Parallel Genehunter: Implementation of a Linkage Analysis Package for Distributed-Memory Architectures

Gavin Conant[*], Steve Plimpton[†], William Old[‡], Andreas Wagner[*], Pam Fain[‡], and Grant Heffelfinger[†]

[*]Department of Biology, The University of New Mexico, [†]Computation, Computers, and Mathematics Center, Sandia National Laboratories, and [‡]Health Sciences Center, The University of Colorado

Abstract

We present a parallel algorithm for performing multipoint linkage analysis of genetic marker data on large family pedigrees. The algorithm effectively distributes both the computation and memory requirements of the analysis. We discuss an implementation of the algorithm in the Genehunter linkage analysis package (version 2.1), enabling Genehunter to be run on distributed memory platforms for the first time. Our preliminary benchmarks indicate reasonable scalability of the algorithm for even small fixed-size problems, with parallel efficiencies of 75% or more on up to a few dozen processors.

1. Introduction

Linkage analysis uses the positions of known genetic loci to locate unknown genes on a chromosome. One of the primary applications of the technique is the identification of loci associated with diseases. The technique works by using the number of recombination events between two loci on the same chromosome as a distance measure (1, 2). Linkage analysis in humans is complicated by high degrees of homozygosity. Fisher, Haldane and Smith, and Morton proposed using maximum likelihood to infer genetic maps from imperfect data (3-5).

The data studied in linkage analysis are generally pedigrees of families affected by some disease. Two algorithms are commonly used for this problem. The algorithm described by Elston and Stewart scales linearly with the number of individuals in the pedigrees, but exponentially with the number of genetic loci, or *markers* (6). In 1987, Lander and Green proposed a complementary algorithm with linear scaling in the number of markers but exponential scaling in both time and memory in the number of individuals in the pedigree (7). Lander and Green's algorithm is widely used for *multipoint* linkage analysis: *i.e.* problems where many markers are considered simultaneously. Unfortunately, many researchers are interested in analyzing datasets which are too large for current implementations of the algorithm. Dwarkadas *et. al.* have previously presented a shared memory parallelization of a linkage analysis code; however, it was not designed to scale to more than a few processors (8). We present here a parallel Lander and Green algorithm that has been implemented in the Genehunter program (9); to our knowledge it is the first distributed-memory implementation of the Lander and Green algorithm.

The Lander and Green algorithm is based on a novel representation of inheritance data. These authors point out that the inheritance of a genetic locus in a pedigree can be completely described by identifying from which parental chromosome each child derives its alleles at that locus. Figure 1 presents a simple example of this principle. Two parents (who are termed *founders* because their parents are not present in the pedigree) have a single offspring. Each parent has two copies of the locus in question. The father (top square) has different alleles at this locus—an **A** from his father and an **a** from his mother. The mother (circle) is homozygous: she received an **A'** from both parents. When considering the offspring (bottom square), we can describe the offspring's two alleles at this locus simply by indicating in binary coding whether he received the allele from his grandfather or his grandmother. In this case, we can unambiguously state that he received an **A** from his paternal grandfather and designate his parental chromosome with a 0. On the maternal chromosome we cannot unambiguously determine whether the offspring received his allele from his maternal grandmother or grandfather. We therefore must consider both

Address for correspondence: Gavin Conant
Department of Biology
167 Castetter Hall
The University of New Mexico
Albuquerque, NM 87131-1091
Email: gconant@unm.edu

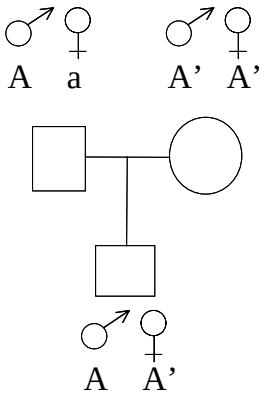

Figure 1: A example pedigree illustrating Lander and Green's approach to representing inheritance patterns as binary strings. Alleles at this locus are represented by the letters A, a, and A'.

possibilities throughout the analysis, leaving this chromosome coded as 0/1.

This example is artificial because we assume that the *phase* of the two parents is known: *i.e.* that we can distinguish paternal from maternal chromosomes. In fact, this is generally not the case; instead, common algorithms assign definitions of maternal and paternal to founders. Since this assignment is arbitrary, it is often referred to as *founder-phase symmetry*.

Lander and Green's algorithm works by representing each possible inheritance pattern for the pedigree as a string of 2*n* bits, where *n* is the number of non-founding individuals in the pedigree (a single offspring in the example above). In fact, we can make use of the founder symmetry described above, so that instead of considering all *2n* bits, by picking a definition of maternal and paternal for each founder, we can reduce the size of the representation to *2n-f*, where *f* is the number of founders in the pedigree. For details of this modification, see (9).

Although any given inheritance pattern can be represented in *2n-f* bits, uncertainties about the actual pattern of inheritance at each locus (as in the maternal chromosome above) means that no single pattern will represent the data exactly. Instead, a (possibly zero) probability is assigned to each of the 2^{2n-f} possible inheritance patterns (an *inheritance vector of probabilities*) at each marker in the map. In multipoint linkage analysis, it is assumed that one has a genetic map containing the recombination distance between each pair of markers. Using this map information, it is conceptually straight-forward to use a Markov-chain approach to calculate the probability of each marker, conditional on all of the markers before or after it on the map. Consider two markers m_1 and m_2 separated by a recombination distance θ (in other words, two markers which undergo recombination between each other with probability θ). For each of the possible inheritance patterns *i* in m_2, define a distance *d(i,j)* between pattern *i* and each possible pattern *j* in m_1. Any bit position where *i* and *j* differ implies a cross-over event. Thus we compute *d(i,j)* as the Hamming distance between *i* and *j*. The probability of the transition between pattern *j* at m_1 and pattern *i* at m_2 is given by

$$\theta^{d(i,j)} \cdot (1-\theta)^{2n-f-d(i,j)} \qquad (1)$$

Using this formula, one can create a transition probability matrix *M(i,j)* where the i,j^{th} entry gives the probability of the transition from inheritance pattern *i* to *j*, as calculated by (1). Given *inheritance probability vectors* P_1 and P_2 (containing the probability of every inheritance pattern at marker 1 and 2, respectively), we can calculate $P_{2|1}$ (vector of inheritance pattern probabilities at marker 2 conditional on the probabilities at marker 1) by:

$$P_{2|1} = P_2 \circ (M \cdot P_1) \qquad (2)$$

where ^{o} represents a component-wise vector product. (2) can be then applied recursively to calculate any required conditional probability vector. This Markov-chain approach is an $O\left(\left(2^{2n-f}\right)^2\right)$ time algorithm, but the structure of matrix *M* allows the matrix-vector multiplication to be performed as an FFT, reducing the complexity of the Genehunter algorithm to $O\left(2^{2n-f} \cdot \log^2\left(2^{2n-f}\right)\right)$ (10). It is important to note that although *M* is a convenient mathematical description of the transition probabilities, its structure is such that there are only *2n-f+1* distinct entries; no object of size $2^{2n-f}x2^{2n-f}$ need ever be stored in the linkage analysis computation.

2. Genehunter Computation:

The Genehunter 2.1 software (11) uses the above algorithm to compute likelihood and non-parametric scores for the occurrence of a disease gene at a number of user-requested locations in a genetic map. Genehunter's computation proceeds in three distinct stages:

1. Calculation of the probability of each possible *2n-f* -bit inheritance pattern for each marker and for the disease gene. Calculation of a nonparametric statistic for each inheritance pattern.
2. Calculation of the conditional inheritance probabilities for each marker, conditioned on all markers to the right of it in the map and on all markers to the left. We refer to this operation as

"walking" up or down the map, using at each marker the results of the last marker to calculate conditional probabilities using equation 2.

3. Calculation of likelihood and non-parametric scores for the requested disease location(s). In Genehunter, scores are calculated for placing the disease gene at each marker, and at a default of five evenly-spaced points between every pair of markers.

In addition to the FFT mentioned above, Genehunter 2.1 introduced an important improvement, based on the insight that some of the 2^{2n-f} possible inheritance patterns will be precluded by the data and can be ignored. Recall in the example above that the offspring's allele on the paternal chromosome could not have been inherited from the paternal grandmother. Therefore inheritance patterns of the form 1* can be ignored. Clearly, each restriction of this kind reduces the number of possible inheritance patterns by half, since each restriction excludes one of the two settings at a bit position. Thus, for many pedigrees, these restrictions substantially reduce the problem size. (For the full details of this improvement, see (11)). This improvement reduces the size of the vectors used to store inheritance probabilities from $O(2^{2n-f})$ to $O(2^{2n-f-k})$, where k is the number of inheritance bits that can be unambiguously determined, or *fixed*. There is also a similar effect on running time.

3. Memory requirements in Genehunter:

On many computers, linkage analysis problems are limited by the amount of available physical memory, rather than running time (unpublished data). The memory requirements for Genehunter consist of two distinct parts: the memory needed to store the inheritance probability vectors for the markers and the memory required to store the inheritance probability vectors for the disease phenotypes. This second vector stores the probability of seeing the observed disease phenotypes in the pedigree for each inheritance pattern. Because of the non-deterministic mapping of genotype to disease phenotype, it is impossible to definitively exclude any inheritance patterns. As a result, the vector of inheritance probabilities for the disease always requires $O(2^{2n-f})$ memory. (It is possible to perform *non-parametric* linkage analysis which does not always require storing a vector of this size; but the computation has an identical form and we will not discuss this here). In the worst case, the amount of memory required for all the marker probability vectors could be as high as $O(m2^{2n-f})$, where m is the number of markers. However, the presence of fixed bits in the dataset will almost always mean that the actual memory requirements for a given dataset are significantly lower.

4. Parallelization Approach:

In order to allow larger problems to be solved, a scalable parallelization scheme for Genehunter must partition both the computations and memory. We discuss the parallelization of each of the 3 steps of the previous section separately. The input and output files for Genehunter are typically quite small (no more than a few hundred lines of text and postscript), meaning that parallel I/O is not a significant bottleneck.

4.1: Step 1:

Step 1 is the most straight-forwardly parallelizable part of Genehunter. The purpose of step 1 is to calculate, for each inheritance pattern, the probability of that pattern at each marker, the probability of that pattern given the disease phenotype data, and the non-parametric score for that pattern. The non-parametric scores and disease probabilities are independent; thus they can simply be divided evenly across processors, so that each processor owns and operates on inheritance probability sub-vectors of length $2^{2n-f}/P$, where P is the number of processors.

The distribution of the marker vectors is slightly more tricky: each marker has a vector of size 2^{2n-f-k}, where k is the number of fixed bits for that particular marker. One possible approach would be to store entire inheritance probability vectors on each processor, *i.e.* each processor would be assigned a fraction m/P of the m marker vectors. However, the size of each vector varies considerably from marker to marker depending on k (the number of fixed bits), making load-balancing with this approach problematic. Our strategy of having each processor store a fraction $2^{2n-f-k}/P$ of every marker vector is better balanced.

4.2: Step 2:

Step 2 consists of calculating, for each marker i, the vector of inheritance probabilities $P_{i|i-1..0}$ (probability of each inheritance pattern at marker i conditioned on markers 1 through i-1) and the vector $P_{i|i+1..m}$ (inheritance pattern probabilities conditioned on markers $i+1$ through m). This calculation is performed using FFTs in the conceptual manner of equation 2.

For the moment, assume that each processor has all the elements of the conditional probability vector at marker $i-1$ needed to compute the conditional probabilities at i. (Since the vectors at markers $i-1$ and i

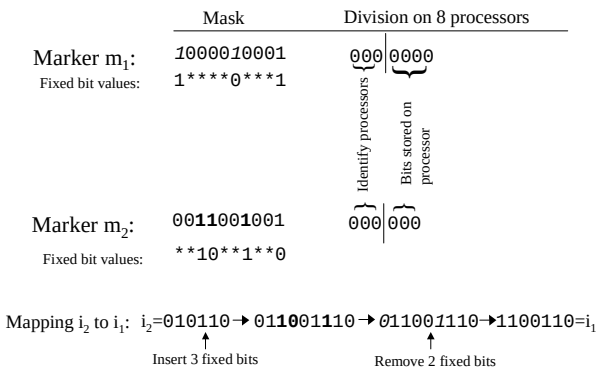

Figure 2: Examples of fixed bits for two markers, m_1 and m_2. Masks for the fixed bits are shown, with the corresponding fixed bit values shown underneath. The lower half of the figure shows the mapping of an index in m_2 representation to m_1 representation.

are not typically the same length this is not a valid assumption; we deal with this additional data movement complexity below.) The calculation itself consists of first using an FFT to compute a matrix-vector product similar to that seen in (2). The presence of *k* fixed bits at a marker means that the size of the probability vector is 2^{2n-f-k} and the matrix has effective dimension 2^{2n-f-k}x2^{2n-f-k}

In Genehunter, the matrix-vector multiply is replaced by an FFT-based convolution, with 2 forward 1d FFTs on vectors of length $(N=)2^{2n-f-k}$, followed by an element by element multiplication and an inverse FFT. Note that for large n, these 1d FFTs are still quite computationally intensive. Note also that the elements of each N-length vector are distributed across the P processors in contiguous chunks. Conceptually this data layout can be viewed as a 2d matrix of values with P rows and N/P elements in each row, and each processor owning a row of the matrix. The FFT operation can then be parallelized the same way that a 2d FFT is performed on a distributed memory parallel machine. N/P-length 1d FFTs are first performed within each row (an on-processor computation), then a matrix transpose is performed which requires all-to-all communication between the processors, followed by a series of P-length 1d FFTs on data that is now local to each processor. The inverse FFT simply reverses this process.

The result of the FFT calculation is a new conditional probability vector of the same size as the original, still distributed evenly among the processors. The remaining calculation is a component-wise vector product between every element of the probability vector at marker *i* and the corresponding element in this new conditional probability vector, yielding a vector of the size of the original vector at marker *i*. This calculation can be done very efficiently in parallel with each processor calculating a component-wise product with its particular portion of the probability vector.

4.3: Step 3:

Conceptually, step 3 is very similar to step 2. The major difference is that in step 2, we calculated the conditional probability of all of the inheritance patterns at a marker given the markers to the left or right, while in step 3 we are calculating the probability of the disease gene being at position *x* in the map, given the markers to the left and right of *x*. This probability can be written as

$$p = P_D \cdot P_{x|1..i} \cdot P_{x|i+1..m} \qquad (3)$$

where P_D is the disease vector and $P_{x|1..i}$ and $P_{x|i+1..m}$ are calculated in the manner of equation 2. Non-parametric scores are calculated in a similar manner using the non-parametric scores rather than P_D. An FFT is used with the conditional probabilities calculated in step 2 to calculate the conditional probability of each inheritance pattern at the point *x* from the marker at left and at right. In this case, θ is given by the distance between the marker and *x*. Once again, the calculation of this dot-product can be done efficiently in parallel once each processor has the data needed for its part of the calculation.

4.4: Redistribution of marker vectors for computations of vector products:

In the above discussion we ignored one very important complication. If all marker vectors were of size 2^{2n-f}, the computation of dot and component-wise vector products between markers could be trivially distributed among processors, because element *i* in one marker could be mapped directly to the same element *i* in any other marker. However, the introduction of *k* fixed bits at a particular marker location means that a particular marker vector is actually of length 2^{2n-f-k}. Since the values of *k* for adjacent markers are often different, the data layout of the 2 marker vectors is also different and that redefines the mapping between inheritance probability vectors. Fixed bits are generally represented as bit masks of length *2n-f* with 1s at positions where fixed bits occur. Figure 2 gives a possible configuration of the fixed bit masks for two adjacent markers m_1 and m_2. It is important to note that each fixed bit has an associated value (also shown in figure 2).

Only non-fixed bits are stored in inheritance probability vectors. Thus, in figure 2, marker m_1

	Mask in target representation
Marker m_1:	100100
Fixed bit values:	1**0**

	Mask in source representation
Marker m_2:	0110100
Fixed bit values:	*10*1**

Figure 3: Example showing the representation of the masks in figure 2 in each other's representation.

would have a size of 2^7 and marker m_2, 2^6. Suppose we are mapping inheritance pattern i (represented as a binary sequence of $2n\text{-}f$ digits) from m_2 to m_1. We first remove any fixed bits k_1 that are common to both markers (such as the last bit in figure 2). We now have two indexes i_1 and i_2 each of length $2n\text{-}f\text{-}k_1$. We now define two new operations for the mapping of indices between markers: (1) If m_2 has k_2 fixed bits not present in m_1, (shown as **bold** in figure 2) we look up the value of those fixed bits and insert them in the appropriate locations of i_1. Thus index i_1 now has length $2n\text{-}f\text{-}k_1+k_2$. (2) If m_1 (the target) has fixed bits not present in m_2 (shown as *italics* in figure 2), we drop those k_3 bits. The final size of i_1 is therefore $2n\text{-}f\text{-}k_1+k_2\text{-}k_3$.

Consider the example index i_2=010110. We can use the masks in figure 2 to create the corresponding element i_1 in m_1. The fixed bit in the ones position has already been removed from m_2, and can be ignored. First, we insert three fixed bits into i_1 at the locations specified in the mask. This gives us 011001110 (figure 2 shows the inserted bits in **bold**). We now drop the two fixed bits present at m_1 but not m_2 (shown in *italics* in figure 2). The result is i_1=1100110.

Unfortunately, the above process of adding and removing bits may result in the need to access indices (and the associated data) that are owned by other processors. We can visualize the processor distribution of each marker by writing a bit-string of the length of each marker and drawing a line through it after $log_2(P)$ bits (assuming we have 2^P processors for some integer P). This operation is shown in figure 2 for an eight-processor (3-bit) distribution. For our example index above, we see that i_2=010|110, meaning that i_2 is located on processor 010 (4). However, we find that i_1=110|0110, meaning i_1 is located on processor 110 (7). Clearly, we may need to redistribute probability vectors when we compute the dot or component-wise products between markers.

This redistribution may at first appear to be costly, but the bit patterns in the data allow the construction of reasonably efficient communication routines. It is first convenient to represent the source and target masks in their partner's variable bit space. Thus, the target mask in the source representation shows all the locations in the target (m_2) where there are fixed bits not present in the source (m_1). The symmetric situation applies for the target mask represented in the source configuration. Figure 3 gives examples for the masks in figure 2.

We can now use these two new masks and apply the same procedure of considering only the highest $\log_2(2^p)$-order (*processor-order*) bits in each mask. For eight processors, m_1 in m_2's representation is 100|100 and m_2 in m_1's representation is 011|0100. The first thing to note is that cases where the processor-order bits are all zeros require no communication. This case is actually fairly common when the number of processors is small relative to $2n\text{-}f$. There are, however, two other cases to consider.

First, there may be fixed processor-order bits in the target not in the source. As described in operation (1) above, these cases require looking up indices in the source that match the values of the fixed bits in the target. In processor-order bits, this implies that processors whose value at that position do not match the fixed bits will be idled. For instance, in figure 3, only processors with ranks of the form *10 will contribute to the computations in the source. The communication algorithm based on this observation is straight-forward: the subset of vectors contained on non-idled processors are evenly redistributed on all processors using a logarithmic time scatter-type operation. For the example in figure 3, processors 010 and 110 would each split their data with their three neighbors: 010 with 000-010 and 110 with 011-111. In doing this communication, we can use any non-processor-order fixed bits to realize a time-savings. Note that each non-processor-order fixed bit reduces the size of the vector that must eventually be split among processors by ½, so instead of distributing the complete source vector on a processor, we need only distribute a compacted version where the elements that match the fixed values are included.

The second case is when there are processor-order fixed bits in the source not in the target. As noted in operation 2 above, this situation results in reuse of elements in the source vector. In this case the processor-order fixed bits create equivalence classes of processors which all need identical data. Every such fixed bit doubles the size of the resulting data vectors and halves the number of equivalence classes. For instance, in figure 3, each processor is paired with the processor that differs from it only at the highest order

Source Mask: 100|0000

Processors:	000	001	010	011	100	101	110	111
Block:	00	01	10	11	00	01	10	11
Block Rank:	0	0	0	0	1	1	1	1
Original Block:	00	00	01	01	10	10	11	11
Original Block Rank:	0	1	0	1	0	1	0	1

Figure 4: Example showing the distribution of a marker with mask 1000000 onto a target with mask 0000000. Processor ranks are shown in standard binary representation. The following two lines indicate the block which that processor will need and the its rank in that block. The remaining lines show how the blocks are initially distributed on processors.

bit position. Thus, 000 and 100 will have identical data, as will 001 and 101 and so forth. We refer to these equivalence classes as "blocks". We can also calculate the "original" block for each processor (which is just its rank divided by the number of blocks) and its "original block rank" (which is the modulus of its rank with the number of blocks). When we have this information, we can create block ranks for each processor. This is done in the following way:

- If the processor's original block and correct block are the same (as for processor 0 in figure 4), then that processor's block rank is set to its original block rank
- The remaining block ranks are assigned sequentially in rank order.

Once block ranks are assigned, we use a unit time exchange operation to put the block pieces on the processor with the corresponding block rank. The above assignment of block ranks avoids unnecessary communication when processors already have a piece of the block they will eventually need. Once this exchange has taken place, the communication is simply a logarithmic time all-gather operation *for that block* using block ranks.

In step 3, the target is the disease probability vector, which has no fixed bits, meaning that we only need to consider fixed bits in the source (the second case above). In step 2, there can be arbitrary combinations of target and source fixed bits. Thus, the two communication schemes described above must be slightly modified. Instead of the blocks being originally distributed on all processors, they are only distributed on the non-idled ones (where the processor ranks match the fixed bit values). This means that one processor may hold more than one block piece (or indeed more than one block). Thus, in the first algorithm above, instead of every processor sending its block piece to another processor, only the live (non-idled) processors send data. If more than one block is present on each live processor, the communication loops over the number of blocks per processor, at each step sending a block to a processor that will need it. Otherwise, the communication takes place in a single step, with each live processor that has a block piece it will not need sending it to a processor that will. One of several situations may result from this first communication step:

- If only the target processor-order bits were non-zero, communication is complete.
- If there were more target processor-order bits than in the source, then each block is currently completely stored on a single processor with block rank 0.
- If there were more source processor-order fixed bits than in the target, then each block is either partially or completely distributed on the processors that will need it.

The second case above (target processor-order bits > source processor-order bits) can be simply handled with a logarithmic broadcast operation. The last case is more complicated. First, all processors that have some piece of the block perform a logarithmic all-gather to obtain the complete block. However, there may be some processors with no part of the block. The full block is then broadcast to these processors, using all available processors that have the block.

4.5: Founder symmetry lookups:

There is one final wrinkle to the communication routines above involving the founder-phase symmetry state space reduction. Founder symmetry occurs in *sibships*, or groups of siblings. Essentially, in a sibship of *s* siblings where one parent is a founder, one of the *s* bits in that sibship for the founder's chromosome can be eliminated. However, if some of the other *s-1* bits for the sibship are fixed, we must also consider the complementary assignment of the founder-phase. Thus, if there is one sibship with a fixed bit, for each inheritance pattern calculation, we will need to look up two indices: one with the original founder-phase assignment and one with the complementary assignment. This operation corresponds to "flipping" all of the variable bits in that sibship. When we distribute this calculation, we must check to see if any of these founder sibling "flips" intrude into the processor-order bits. When this occurs, it means that each processor will require another block in addition to the one it already

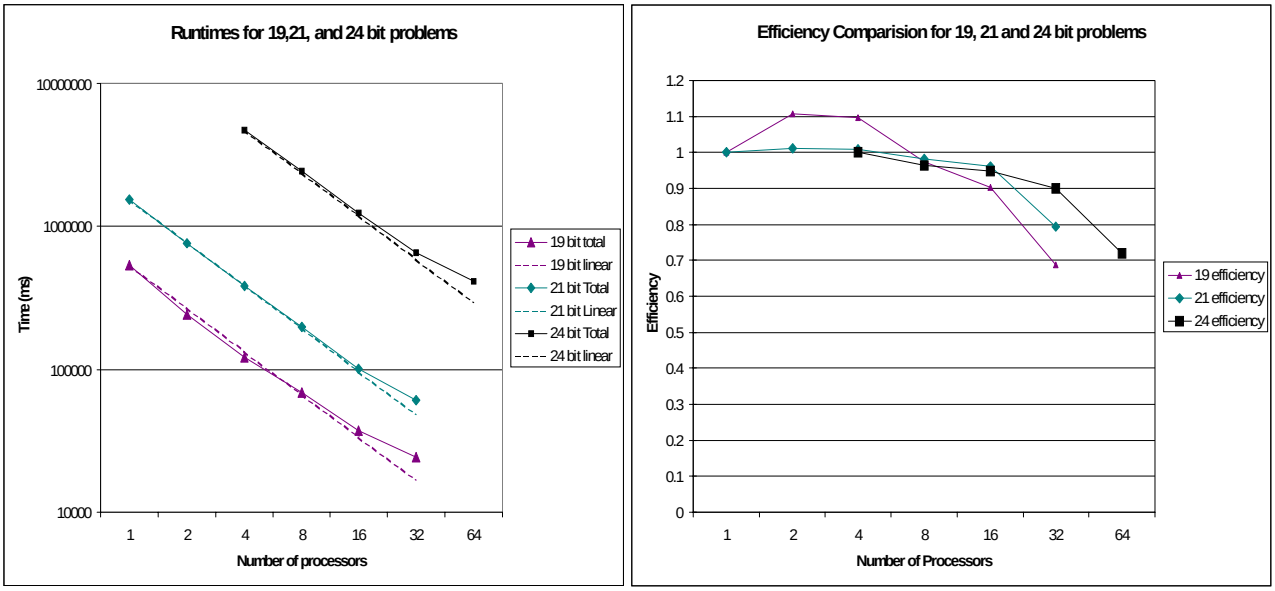

Figure 5: Example algorithm scaling for 19, 21 and 24 problems. A: Running times for different problem sizes by processor counts. (Linear scaling curves are shown for reference.) B: Efficiency for different problem sizes.

has. We handle this by looping over the above communication routines for each founder-symmetry case required. It is important to note that these flips may change the values of the fixed bits, but this is handled transparently by the above algorithm.

5. Performance:

We have analyzed the performance of our algorithm on Sandia National Laboratories' Cplant cluster. Cplant consists of several hundred DEC Alpha EV6 processors connected via Myrinet. We ran three problem sizes, ($2n$-f =)19 and 21 bits of one dataset and a 24 bit problem from second dataset. A 24-bit problem means that the largest vectors Genehunter operates on are of length 2^{24}. The 19 and 21 bit datasets are from a 10cM chromosome 1 genotype screen of a 51 member family with a genetic skin disease, vitiligo. Family members were genotyped using the Prism Linkage Mapping Set Version 2 (LMSv2-MD10) panel of microsatellite markers from Applied Biosystems. The 24 bit dataset is a 5cM chromosome 1 genotype screen from the same family, with genotype data from one additional family member. Figure 5 shows the overall runtime performance of these problems on different number of processors and the efficiency of each problem size on different numbers of processors. We note that the 24-bit problem, which runs in 11 minutes on 32 processors, would require roughly five hours to run on an equivalent single processor system. A line of perfect scaling for each problem size is included for reference in figure 5a, which would correspond to 100% efficiency in figure 5b. Although scaling eventually drops off in each case, there are ranges of processor counts which scale well for each problem size. This observation is of importance when we consider the possibility of scaling to larger problems and processor counts, since it implies that for most problems we can hope to find a range of processor numbers where efficient use of computational resources is made.

There are several factors limiting scalability. One is the disparate sizes of the different marker vectors. For our datasets, markers sizes may range from less than 2^{10} up to full size (2^{2n-f}). To avoid dividing a marker onto more processors than it has bits, we have set a limit L(=2^9 for figure 5), such that any markers with fewer than L bits are analyzed in serial. At some point, these serial markers make begin to have an impact on running time. Of course, although the communication routines described above are fairly efficient, they also impose an overhead cost that may limit scalability.

6. Future Directions:

Genehunter 2.1 as implemented has a built-in limit of $2n \leq 32$ bits, because it uses 32-bit integers as masks. We currently in the process of increasing this limit to $2n \leq 64$ by replacing the integer masks with the c datatype **long long int**. This modification is required because the current limit may impede on problems as small as $2n$-f

$=26$ bits, if $f>6$, not an unreasonable value. By increasing the limit, we will allow the analysis of larger problems on clusters and will be able to test the scalability of our algorithmic approach on larger numbers of processors.

7. Acknowledgments:

G. Conant is supported by the Department of Energy's Computational Sciences Graduate Fellowship and this project was part of his practicum program for this fellowship. Sandia National Laboratories received additional support for this project from the Department of Energy's Office of Biological and Environmental Research.

8. References:

1. Morgan, T. H. (1911) *Journal of Experimental Zoology* **11**, 365-413.
2. Sturtevant, A. H. (1913) *Journal of Experimental Zoology* **13**, 43-59.
3. Fisher, R. A. (1935) *Annals of Eugenics* **6**, 187-201.
4. Haldane, J. B. S. & Smith, C. A. B. (1947) *Annals of Eugenics* **14**, 10-31.
5. Morton, N. (1955) *American Journal of Human Genetics* **7**, 277-318.
6. Elston, R. C. & Stewart, J. (1971) *Human Heredity* **21**, 523-542.
7. Lander, E. S. & Green, P. (1987) *Proceedings of the National Academy of Sciences, U.S.A.* **84**, 2363-2367.
8. Dwarkadas, S., Schäffer, A. A., Cottingham, R. W., Cox, A. L., Keleher, P. & Zwaenpoel, W. (1994) *Human Heredity* **44**, 127-141.
9. Kruglyak, L., Daly, M. J., Reeve-Daly, M. P. & Lander, E. S. (1996) *American Journal of Human Genetics* **58,** 1347-1363.
10. Kruglyak, L. & Lander, E. S. (1998) *Journal of Computational Biology* **5**, 1-7.
11. Markianos, K., Daly, M. J. & Kruglyak, L. (2001) *American Journal of Human Genetics* **68**, 963-977.