BIOINFORMATICS ORIGINAL PAPER

Vol. 29 no. 4 2013, pages 413–419
doi:10.1093/bioinformatics/bts704

Genome analysis

Advance Access publication December 13, 2012

SwiftLink: parallel MCMC linkage analysis using multicore CPU and GPU

Alan Medlar[1,2,*], Dorota Głowacka[3], Horia Stanescu[1], Kevin Bryson[4] and Robert Kleta[1]

[1]Division of Medicine, University College London, London WC1E 6BT, UK, [2]Institute of Biotechnology, University of Helsinki, Helsinki 00014, Finland, [3]Helsinki Institute for Information Technology, Department of Computer Science, University of Helsinki, Helsinki 00014, Finland and [4]Department of Computer Science, University College London, London WC1E 6BT, UK

Associate Editor: Martin Bishop

ABSTRACT

Motivation: Linkage analysis remains an important tool in elucidating the genetic component of disease and has become even more important with the advent of whole exome sequencing, enabling the user to focus on only those genomic regions co-segregating with Mendelian traits. Unfortunately, methods to perform multipoint linkage analysis scale poorly with either the number of markers or with the size of the pedigree. Large pedigrees with many markers can only be evaluated with Markov chain Monte Carlo (MCMC) methods that are slow to converge and, as no attempts have been made to exploit parallelism, massively underuse available processing power. Here, we describe SWIFTLINK, a novel application that performs MCMC linkage analysis by spreading the computational burden between multiple processor cores and a graphics processing unit (GPU) simultaneously. SWIFTLINK was designed around the concept of explicitly matching the characteristics of an algorithm with the underlying computer architecture to maximize performance.

Results: We implement our approach using existing Gibbs samplers redesigned for parallel hardware. We applied SWIFTLINK to a real-world dataset, performing parametric multipoint linkage analysis on a highly consanguineous pedigree with EAST syndrome, containing 28 members, where a subset of individuals were genotyped with single nucleotide polymorphisms (SNPs). In our experiments with a four core CPU and GPU, SWIFTLINK achieves a 8.5× speed-up over the single-threaded version and a 109× speed-up over the popular linkage analysis program SIMWALK.

Availability: SWIFTLINK is available at https://github.com/ajm/swiftlink. All source code is licensed under GPLv3.

Contact: alan.j.medlar@helsinki.fi

Received on October 24, 2012; revised on December 1, 2012; accepted on December 4, 2012

1 INTRODUCTION

Exact linkage analysis algorithms scale exponentially with the size of the input. Beyond a critical point, the amount of work that needs to be done exceeds both available time and memory.

The Elston–Stewart algorithm (Elston and Stewart, 1971) calculates successive conditional likelihoods between family members, permitting the algorithm to scale linearly with the size of the pedigree (more precisely, the number of meioses), but can only analyse a handful of markers jointly.

The Lander–Green hidden Markov model (HMM) approach (Kruglyak *et al.*, 1995, 1996; Lander and Green, 1987), on the other hand, can analyse many markers, but only in pedigrees of limited size.

In between these two extremes, approaches using Bayesian networks, for example SUPERLINK (Fishelson and Geiger, 2002, 2004), scale to greater numbers of markers than the Elston–Stewart algorithm and to larger pedigrees than the Lander–Green algorithm, but still cannot handle both large pedigrees and many markers.

In the situation where we have a large, perhaps consanguineous, pedigree typed with many markers, we are forced to either abbreviate the input or else use an approximation like Markov chain Monte Carlo (MCMC).

Although MCMC helps make the problem tractable, it can take a long time to converge. The problem of slow running time is compounded by the fact that existing software is single-threaded and, therefore, not designed to take advantage of all the processing power available in even a single modern PC.

This underuse will become more acute as multicore processors feature increasing numbers of cores and graphics processing units (GPUs) become more general purpose.

Parallelism has previously been applied to accelerate exact linkage algorithms in the applications FASTLINK, across networks of workstations (Dwarkadas *et al.*, 1994; Gupta *et al.*, 1995), PARALLEL-GENEHUNTER, on computer clusters (Conant *et al.*, 2002) and SUPERLINK-ONLINE, on grid middleware (Silberstein *et al.*, 2006).

Although parallel implementations of exact linkage algorithms perform analyses faster and can scale to larger problems owing to accessing memory in multiple machines, they still have the same fundamental scaling properties as their underlying algorithms.

Problems outside of the scope of exact algorithms must be analysed with MCMC-based linkage programs such as

*To whom correspondence should be addressed.

SIMWALK2 (Sobel and Lange, 1996; Sobel *et al.*, 2001) and MORGAN (George *et al.*, 2003, 2005; Heath *et al.*, 1997), which have both been shown, where possible, to produce results of comparable accuracy to exact algorithms (Wijsman *et al.*, 2006).

To the best of our knowledge, there have been no published attempts to parallelize MCMC-based linkage analysis.

In general, parallelizing MCMC is considered a hard problem because the states of the Markov chain form a strict sequential ordering. Successful parallel MCMC implementations tend to focus either on parallelizing expensive likelihood calculations (for example, Suchard and Rambaut, 2009) or on multiple chain schemes (for example, Lee *et al.*, 2009).

The approach we took with SWIFTLINK maximizes the use of hardware resources by combining both previous approaches, allowing different parts of the calculation to be distributed across both CPU and GPU.

The Markov chain is run in parallel on the CPU, where each step in the chain is performed by one of two multithreaded block Gibbs samplers based on the locus sampler (Heath, 1997; Kong, 1991) and the meiosis sampler (Thompson and Heath, 1999). Expensive likelihood calculations required to estimate LOD (logarithm of odds) scores are performed on the GPU, if one is available. High performance is achieved on the GPU by maximizing hardware use, requiring us to identify substantial independent calculations.

This article is organized as follows: we elaborate on the details of the Gibbs samplers used to form the Markov chain, what aspects of each sampler were changed to facilitate execution on parallel hardware and how SWIFTLINK orchestrates these actions to maximize hardware use. We report experimental results comparing SWIFTLINK with other MCMC-based linkage analysis software on a highly consanguineous pedigree and end with discussion and an outline of future work.

2 METHODS

SWIFTLINK was designed to explicitly harness not only multiple processor cores, but also a GPU if one is present. We describe three core components for performing MCMC in the context of linkage analysis: LOD score calculation, the locus sampler and the meiosis sampler. For each of these modules, we highlight where independent calculations can be used to exploit parallelism and which hardware platform would be best suited to the algorithm in question. In addition, the structure of the application is such that we can safely use both CPU and GPU concurrently.

2.1 Overview

To simulate the inheritance of genetic material through a pedigree, the Markov chain operates on a *descent graph*, S. The descent graph is made up of two meiosis indicators per non-founder of the pedigree, per loci. A meiosis indicator is a binary variable stating which grandparent that allele was inherited from (Kruglyak *et al.*, 1996; Sobel and Lange, 1996). We denote a particular meiosis indicator $S_{i,j}$, as meiosis i at locus j. An example of a descent graph for a simple pedigree is shown in Figure 1.

The Markov chain is composed of two Gibbs samplers, the locus sampler and the meiosis sampler. At each step of the Markov chain, we run the locus sampler with probability $P(\textit{locus sampler})$, otherwise we run the meiosis sampler. A single iteration is made up of a complete scan of all markers, in the case of the locus sampler; or all meioses, in the case of the meiosis sampler. The locus sampler is computationally more expensive than the meiosis sampler, but necessary as the meiosis sampler suffers from irreducibility issues on its own (Lin and Speed, 1996). Although the locus sampler is known to have poor mixing, it is guaranteed to produce an irreducible Markov chain, given all recombination fractions between consecutive markers are positive. By combining the two samplers, we can therefore ameliorate their respective shortcomings (Heath and Thompson, 1997). The Markov chain is thinned by only scoring every n^{th} sampled descent graph, and LOD scores are calculated at k equidistant points between consecutive markers. Dependent on the amount of chain burn-in and the values of k and n, LOD score calculations can easily take up the majority of the total runtime.

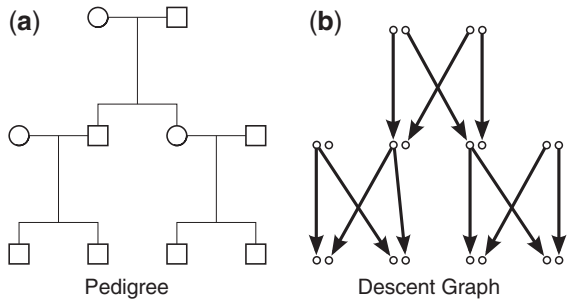

Fig. 1. Simple three-generation pedigree (**a**) and one of the possible descent graphs explicitly showing the flow of genetic material from founders to leaf nodes (**b**)

2.2 CPU and GPU architecture

A modern commodity multicore CPU will feature, at the time of writing, up to eight homogeneous processor cores with some architecture-specific level of cache sharing. For example, it is common to have a core-local L1 cache, as well as shared L2 cache. Individual processor cores are capable of instruction-level parallelism by the explicit use of SIMD (single-instruction multiple data) instructions by the programmer or compiler. Course-grain parallelism using multiple cores concurrently must be performed programmatically by writing multithreaded code.

There are two dominant APIs for GPU programming: CUDA and OpenCL. SWIFTLINK is implemented in CUDA, so throughout we will use CUDA terminology. GPU architectures feature several multiprocessors, each containing a number of stream processors. A function that runs on the GPU is called a *kernel*. Kernels split work into *blocks* and *threads*. Blocks are assigned to run on multiprocessors, each of which is composed of a programmer-defined number of threads. The number of threads contained in a block need not be identical to the number of stream processors inside a multiprocessor, as they are run in groups of 32 threads called *warps*. Stream processors are akin to CPU cores; however, within the same multiprocessor, they share a single fetch-decode unit. The shared fetch-decode unit forces threads to execute in lock-step, similar to SIMD instructions, but with the distinction that any thread can diverge from the common execution path (with the caveat that *thread divergent* code will cause all other threads within the same warp to stall). Analogous to the L2 cache of a CPU, GPUs feature *shared memory*; however, it must be explicitly populated and managed in software.

Algorithm 1: GPU-based parallel computation of $P(\mathbf{Y}_T|\hat{S}_i)$ used in LOD score calculation

```
begin
    blockDim ← num. threads per block
    blockID ← block index (0 ... num. loci - 1)
    threadID ← thread index (0 ... blockDim - 1)
    N ← number of pedigree members
    P ← pedigree peeling sequence
    prefetch relevant genetic map data
    for p ∈ P do
        prefetch relevant descent graph, pedigree info and previous partial
        likelihood function dependencies into shared memory
        p_shared ← allocate partial likelihood matrix for p in shared memory
        for i ← threadID to 4^|C_i|, increment blockDim do
            p_shared[i] ← H_p(C_p[i])
        end
        flush shared cache to main mem.
        p_main ← p_shared
    end
    LOD summation += P_N
end
```

2.3 LOD scores

Samples of possible descent graphs are generated by the Markov chain for which we calculate the likelihood of the trait locus being at different locations based on observed genotype data, $\mathbf{Y}_M$, and trait data, $\mathbf{Y}_T$. The LOD score at a position d is the likelihood ratio between the disease trait being found at d versus being unlinked. LOD scores can be approximated using the Sobel–Lange estimator (Lange and Sobel, 1991), where the LOD score is defined as

$$LOD = log_{10}\left(\frac{P(\mathbf{Y}_T|\mathbf{Y}_M)}{P(\mathbf{Y}_T)}\right) \tag{1}$$

where

$$\begin{aligned} P(\mathbf{Y}_T|\mathbf{Y}_M) &= \sum_{\hat{S}} P(\mathbf{Y}_T|\hat{S})P(\hat{S}|\mathbf{Y}_M) \\ &\propto \tfrac{1}{n}\sum_{i=1}^{n} P(\mathbf{Y}_T|\widehat{S}_i) \end{aligned} \tag{2}$$

and $P(\mathbf{Y}_T)$ is the probability that the trait locus is unlinked. $P(\mathbf{Y}_T|\widehat{S}_i)$ is calculated via an iterative summation using the Elston–Stewart algorithm (Cannings *et al.*, 1976; Elston and Stewart, 1971) to traverse the pedigree, *peeling* individuals in sequence. The process of peeling an individual involves calculating partial likelihoods conditional on the members of their *cut-set* and the meiosis indicators at flanking loci. The cut-set C is defined as containing all unpeeled individuals adjacent to the graph cut partitioning peeled from unpeeled individuals (C_i, the cut-set of the i^{th} individual in the peeling sequence, will therefore always include i, as it is, as yet, unpeeled). Each iteration of the peeling algorithm is based on the following recursion:

$$H_i(C_i) = \sum_{g_i} P(Y_i|g_i)P(g_i)P(g_i|\hat{S}, g_m, g_p)\prod_{i\in C_j} H_j(C_i) \tag{3}$$

where $H_i(C_i)$ peels the i^{th} individual given each possible assignment of disease phenotypes to members of i's cut-set $C_i = (c_0, c_1, \ldots c_C)$. For outbred pedigrees, the optimal cut-set size will never exceed 3; however, for inbred pedigrees, even the optimal ordering of variables can contain much larger cut-sets. The amount of work required to peel an individual increases exponentially with cut-set size as the number of disease phenotype assignments required to calculate each partial likelihood is $4^{|C|}$.

2.3.1 Parallel computation The calculations of LOD scores at different positions are independent of one another and only dependent on the underlying descent graph being scored. This workload is trivial to parallelize by dividing the total number of LOD scores between all CPU cores; however, this naïve strategy will not be suitable for the GPU. In our approach on the GPU, outlined in Algorithm 1, we assign each LOD score to a block and additionally parallelize the internal sum-of-products calculations from each step of the peeling sequence by assigning a GPU thread to calculate the likelihood of each disease phenotype assignment.

Peeling a pedigree of p members requires p separate peeling operations with different sized cut-sets, requiring different numbers of threads to perform optimally. Unfortunately, making a separate call to the GPU per peel operation incurs substantial overhead (in our measurements as much as twice the runtime to calculate all LOD scores for one sample). It is more efficient to use a constant number of threads despite the fact many threads will sit idle at different stages of the computation. To calculate the data-dependent optimal number of threads prior to each simulation, `SWIFTLINK` performs several timing runs with different numbers of threads per block to ensure the best performance. We found that empirically discovering the optimal number of threads was far more robust than an analytical approach, as, despite using the same input data and graphics card, the optimum number of threads was also dependent on which version of CUDA was being used.

GPU applications are sensitive to memory access patterns, and maximizing performance requires caching via shared memory and ensuring the GPU can batch ('coalesce' in the CUDA literature) time-consuming reads and writes to global memory. High memory latency is balanced by ensuring each kernel has a high occupancy, i.e. each multiprocessor is assigned as many blocks as possible, so work can be done by other warps when I/O operations are being waited on. Given the strong dependencies between matrices for intermediate calculations, we make extensive use of shared memory by opportunistically caching as many of the intermediate matrices needed for each iteration of peeling up to a total size of 4 kilobytes. GPUs supporting CUDA compute capacity 2.0 or above have 48 kilobytes shared memory per multiprocessor, which, as there can only be eight active blocks at once, is within this bound. Any memory requirements above the 4 kilobyte limit default to being directly accessed from main memory, however, that was unnecessary for any of the example pedigrees we have tested so far. We ensure the memory layout is such that a majority of reads and writes to main memory are made for contiguous addresses when populating software caches in shared memory.

2.4 Locus sampler

The locus sampler (Heath, 1997; Kong, 1991) similarly builds on the Elston–Stewart algorithm, but unlike LOD scores, which are calculated at points between markers, the locus sampler samples a realization of the meiosis indicators at a given genetic marker:

$$P(S_{*,j}|S_{*,j-1}, S_{*,j+1}, Y_l) \tag{4}$$

Sampling proceeds in two steps: (i) we use *reverse peeling* to sample ordered (or phased) genotypes and (ii) ordered genotypes are converted to appropriate meiosis indicators. To sample ordered genotypes, we first sample from the marginal distribution of the final individual in the peeling sequence and work backwards through the peeling sequence conditioning on those genotypes already sampled. Conversion to meiosis indicators is unambiguous for heterozygous parents, but the meiosis indicators to homozygous parents must be sampled with respect to the flanking loci in the descent graph. The locus sampler, like the calculation of LOD scores, can be expensive for large inbred pedigrees and reverse peeling must be performed sequentially. Each iteration of the locus

sampler involves sampling a new descent graph at all loci in a pseudo-random order.

2.4.1 Parallel computation To parallelize the locus samplers execution, we maintain a queue containing all loci that are shuffled into a pseudo-random order. The queue is consumed by a pool of worker threads equal to the number of CPU cores, with the special consideration that loci adjacent to those currently being sampled by other threads are avoided. Multithreading is performed using OpenMP.

We do not use the GPU to parallelize the locus sampler because sampling loci in a pseudo-random order while achieving high occupancy would be difficult without affecting the mixing properties of the Markov chain. Additionally, the sampling phase is relatively long and inherently sequential, which would be better run on the CPU that can run sequential code faster than a single thread on a GPU.

2.5 Meiosis sampler

The meiosis sampler (Thompson and Heath, 1999) is somewhat analogous to the Lander–Green algorithm (Lander and Green, 1987), but whereas the latter performs the forward–backward algorithm on the complete hidden Markov model (HMM), the meiosis sampler only does so in a limited fashion sampling a single meiosis on one individual in the pedigree conditional on all other meioses. Formally, we are aiming to compute:

$$P(S_{i,*}|\{S_{k,*}, k \neq i\}, \mathbf{Y}) \tag{5}$$

The forward component of the algorithm involves computing the cumulative probability for the meiosis indicator $S_{i,l}$ given loci up to and including locus l. The first locus does not depend on any previous loci:

$$Q_1(S_{i,1}) \propto P(Y_1 \mid S_{*,1}) \tag{6}$$

and at all proceeding loci:

$$Q_l(S_{i,l}) \propto P(Y_l \mid S_{*,l}) P(S_{i,l} \mid S_{i,l-1}) \tag{7}$$

The probability of recombination $P(S_{i,l}|S_{i,l-1})$ is defined as:

$$Q_{l-1}(S_{i,l-1})(1-\theta_{l-1}) + Q_{l-1}(1-S_{i,l-1})\,\theta_{l-1} \tag{8}$$

where θ_{l-1} is the recombination fraction between locus $l-1$ and l. After the forward phase, the meiosis indicators are sampled in reverse order from locus L to 1, this time taking into account the likelihood of recombination between loci l and $l+1$. During a single iteration, the meiosis sampler is run on all meioses in a pseudo-random order.

2.5.1 Parallel computation Above we described a sequential algorithm, but we note that the calculations of $P(Y_l|S_{*,l})$ are independent under the assumption of linkage equilibrium. In addition, the amount of work this involves increases as a function of the number of loci considered jointly (in our experiments, this accounted for $>90\%$ of the total runtime of the meiosis sampler). Each calculation of $P(Y_l|S_{*,l})$ involves the construction and evaluation of a founder allele graph (Kruglyak *et al.*, 1996; Sobel and Lange, 1996). Graph algorithms can be difficult to parallelize on a GPU, as the structure of the graph is dependent on the genotypes of different individuals in the pedigree and differences between individuals will lead to divergent thread execution paths, which are unsuited to current GPU architectures. For these reasons, in addition to the slow sequential sampling, the meiosis sampler will be run on the CPU with the parallel calculation of $P(Y_l|S_{*,l})$ at each loci being equally divided between all processor cores using OpenMP.

2.6 Coordination

Both samplers used by `SWIFTLINK` are run on the CPU, and LOD score calculations are performed on the GPU, if one is present. `SWIFTLINK` has a master thread that coordinates between the two hardware platforms. On the CPU, the master thread selects which sampler to run at each iteration and manages a thread pool to execute that sampler in parallel using the strategies described above, whereas on the GPU, it copies sampled descent graphs from host to device memory and executes the GPU LOD score kernel.

`SWIFTLINK` is able to use both CPU and GPU concurrently because of how code is run on the GPU. GPU kernels are run asynchronously, that is, the call to run code on the GPU does not block until completion, but returns control to the caller immediately. A CUDA application, after launching a kernel, calls a separate *wait* function to be signalled when results are ready to be copied from the device to host RAM. LOD score calculations are performed on the GPU by copying the current descent graph produced by the Markov chain from host to device memory and then starting the LOD score kernel. As the LOD score kernel operates on a copy of the descent graph held in GPU memory, the CPU is free to run the Markov chain independently without interfering. When the next sample is produced, before we copy it over to the GPU, we must call *wait* to ensure that the previous LOD score calculations have completed.

3 RESULTS

The single-threaded variants of both locus and meiosis samplers have been shown previously to perform well on simulated datasets that can be analysed using exact linkage algorithms (Wijsman *et al.*, 2006). In this article, we focus on a real-world case study to gain an appreciation for the runtimes inherent in linkage studies that require MCMC.

Our case study, shown in Figure 2, is a highly consanguineous six-generation pedigree with EAST syndrome. EAST syndrome is an autosomal recessive monogenic disorder related to improper renal tubular salt handling where patients additionally present with the following symptoms: infantile-onset seizures, ataxia and sensorineural deafness. Genotyping with SNPs (single nucleotide polymorphisms) was performed on six members of the pedigree using Affymetrix GeneChip Human Mapping 10K SNP chips including all affected individuals and the parents of the three affected siblings (individuals 23–28 in Fig. 2). In this article, we only consider chromosome 1, which included 780 SNPs (0.35 cM average spacing). The disease trait is

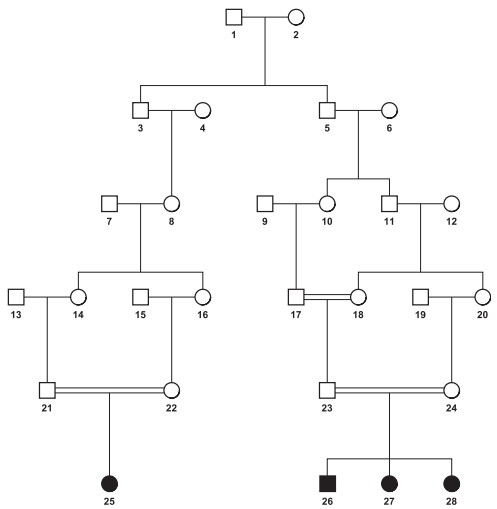

Fig. 2. EAST syndrome pedigree from Bockenhauer *et al.*, 2009 containing 28 members over six generations

assumed to have an allele frequency of 0.0001 and to have complete penetrance.

We compare the runtime performance of SIMWALK (version 2.91), MORGAN (version 3.1, *lm_linkage* program) and SWIFTLINK in several configurations. SIMWALK was always run with default parameters and, as the runtime can be prohibitively slow for large numbers of markers, each dataset was run in windows of 50 markers (this size was chosen based on experience with past projects). Even with this advantage, SIMWALK still had to be run on a computer cluster, whereas MORGAN and SWIFTLINK were run on a single desktop PC. We report SIMWALK's total runtime assuming sequential execution. MORGAN, by default, is compiled at optimization level 0, causing unnecessary slowdown. To ensure a fair comparison, we recompiled MORGAN at optimization level 2, which we refer to as MORGAN Fast. Following the parameters used in a previous study (Wijsman *et al.*, 2006), both MORGAN and SWIFTLINK were run for a total of 100 000 iterations, comprising 10 000 iterations of burn-in and 90 000 iterations of simulation. *P*(*locus sampler*) was set to 0.5. The Markov chain was initialized with 1000 runs of sequential imputation. During simulation, we scored every 10th sample and calculated LOD scores at 10 equidistant points between each consecutive pair of markers. SWIFTLINK and MORGAN were both run on the same computer running Ubuntu 12.04.1 LTS edition with Linux kernel 3.2.0-30 for 64-bit. The computer has an AMD Phenom II with four processor cores (clocked at 3.2 GHz) and 4 GB RAM (clocked at 1.6 GHz). GPU performance was tested with CUDA 5.0 (Release Candidate 1) on an Nvidia GTX 580. Each experiment was repeated 10 times, all results are averages over all runs.

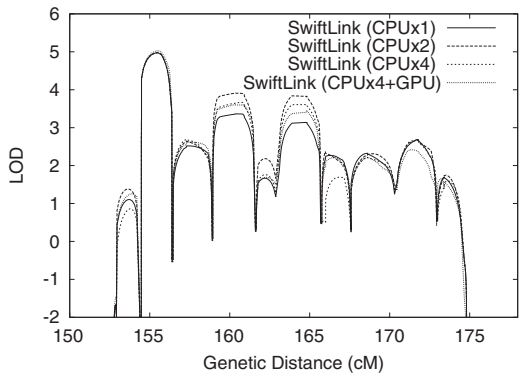

Fig. 3. Results from all four tested versions of SWIFTLINK showing region of interest from chromosome 1 of the EAST pedigree

Figure 3 shows typical results for the region of interest for the EAST pedigree on chromosome 1 from each of the four parallel versions of SWIFTLINK tested. The most likely linked region found by all programs corresponded to the 840 kilobase region identified in the original study containing the KCNJ10 gene with similar maximum LOD scores for the region of interest (Table 1).

3.1 Runtime

Table 1 shows the different performance characteristics of SIMWALK, MORGAN and SWIFTLINK. In addition to these results, we anecdotally ran SIMWALK once with all markers, which took ~42 days, making SIMWALK by far the slowest. Even with the advantage of running in 50 marker windows, SIMWALK had the worst runtime of the three programs tested.

All versions of SWIFTLINK were faster than both MORGAN and MORGAN Fast, with single-threaded SWIFTLINK showing a 1.6× speed-up over MORGAN Fast. This difference in runtime between single-threaded programs can probably be attributed to differences in the internal representation used for pedigree peeling; SWIFTLINK peels a genotype network, whereas MORGAN appears to peel an allele network. While allele networks ultimately scale better than genotype networks, in our experience, a majority of pedigrees will be smaller than is necessary for this to make a difference. At its fastest, using four CPU cores and GPU, SWIFTLINK is 13.6× faster than MORGAN Fast and 109.4× faster than SIMWALK analysing the EAST pedigree (Fig. 4).

SWIFTLINK achieves a high degree of parallelism, as $<3\%$ of the program's total runtime was spent on serial tasks that could not be parallelized. Using all four cores of the CPU and the GPU, we achieve up to almost an order of magnitude (8.5×) speed-up over the single-threaded implementation.

Compared with four CPU cores, the addition of a GPU provides a 2.4× speed-up. We had expected higher performance, but the CPU is a bottleneck which does not run the Markov chain fast enough to keep up with the GPU. We expect the situation to improve with CPUs with more cores and with improvements to our CPU code (so far we have not used SIMD instructions). To demonstrate how underused the GPU is, the LOD scoring code in isolation performs 24× faster than a single-threaded CPU implementation.

Table 1. Runtime and RAM usage for different MCMC-based linkage analysis programs operating on EAST pedigree

Program	Mean Max. LOD score	Mean Runtime	Std. dev.	Peak memory usage (MB)
SimWalk2	4.96	1356 min 02 s	11 min 13 s	5
Morgan 3.1	4.97	351 min 53 s	6 min 10 s	10
Morgan 3.1 Fast	4.97	168 min 20 s	2 min 43 s	10
SwiftLink (CPU × 1)	4.98	105 min 10 s	2 min 43 s	19
SwiftLink (CPU × 2)	4.97	55 min 01 s	1 min 12 s	19
SwiftLink (CPU × 4)	4.92	28 min 04 s	0 min 38 s	19
SwiftLink (CPU × 4 + GPU)	4.97	12 min 24 s	0 min 24 s	153 (+390 GPU)

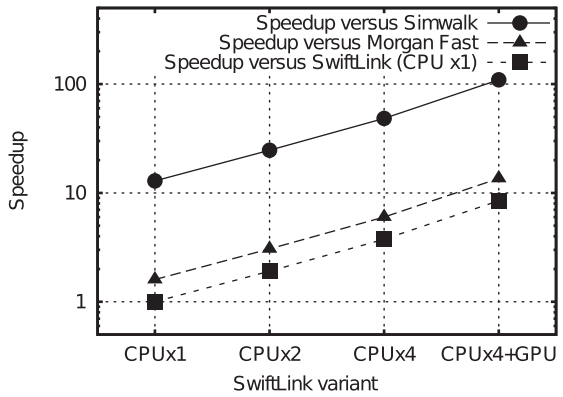

Fig. 4. Speed-up achieved by all four tested versions of SWIFTLINK versus SIMWALK, MORGAN and single-threaded SWIFTLINK

3.2 Memory usage

RAM usage was lowest in SIMWALK, though this is unrelated to the size of the input, as all memory must be statically allocated in Fortran '77. When SWIFTLINK is run without using the GPU, memory usage is twice that of MORGAN. Using a GPU in SWIFTLINK increases memory usage on the host PC 8-fold, partly due to converting C++ data structures into a more appropriate format for the GPU and partly due to GPU driver overhead. RAM usage on the GPU is high, but most modern GPUs feature at least 1 GB and we assume exclusive access to the device.

4 DISCUSSION

The methods presented in this article produce almost an order of magnitude speed-up of MCMC linkage analysis compared with an optimized single-threaded implementation on fairly minimal hardware. Compared with existing MCMC linkage software, SWIFTLINK achieves up to two orders of magnitude speed-up when using four CPU cores and a GPU concurrently. We believe SWIFTLINK will be especially useful to researchers in, for example, clinical research settings, where access and/or experience with computer clusters is limited. In addition, SWIFTLINK has been specifically designed to fit into existing workflows by supporting the standard 'LINKAGE' file formats (ped, map and data files) as other popular mulitpoint linkage analysis programs, e.g. Allegro and Genehunter.

A majority of linkage projects include only small-to-medium size pedigrees and can therefore be analysed using exact algorithms. Many large pedigrees can be successfully abbreviated to run using an exact algorithm; however, for cases where there are few genotyped individuals or in cases like EAST syndrome (Fig. 2), where one of the branches only contains a single genotyped individual, any abbreviation will dramatically reduce the power of the study. This is a clear niche where SWIFTLINK can be used to great effect.

Unlike many other articles detailing GPU applications, we did not show results for both 32- and 64-bit. As performance on the GPU is affected so strongly by register usage and therefore occupancy, 32-bit code tends to run faster as pointers take up half the space and more blocks can be run concurrently. However, CPU code in general performs faster in 64-bit thanks to wider instructions. In the current version of SWIFTLINK, the GPU is underused because the CPU is a bottleneck, therefore it only makes sense for us to run in 64-bit mode. In the future, we will be aiming to improve SWIFTLINK by removing the bottlenecks from the slower execution of the Markov chain on the CPU. Additional speed-ups could be made by using SIMD instructions as well as multithreading or by offloading some of the excess work to the GPU. It is not clear how to balance the load dynamically between both platforms and whether the additional complexity is warranted for a speed-up that can be provided more simply with additional CPUs, e.g. with more cores locally or in a computer cluster.

ACKNOWLEDGEMENTS

The authors acknowledge the use of the UCL Legion High Performance Computing Facility, and associated support services, in the completion of this work.

Funding: R.K. is supported by St. Peter's Trust for Kidney, Bladder and Prostate Research, The Grocers' Charity and The David and Elaine Potter Charitable Foundation.

Conflict of interest: none declared.

REFERENCES

Bockenhauer,D. *et al.* (2009) Epilepsy, ataxia, sensorineural deafness, tubulopathy, and KCNJ10 mutations. *NEJM*, **360**, 1960–1970.

Cannings,C. *et al.* (1976) The recursive derivation of likelihoods on complex pedigrees. *Adv. Appl. Probab.*, **8**, 622–625.

Conant,G. *et al.* (2002) Parallel Genehunter: Implementation of a linkage analysis package for distributed memory architectures, Proceedings IEEE HiCOMB'02.

Dwarkadas,S. *et al.* (1994) Parallelization of general linkage analysis problems. *Hum. Hered.*, **44**, 127–141.

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

Fishelson,M. and Geiger,D. (2002) Exact genetic linkage computations for general pedigrees. *Bioinformatics*, **18** (**Suppl. 1**), S189–S198.

Fishelson,M. and Geiger,D. (2004) Optimizing exact genetic linkage computations. *J. Comput. Biol.*, **11**, 263–275.

George,A.W. *et al.* (2003) Approaches to mapping genetically correlated complex traits. *BMC Genet.*, **4**, S71.

George,A.W. *et al.* (2005) MCMC multilocus lod scores: application of a new approach. *Hum. Hered.*, **59**, 98–108.

Gupta,S.K. *et al.* (1995) Integrating parallelization strategies for linkage analysis. *Comput. Biomed. Res.*, **28**, 116–139.

Heath,S.C. (1997) Markov chain Monte Carlo segregation and linkage analysis for oligogenic models. *Am. J. Hum. Genet.*, **61**, 748–760.

Heath,S. and Thompson,E.A. (1997) MCMC samplers for multilocus analyses on complex pedigrees. *Am. J. Hum. Genet.*, **61** (**Suppl.**), A278.

Heath,S.C. *et al.* (1997) MCMC Segregation and linkage analysis. *Genet. Epidemiol.*, **14**, 1011–1016.

Kong,A. (1991) Analysis of pedigree data using methods combining peeling and Gibbs sampling. *Computer Science and Statistics: Proceedings of the 23rd Symposium on the Interface*, 379–385.

Kruglyak,L. *et al.* (1995) Rapid multipoint linkage analysis of reccessive traits in nuclear families including homozygosity mapping. *Am. J. Hum. Genet.*, **56**, 519–527.

Kruglyak,L. *et al.* (1996) Parametric and nonparametric linkage analysis: a unified multipoint approach. *Am. J. Hum. Genet.*, **58**, 1347–1363.

Lander,E.S. and Green,P. (1987) Construction of multilocus genetic maps in humans., *Proc. Natl Acad. Sci. USA*, **84**, 2363–2367.

Lange,K. and Sobel,E. (1991) A random walk method for computing genetic location scores. *Am. J. Hum. Genet.*, **49**, 1320–1334.

Lee,A. *et al.* (2009) On the utility of graphics cards to perform massively parallel simulation of advanced monte carlo methods. *J. Comput. Graph. Stat.*, **19**, 4.

Lin,S. and Speed,T.P. (1996) Incorporating crossover interference into pedigree analysis using the χ^2 model. *Hum. Hered.*, **46**, 315–322.

Silberstein,M. *et al.* (2006) On-line system for faster linkage analysis via parallel execution on thousands of personal computers. *Am. J. Hum. Genet.*, **78**, 922–935.

Sobel,E. and Lange,K. (1996) Descent graphs in pedigree analysis: applications to haplotyping, location scores, and marker-sharing statistics. *Am. J. Hum. Genet.*, **58**, 1323–1337.

Sobel,E. *et al.* (2001) Multipoint estimation of identity-by-descent probabilities at arbitrary positions among marker loci on general pedigrees. *Hum. Hered.*, **52**, 121–131.

Suchard,M.A. and Rambaut,A. (2009) Many-core algorithms for statistical phylogenetics. *Bioinformatics*, **25**, 1370–1376.

Thompson,E.A. and Heath,S.C. (1999) Estimation of conditional multilocus gene identity among relatives. *Lect. Notes Monogr. Ser.*, **33**, 95–113.

Wijsman,E.M. *et al.* (2006) Multipoint linkage analysis with many multiallelic or dense diallelic markers: Markov chain-Monte Carlo provides practical approaches for genome scans on general pedigrees. *Am. J. Hum. Genet.*, **79**, 846–858.