags, the program records how many calls are made to each subroutine and very roughly how much time is spent in each subroutine. The second useful UNIX tool is called **`tcov`**. It records how often each basic block of code is executed in a particular execution. The UNIX implementations we are familiar with have several tools similar to **`gprof`** and **`tcov`**.

The timing experiments were run on various Sun SPARCstation 2 computers with 32Mbytes of RAM. These machines run the operating system SunOS, version 4.1.2, which is an implementation of UNIX. For each test, we ran both the old and new programs on the same machine to avoid any inter-machine variation. To compile both the old and new versions of the LINKAGE programs we used the **`gcc`** (short for gnu cc) compiler, version 2.2.2 with the optimization flag -O (Stallman 1992).

Algorithmic Improvements

This section reviews the ideas behind our improvements to LINKAGE and gives some examples of how those ideas are applied. We first summarize two basic biological principles and four basic computer science techniques that are crucial to our improvements. The general paradigm for our more interesting improvements is that one biological principle exposes a part of the computation that can be sped up, and then one or more computer science techniques are used to speed up that part. Our key point is that a *synthesis of ideas from the two disciplines is essential* to the improvements we have obtained in the LINKAGE programs.

Biological Principles

Sparsity. An array or matrix is *sparse* if a significant fraction of the entries are 0. In most cases the array **`genarray`** (containing the probabilities for each joint genotype) is sparse. It is always sparse when the complete genotype is known, and will usually be sparse even if some partial phenotype information is known. The reason is simple from a biological point of view: knowledge of the phenotype severely restricts the possible genotypes, although it may not determine the genotype completely.

Isozygotes. The biological theory of linkage and recombination, as described by Ott (Ott 1991), for example, suggests that inheritance events which differ only by whether or not recombination has occurred, can be treated similarly. We make more precise what we mean by "similar" with a generic example of two-locus inheritance.

We call two joint genotypes *isozygotes* if they can pass on to a child precisely the same set of haplotypes. We use the notation $H_1 \| H_2$ to mean the *ordered* joint genotype where H_1 is on the left strand and H_2 is on the right strand. For example, if $A_1 \neq A_2$ and $B_1 \neq B_2$, then the ordered, phase known joint genotypes

$$\begin{array}{c} A_1 \\ B_1 \end{array} \left\| \begin{array}{c} A_2 \\ B_2 \end{array} \right. \qquad \begin{array}{c} A_2 \\ B_2 \end{array} \left\| \begin{array}{c} A_1 \\ B_1 \end{array} \right. \qquad \begin{array}{c} A_1 \\ B_2 \end{array} \left\| \begin{array}{c} A_2 \\ B_1 \end{array} \right. \qquad \begin{array}{c} A_2 \\ B_1 \end{array} \left\| \begin{array}{c} A_1 \\ B_2 \end{array} \right.$$

are isozygotes because they can all pass on the haplotypes

$$\begin{array}{c} A_1 \\ B_1 \end{array} \qquad \begin{array}{c} A_1 \\ B_2 \end{array} \qquad \begin{array}{c} A_2 \\ B_1 \end{array} \qquad \begin{array}{c} A_2 \\ B_2 \end{array}$$

There is an obvious symmetry between the first and second genotypes and between the third and fourth obtained by just swapping the two strands. This symmetry is already exploited in the original LINKAGE code and is the reason for the 2 in the denominator of the number of joint genotypes. Thus from the point of view of LINKAGE there are just two isozygotes in this example, the first and third genotypes.

Suppose a child of the above parent inherits the joint haplotype

$$\begin{matrix} A_1 \\ B_1 \end{matrix}$$

Then with the first genotype no recombination has occurred, while with the third genotype a recombination has occurred. Otherwise, there is no difference. Similar facts hold for the other three joint haplotypes.

The set of all joint genotypes can be partitioned into equivalence classes of isozygotes. By *partition* we mean that each joint genotype belongs to exactly one equivalence class of *isozygotes*; this is the mathematically precise notion of "similarity" we sought. The principle that isozygotes are similar is useful computationally because it reveals that the most we need to know about a genotype is which haplotypes can be passed on to a child and, thus, some pieces of the computation on different isozygote genotypes can be combined.

The definition of isozygotes can be extended naturally to any number of loci; in fact, the concept becomes more important computationally as the number of loci increases. The *size* of an isozygote class is the number of joint genotypes in the class that are not symmetric by switching strands. For the case of two loci, each doubly heterozygous genotype will be in a class of size 2, while singly heterozygous and doubly homozygous genotypes will be in classes of size 1. This is shown by fixing the position of the two alleles for the first locus and then alternating (switching strands of) the two alleles for the second locus. This alternation produces 2 different joint genotypes if and only if they are doubly heterozygous at both loci. So the size of the equivalence class in a particular instance can be determined by conducting this analysis. In a 3-locus case, for example, triply heterozygous genotypes form isozygote classes of size 4. This is because we again fix the left-right choice at the first locus and then flip the alleles at the other two loci one at a time. In general, if a genotype is heterozygous at $k > 0$ loci, it belongs to an isozygote class of size 2^{k-1}. What distinguishes the different elements of an isozygote class is recombination if it exists. The size of the class in relation to the number of possible joint genotypes is an indication of the potential performance improvement.

The notion of isozygotes can be further extended to two parents in a straightforward way. The ordered pairs (G_1, G_2) and (G_3, G_4) are isozygotes if G_1, G_3 are in the same isozygote class for one parent and G_2, G_4 are in the same isozygote class for the other parent.

The symmetry that isozygotes share is quite different from the symmetry of recombination classes described in (Lathrop and Lalouel 1988) that was previously implemented in LINKAGE, which we still use. In recombination classes, the elements in the same class share the same *recombination pattern* but have different alleles and haplotypes (left or right haplotype of each parent). In isozygote classes, the elements have different recombination patterns and different haplotypes, but use the same alleles and can pass on a fixed set of haplotype combinations to a child. Another way of viewing this distinction is that recombination classes exploit a symmetry of child genotypes, while isozygote classes exploit a symmetry of the parental genotypes.

Computer Science Techniques

Common Subexpression Elimination. Our first computer science technique is the elimination of common subexpressions. A traditional example is the evaluation of the expression

$$a_1a_3 + a_2a_3 + a_1a_4 + a_2a_4.$$

We can replace this by the equivalent expression

$$(a_1 + a_2) \times (a_3 + a_4)$$

which replaces an expression that has 3 additions and 4 multiplications with an expression that has 2 additions and 1 multiplication. The key idea is to first determine the *common subexpressions* $(a_1 + a_2)$ and $(a_3 + a_4)$.

In fact, we use a generalized version of precisely this substitution to speed up the computation of how haplotypes are transmitted. Using the notation of **segdown**, suppose the parents are p, q and their only child is r. Suppose that their genotypes are G_p, G_q and G_r, and G_r consists of haplotypes H_{r1} and H_{r2}. In the original version of **`segdown`** the probability that r gets genotype G_r is computed as

$$\begin{array}{ll}\sum_{G_p}\sum_{G_q} & (P(p \text{ passes on } H_{r1} \mid G_p) \times P(q \text{ passes on } H_{r2} \mid G_q))+ \\ & (P(p \text{ passes on } H_{r2} \mid G_p) \times P(q \text{ passes on } H_{r1} \mid G_q))\end{array}$$

This can be sped up by first computing separately the probability that each parent passes on each haplotype.

$$P(p \text{ passes on } H_{r1}) = \sum_{G_p} P(p \text{ passes on } H_{r1} \mid G_p) \times P(G_p)$$

Then we compute the probability that r gets genotype G_r as

$$(P(p \text{ passes on } H_{r1}) \times P(q \text{ passes on } H_{r2})) + (P(p \text{ passes on } H_{r2}) \times P(q \text{ passes on } H_{r1}))$$

Thus instead of a long double sum over pairs of joint genotypes for each recombination class, we do four short single sums for each haplotype and then just two more multiplications and one addition for each recombination class.

We will show later that this simplification can be further improved by applying the principle of sparsity.

Caching. Storing expressions that are frequently recomputed is our next computer science technique. This is sometimes called *caching*, similar to the term *cache*, which is used to described the fast internal memory of a computer. A good memory management strategy tries to put frequently accessed variables in the cache. In our context, caching means simply defining extra variables or arrays to store intermediate results that are frequently needed.

Replacing Top-Down Computation with Bottom Up. Our third computer science technique is replacing top-down computation with bottom-up. This is a general technique that can be used to traverse trees or evaluate recursive functions. The general idea is that one should do a bottom-up traversal when starting at the bottom of the tree will eliminate or combine many of the options that would be separately explored if one started at the top.

One of the important ideas in the Elston-Stewart algorithm is that one should compute the conditional probabilities traversing the pedigrees from children to parents. While this is possible in simple pedigrees (and is done in LINKAGE), there are complicated pedigrees that arise in practice where it is necessary to update some children's probabilities based on their parents.

It is very interesting that Elston and Stewart independently discovered the benefit of replacing top-down with bottom-up in the context of linkage analysis. We extend the application of this principle beyond the order of pedigree traversal to the sequence of computations for a particular nuclear family.

Reduction in Operator Strength. Our fourth computer science technique is called *strength reduction* or more formally *reduction in operator strength*. The idea is to replace a slower primitive operation with a faster one, especially inside frequently executed loops. An important example, relevant to LINKAGE, is that floating point multiplication is generally considerably slower than other operations such as addition or boolean comparisons.

Good optimizing compilers apply all four of these techniques to some extent in an effort to make the assembly code run faster. However, in our experience with LINKAGE we found many cases where these techniques apply that even state-of-the-art compilers cannot detect. Sometimes, the reason is that the logic needed to apply the technique is just too complicated—for example, when it requires restructuring code across several nontrivial loops or across procedure boundaries. Other times, the compiler cannot detect that applying the technique is advantageous because this requires knowledge of the underlying biology, in particular the principles of sparsity and isozygotes.

Synthesis - Putting All the Ideas Together

We now briefly describe seven improvements we have made in the probability updating algorithms of **`segup`**, **`segdown`**, and **`segfun`**. The reader who is not at all algorithmically inclined may find this subsection overly technical and may prefer to skip ahead to the next section.

List of genotypes that have nonzero probability. For each individual we keep a list of genotypes that are possible; a genotype is possible if the corresponding *genarray* entry is not 0.0. A similar list is constructed in MENDEL using more complicated logic to determine what is possible (Lange and Goradia 1987). This is an example of combining the principle of sparsity with the technique of caching.

Instead of iterating over all genotypes, we can iterate over just the possible ones. Here is one simple example of how this speeds up the computation. Expressed in pseudocode, the original two outer loops in **`segdown`** and **`segup`** look like:

```
For father's genotype = 1 to number of joint genotypes do
 if this genotype is possible for the father then
  for mother's genotype = 1 to number of joint genotypes do
   if this genotype is possible for the mother then

                 ...............
```

Notice that this code does test for sparsity—if a joint genotype is not possible, the computation in that loop iteration stops. However, the tests for sparsity on the mother's **`genarray`** are very excessive. The reason is that the set of joint genotypes for the mother that are possible and pass the second **`if`** test is the *same for every choice of the father's joint genotype*. Therefore, we precompute and cache the set of possible joint genotypes for the mother and change the code to look like:

```
For father's genotype = 1 to number of joint genotypes do
 if this genotype is possible for the father then
  for each joint genotype that is possible for the mother

        ...............
```

Sparsity pattern of the child's `genarray` at the beginning of `segdown`. We have found a way to apply the basic Elston-Stewart bottom-up principle to **`segup`** and **`segdown`** when the number of children is 1 (a very important and frequent special case). With the sparsity modification described above, all the sparsity testing was being done on the parents' joint genotype. But as outlined in the summary of the LINKAGE package above, the new value of `genarray`$[j]$ is the old value multiplied by some factors depending on the parent's probabilities and the recombination fraction. This means that if the old child probability was 0.0 (genotype not possible), it will remain 0.0, and there is no point in updating it. The other key observation is that if we know some information about the child's phenotype, then (in most cases) the number of possible genotypes for the child that are consistent with the known phenotype is very much smaller than the number of possible joint genotypes that the parents can give to a child based on their genotypes. To put this in the other direction, there are many joint genotypes j, such that the parents could potentially have a child with joint genotype j, but the one child cannot have joint genotype j because it is not consistent with the child's known phenotype.

Boolean expressions for 0 testing. It has been understood since the time of Haldane that logical analysis of the genotypes and inheritance patterns can be used to derive genotype information. An interesting formal, but theoretical treatment of this idea is given by Wijsman (Wijsman 1987).

We have found that Wijsman's suggestion of using Boolean logic to reason about the pedigrees has some very practical uses. To do this we combine the principle of sparsity with the technique of strength reduction, and in some cases we also use caching. We describe our most interesting use of boolean logic here. The next improvement shows another, simpler use.

For some joint genotypes that are possible for two parents, many meiotic combinations may not be possible for a child, due to what is known about the child or its siblings. Therefore, many calls to **`segfun`** yield 0.0. We want to avoid these calls or reduce their cost if at all possible. We can use the facts that: 1) testing whether a bit is 0 is a lot faster than testing whether a floating point number is 0.0 and 2) all the numbers are nonnegative and the only arithmetic operations are addition and multiplication, so we cannot get 0.0 by adding a positive number and its negative complement.

To put these ideas in practice, consider a real number expression using only addition and multiplication. Replace every 0.0 value by the boolean value 0 (FALSE), replace every nonzero value by the boolean value 1 (TRUE), replace every + by the boolean operator OR, and replace every $\times$ by the boolean operator AND. Provided all the original values are nonnegative, this replacement yields a boolean expression whose value is 0 if and only if the original expression has value 0.0. Thus we replace the pseudocode:

```
Compute algebraic expression E
```

by the pseudocode:

```
If the corresponding Boolean expression evaluates to 1
then compute algebraic expression E
```

This is a major improvement when the terms in the algebraic subcomputations are repeatedly reused, and we just need to precompute once and *cache* which simple terms are 0.0 and which simple terms are positive, to set up the boolean expressions. For example, we augmented the routine **`segfun`** with its boolean counterpart **`lsegfun`** (**`l`** stands for logical), so that whenever the previous program called **`segfun`**, we call **`lsegfun`** first and call **`segfun`** only when it is known to return a nonzero value. We later replaced **`segfun`** with a more useful alternative, but we still use **`lsegfun`** before calling that alternative.

The cost of the boolean expression is mitigated by the fact that both C and PASCAL can evaluate the logical operators AND and OR conditionally (this is the default in C), so that as soon as we find a 0 argument to AND we return 0, and as soon as we find a 1 argument to OR we return 1. Thus we usually do not need to evaluate the full boolean expression.

Boolean flags for haplotypes. This idea combines the common subexpression simplification for haplotypes with the idea of using boolean logic to apply the sparsity principle. We keep an array of boolean flags that indicates for each haplotype H whether the child can have any genotype that includes H as one haplotype. For all the haplotypes that the child cannot have, we need not compute the probability that each parent passes on that haplotype. When significant phenotype information is known, the vast majority of haplotypes can be ruled out, so we compound the effect of extracting common subexpressions.

Caching haplotypes that can be passed on for each genotype. The principle of isozygotes tells us that from a computational point of view what matters most about a parent's genotype is what haplotypes can be passed on to the child. Therefore we precompute this set for all joint genotypes. In the previous code this set was repeatedly recomputed inside multiple loops using two levels of indirection (two array accesses). In our version we need just one level of indirection to retrieve the set.

This use of caching trades off substantial memory usage to gain speed. In complex linkage analysis speed is a problem, so this is a trade off we willingly make. In this case the cache contains space for $3 \times 2^{k-1} \times (maxhap + 1) \times maxhap/2$ integer values, where k is the number of loci, and maxhap is the product of the number of alleles. For 3 loci with 6 alleles each, maxhap is 216 and the cache has 281,232 total locations. In exchange, we avoid recomputing these haplotypes repeatedly inside a double loop.

Remove probability multiplications from `segfun`. We reorganized the computations in **`segfun`**, so that some of them are now done in **`segdown`** and **`segup`**. The rest are done in a new routine called **`segsum`**. The essence of the computation within the routine **`segfun`**

can be described in two stages. In the first stage, we determine which recombination classes can be produced by a given pair of parents, and sum up the probabilities of each child under each recombination class. In the second stage, we take a weighted average of these sums based on the recombination probabilities for each class. The principle of isozygotes revealed that among the children the first stage was being repeated for each isozygote because each isozygote produces that recombination class with a different probability. Since the first stage is the same for all parent pairs in the same isozygote class, we compute it only once for each isozygote class in the routine **`segsum`** and save these sums in a cache. For the second stage, we multiply the values in the cache with the appropriate recombination probabilities and take the weighted average in **`segup`** or **`segdown`** itself.

One call to `lsegfun` for each combined parent's isozygote class. Recall that by incorporating Boolean flags we guarded our calls to **`segsum`** (replacement for **`segfun`**) with a preliminary call to the logical routine **`lsegfun`**, so that if **`lsegfun`** can prove that **`segsum`** will return all zeroes, we avoid the call to **`segsum`**. Now we apply the principle of isozygotes to **`lsegfun`**. Our criterion for returning 0.0 is precisely that at least one child is not genetically compatible with the proposed joint genotypes of the parents. More specifically this means that for the parental genotypes (G_f, G_m), none of the haplotype combinations that these parental genotypes can pass on to a child can be a genotype for the child. Now suppose (G'_f, G'_m) is an isozygote of (G_f, G_m). This means that the set of haplotypes that can be passed on by (G'_f, G'_m) is the same as that for (G_f, G_m). Therefore, if a child is incompatible with parental genotypes (G_f, G_m), the child must also be incompatible with parental genotypes (G'_f, G'_m). Thus we can just pick one representative of the isozygote class to test **`lsegfun`** and the same answer (0 or 1) applies to all the other members of the class. This is an example of an improvement that is much more powerful as the number of loci increases. With three loci, for example, in the case where both parent genotypes are triply heterozygous, each single parent isozygote class is of size 4 as explained above, so the combined classes are of size $4 \times 4 = 16$. This means that we cut down on the number of calls to `lsegfun` by a factor of 16 in this case.

Validation of Speedup

To validate our improvements, we report the running times of the old programs and the new programs on three sample data sets.

The data sets are:

- CEPH: data from the CEPH Database for chromosome 6 on the the standard family panel of 65 three generation families (Dausset et al. 1990).

- 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 192 individuals containing 2 marriage loops (Blanton et al. 1991). As shown in (Blanton et al. 1991), this pedigree had to be split into three pieces because computation on the whole family together was prohibitively long. In the Tables of Results, RP01-3 denotes analysis with the family split in three pieces. RP01 denotes analysis of the whole family as a single pedigree (fig. 1).

- 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 (Law et al. 1992).

The CEPH and RP01 data sets are two extremely different kinds of pedigree structures and are representative of the range of pedigree situations found in genetic disease linkage studies. These two data sets contain many loci, so we chose various subsets for our experiments. In the case of RP01, the locus with 2 alleles is always the disease locus. The BAD data set represents a large single family with a disease, like RP01. However, it is not as deep and we only have data for 3 loci.

All these data sets represent autosomal inheritance, but we have also implemented our improvements for the sex-linked analogues of **`segup`**, **`segdown`**, and **`segfun`**, and they work there too. The problem of long runs is not as serious for sex-linked data because updating the probabilities for a male child is relatively easy as compared to the autosomal case, and there is no recombination in the sex chromosomes of a father.

The times recorded are the "user" times reported by the **`time`** command on our systems. We have rounded the times to the nearest second or minute to simplify the data and to acknowledge that times will vary. In fact, we observed variations in running time of as much as 10% on the same run because of inherent variations in the load of the other processes running simultaneously on the computer. We have also rounded the speedup quotients to the nearest integer for simplicity.

Table 1 shows sample performance improvements on 2-point analyses using LODSCORE, and Table 2 shows sample improvements on 3-point analyses using ILINK. Table 3 shows a few sample improvements on 4-point analyses using ILINK.

In some of the test cases the computation is so long that we ran the old, slow program for just one function evaluation of each order tried. In these cases, we ran the fast new program to completion and then divided its running time by the number of function evaluations. This is reasonable because our changes should have roughly the same effect on each function evaluation of a given order. We found in practice that comparing one function evaluation of the old program with the average on the new program *underestimates* the speedup slightly, because the old program modified for one function evaluation, avoids some overhead at the end of the run. In the two 3-point runs for the full RP01 we report here the speedups based one function evaluation of each program. In the three 4-point tests on RP01-3 and the two

3-point tests on the full RP01, we tried only one order. The four runs withs $2 \times 6 \times 9$ alleles represent two different choices each for the 6-allele and for the 9-allele locus.

The speedup numbers in the rightmost column are substantial in all but the shortest runs. In the short runs, such as the $2 \times 3 \times 3$ allele run, the overhead of various initializations and procedure calls is not negligible, and the part of the code we have speeded up is not as dominant as it becomes in longer runs.

The numbers suggest that as the amount of computation increases above some pedigree-dependent threshold, our improvements become more effective. This is important because we are primarily concerned with longer runs, since these are where the most absolute time can be saved using our faster programs. In practice, the user must do some amount of thinking and hypothesis formulation before running the LINKAGE programs. If the slow version already takes substantially less time than the user needs to think up the hypothesis and set up the program, then LINKAGE is not the bottleneck. We are concerned with cases, such as the longer runs on the RP01 data set, where the slow speed of the old LINKAGE programs is the bottleneck in the user's work.

The reader may be curious as to why the CEPH run with 7×9 alleles exhibits so much more speedup than the other two CEPH examples. The reason is there are many fewer unknowns for the 6-allele locus than for the other two loci. We can quantify this partially as follows. There are 137 individuals whose 6-allele gene is known, but whose 7-allele and 9-allele genes are not known. The are only 17 individuals whose 7-allele gene is known, but whose 6-allele and 9-allele genes are not known. There are only 5 individuals whose 9-allele gene is known, but whose 6-allele and 7-allele genes are unknown. Most of these unknowns are grandparents. This results in many more grandparental genotype possibilities and a significantly longer run. However, by doing the sparsity analysis bottom-up in the new program, we take tremendous advantage of what is known about the parent to simplify the update for the grandparent.

The reader may also be curious as to why we got so much more speedup on the full RP01 than on the split RP01-3. The answer is that from the perspective of the computation these two pedigrees are quite *different*. Using **`gprof`** we observed that at least 80% of the computation time is spent updating the probabilities in the top two generations. The split considerably simplifies the top generations by reducing the number of children in several nuclear families and removing one loop (involving a child of parents in the second generation). The fact that the lower generations of RP01 and RP01-3 are very similar is insignificant computationally.

The previous two paragraphs give some indication of the difficulty of predicting speed effects. There is some discussion about running time in (Lange and Elston 1975), but it does not take sparsity into account. As we mentioned above, even the old LINKAGE programs took some advantage of sparsity. There are many variables that can affect the amount of sparsity, e.g., graph structure of the pedigree, number of loci, number of alleles, number of unknows for those alleles, and pedigree traversal order. Of course, their effects are not independent.

Another important variable factor in total running time is the number of function evaluations. In the runs reported above, the programs took anywhere from 15 to 55 function evaluations to converge for a single order. As mentioned previously, the number of function evaluations is the same for the old and new programs. We do not see a clear pattern that could help predict how many evaluations would be required for a given run.

Similarly, it is very hard to quantify the separate effects of the changes we made. Like the basic factors affecting running time, the changes interact in complicated ways, but we give some anecdotal evidence.

For pedigrees like CEPH, the most useful changes are sparsity-based, and the switch to bottom-up computation in **`segup`** and **`segdown`** is the most useful. The reason appears to be that the pedigrees have two or three generations with most of the unknowns in the top generation of the three-generation pedigrees. Switching to bottom-up evaluation eliminates most of the candidate genotypes in the grandparent generation. We implemented the sparsity-based changes first and we observed most of the speedup reported in Table 1 then, before the isozygote-based changes were implemented.

For the disease pedigrees, both the sparsity-based changes and the isozygote-based changes contribute significantly. Intuitively, the sparsity-based changes work as in the CEPH pedigree to drastically reduce the computation caused by the youngest generation or two with unknowns. If there are more generations with unknowns, their **`genarray`**s may not be very sparse. For example, using a debugger and/or **`tcov`** we can check that the **`genarray`** for the top male and female in RP01-3 for the $2 \times 9 \times 9$ run are not very sparse. Most of the genotypes that are consistent with their disease status and the disease status of their offspring cannot be ruled out. The rearrangement of the probability summations based on isozygotes significantly cut the computation time of updating the probability arrays by at least a factor of 2.

These performance improvement results compare the new C program with the old C program generated by **`p2c`**. Therefore these are valid comparisons of the results of our work. However, to summarize, and put the total results in perspective, we ran the $2 \times 9 \times 9$ ILINK run on the RP01 data with the new ILINK and the original Pascal version of ILINK compiled with flag -O4. Because this run would take so long we ran just the first function evaluation for each program to compare the speed. We found that the new program is 40 times faster on this problem. Then the new program was rerun to completion, which took 6 days on a Sun SPARCstation 2. So the comparable run under Pascal would have taken 8 months!

Discussion

The combinatorial complexity of genetic linkage computations and the slow speed of linkage programs are rapidly becoming a bottleneck in genetic linkage analysis. Rather than trying to throw more and better computer hardware at the problem, we have systematically improved the basic algorithms used in the LINKAGE package.

The algorithmic approach to speeding up the programs is sequential and completely different from the complementary approach of using more hardware such as vector or parallel computers, which has been shown to be successful (Miller et al. 1991; Goradia et al. 1992). According to these papers, one can achieve a speedup of roughly one order of magnitude if enough processors are available. As shown by our results, the algorithmic improvements described also achieve a speedup of one order of magnitude, and sometimes more, but without using any extra computer hardware. Both types of improvements make a qualitative change in the complexity of useful genetic linkage problems that can be solved in a reasonable amount of time. It will be interesting to combine our approach of better algorithms and the approach of using parallel computation to try and achieve a speed improvement of two or more orders of magnitude.

The data in the previous section show that the improvements are practical, substantial, and qualitative. With our improved algorithms we can reach solutions in a week that would have taken almost a year, and therefore were impossible. Nevertheless, we recognize that no matter what algorithmic improvements we achieve, geneticists will want to solve larger and more difficult linkage problems. Therefore, the complementary approach of using better hardware, including parallel computers, should be combined with our algorithmic approach to speed up linkage computations.

On a more abstract level our work is one more demonstration that the study of human genetics is becoming more computational. Much more synthesis of biological theory and computer science techniques is needed to produce programs than can efficiently carry out the increasingly complex computations that geneticists want to do.

Acknowledgments

We thank Dr. Stephen P. Daiger and Dr. Lori A. Sadler for contributing the disease family data for our experiments and generously allowing us to present the pedigree and results. Development of these data was supported by grants from the National Retinitis Pigmentosa Foundation and the George Gund Foundation. Also thanks to Drs. David R. Cox and Richard M. Myers and their colleagues for their contribution of the Amish family data developed with the support of a grant from the National Institutes of Health. And thanks to Professor Alan Cox for help with details regarding the SPARC architecture and `gcc` . Special thanks to Professors Ken Kennedy (Rice) and Charlie Lawrence (Baylor) for helping us get our collaboration started. This work was supported by grants from the Human Genome Program of the National Institutes of Health (RWC), the W. M. Keck Foundation (RWC and RMI), and from the National Science Foundation (AAS).

References

[1] Blanton SH, Heckenlively JR, Cottingham AW, Friedman J, Sadler LA, Wagner M, Friedman LH, et al. (1991) Linkage mapping of autosomal dominant retinitis pigmentosa (rp1) to the pericentric region of human chromosome 8. Genomics 11:857–869

[2] Dausset J, Cann H, Cohen D, Lathrop M, Lalouel J-M, White R (1990) Centre d'Etude du Polymorhisme Humain (CEPH): Collaborative genetic mapping of the human genome. Genomics 6:575–577

[3] Elston RC, Stewart J (1971) A general model for the analysis of pedigree data. Hum Hered 21:523–542

[4] Goradia TM, Lange K, Miller PL, Nadkarni PM (1992) Fast computation of genetic likelihoods on human pedigree data. Hum Hered 42:42–62

[5] Lalouel J-M (1979) GEMINI - a computer program for optimization of general nonlinear functions. Technical report no 14, Department of Medical Biophysics and Computing, University of Utah, Salt Lake City

[6] Lange K, Elston RC (1975) Extensions to pedigree analysis. I. Likelihood calculation for simple and complex pedigrees. Hum Hered 25:95–105

[7] Lange K, Goradia TM (1987) An algorithm for automatic genotype elimination. Am J Hum Genet 40:250–256

[8] Lathrop GM, Lalouel, J-M (1984) Easy calculations of lod scores and genetic risks on small computers. Am J Hum Genet 36:460–465

[9] Lathrop GM, Lalouel J-M (1988) Efficient computations in multilocus linkage analysis. Am J Hum Genet 42:498–505

[10] Lathrop GM, Lalouel J-M, Julier C, Ott J (1984) Strategies for multilocus linkage analysis in humans. Proc Natl Acad Sci USA 81:3443–3446

[11] Lathrop GM, Lalouel J-M, White RL (1986) Construction of human genetic linkage maps: likelihood calculations for multilocus linkage analysis. Genet Epidemiol, 3:39–52

[12] Law A, Richard CW III, Cottingham RW Jr, Lathrop GM, Cox DR, Myers RM (1992) Genetic linkage analysis of bipolar affective disorder in an Old Order Amish pedigree. Human Genetics, 88:562–568

[13] Miller PL, Nadkarni P, Gelernter JE, Carriero N, Pakstis AJ, Kidd KK (1991) Parallelizing genetic linkage analysis: A case study for applying parallel computation in molecular biology. Comp Biomed Res 24:234–248

[14] NCHGR article (1991) NCHGR begins unified framework map effort. Human Genome News 3(2):1

[15] Ott J (1974) Estimation of the recombination fraction in human pedigrees– efficient computation of the likelihood for human linkage studies. Am J Hum Genet 26:588–597

[16] Ott J (1991) Analysis of Human Genetic Linkage, revised edition. The Johns Hopkins University Press, Baltimore and London, 1991

[17] Stallman RM (1992) Using and porting gnu cc. Manual provided by the Free Software Foundation to document `gcc`

[18] Wijsman, EM (1987) A deductive method of haplotype analysis in pedigrees. Am J Hum Genet 41:356–373

Data Set	Number of Alleles	New Program Time	Old Program Time	Speedup
CEPH	6×7	17sec	105sec	6
CEPH	6×9	22sec	647sec	30
CEPH	7×9	46sec	17279sec	375
RP01-3	6×6	436sec	4234sec	9
RP01-3	6×9	1357sec	14953sec	11
RP01-3	9×9	490sec	5452sec	11^*

Table 1: Some sample 2-point analyses done with LODSCORE. * means that the old program was run for only one function evaluation.

Data Set	Number of Alleles	New Program Time	Old Program Time	Speedup
RP01-3	$2 \times 3 \times 3$	60sec	320sec	5
RP01-3	$2 \times 6 \times 6$	61min	586min	9
RP01-3	$2 \times 6 \times 9$	190sec	1602sec	8^*
RP01-3	$2 \times 6 \times 9$	200sec	1694sec	8^*
RP01-3	$2 \times 6 \times 9$	183sec	1682sec	9^*
RP01-3	$2 \times 6 \times 9$	200sec	1771sec	9^*
RP01-3	$2 \times 9 \times 9$	15min	130min	9^*
RP01	$2 \times 6 \times 6$	299sec	5541sec	$19^\dagger$
RP01	$2 \times 9 \times 9$	99min	3013min	$31^\dagger$
BAD	$2 \times 4 \times 4$	48sec	691sec	14^*

Table 2: Some sample 3-point analyses done with ILINK. * means that the old program was run for only one function evaluation; † means that both programs were run for one function evaluation.

Data Set	Number of Alleles	New Program Time	Old Program Time	Speedup
RP01-3	$2 \times 3 \times 3 \times 4$	92sec	821sec	9^*
RP01-3	$2 \times 3 \times 3 \times 6$	345sec	4131sec	12^*
RP01-3	$2 \times 3 \times 4 \times 5$	587sec	6085sec	10^*

Table 3: Three sample 4-point analyses done with ILINK. * means that the old program was run for only one function evaluation.

Figure 1 The UCLA-RP01 pedigree containing 192 individuals over 7 generations. Double bars indicate the two marriage loops. The dashed boxes show how the pedigree was split into three pieces when required, and * indicates the individuals coded in two sub-pedigrees. Shading indicates affected individuals (solid: 100%, forward slashes: 90%, back slashes: 80% probability of being affected). The ? shows at-risk individuals whose disease status is unknown.

Keywords:

genetic linkage analysis

LINKAGE

algorithms

computer science

recombination

sparsity