Available at www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT•

J. Parallel Distrib. Comput. 63 (2003) 674–682

Journal of Parallel and Distributed Computing

http://www.elsevier.com/locate/jpdc

Parallel Genehunter: implementation of a linkage analysis package for distributed-memory architectures

Gavin C. Conant,[a,*] Steven J. Plimpton,[b] William Old,[c] Andreas Wagner,[a] Pamela R. Fain,[d] Theresa R. Pacheco,[d] and Grant Heffelfinger[b]

[a] *Department of Biology, 167 Castetter Hall, The University of New Mexico, Albuquerque, NM 87131-1091, USA*
[b] *Computation, Computers, and Mathematics Center, Sandia National Laboratories, Albuquerque, NM, USA*
[c] *Agilent Laboratories, Fort Collins, CO, USA*
[d] *Health Sciences Center, The University of Colorado, Fort Collins, CO, USA*

Received 4 December 2002; revised 15 April 2003; accepted 16 May 2003

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 run on distributed-memory platforms for the first time. Our preliminary benchmarks indicate reasonable scalability of the algorithm even for fixed-size problems, with parallel efficiencies of 75% or more on up to 128 processors. In addition, we have extended the hard-coded limit of 16 non-founding individuals in Genehunter 2.1 to a new limit of 32 non-founding individuals.
© 2003 Elsevier Inc. All rights reserved.

1. Introduction

Linkage analysis attempts to locate genes responsible for a genetic disease by performing computations on genetic data from a group of related individuals with a high incidence of that disease. Preliminary analysis can usually suggest a rough location for a disease gene(s) and limit the search to a portion of a single chromosome. Blood samples can then be taken from the (available) individuals in the extended family to determine their genetic profile (or genotype) at a number of *markers* within this region of the chromosome. Markers are identifiable locations on a chromosome (often microsatellite positions or single-nucleotide polymorphisms) where individual humans are known to show genetic differences. Because the DNA molecule is linear, these markers can be ordered into a list. Each individual has two copies of every marker, one from his or her father and one from his or her mother. Each copy can also be in two possible states, depending on whether the parent transmitted the marker from his or her mother or father (the grandparents of the individual).

In general, if a given marker was received (for instance) from the paternal grandfather, it is very likely that the next marker in the list was also received from the paternal grandfather. However, meiosis, the process of cell division which produces eggs and sperm, can create other inheritance patterns. During this process, the parent's two chromosomes can "cross-over," resulting in two new chromosomes which each contain a portion of the original two. Cross-over, also known as *recombination*, occurs when two DNA helices with similar sequences intertwine and base pair with each other. Under these circumstances, the phosphate backbone of one of the helices can break and join with the that of the other DNA molecule. Because recombination occurs between homologous regions of the original chromosomes, both new chromosomes have a complete set of markers, but the ancestry of these markers is mixed between the grandfather and grandmother of the offspring.

The closer two markers are on the chromosome the more likely it is that they will be inherited together, since the probability that a recombination event will separate them is low. Two markers that are widely separated on a chromosome will almost always have at least one recombination event occur between them. As a result,

*Corresponding author.
E-mail address: gconant@unm.edu (G.C. Conant).

two such markers will be inherited together only 50% of the time (i.e. when an even number of recombination events occurs between them); a pair inherited together more often than this is said to be *linked* [10,11]. Thus, the number of recombination events that have occurred between two markers is a distance measure which can be statistically correlated with the pattern of disease incidence in the extended family. This correlation gives insight into where on the chromosome the disease gene is located and is referred to as linkage analysis. The mathematical theory behind linkage analysis dates to work done by Fisher, Haldane and Smith, and Morton, who used maximum likelihood to infer genetic maps from imperfect data [3,4,9]. As genetic marker data become increasingly prolific and cheap to obtain, computational techniques like linkage analysis will become correspondingly more powerful.

Two algorithms are commonly used for the linkage analysis problem. The algorithm described by Elston and Stewart scales linearly with the number of individuals in the pedigree, but exponentially with the number of markers [2]. 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]. The Lander and Green algorithm is widely used for *multipoint* linkage analysis: i.e. problems where many markers are correlated simultaneously. Unfortunately, many researchers are interested in analyzing datasets which are too large for current implementations of the algorithm. As we discuss below, the code we have modified, Genehunter 2.1 [5,8], sets a hard-coded limit of 16 offspring individuals in an analysis. In addition, even problems smaller than this can require significant amounts of memory and CPU time on single-processor systems. For instance, a dataset we have analysed with 14 individuals requires 1 GB of memory and runs for 2.6 h on a 650 MHz Pentium III. When one realizes that adding even a single person to this analysis (15 individuals) quadruples both time and memory requirements, the scope of the problem becomes clear.

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 [1]. Here, we present a parallel version of the Lander and Green algorithm that has been implemented in the Genehunter program; to our knowledge it is the first distributed-memory implementation of the algorithm. Because the Lander and Green approach scales exponentially, very large pedigrees will always remain out of reach with this algorithm. However, we show that our parallel version of Genehunter extends the range of possible analyses up to 20–22 individuals. In addition, because of the flexibility of our message-passing implementation, this new version of Genehunter can be run on diverse hardware platforms, from shared-memory workstations to massively parallel supercomputers.

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. Fig. 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 (versions of a gene) 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 possibilities throughout the analysis, leaving this chromosome coded as 0/1.

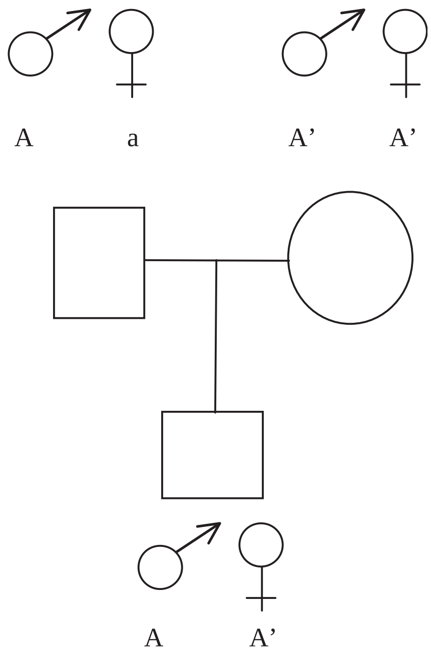

Fig. 1. An example pedigree used to illustrate the Lander and Green approach to representing inheritance patterns as binary strings. Alleles at this locus are represented by the letters A, a, and A′. Symbols above each allele indicate whether that allele was inherited from the mother or father of the individual in question.

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 referred to as *founder-phase symmetry*.

The Lander and Green algorithm works by representing each possible inheritance pattern for the pedigree as a string of $2n$ 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, picking a definition of maternal and paternal for each founder allows us to 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 [5].

One limitation of Genehunter 2.1 as written is that 32-bit integer data types are used as binary masks by the code. Thus (on standard platforms), the code has a hard limit of $2n \leqslant 32$, or 16 non-founding individuals, which is also roughly the limit of what is computationally feasible on modern single processor systems. Note that $2n$ and not $2n - f$ is limited, due to constraints in the manner in which inheritance patterns are stored. In order to make use of the increased computing power and memory available on distributed-memory machines, we have increased this limit to $2n \leqslant 64$ by replacing the 32-bit integers with a 64-bit integer type. Use of the 64-bit integer rather than a custom data type was necessary to maintain the structure of the code as written, but this new upper limit should be sufficient for any computationally feasible analysis.

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 j in m_1, define a distance $d(i,j)$ between pattern j and each possible pattern i in m_2. 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

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

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

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

where $\circ$ represents a component-wise vector product. (2) can be then applied iteratively to calculate any required conditional probability vector (i.e. using $P_{i|i-1}$ to compute $P_{i+1|i}$). This Markov-chain approach is an $O((2^{2n-f})^2)$ 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 $O(2^{2n-f} \cdot \log_2(2^{2n-f}))$ [6]. 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 $2^{2n-f} \times 2^{2n-f}$ need ever be stored in the linkage analysis computation.

2. Genehunter computation

The Genehunter 2.1 software [8] 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 non-parametric 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 with Eq. (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 observed genotype data and can be ignored. Recall that in the example in Fig. 1, 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 [8]). 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 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* (i.e. which individuals are affected with the disease). 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 (NPL) analysis which does not always require storing a vector of this size; however, the computation has an identical form, and we will not discuss it 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 computation and memory. We discuss the parallelization of each of the three steps described above separately. The I/O requirements for Genehunter are typically quite small (no more than a few hundred lines of text input and postscript output), 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 m_i, the vector of inheritance probabilities $P_{i|1..i-1}$ (probability of each inheritance pattern at m_i conditioned on markers m_1 through m_{i-1}) and the vector $P_{i|i+1..m}$ (inheritance pattern probabilities conditioned on markers m_{i+1} through m_m). This calculation is performed using FFTs in the conceptual manner of Eq. (2).

For the moment, assume that each processor has all the elements of the conditional probability vector at m_{i-1} needed to compute the conditional probabilities at m_i. (Since the vectors at m_{i-1} and m_i 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 $2^{2n-f-k} \times 2^{2n-f-k}$.

In Genehunter, the matrix–vector multiply is replaced by an FFT-based convolution, with 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. 1d FFTs of length N/P 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 1d FFTs of length P on data that are 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 m_i and the corresponding element in this new conditional probability vector, yielding a vector of the size of the original vector at m_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

$$p = P_D \cdot P_{x|1..i} \cdot P_{x|i+1..m}, \tag{3}$$

where P_D is the disease vector and $P_{x|1..i}$ and $P_{x|i+1..m}$ are calculated in the manner of Eq. (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.

The above description must be modified somewhat for certain values of x. In general, x is at some point between two markers. However, x can also be positioned at a single marker. This difference does not affect the overall form of the computation shown in (3), but, because the conditional probabilities at each marker are already known, the on-marker case does not require an FFT to calculate them.

4.4. Redistribution of marker vectors for computation 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, redefining the mapping between inheritance probability vectors. Fixed bits are represented using bit masks of length $2n-f$ with 1's at positions where the fixed bits occur. Fig. 2 gives a possible configuration of the fixed-bit masks for two adjacent markers m_1 and m_2. Note that the value at which each of these bits is fixed is also required and is shown in Fig. 2.

Only non-fixed bits are stored in inheritance probability vectors. Thus, in Fig. 2, marker m_1 would have a size of 2^6 and marker m_2, 2^7. Suppose we have an inheritance pattern i_1 at marker m_1 for which we would like to know what inheritance pattern i_2 that it corresponds to in the variable bit space of marker m_2. Starting with $i_2 = i_1$, we first note that any fixed bits k_s that are common to both markers (such as the last bit in Fig. 2) are already absent from i_1 and can be ignored from this point on. We next define two new operations for the mapping of indices between markers: (1) If m_1

Marker	Mask • 1=Fixed • 0=Variable	Fixed bit values	Marker size	Division of variable bits on 8 processors	Indices per processor
m_1	$00\boxed{11}00\boxed{1}001$	**10**1**0	2^6	000 | 000	2^3
m_2	$\underline{1}0000\underline{1}0001$	1****0***1	2^7	000 | 0000	2^4

Using the above, we can map an index from m_1 (i_1) to m_2 (i_2)

$$i_2=i_1=010110 \rightarrow 01\boxed{10}011\boxed{1}0 \rightarrow \underline{0}1100\underline{1}110 \rightarrow 1100110 = i_2$$

Insert 3 fixed bits   Remove 2 fixed bits

Fig. 2. Examples of fixed-bit masks for two markers, m_1 and m_2. Masks for the location of fixed-bits are shown, with the corresponding fixed-bit values in the next column. Boxed and underlined fixed bits indicate cases where that bit is fixed only on the source or target, respectively. The lower half of the figure shows the mapping of an index in m_1 representation into m_2 representation.

has k_1 fixed bits not present in m_2, (boxed in Fig. 2) we look up the value of those fixed bits and insert them in the appropriate locations of i_2. Thus, index i_2 now has length $2n-f-k_s+k_1$. (2) If m_2 has fixed bits not present in m_1 (shown underlined in Fig. 2), we drop those k_2 bits from i_2. The final size of i_2 is, therefore, $2n-f-k_s+k_1-k_2$.

Consider converting the example index $i_1 =$ `010110` at marker m_1 to i_2 at m_2. The fixed bit in the 1's position has already been removed from i_2, and can be ignored. First, we insert three fixed bits into i_2 at the locations specified in the mask. This gives us `011001110` (Fig. 2 shows the inserted bits boxed). We now drop the two fixed bits present at m_2 but not m_1 (shown underlined in Fig. 2). The result is $i_2 =$ `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. If we assume we have 2^p processors for some integer p, 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 p bits. This operation is shown in Fig. 2 for an eight-processor (3-bit) distribution. For our example index above, we see that $i_1 =$ `010|110`, meaning that i_1 is located on processor `010` (2). However, we find that $i_2 =$ `110|0110`, meaning i_2 is located on processor `110` (6). 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 source mask represented in the target configuration. Fig. 3 gives examples for the masks in Fig. 2.

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

	Mask in target representation
Marker m_1:	0110100
Fixed bit values:	*10*1**

	Mask in source representation
Marker m_2:	100100
Fixed bit values:	1**0**

Fig. 3. Example showing the masks in Fig. 2 in each other's representation (see text).

First, there may be fixed processor-order bits in the target not in the source. This implies that processors whose ranks do not match the processor-order values of those fixed bits will be idled. For instance, in Fig. 3, only processors with ranks of the form `1**` will contribute source probabilities to the computation. The communication algorithm based on this observation is straightforward: the subset of vectors contained on non-idled processors is evenly redistributed on all processors using a scatter operation. For the example in Fig. 3, processors `100` through `111` will send their data to processors `000−001`, `010−011`, `100−101`, and `110−111`, respectively.

The second case is when there are processor-order fixed bits in the source not in the target. 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 Fig. 3, processors are grouped into two equivalence classes based on the value of their rank's highest order bit position (thus, one equivalence class is `000−011` and the other is `100−111`). We refer to these equivalence classes as "blocks". Each member of a block will need the complete set of probabilities common to all members of the block.

Using the current data layout and mask values, we invoke a unit-time exchange operation where processors put their current data onto a processor that will need it. Once this exchange has taken place, the data for each block are evenly distributed among the block members and can be collected using a logarithmic time gather operation within each block.

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 this step, there are actually two communication schemes required, depending on whether x is positioned on a marker or between two markers:

- For the general case when x is between two markers, we use the above scheme: a unit-time point-to-point exchange followed by a gather that takes place within each block and results in each processor obtaining a complete copy of that block.
- If x is on a marker the situation is simpler. For this on-marker calculation, any inheritance pattern that

does not have appropriate fixed-bit values for the marker can be ignored. Thus, we need only consider a subset of size 2^{2n-f-k} of the disease vector, which is distributed according to the fixed-bit mask for that marker. If no fixed bits intrude into the processor-order bits, communication is unnecessary. Otherwise, any processor with a rank that does not have the correct value of those fixed bits is idled. The remaining non-idle processors R each require $2^{2n-f-k}/R$ of the marker vectors. This distribution can be easily accomplished with a gather operation, where the marker vectors on P/R processors are gathered onto the single processor that requires them.

Unfortunately, since some processors are idled, this on-marker communication scheme introduces load imbalance. However, the on-marker computation is fast relative to that used between markers, and the communication required to evenly distribute data for the on-marker calculation would be prohibitively expensive.

The parallel complexity of step 3 depends on several parameters. The total size of a marker vector is $N = 2^{2n-f-k}$, of which N/P is stored on each processor. For the parallel FFT (Section 4.2), the serial runtime is $O(N \cdot \log N)$, which in parallel is decreased to $O(N/P \cdot \log N)$. Our implementation of the parallel FFT requires each processor to exchange half its data with another processor $\log(P)$ times, incurring a bandwidth cost of $O(\log P \cdot N/P)$ and a latency cost of $O(\log P)$. The second part of the parallel complexity of step 3 is the communication routines required to assemble the marker vectors for the computation of the likelihood value (see Section 4.3). This communication step is of course unnecessary in the serial code, but the trade-off is that the serial computation of the likelihood requires $O(2^{2n-f})$ work. In parallel, this computation is reduced to $O(2^{2n-f}/P)$ work, but requires a communication overhead. The size of this overhead depends on k': the number of processor-order fixed bits for the marker (in Fig. 2, k' is 1 for marker m_1). The required communication is essentially a series of gathers over groups of $2^{k'}$ processors and thus requires $O(2^{k'} \cdot N/P)$ bandwidth with a latency of $O(k')$. These values hold both for the between- and on-marker calculation.

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 a much larger fraction of the necessary data. Thus, in the first scheme above, instead of every processor sending its current data to another processor, only the non-idled processors send data. Once this exchange is finished, any further collect or broadcast operations needed to complete the communication are performed.

The parallel complexity of step 2 is rather more elaborate than that of step 3, as it depends on both $k_1{}'$ and $k_2{}'$ (the processor-order fixed bits at the current marker and the next marker, respectively). Recall that the marker data are of size $N = 2^{2n-f-k_1}$. In addition to the complexity of the parallel FFT already discussed, there are two situations to consider. First, if $k_2{}' > k_1{}'$, there is point-to-point communication requiring $O(N/(P \cdot 2^{k_2-k_2{}'}))$ bandwidth and $O(2^{k_2{}'-k_1{}'})$ latency. This communication is followed by a broadcast requiring $O((k_1{}' \cdot 2^{k_1{}'} \cdot N)/(P \cdot 2^{k_2}))$ bandwidth and $O(k_1{}')$ latency. If, on the other hand, $k_1{}' \geqslant k_2{}'$, then there are gather operations requiring $O((2^{k_1{}'} \cdot N)/(P \cdot 2^{k_2}))$ bandwidth and $O(k_1{}' - k_2{}')$ latency. This is followed by a broadcast requiring $O((k_2{}' \cdot 2^{k_1{}'} \cdot N)/(P \cdot 2^{k_2}))$ bandwidth and $O(k_2{}')$ latency. The extra $O(k_1{}')$ or $O(k_2{}')$ bandwidth requirements in broadcast operations are inefficiencies required by these special data layouts.

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 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.

4.6. Entropy calculation

Each inheritance pattern has an associated NPL score. However, these scores are not in general unique (meaning that several inheritance patterns may yield the same NPL score). Genehunter maintains a list of all unique NPL score values and uses this list to calculate positional entropies during the computation. The list of unique NPL scores is implemented in Genehunter as an array which must be linearly searched 2^{2n-f} times at every position in step 3. To improve performance, we have replaced the array with a standard binary search tree (BST), which offers reasonable performance improvements even in the serial code (data not shown).

In the parallel code, each processor maintains a BST of all the unique NPL scores stored on that processor. The proportion of the total probability accounted for at each position by each of these scores is also stored locally. However, to actually compute the entropy, we must make sure that each unique NPL score is represented exactly once over all processors. The difficulty is that the list of unique NPL scores could potentially be too long to store on a single processor. We have therefore implemented a progressive reduction strategy. It starts by broadcasting the first processor's list of unique scores to all other processors. Any other processor that also has one of these scores removes it from its BST and returns the associated probabilities to processor 0. This process is repeated for each processor, broadcasting scores only to those processors with higher rank. The result is that each score is represented exactly once, although the progressive nature of the operation means that, on average, lower ranked processors have more scores to handle. This load imbalance is not generally large (unpublished data).

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. Two different types of performance analysis were carried out. First, we used the new 64-bit version of the code to run a problem of size $2n = 42$, which is a thousand times larger than was possible with the original 32-bit version of Genehunter, even assuming unlimited time and memory were available. (The dataset used for this example was a scaled-up version of the chromosome 6 data discussed below.) On 128 processors, this problem runs in 5 h, requires approximately 34 GB of memory, and demonstrates that our code has extended the limits of possible analyses with Genehunter.

We next analyzed the parallel performance of our code. We ran two different datasets: a 10 cM chromosome 1 genotype screen of a 51-member family with a genetic skin disease, vitiligo (genotypying done using the Prism Linkage Mapping Set Version 2 (LMSv2-MD10) panel of microsatellite markers from Applied Biosystems) (analysis size was $2n - f = 21$ bits) and a 5 cM chromosome 6 genotype screen of a 29-member family having 13 members affected with multiple lentigenes syndrome, an autosomal dominant disorder (genotypying done using the ABI Linkage Mapping Set version 2.5-MD10, Applied Biosystems, Foster City, CA and ABI Linkage Mapping Set version 2.5-MD5). For this second dataset, we ran two problem sizes: a $(2n - f = 19)$-bit problem and a $(2n - f = 23)$-bit problem. The 23-bit problem has $2n = 34$, meaning that it could not be run with the original version of Genehunter. Fig. 4 shows the overall runtime performance of these problems on different numbers of processors and the corresponding efficiency of each problem for these processor counts. A line of perfect scaling for each problem size is included for reference in Fig. 4a, which would correspond to 100% efficiency in Fig. 4b.

As mentioned above, pedigree analyses are often limited by available memory rather than CPU time. We, therefore, also show how the memory usage in the largest problem size in Fig. 4 scales with processor counts (Fig. 5). Although memory scaling is imperfect as high processor counts are reached, the algorithm nonetheless shows reasonable scalability in memory.

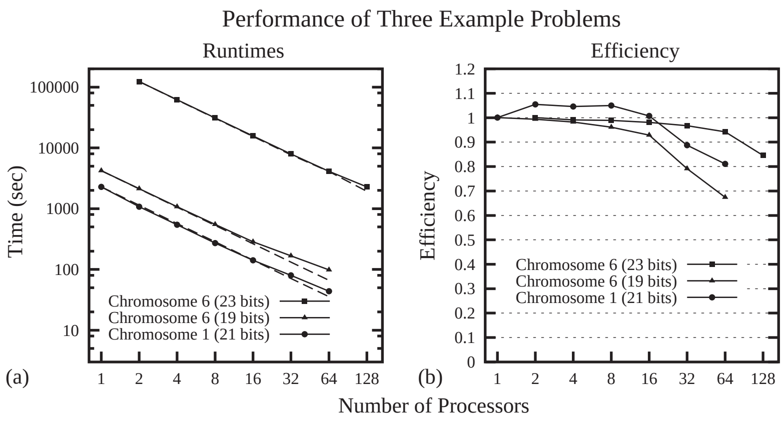

Fig. 4. Algorithm scaling for three problems, one on chromosome 1 and two on chromosome 6 (see text for details). (A) Running times for different problem sizes by processor counts. (Linear scaling curves are shown for reference.) (B) Efficiency for different processor counts.

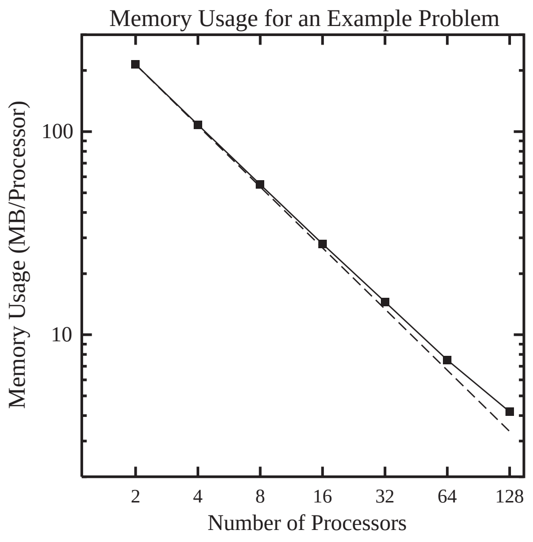

Fig. 5. Memory scaling for the largest problem analyzed in Fig. 4 (23 bits, chromosome 6). A linear scaling curve is shown for reference.

Fig. 4 raises the question of where scaling is falling off in the algorithm. For the chromosome 6, 19-bit dataset (the largest for which we have single-processor timings) fully 60% of the difference between the ideal and actual runtimes is accounted for in the entropy reduction step (see Section 4.6), a communication routine not required by the serial code. Five percent of the remaining difference appears to be due to communication costs in the parallel FFT (communication and computation are closely interleaved in this routine, preventing us from explicitly timing the FFT communication). Remaining parallel costs include the step 2 and 3 marker reorganizations ($<1\%$), start-up and I/O costs (11%) and re-computation in step 1 (8%). There are also other factors which may limit 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 $L(=2^8$ for the 19- and 21-bit datasets and $=2^{11}$ for the 23 bits dataset in Fig. 4), such that any markers with fewer than L bits are analyzed in serial. At some point, these serial markers may begin to have an impact on running time.

The above problems are good tests of our parallel algorithm because they have different runtime features. In particular, the chromosome 1 dataset is memory-limited, which is likely why super-linear speedup is seen between two and 16 processors. The chromosome 6 dataset is not memory limited, and thus its scaling falls off more rapidly for the 19-bit analysis as runtimes become short (the 64-processor running time for the 19-bit dataset was 98 s). Because the chromosome 6 dataset has a large number of fixed bits which reduce its memory requirements, steps 2 and 3 also require more communication and hence do not scale as well as the chromosome 1 dataset. Clearly, however, the larger version of the chromosome 6 problem (23 bits) scales quite well (single-processor times for this problem are not available due to memory constraints). Although scaling does drop off in each of these cases, it is encouraging that our largest problem scales well even on 64 and 128 processors, suggesting that this code should extend to very large problems in a reasonable way.

Acknowledgments

We thank Bill Hart and Bruce Hendrickson of Sandia National Laboratories for many helpful insights during the development of this project. GCC is supported by the Department of Energy's Computational Sciences Graduate Fellowship (administered by the Krell Institute), and this project was part of his practicum program for this fellowship. Sandia received additional support for this project from the Department of Energy's Office of Biological and Environmental Research.

References

[1] S. Dwarkadas, A.A. Schffer, R.W. Cottingham, A.L. Cox, P. Keleher, W. Zwaenpoel, Parallelization of general-linkage analysis problems, Hum. Hered. 44 (1994) 127–141.

[2] R.C. Elston, J. Stewart, A general model for the genetic analysis of pedigree data, Hum. Hered. 21 (1971) 523–542.

[3] R.A. Fisher, The detection of linkage with ''dominant'' abnormalities, Ann. Eugenics 6 (1935) 187–201.

[4] J.B.S. Haldane, C.A.B. Smith, A new estimate in the linkage between the genes for color-blindness and haemophilia in man, Ann. Eugenics 14 (1947) 10–31.

[5] L. Kruglyak, M.J. Daly, M.P. Reeve-Daly, E.S. Lander, Parametric and nonparametric linkage analysis: a unified multipoint approach, Amer. J. Hum. Genet. 58 (1996) 1347–1363.

[6] L. Kruglyak, E.S. Lander, Faster multipoint linkage analysis using fourier transforms, J. Comput. Biol. 5 (1998) 1–7.

[7] E.S. Lander, P. Green, Construction of multilocus genetic linkage maps in humans, Proc. Nat. Acad. Sci. USA 84 (1987) 2363–2367.

[8] K. Markianos, M.J. Daly, L. Kruglyak, Efficient multipoint linkage analysis through reduction of inheritance space, Amer. J. Hum. Genet. 68 (2001) 963–977.

[9] N. Morton, Sequential tests for the detection of linkage, Amer. J. Hum. Genet. 7 (1955) 277–318.

[10] T.H. Morgan, An attempt to analyze the constitution of the chromosomes on the basis of sex-limited inheritance in *Drosophila*, J. Exp. Zoo. 11 (1911) 365–413.

[11] A.H. Sturtevant, The linear arrangement of six sex-linked factors in *Drosophila*, as shown by their mode of association, J. Exp. Zoo. 13 (1913) 43–59.