Rapid Computation of Distance Estimators from Nucleotide and Amino Acid Alignments

M. Simonsen
Bioinformatics Research Center
C. F. Møllers Allé 8
DK-8000 Aarhus C, Denmark
zxr@birc.au.dk

C. N. S. Pedersen
Bioinformatics Research Center
C. F. Møllers Allé 8
DK-8000 Aarhus C, Denmark
cstorm@birc.au.dk

ABSTRACT

Distance estimators are needed as input for popular distance based phylogenetic reconstruction methods such as UPGMA and neighbour-joining. Computation of these takes $O(n^2l)$ time for n sequences with length l which is usually fast compared to reconstructing a phylogenetic tree of n taxa. However, with the introduction of fast search heuristics for distance based phylogenetic reconstruction methods, the computation of distance estimators has become a bottleneck especially for long sequences. Elias et al. have shown how distance estimators can be computed efficiently from unaligned nucleotide sequences using vectorisation of code. In this paper we extend their method to allow efficient computation of distance estimators from aligned nucleotide and amino acid sequences using vectorisation of code and parallelisation on both CPUs and GPUs. Experiments are presented which show an increase in performance of up to 36x and 8x relative to the naive approach when computing distance estimators from nucleotides and amino acids alignments respectively.

Categories and Subject Descriptors

I.m [**Computing Methodologies**]: Miscellaneous;
J.3 [**Computer Applications**]: Life and Medical Sciences—*Biology and genetics*

General Terms

Algorithms

Keywords

Phylogenetic distance estimator, Phylogenetic inference, Parallelization, Vectorization, GPU.

1. INTRODUCTION

Distance based methods are widely used for phylogenetic tree reconstruction and are in general computationally efficient. These methods use estimates of the pairwise evolutionary distance between a set of sequences to reconstruct trees. Such distance estimates are normally based on the number of mutational events between sequences and counting these is the most time consuming step in computing most distance estimators. Computing the number of mutational events between each pair of n sequences takes time $O(n^2l)$ where l is the length of the sequences. Compared to the $O(n^3)$ running time of the two most popular distance based phylogenetic reconstruction methods, UPGMA [15] and neighbour-joining [16], this is fast and can be done with a naive implementation. However, by using fast search heuristics such as FastPair [5] and RapidNJ [17] the running times of UPGMA and neighbour-joining are significantly reduced hereby making computation of distance estimators a bottleneck even when l is relatively small.

Elias et al. [4] have shown how the number of mutational events between two nucleotide sequences can be computed efficiently using the SSE2 instruction set combined with a divide and conquer algorithm for counting the number of 1-bits in a vector. In this paper we extend this method to work on multiple aligments of nucleotides and we also present a new method for computing distance estimators from amino acid sequences and alignments of these.

SAC'11 21-MAR-2011, TaiChung, Taiwan.

2. BACKGROUND

2.1 Computing the evolutionary distance between sequences

The evolutionary distance between two nucleotide or amino acid sequences can be computed as the observed number of substitutions, i.e. number of substitutions needed to transform one sequence into the other. However, this is usually an underestimate as multiple substitutions could have occurred at any site in the two sequences. Sequence evolution models such as the popular Kimura's two-parameter (K2P) model [12] correct for these hidden substitutions based on the observed number of substitutions.

The K2P model takes into account that transitions (A↔T and C↔G) and transversions (all other substitutions) occur at different rates. The distance, d, between two nucleotide sequences can be computed under the K2P model as

$$d = \frac{1}{2} ln \left(\frac{1}{1-2P-Q} \right) + \frac{1}{4} ln \left(\frac{1}{1-2Q} \right) \quad (1)$$

where P and Q are the frequency of transitions and transversions respectively. With the number of transitions and transversions it is also possible to use both the Jukes-Cantor [10]

Table 1: Nucleotide SIMD operations.

$r \leftarrow u \oplus v$
$tv \leftarrow (r \gg 1)\ \&\ m$
$ts \leftarrow (r\ \&\ m)\ \&\ (\sim tv)$

Table 2: SIMD operations for handling gaps

$g \leftarrow g_v\ \&\ g_u$
$tv \leftarrow tv\ \&\ g$
$ts \leftarrow ts\ \&\ g$

and Tamura-Nei [11] models for estimating the evolutionary distance.

In case of amino acid sequences, empirical score matrices such as PAM or BLOSUM matrices are often used to estimate the distance between sequences. PAM distances can be approximated using Kimura's distance [13] as

$$d = -ln\left(1 - p - \frac{1}{5}p^2\right) \quad (2)$$

where p is the proportion of substitutions between two amino acid sequences. Compared to distance estimates computed with score matrices, the Kimura distance is more rough as all types of substitutions are treated equal. Even so, the Kimura distance is used in phylogenetic software to provide fast distance estimates.

Equations 1 and 2 are computationally efficient once the number of substitutions has been obtained. A naive approach for counting substitutions is to simply compare each pair of nucleotides or amino acids one by one as ASCII characters. By taking advantage of modern processors Single Instruction Multiple Data (SIMD) capabilities, 2x128 bits or 16 characters can be compared in each iteration which significantly reduces the time required to count the number of substitutions between sequences [4].

2.2 Counting transition and transversions using SIMD instructions

In [4] SIMD instructions, available in the SSE2 instruction set, are used to count the number of transitions and transversions between nucleotide sequences which in combination with a compact encoding of nucleotides allows up to 64 nucleotides to be processed simultaneously. The complexity of the algorithm remains $O(n^2 l)$ but the algorithm has a significantly better performance in practice compared to the naive approach.

Each nucleotide is encoded using a 2-bit encoding where $A = 00$; $C = 11$; $G = 01$; $T = 10$. Given two 128-bit vectors, u and v, with encoded nucleotides, the number of transitions and transversions are computed using the SIMD instructions in Table 1 as follows. The exclusive or (XOR) of v and u is computed and stored in a 128-bit vector r. Each 2-bit block of r now contains one of the following bit patterns: <00> indicating that no substitution has occurred, <01> indicating that a transition has occurred or either <10> or <11> both indicating that a transversion has occurred. A 128-bit vector tv is computed by performing a logical bit-wise right-shift of r by one bit and computing the bit-wise *AND* with a 128-bit mask, m, containing the bit-pattern <010101...>. The bit-vector tv now contains a 1-bit for each transversion that has occurred. Another 128-bit vector, ts, is computed as the bit-wise *AND* of r and m followed by the bitwise *AND* with the negation of tv. As a result, ts contains a 1-bit for each transition that has occurred. The number of transitions and transvertions between the encoded nucleotides in u and v can now be computed as the number of 1-bits in ts and tv respectively. Counting the number of 1-bits in a bit-vector can be done by counting them one by one. This is, however, inefficient, but the SSE2 instruction set does not provide an immediate way of doing this more efficiently[1].

A divide and conquer algorithm for counting the number of 1-bits in ts and tv efficiently is described in [4] and summarised in the following. Each 2-bit block in ts and tv contains either 00 or 01 corresponding to the integer values 0 and 1. Up to three of these 128-bit vectors can be added with a SIMD add operation without causing an overflow in the 2-bit blocks. Using a series of SIMD instructions, neighbouring 2-bit blocks can be merged into a 4-bit block containing the sum of the merged blocks. The result is a 128-bit vector containing 32 4-bit blocks each representing an integer $\leq$6. Now the process can be repeated by adding vectors until we risk overflowing a 4-bit integer in which case neighbouring 4-bit blocks are merged into 8-bit blocks. When only one vector remains or the block size reaches 32 bits and there is a risk of overflowing, the integers are extracted and added using scalar instructions.

3. METHODS

3.1 Computing distance estimators from multiple alignments of nucleotide sequences

The method presented in [4] does not handle gaps and is therefore not suited for multiple alignments of nucleotide sequences. To enable computation of distance estimators from multiple alignments, the encoding of nucleotides need to be extended with at least one bit for representing gaps. We use a 4-bit encoding which results in a cleaner and faster code compared to a 3-bit encoding. Each nucleotide in the alignment is encoded using the 2-bit encoding introduced in Sec. 2.2 and gaps are encoded as Adenine, i.e. $A = 00$ (but could be encoded as any one of the four nucleotides). The remaining two bits are used to create a *gap-filter* for each sequence. The purpose of the gap-filter is to ignore substitutions between two sequences where one or both sites contain a gap which is a common strategy for handling gaps. For each sequence a gap-filter is created as a number of 128-bit vectors where each 2-bit block is set to <00> if the sequence contains a gap at the corresponding site and <01> otherwise. Given two 128-bit vectors of encoded nucleotides, v and u, we compute ts and tv using the operations in Table 1. Let g_v and g_u be two 128-bit vectors containing the sections of gap-filters corresponding to v and u respectively. The operations in Table 2 can now be used to zero the number of transitions and transvertions at sites containing gaps in ts and tv.

Using the divide and conquer method described in Sec. 2.2, the number of transitions and transvertions can be obtained

[1]Instructions for counting the number of 1-bits in a vector is available in the SSE4.2 instruction set. However, the latency of these instructions are high which leads to a decrease in performance.

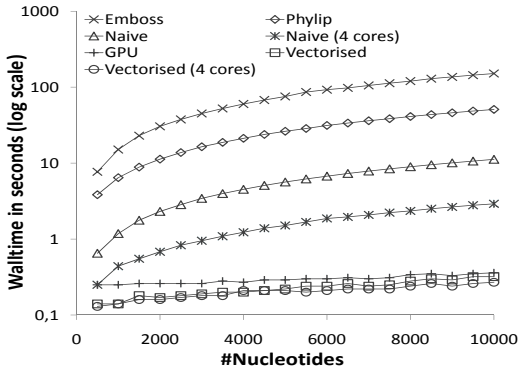

Figure 1: Running times on simulated datasets containing 500 nucleotide sequences.

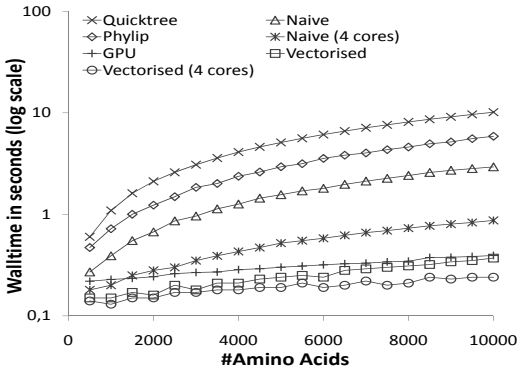

Figure 2: Running times on simulated datasets containing 500 amino acid sequences.

Table 3: Amino acid SIMD operations.

$r \leftarrow u \ == v$
$r \leftarrow (\sim r)$ & m_8
$g \leftarrow ((u \ == \ m_g) \parallel \ (v \ == \ m_g))$ & m_8
$r \leftarrow (\sim g)$ & r

by counting the number of 1-bits in ts and tv respectively. To compute a distance estimate of two aligned sequences, the length of the two sequences where columns containing gaps have been removed is also needed. The length is obtained by simply counting the number of 1-bits in the g-vectors.

3.2 Computing distance estimators from amino acid sequences

Computing the Kimura distance between two amino acid sequences requires the number of substitutions between these sequences and the length of the sequences without gaps. As with nucleotide sequences, this can be done efficiently using the SSE2 instruction set as follows. Each amino acid is encoded using the standard one-letter code in 8-bit ASCII format thus allowing 16 amino acids to be processed simultaneously using the SSE2 instruction set. Gaps are encoded as the '-' character but any character not used by the amino acids is usable. Given two 128-bit vectors of encoded amino acids, v and u, the number of mutations can be counted using the SIMD operations in Table 3.

The bit-vectors u and v are compared using the *pcmpeqb* instruction from the SSE2 instruction set which compares each byte in u and v for equality. The return value of the comparison is a new 128-bit vector, r, where all bits in a byte are set to 1 if the two bytes were equal, and 0 otherwise. Next, the bitwise negation of r is computed and used to compute the bitwise AND with a mask, m_8, where each byte contains the bit-pattern <00000001>. r now contains a 1-bit for each substitution in the two vectors but possibly also a 1-bit for positions with gaps. To ignore positions with gaps, each byte in v and u are compared with a mask, m_g, where each byte has the bit-pattern <00101101> corresponding to the ASCII code for '-'. The result of the comparison is then combined using a bitwise OR and, subsequently, a bitwise AND with the m_8 mask. The bytes in g now contain <00000001> if the corresponding position in either v or u contained a gap and <00000000> otherwise. The last step is to zero all positions in r which contained a gap using g. The number of mutations between the two substrings in v and u and the length of the two sequences without gaps is obtained by counting the number of 1-bits in all r and g respectively.

3.3 Computation of distance estimators using GPUs

General Purpose Graphics Processing Units (GPU) are available in most modern desktop computers and often have significantly higher peak performance compared to CPUs. However, problems must be massively parallelisable to take advantage of GPUs which often contains more than 100 cores. When counting the number of substitutions between n sequences, each pair of sites can be processed independently making this problem well suited for GPUs.

We have used NVIDIAs Compute Unified Device Architecture (CUDA) SDK to implement the methods in Sec. 3.1 and 3.2. CUDA uses a multi-threading scheme to parallelise workloads where threads are organised in *blocks* and blocks are organised in grids. Using this organisation of threads developers can define parallelisation schemes for a specific problem. Several approaches for parallelising the computation of distance estimators were investigated where the following approach showed the best overall performance.

Given n sequences a grid containing $n \times (n-1)$ blocks is used to compute distance between all pairs of sequences. Each block is assigned to a pair of sequences, (i, j), and contains 128 threads where each thread is used to compute the number of substitutions between $\lceil l/128 \rceil$ sites in sequence i and j. Each thread store intermediate results in shared memory. To obtain the total number of substitutions we use *parallel reduction* [8] to sum all intermediate results efficiently using all threads in a block and shared memory.

CUDA does not make use of SIMD instructions directly but as GPU cores are essentially vector processors many SSE2 instructions can be implemented using a number of threads executing the same sequence of instructions on vectors of data. In case of nucleotide sequences, the 128-bit SIMD operations in Table 1 and 2 can be implemented using four threads executing the same scalar operation on 32-bit of data. It is, however, not possible to implement the *pcmpeqb* instruction used in Table 3 with a single operation. Consequently, each pair of amino acids must be compared one at a time.

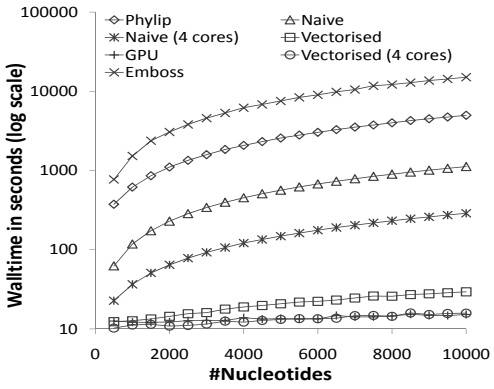

Figure 3: Running times on simulated datasets containing 5000 nucleotide sequences.

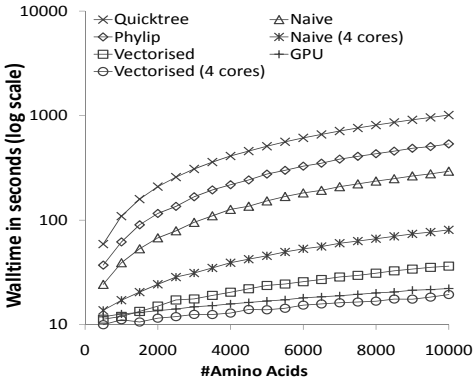

Figure 4: Running times on simulated datasets containing 5000 amino acid sequences.

4. RESULTS AND DISCUSSION

To assess the performance of the methods described in Sec. 3 they were implemented in a tool using C++ and CUDA 3.0. We will refer to our implementations as *vectorised* for the SSE2 based solution and *GPU* for the CUDA based solution. An optimised version of the naive approach, referred to as *naive*, was also implemented. Both the vectorised and naive implementation were parallelised using POSIX threads. Source code for all implementations is available at *http://birc.au.dk/research/software/rapidnj/* (as a part of the RapidNJ tool).

The running times of the different methods were compared against those of three popular tools. For nucleotide sequences we compared with *dnadist* from the *Phylip* package [6] and distmat from the *EMBOSS* package [3]. For amino acid sequences *protdist* from the Phylip package and *QuickTree* [9] were used. The K2P model (Eq. 1) or the Kimura distance (Eq. 2) were employed by all tools to compute distances for nucleotide sequences and amino acid sequences respectively. All running times include the time used for loading data, encoding sequences where applicable and outputting the result. The experiments were performed on an 2.66 GHz Intel Core 2 quad (Q9450) CPU with 4GB RAM running Ubuntu 9.04 equipped with a Geforce 8800GT GPU.

4.1 Experiments on simulated data

In Fig. 1 and 2 the running times on alignments containing 500 simulated (random nucleotides and amino acids) sequences are shown. The unparallelised vectorised implementation is up 35x faster on nucleotide alignments compared to our own naive implementation. On amino acid alignments the vectorized implementation can only process 16 sites concurrently compared to 64 on nucleotide alignments which results in a smaller speedup of 8x compared to our naive implementation. On larger alignments with 5,000 sequences, the unparallelised vectorized implementations also achieve significant speedups of up to 36x on nucleotide alignments and 8x on amino acid alignments as shown in Fig. 3 and 4.

Neither of the three reference tools (Phylip, EMBOSS, QuickTree) utilise the parallel capabilities of modern processors which makes a direct comparison to the massive parallel GPU implementation infeasible. However, as our unparallelised naive implementation has better performance than the three reference tools, we use the parallelised version of this implementation to assess the performance of the parallel vectorised implementation and the GPU implementation. On short alignments, the naive implementation does not benefit significantly from parallelisation whereas a near 4x speedup is achieved on longer alignments using 4 cores. Still, both the unparallelised and parallelised vectorised implementations are significantly faster than the parallelised naive implementation. The performance of the GPU implementation and the parallelised vectorised implementation is similar even though the GPUs theoretical peak performance exceeds that of the CPU by more than a factor 5. As shown in [18], it is often hard to fully utilise all clock cycles in GPUs because of e.g. small cache sizes and poor performance of scalar code. The cache size is especially important here as each sequence is involved in $O(n)$ comparisons. The limited amount of cache available in the GPU only allows small chunks of sequences to be stored and to take advantage of cached data (in shared memory) the number of thread-blocks has to be reduced to a point where the GPU becomes underutilised for most reasonable data set sizes. Experiments with caching sequence data in shared memory and the use of texture memory did not give rise to better performance.

4.2 Experiments on biological data

The size of the simulated alignments used in Sec. 4.1 were chosen to represent the size of real biological alignments which are currently available in various databases. Here we compare running times of the vectorised implementation against those of the naive implementation on large alignments found in online databases. From the EMBL-Align database [14] we used an alignment of 16S rRNA genes [2] mainly from Microsporidia, and from the HIV sequence database [1] we used a large alignment of HIV and SIVcpz sequences. We also used two amino acid alignments found in the Pfam [7] database which are identified by their Pfam-id in Table 4. In all experiments we used unparallelised implementations for computing K2P and Kimura corrected distances. The GPU implementation was not used in these experiments as the performance is similar performance to that of the vectorised implementation.

In Table 5 the running times for 1000 bootstrapping operations using neighbour-joining are shown. The Phylip tool uses a standard implementation of the neighbour-joining method while both the naive and vectorised implementa-

Table 4: Running times for computing distance estimators from biological data and the minimum speedup achieved by the vectorised implementation.

Dataset	n	l	Phylip	Naive	Vectorised	Speedup
16S (DNA)	338	1884	9, 19s	0, 7s	0, 09s	x7.8
HIV (DNA)	1257	11848	20m 14s	37, 97s	1, 87s	x20.3
PF00722.13 (protein)	1053	1470	2, 83s	2, 48s	0, 54s	4.6x
PF01370.13 (protein)	10338	1793	4m 58s	4m 28s	57, 91s	4.3x

Table 5: Running times when bootstrapping biological data and the speedup of the vectorised implementation compared to the naive implementation.

Dataset	Phylip	Naive	Vec.	Speedup
HIV	>24h	16h 16m	31m 25s	31,1x
PF00722.13	3h 11m	40m 3s	5m 20s	7,5x

tions use RapidNJ [17] to speed up phylogenetic reconstruction. The Phylip tool spends 76% of the total running time on reconstructing phylogenies when bootstrapping the PF00722.13 dataset compared to 10% in the naive implementation where RapidNJ is used. Clearly, the bottleneck in the naive implementation is the computation of distance estimators.

From the results in Tables 4 and 5 it is evident that vectorisation have a significant impact on the time required to compute distance estimators and hereby the reconstruction of phylogenetic trees. In particular the time required to perform a large number of bootstrapping evaluations is reduced from hours to minutes.

5. CONCLUSION

We have presented methods for computing the number of substitutions between aligned nucleotide and amino acid sequences which extends the work in [4] by handling gaps, amino acid sequences and parallelisation of the workload on both CPUs and GPUs. Our experiments showed that vectorisation of code gave a significantly higher performance increase than parallelisation of the naive approach on a CPU. When the code was both vectorised and parallelised on a CPU with 4 cores the performance was equal to that of an efficient implementation running on a comparable GPU. The methods presented here significantly reduce the time required to compute distance estimators using e.g. the K2P model and the Kimura distance and hence reduce the total time required to reconstruct phylogenetic trees using distance based methods.

The most time consuming step in reconstruction of phylogenies remains the computation of multiple alignments. Still, minimising the time spend on reconstructing phylogenies from a multiple alignment is important as it allows e.g. fast bootstrapping and fast analysis of readily available alignments.

6. REFERENCES

[1] Hiv databases. *http://www.hiv.lanl.gov*.

[2] H. E. McClymont. Molecular phylogeny of microsporidian parasites with special attention to mollusc-infective species 2006.

[3] T. Carver. distmat - the European Molecular Biology Open Software Suite (EMBOSS). *http://emboss.sourceforge.net/index.html*.

[4] I. Elias and J. Lagergren. Fast computation of distance estimators. *Bioinformatics*, 8:89, 2007.

[5] David Eppstein. Fast hierarchical clustering and other applications of dynamic closest pairs. *Journal of Experimental Algorithmics*, 5:1, 2000.

[6] J. Felsenstein. Phylip 3.69. *http://evolution.genetics.washington.edu/phylip.html*.

[7] R. D. Finn et al. Pfam: clans, web tools and services. *Nucleic Acids Research*, Database Issue 34:D247–D251, 2006.

[8] Mark Harris, S Sengupta, and JD Owens. Parallel prefix sum (scan) with CUDA. *GPU Gems*, (April), 2007.

[9] K. Howe, A. Bateman, and R. Durbin. QuickTree: Building huge neighbour-joining trees of protein sequences. *Bioinformatics*, 18(11):1546–1547, 2002.

[10] T. H. Jukes and C. R. Cantor. Evolution of protein molecules. *Mammalian Protein Metabolism*, pages 21–123, 1969.

[11] M. Nei K. Tamura. Estimation of the number of nucleotide substitutions in the control region of mitochondrial dna in humans and chimpanzees. *Molecular Biology and Evolution*, 10(3):512–526, 1993.

[12] M. Kimura. A simple model for estimating evolutionary rates of base substitutions through comparative studies of nucleotide sequences. *Journal of Molecular Evolution*, 1980.

[13] M. Kimura. The Neutral Theory of Molecular Evolution. *Cambridge University Press*, 1983.

[14] V. Lombard et al. Embl-align: a new public nucleotide and amino acid multiple sequence alignment database. *Bioinformatics*, 18(5):753–764, 2002.

[15] C. D. Michener and R. R. Sokal. A quantitative approach to a problem in classification. *Evolution*, 11:130–162, 1957.

[16] N. Saitou and M. Nei. The neighbor-joining method: A new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4:406–425, 1987.

[17] M. Simonsen, T. Mailund, and C. N. S. Pedersen. Rapid neighbour-joining. *Algorithms in Bioinformatics, Proceedings 8th International Workshop*, volume 5251, pages 113–123, 2008.

[18] V. W. Lee et al. Debunking the 100X GPU vs CPU myth an evaluation of throughput computing on CPU and GPU *Proceedings of the 37th annual international symposium on Computer architecture* 2010