Avoiding Recomputation in Linkage Analysis

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

Sandeep K. Gupta [†]
Department of Computer Science
Rice University
Houston

K. Shriram [‡]
Department of Computer Science
Rice University
Houston

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

January 3, 1994

Keywords: Genetic linkage analysis, algorithms, recombination classes, check-pointing, crash-recovery.

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

[*] schaffer@cs.rice.edu
[†] skgupta@owlnet.rice.edu
[‡] shriram@cs.rice.edu
[§] bwc@bcm.tmc.edu

Abstract

We describe four improvements we have implemented in a version of the genetic linkage analysis programs in the LINKAGE package: subdivision of recombination classes, better handling of loops, better coordination between the optimization and output routines, and a checkpointing facility. The unifying theme for all the improvements is to store a small amount of data to avoid expensive recomputation of known results. The subdivision of recombination classes improves on a method of Lathrop and Lalouel [Amer. J. Hum. Genetics 42(1988), pp. 498–505]. The new method of handling loops extends a proposal of Lange and Elston [Hum. Hered. 25(1975), pp. 95–105] for loopless pedigrees with multiple nuclear families at the earliest generation. From a practical point of view, the most important improvement may be the checkpointing facility which allows the user to carry out linkage computations that are much longer than the mean-time-to-failure of the underlying computer.

1 Introduction

Linkage analysis was fully thrust into the computer age by the discovery of the Elston-Stewart [3] algorithm for pedigree traversal and its implementation in LIPED [11]. From a computer science perspective, the reason the Elston-Stewart algorithm works efficiently is that it *avoids recomputation* on subtrees of the pedigree by traversing bottom-up towards the root. In this paper we further advance the theme of avoiding recomputation in linkage analysis by describing four improvements of that genre that we have implemented in programs in the LINKAGE package.

Although the Elston-Stewart algorithm made a host of linkage problems computationally tractable for the first time, geneticists still want to solve computationally intractable linkage problems. The computational requirements caused by better data collection methods and the desire to do multilocus analysis have grown at a much greater rate than the speed of readily available computers.

We continue an investigation started in [2] concerning better sequential algorithms for linkage analysis. That paper showed significant hope that better algorithms can make much bigger linkage analysis problems computationally tractable. As in [2], we demonstrate the improvements described herein by implementing them in some of the programs in the LINKAGE software package and show the improvements in computation time experimentally. LINKAGE [7, 9] is one of the most popular linkage analysis packages and is particularly useful for multilocus analysis of large disease pedigrees.

The four improvements described herein are:

1. Subdivision of recombination classes.

2. More efficient multiple traversals of pedigrees with loops.

3. Better coordination between the optimization routine and the output routine.

4. A checkpointing facility that allows the user to restart a "crashed" computation near the point where the crash occurred.

The first two improvements can be viewed as general algorithmic improvements based on proposals in the literature, while the latter two improvements correct weaknesses specific to programs in LINKAGE. The subdivision of recombination classes improves on a proposal of Lathrop and Lalouel [8] that was implemented in LINKAGE. The new loop algorithm extends a proposal of Lange and Elston [6] for handling loopless pedigrees with multiple nuclear families at the earliest generation (Lange and Elston called such a pedigree *complex*).

All four improvements help by avoiding recomputation of already known values. The first three improvements speed up uninterrupted runs. The checkpointing facility avoids significant recomputation when the computer crashes.

In terms of practical benefit to the LINKAGE user, the checkpointing facility may be the most significant of the improvements, because it drastically increases the size of linkage problems that can be solved reliably on inherently unreliable computers.

We have implemented the changes described herein in the already improved versions of the LINKAGE programs described in [2], however they are essentially independent of our previous work. Our modified programs are in C. The improved LINKAGE programs are available by anonymous FTP from a computer at Rice University (Current instructions: ftp softlib.cs.rice.edu, cd linkage/fastlink, get files from that subdirectory; contact the first author by e-mail at schaffer@cs.rice.edu for further assistance).

The rest of this paper is organized as follows. Section 2 gives relevant background on the LINKAGE programs. Sections 3 through 6 describe the four improvements. Sections 7 and 8 validate the improvements. We conclude with a short discussion.

2 Summary of LINKAGE and Related Work

A thorough treatment of genetic linkage analysis, including a summary of the LINKAGE programs, is given in Ott's monograph [12]. In this section we review a few facts about LINKAGE relevant to this paper. The most fundamental goal in linkage analysis is to compute the probability, θ, that a recombination occurs between two genes G_1 and G_2.

The LINKAGE package contains four linkage analysis programs: LODSCORE, ILINK, LINKMAP, and MLINK. The improved sequential algorithms in [2] are applicable to all the programs. The four changes described here are applicable to LODSCORE and ILINK. The improved handling of recombination classes and loops are also applicable to LINKMAP and MLINK.

The LODSCORE program searches for the maximum likelihood estimate, $\hat{\theta}$, of the recombination probability between two genes. For each candidate θ, its likelihood is computed with respect to the input pedigree(s). Given a set of loci, LODSCORE will estimate θ for each pair of loci.

The notion of recombination can be generalized to more than two loci. Suppose $G_1, G_2, \ldots, G_n$ are multiple gene loci occurring in that order. Then we can define a vector $(\theta^1, \theta^2, \ldots, \theta^{n-1})$, where the component θ^i is the recombination fraction between loci G_i and G_{i+1}. We use superscripts here because later $\hat{\theta}_i$ will represent the i^{th} estimate, in a sequence of estimates, to the θ vector. We will use n to represent the number of loci. The ILINK program searches for the maximum likelihood estimate of the multilocus θ vector.

Both LODSCORE and ILINK start from an initial solution and use an iterative procedure called `gemini` [5] to find the estimate $\hat{\theta}$. Like many iterative optimization procedures, `gemini` can only guarantee to find a *local* optimum and *not a global* optimum. After `gemini` computes a locally optimal $\hat{\theta}$, an output procedure `outf` is called to report $\hat{\theta}$, the likelihood, and a variety of other statistics. The interaction between `gemini` and `outf` will be discussed more in Section 5.

The function that computes a combined likelihood for all the pedigrees and reports the value $-2 \times \log(\text{likelihood})$ (offset by an additive scaling term) is called `fun`. The programs try to maximize the value of the likelihood, but because of the minus sign, this corresponds to *minimizing* the value of `fun`. Each time a new candidate vector $\hat{\theta}_i$, provably better than previous candidates $(\hat{\theta}_1, \ldots, \hat{\theta}_{i-1})$, is generated this is called an *iteration* and the value of `fun` at the new $\hat{\theta}_i$ is reported on the screen along with some diagnostics.

In different iterations there may be a variable number of calls to `fun`. We denote a call to `fun` by FE, short for *function evaluation*. Each iteration of `gemini` has two phases. The first phase searches for a new estimate to $\hat{\theta}$ that improves on the estimate from the previous iteration. The second phase estimates the gradient at the the current $\hat{\theta}$ by perturbing one dimension at a time. Our experiments suggest that the first phase often takes two FEs, but there is no upper bound on the number of FEs needed. The gradient estimation usually takes $n-1$ FEs if forward differences are used and $2*(n-1)$ if central differences are used. The two gradient estimation methods are discussed in [8]. There are a few exceptions; e.g., if θ for males and females are assumed to be distinct then the number of FEs for the second phase is doubled.

In contrast to LODSCORE and ILINK, LINKMAP and MLINK take multiple values of the θ vector and computes the likelihood for each one. The computation of the likelihood for each input value and for each pedigree are essentially independent except for some shared input/output. Thus if the computation crashes in the middle, all the likelihood calculations that were completed do not need to be redone (provided the results were written to a file before the crash); This explains why there is little need to use a checkpointing scheme within LINKMAP and MLINK.

The basic structure of the likelihood computation is outlined in the section on Numerical and Computerized Methods in [12]. Inside the loop over pedigrees, the programs traverse a pedigree updating the probabilities of each joint genotype for each individual. There are several different updating routines, but they all start with a double nested loop over the possible genotypes for one parent and then the other parent.

Associated with each individual is an array `genarray` indexed by joint genotype numbers. We assume that the alleles at each locus, the haplotypes, and the genotypes are encoded by numbers. The specific encoding scheme used in the programs is not important here. We denote the number of haplotypes by h. The number of genotypes is denoted by $g = h \times (h+1)/2$. The reason for this formula is that a genotype is formed by *choosing* two haplotypes to make the two strands. If the two strands are different and we think of drawing the genotype as a diagram, then the strand that has the *lower* allele number at the first heterozygous locus is declared by convention to be the *left* strand; and the strand with the higher allele number at the first heterozygous locus is declared to be the *right* strand.

The size of a `genarray` is g. For our first improvement it is useful to understand that generally h is at most in the hundreds, even on long runs, but g may be in the tens or hundreds of thousands. The entry `genarray[`j`]` initially stores a scaled version of the probability that the individual has the phenotype associated with genotype j given the joint genotype j. 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 relatives already visited in the traversal and the recombination fraction. Following the notation of Lathrop and Lalouel [8], suppose we are updating the probability that individual with phenotype X has joint genotype G, conditioned on the collective joint genotypes Y of the relatives that have already been traversed and a candidate θ. Then the update rule is:

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

If we could compute with arbitrary precision, then $P(X \mid G)$ would be the contents of `genarray`$[G]$ before the update, and $P(X, G \mid Y, \theta)$ would be the contents after. However, the probability computed is generally so small that underflow is a danger. Therefore, LINKAGE scales all the values in `genarray` and undoes the scaling at the very end of the likelihood computation.

3 Subdividing Recombination Classes

In this section, we describe how we improved the use of recombination classes, as previously proposed in [8]. We first need to review what recombination classes are and how they are used inside the likelihood computation. Throughout this section, we assume that all sample parental genotypes are *heterozygous* at all loci. In the issues discussed here, the homozygous loci can be ignored since we know what the child will inherit at homozygous loci. For this section we assume we are working on a computation for an autosomal chromosome.

We define a *recombination pattern* to be a vector of length $n-1$ with value 0 or 1 at each position. Given a genotype for a parent, the recombination pattern restricts which haplotypes the child may inherit. If we fix the allele inherited at locus 1, we then interpret 0 in position i to mean that the child inherits the allele from locus $i+1$ from the *same* strand as the allele from locus i; i.e., no recombination occurs between loci i and $i+1$. We interpret 1 in position i to mean that recombination does occur between loci i and $i+1$.

For example, suppose that the parent has a 3-locus phase known genotype of $(1 \mid 2, 3 \mid 4, 5 \mid 6)$. If inheritance follows the recombination pattern $(1, 0)$, then the child could inherit either of the haplotypes $(1, 4, 6)$ or $(2, 3, 5)$ depending on the choice of allele at the first locus.

Recall from Section 2 that the update of a nuclear family starts with two loops over the possible genotypes of each parent. Let G_1 be a possible genotype for parent 1 and G_2 a possible genotype for parent 2. Let R_1 and R_2 be recombination patterns.

Let the two haplotypes that an untyped child can inherit from G_1 consistent with pattern R_1 be H_{1l} and H_{1r}. with the second subscript indicating whether the allele at locus 1 is inherited from the left strand or the right. Similarly, let H_{2l} and H_{2r} be the two haplotypes that the child can inherit from G_2 consistent with R_2.

Lathrop and Lalouel defined a *recombination class* to be the set of four genotypes formed from the haplotype pairs $\{H_{1l}H_{2l}, H_{1l}H_{2r}, H_{1r}H_{2l}, H_{1r}H_{2r}\}$. These are the four genotypes that a child might inherit from parents with genotypes G_1 and G_2, if recombination patterns R_1 and R_2 apply.

For each recombination class, inside an inner loop we need to compute the sum

$$\texttt{genarray}_c[H_{1l}H_{2l}] + \texttt{genarray}_c[H_{1l}H_{2r}] + \texttt{genarray}_c[H_{1r}H_{2l}] + \texttt{genarray}_c[H_{1r}H_{2r}],$$

where $\texttt{genarray}_c$ is the array of genotype probabilities for a child c. Such a sum is done for each child in a nuclear family, except possibly the first. Many recombination classes (viewed as unordered sets) contain exactly the *same* genotypes. Therefore, in very early versions of the LINKAGE programs the same sum was being *redone* for every identical class. By rearranging the order of the computation, Lathrop and Lalouel did all the identical classes consecutively, so that for each different class the four-term sum is done only once.

It should be emphasized that the above sum involves 3 floating-point additions and is in a deeply nested loop; versions of the sum (with different array indices and different children) are done tens of millions of times, or more, on long LINKAGE computations. Therefore, we look at the sum very, very closely and find more opportunity to eliminate duplicate additions. In the sex-linked case, the four-term sum becomes a two-term sum and there is no better way to compute it.

Suppose we consider a different possible genotype G'_2 for the second parent, while keeping the first parent the same. In practice, such a change happens frequently because, we loop over all possible genotypes for parent 2 is *inside* the genotype loop for parent 1. The new 4-term sum will be

$$\texttt{genarray}_c[H_{1l}H'_{2l}] + \texttt{genarray}_c[H_{1l}H'_{2r}] + \texttt{genarray}_c[H_{1r}H'_{2l}] + \texttt{genarray}_c[H_{1r}H'_{2r}].$$

The change from G_2 to G'_2 may change all four *genotypes* in the recombination class, but it can be more usefully viewed as a substitution of the *haplotype* H'_{2l} for H_{2l} and a substitution of the *haplotype* H'_{2r} for H_{2r}. To promote the latter view, let us reorganize the previous quadruple sum as follows

$$(\texttt{genarray}_c[H_{1l}H'_{2l}] + \texttt{genarray}_c[H_{1r}H'_{2l}]) + (\texttt{genarray}_c[H_{1l}H'_{2r}] + \texttt{genarray}_c[H_{1r}H'_{2r}]).$$

Notice that if we fix G_1, R_1, and c, each parenthesized pair sum depends only on a *haplotype* for parent 2. Three important consequences are:

1. Even though, each quadruple sum like our first sum is done only once, the pair sums parenthesized in our third sum occur in many quadruple sums and are being redone many times.

2. If we could precompute each possible pair sum, the sum in the inner loop would become a *single floating point addition* of two terms rather than a *triple floating point* addition of four terms.

3. If we fix G_1, the number of possible pair sums needed is at most (number of recombination patterns $\times$ number of children $\times$ `maxhap`), which is reasonable as long as the number of loci is 5 or less.

Therefore, once inside the first loop, where G_1 is fixed, we compute a table of all the possible pair sums. Inside the innermost loop, the triple sum becomes a sum of two terms from our table.

In pseudocode: the old algorithm looks like:

For each possible genotype G_1 of parent 1
    For each possible genotype G_2 of parent 2
        Do probability update with many 4-term sums

and the new algorithm (calling the table for child c T_c) looks like:

For each possible genotype G_1 of parent 1
    For each haplotype H
        $T_c[H] \leftarrow \texttt{genarray}_c[H_{1l}H] + \texttt{genarray}_c[H_{1r}H]$
    For each possible genotype G_2 of parent 2
        Do probability update with 2-term sums

where the typical 4-term sum:

$\texttt{genarray}_c[H_{1l}H_{2l}]+\texttt{genarray}_c[H_{1l}H_{2r}]+\texttt{genarray}_c[H_{1r}H_{2l}]+\texttt{genarray}_c[H_{1r}H_{2r}]$,

becomes the 2-term sum:

$$T_c[H_{2l}] + T_c[H_{2r}]$$

In this simplified pseudocode, if there are h haplotypes and q quadruple sums to be computed for a given choice of G_1, then the old code does $3q$ sum operations, while the new code does $h+q$, where h are done to build table T and q are done in the inner loop. In the actual code, there is a second index to the table depending on the choice of recombination pattern for parent 1 (which determines the values of H_{1r} and H_{1l}.

The number of entries in the table is $2^{(n-1)}\times$ (number of haplotypes) $\times$ (number of children). For example, in 4-locus problem, with allele product (i.e., number of haplotypes) $2\times3\times4\times5 = 120$, and at most 10 children in a family, the table would have $8 \times 120 \times 10 = 9600$ entries.

4 Handling Loops

One of the causes of long LINKAGE runs is loops in the input pedigree(s). The algorithm used in LINKAGE to handle loops was first proposed by Lange and Elston [6] and is also described on pages 170–171 of [12]. We review it here to keep the paper self-contained.

Suppose for simplicity that there is just one one loop in the pedigree; this is by far the most common case. LINKAGE expects the preparer of the input to designate one individual in the loop as the *loop breaker*, b. This is done by making two copies of the individual b, which we call b_1 and b_2. Person b_1 is shown as the child of b's parents, but b_1 has no spouse or children. Person b_2 has b's spouse and children, but has no parents.

If person b can have only one joint genotype then the modified pedigree is computed in a manner equivalent to having no loop declared. If person b can have several genotypes, $\{G_1, G_2, \ldots, G_p\}$ then the algorithm iterates over all these genotypes. For each possible genotype G_i, the likelihood is computed conditioned on both b_1 and b_2 having genotype G_i. Ott [12] denotes this by $P(x, G_i)$, where x stands for the phenotype data observed on the pedigree members. Then the overall likelihood is given by

$$\sum_{i=1}^{p} P(x, G_i)$$

Each computation of $P(x, G_i)$ does a fresh traversal of the entire pedigree updating `genarrays`. In each nuclear family, there is a single representative, r, such that the `genarray` of r is updated during the traversal; the update for r is conditioned on the probabilities for the other members of the nuclear family, so that their probability updates are done implicitly. We observed the following recomputation happening and found a simple way to avoid it. Suppose that the loop does not encompass the entire pedigree. Then there may be certain nuclear families where the `genarray` values are computed independently of b's genotype. These nuclear families do not need to be revisited on each traversal, since the update done will be the same for each G_i. We have modified the pedigree traversal algorithm, so that during the traversal for the first possible genotype of b, which we called G_1, the set of nuclear families that do not have to be revisited is identified.

There is a colorful biochemical metaphor that captures precisely how the decision of whether a nuclear family needs to be revisited is made. Imagine that we take some radioactive dye that we can "inject" into the pedigree, so that some members will be radioactive and others not. If a nuclear family representative r is radioactive and is updated on the first traversal, then that nuclear family must be revisited, and r updated again, for each G_i. If r is not radioactive, then its nuclear family does not need to be revisited. Here are the rules we use for updates:

Rule 1. Before the first traversal for G_1: b_1, b_2, and all the descendants of b_2 (b_1 has no descendants, by definition) are made radioactive.

Rule 2. During the traversal for G_1: for each nuclear family, if any member of the nuclear family is radioactive, then the representative is made radioactive.

Rule 3. During later traversal for $G_2, G_3, \ldots$: we update the representative of a nuclear family if and only if the representative is radioactive.

The reason for Rule 1 is that LINKAGE takes into account which genotypes an individual can inherit when initializing `genarray`. Thus for b_1, b_2, and all the descendants of b_2, the initial `genarray` for each traversal depends on the choice of genotype for b_1 and b_2 and must be recomputed.

When there are multiple loops, each loop is broken separately. The algorithm iterates over all vectors of genotypes, where component j of the vector is a possible genotype for the j^{th} loop breaker. We apply our traversal algorithm to the loop whose component is the least significant or innermost in the vector iteration. This is the component that always changes from iteration to iteration.

The ordering of loops from inner to outer is determined by numbers that the user supplies in the input pedigree. Running times for all versions of LINKAGE can vary widely depending on the loop order chosen by the user.

There is another user-specified parameter that can cause the running time to vary significantly. This parameter is the *root* (in LINKAGE this is referred to as the *proband*) of the pedigree. The traversal order "peels" the pedigree towards the root so that the root belongs to the last nuclear family updated. A variety of peeling orders are possible [3, 12]. We have not changed the peeling order in LINKAGE, but we make no claim that the current order always minimizes the running time. A good, but overly simplistic heuristic to minimize running time, is to choose the root as high as possible and the loop breaker(s) as low as possible in the pedigree, but this choice is not provably optimal in all cases.

Automating the choice of proband and loop breakers, and varying the peeling order are interesting research questions beyond the scope of this paper. Nevertheless, our idea of not updating pedigree members whose probability update does not depend on the genotype of the loop breaker(s) will save time on some pedigrees, for any method of choosing root, loop breaker(s), and traversal order. However, the time saved may depend significantly on those choices.

5 Coordinating Optimization and Output

The previous versions of ILINK and LODSCORE do several wasteful FEs. We identified three distinct sources of wastefulness:

1. When it is decided that the previous estimate $\hat{\theta}_i$ is better than the new estimate $\hat{\theta}_{i+1}$, the gradient at $\hat{\theta}_{i+1}$ is still calculated but never used.

2. The output function evaluates `fun` at the optimal $\hat{\theta}$, even though this was already done by `gemini`.

3. The function which reports lodscores reevaluates the likelihood for each pedigree at the optimal $\hat{\theta}$, even though this was already done inside `fun` when `gemini` called `fun` with the optimal $\hat{\theta}$.

The extra FEs for cause 1 can be eliminated because the gradient at the nonoptimal $\hat{\theta}_{j+1}$ is not needed.

The extra evaluations from causes 2 and 3 can be eliminated simply by storing the values when the same calls are made from `gemini`. The only subtlety is that `gemini` does not decide until shortly after the call to `fun` whether the value used in the most recent call is best so far. Thus the likelihood and `fun` values are stored provisionally pending that decision.

The number of FEs in cause 1 is typically $n - 1$ if forward differences are used to compute the gradient and $2(n-1)$ if central differences are used. Which method is used, and whether the number of FEs matches the typical number, both depend on some program constants and the specific structure of the problem instance.

Cause 2 saves 1 FE and cause 3 saves an amount of computation which is almost the same as for an FE. It should be noted that the old versions of the LINKAGE programs did not count the computations for output when they reported a number of FE's (in the file `final.dat`). Thus when comparing the files produced by old and new versions only the savings from cause 1 are seen in the reduced number of FE's.

In the course of testing our code to save these FE's we found and *repaired* a bug in ILINK and LODSCORE. In some situations the old version was deciding correctly that $\hat{\theta}_j$ was the local optimum, but then reporting the slightly non-optimal $\hat{\theta}_{j+1}$ and its lodscore as the final value. Our new code to coordinate optimization and output corrects this problem.

6 Checkpointing

In this section we describe how we save partial computations of LODSCORE and ILINK, so they can be restarted if the computer crashes. For the purposes of checkpointing, these two programs are almost identical. LODSCORE is slightly more complex because we must store which pair of loci we are working on, while ILINK only works on one locus ordering per run.

There are publically available packages that can checkpoint a general application by taking a core dump of the system state. However, the packages we have seen are designed for specific implementations of UNIX and are not portable. Therefore, we implemented checkpointing by modifying the LINKAGE source code in a portable fashion.

The programs perform checkpointing before some calls to `fun`.

The functions that make calls to `fun` are: `initialize`, `firststep`, `increaset`, `decreaset`, `gforward`, and `gcentral`. After the changes described in the previous section, the output function `outf` no longer makes any calls to `fun`. It does evaluate the likelihood at $\hat{\theta} = 0.5$ for every dimension, to report what is called "Ott's generalized LODSCORE". The function `initialize` is called at the beginning of the first iteration and generates one FE. The function `firststep` is called in every subsequent iteration and generates one FE. In each iteration, either the function `increaset` or `decreaset` is called; they generate a variable number of estimates of $\hat{\theta}$ and call `fun` to evaluate each one. In the vast majority of iterations we have seen `increaset` and `decreaset` generate one or two estimates, but there are exceptions. At the end of each iteration either `gforward` or `gcentral` is called to estimate the gradient by generating FEs with a slightly perturbed $\hat{\theta}$. Both `gforward` and `gcentral` have a loop in which one dimension is perturbed at a time; `gforward` makes one FE per loop iteration and `gcentral` makes two FEs per loop iteration.

We take a checkpoint at the start of `initialize`, `firststep`,`increaset`, `decreaset`, `outf`, each iteration of the loop in `gforward`, and each iteration of the loop in`gcentral`. Our rough goal is to take at least one large checkpoint between every two FEs; the only time this goal is not met is when a call to `increaset` or to `decreaset` makes more than two FEs. When a crash occurs, we can restart the computation from the last place where a checkpoint was completed. Since this is usually at most two calls to FEs away, it means that we can have reasonable hope of doing runs where the mean-time-to-failure of the computer is about the time it takes for two FEs.

Checkpointing is done by storing all the relevant program variables in a file. A typical checkpoint file on the SunOS operating system has length roughly 15,000 bytes, which is very small by modern checkpointing standards. The file includes some control flow information, so we can tell where the checkpoint was taken during the recovery process. The file contains a special string at the end so we can tell if the checkpoint is complete. The reason this is needed is that there is a small chance that the computer may crash while a checkpoint is being written out. To prepare for this situation we keep a backup copy of the preceding checkpoint file after the next checkpoint is taken. If we detect during the recovery process that the principal checkpoint file has been corrupted, then we advise the user as to where to move the backup checkpoint file so we can recover from an earlier point in the execution and still not lose too much.

When the programs complete their linkage computations successfully, the checkpoint files are deleted at the very end of the execution. When the programs start up, they determine whether to start from a checkpoint or start from the beginning depending on whether the checkpoint file exists in the working directory.

Because ILINK estimates the θ vector for only one order of loci per run, it is common practice to make scripts that execute multiple runs of ILINK one for each order. It is also possible to make scripts that execute multiple runs of LODSCORE, although this is not nearly as common because LODSCORE during a single run estimates θ for all pairs of loci specified. If a crash occurs during the fourth run in a script say, we would like to be able to keep the results from the first three runs and have the machine know that when the script is restarted, it should start from a crash recovery for the fourth run. We call this *script-level* checkpointing. It is important to realize that script-level checkpointing is essentially external to the programs themselves, and has been integrated mainly for the convenience of the user, who might not be as aware of the state of the computation as the computer is.

The process introduces a small auxiliary program that we call `ckpt`, which behaves as a front-end to the user desiring such checkpointing. In collaboration with the individual programs ILINK and LODSCORE, the `ckpt` program keeps track of the number of invocations of the respective program made during the current script. This count is written to a disk file. In addition, during execution, the programs store information on script-level output, namely the state of the two key output files `final.out` and `stream.out`, which are needed to get the results. The file final `final.out` stores a description of the run, the final $\hat{\theta}$ and some diagnostics. The file `stream.out` store the final $\hat{\theta}$ and the final likelihood values for each pedigree.

Suppose now that a script prematurely terminates; when execution is resumed, the mechanism knows which invocation of LODSCORE or ILINK in the script was last in force. Thus, the script is executed again from the beginning, but completed invocations of the above programs exit immediately without contributing any output. Further, since the state of the output files was saved before the crash occurred, these files can merely be copied into from the stored versions, thus yielding no indication that the script had to be re-started.

7 Methods

We compared the LINKAGE programs described in [2] to the modified new versions described here on some sample runs. We first installed the three changes that improve the speed of the programs and measured the improvement. Then we installed the checkpointing facility and tried to measure how much it slowed down the programs under crash-free operation, but the slowdown was too small to measure with any precision. The timing experiments were run on an unloaded Sun SPARCstation 2 computer with 32Mbytes of RAM. This machine runs the operating system SunOS, version 4.1.2, which is an implementation of UNIX. To compile all versions of the programs we used the `gcc` compiler, version 2.3.3 using the `-O` flag for optimization. The times reported in the next section are the user time given by the `time` command. Since our test runs were usually the only user processes running, this is a pretty good estimate of the actual time taken.

8 Results

We present results to show the performance obtained by our improved implementation. We did a variety of sample runs always comparing the version of the code reported in [2] with the version described here. We tried to measure the cost of checkpointing by using new versions both with and without checkpointing. We were pleased to find that on our computers, the extra time for checkpointing in any non-trivial run is less than the time variation between different executions of the same run. In a few cases the time we measured for a run with checkpointing was as much as 1% *less* than the time measured without checkpointing.

# Data Set	Program	No. of Alleles	Old Time(s)	New Time(s)	Speedup
RP01	LODSCORE	2×9	1121	468	2.40
RP01	ILINK	$2 \times 3 \times 3$	8671	3589	2.42
RP01	ILINK	$2 \times 3 \times 4$	71145	27687	2.57
RP01-3	ILINK	$2 \times 6 \times 9$	6023	4349	1.38
RP01-3	LINKMAP	$2 \times 3 \times 5 \times 6$	14200	13577	1.05
BAD	ILINK	$2 \times 4 \times 4$	4185	2868	1.46
CLP	ILINK	$2 \times 4 \times 4 \times 4$	7012	6285	1.12
CLP	ILINK	$2 \times 4 \times 4 \times 4$	9705	9067	1.07
CLP	LINKMAP	$2 \times 4 \times 4 \times 4$	3445	3455	1.00

Table 1: Execution Times in Seconds

For simplicity and consistency, we report the times taken by the versions in [2] and the new version with checkpointing. Those are the two versions we have been and will be distributing.

We used the three following disease data sets for our experiments:

- 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 190 individuals containing 2 marriage loops [1]. As shown in [1], this pedigree had to be split into three pieces because desired computations with large allele products on the whole family together took prohibitively long. In the tables of results, RP01-3 denotes 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 [10].

- CLP: Data on 12 families with autosomal dominant nonsyndromic cleft lift 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 [4]. The families include 110 individuals in all. Pedigrees 1000 and 1100 are significantly larger than the rest. Pedigree 1200 has a loop, but it encompasses the entire family, so our new loop algorithm does not help.

In most cases, we achieved noticeable speedups. The speedups here are not nearly as large as the speedups we obtained in going from LINKAGE 5.1 to the faster version in [2], but are still worthwhile. Our changes helped the most on the RP01 pedigree because its second loop covers only a small part of the pedigree. The above computations on the full RP01 pedigree are feasible in part because of the algorithmic changes in [2] and because the allele products for the loci we chose are not too high. Our changes had the least effect on CLP because the loop change is irrelevant there and its `genarray`s are not dense enough to benefit much from subdividing recombination classes.

By comparing some pairs of runs in the above table we can see how the three different changes affect the speedup on specific pedigrees. For example, the last LINKMAP run shows that CLP does not benefit from either of the first two changes; of course, this run cannot benefit from the iteration saving change in LODSCORE and ILINK. The loci used in the last two runs are the same. The non-disease loci in the two CLP ILINK runs are completely different, although they have the same number of alleles. Thus we see that the iteration saving change saves about 700 seconds on each CLP ILINK run. The speedups are different because the two runs have different numbers of iterations, but the absolute amount of computation saved is approximately the same.

Comparing the RP01 runs versus the RP01-3 runs, we see that the new way of handling loops must account for most of the savings on RP01. Since RP01-3 has no loops, it benefits only from the first and third changes.

We see no effective way to estimate on paper what the speedup from the three speed improvements will be for a new run because:

1. The savings from our change to recombination classes depend on the sparsity patterns of the `genarray`s of different individuals in the pedigree, which are hard to determine without running the program.

2. The savings from our better loop traversal algorithm depend on the loop structure, the choice of proband, and the sparsity patterns. Knowing the loop structure and proband, it is possible to compute by hand which nuclear families will not be revisited for each genotype of the loop breaker. However, it is not easy to estimate, how much time will be saved by not visiting those families.

3. The savings achieved by better coordination between optimization and output depend on details of the numerical convergence of the θ estimates. These details cannot be determined without actually doing the run.

9 Discussion

One of us (RWC) recently asked Elston what led to the breakthrough Elston-Stewart algorithm. Elston replied,"That's the way I had always done the [linkage] computations by hand." One of the key principles of successful pencil and paper computation is to write down a few intermediate values, so they do not have to be recomputed and so they can be checked in case of errors. The principle of avoiding recomputation was well understood before the time of Newton; it was the stimulus for the hand-computation and publication of vast logarithm tables, for example. Unfortunately the principle is too often forgotten by late 20th century programmers when they assume *incorrectly* that:

1. Having the computer recompute intermediate results does not add significantly to the computation time because the computer is so fast.

2. If the computer crashes, it is satisfactory to restart the program.

We have corrected three places where the LINKAGE programs were spending nontrivial amounts of time recomputing known values internally. We have made it possible for the LODSCORE and ILINK programs to recover from a crash by restarting from a recently completed likelihood function evaluation.

To stress the importance of the last improvement, consider the following scenario which is realistic based on our experience with LINKAGE. We wish to do a multilocus ILINK run where every function evaluation takes roughly a day. We have available to us a workstation whose mean-time-to-crash is roughly a week. Given that the ILINK computation will surely take at least 20 function evaluations, it does not seem worth trying because the workstation is sure to crash during the 20 days it would take to finish the run.

With the checkpointing facility in place, we now see no significant impediment to doing our desired ILINK run. Because operating systems such as UNIX allow processes to run in the "background" at low priority, we can do long runs without interfering with other high priority computations.

Acknowledgments

We thank Dr. Ramana M. Idury for his substantial contributions to the earlier stages of this research. Thanks to 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 were 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. This work was supported by grants from the National Science Foundation, the Human Genome Program of the National Institutes of Health, 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] R. C. Elston and J. Stewart. A general model for the analysis of pedigree data. *Human Heredity*, 21:523–542, 1971.

[4] J. T. Hecht, Y. Wang, B. Connor, S. H. Blanton, and 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.

[5] 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.

[6] K. Lange and R. C. Elston. Extensions to pedigree analysis. I. Likelihood calculation for simple and complex pedigrees. *Human Heredity*, 25:95–105, 1975.

[7] 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.

[8] G. M. Lathrop and J. M. Lalouel. Efficient computations in multilocus linkage analysis. *American Journal of Human Genetics*, 42:498–505, 1988.

[9] 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.

[10] 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.

[11] 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.

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