Cache and Energy Efficient Alignment of Very Long Sequences

Chunchun Zhao
Department of Computer and Information Science and Engineering
University of Florida
Email: czhao@cise.ufl.edu

Sartaj Sahni
Department of Computer and Information Science and Engineering
University of Florida
Email: sahni@cise.ufl.edu

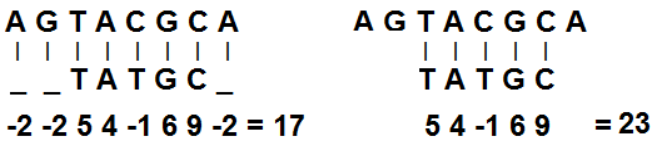
***Abstract*—We develop cache and energy efficient algorithms to align very long sequences. These algorithms were evaluated experimentally on a single node of the IBM Blue Gene/Q. We were able to reduce the run time of the classical Myers and Miller linear space alignment algorithm by up to 43%; energy consumption was reduced by up to 45% on our test data.**

I. Introduction

Sequence alignment is a fundamental and well studied problem in the biological sciences. In this problem, we are given two sequences $A[1:m] = a_1 a_2 \cdots a_m$ and $B[1:n] = b_1 b_2 \cdots b_n$ and we are required to find the score of the best alignment and possibly also an alignment with this best score. When aligning two sequences, we may insert gaps into the sequences. The score of an alignment is determined using a matching (or scoring) matrix that assigns a score to each pair of characters from the alphabet in use as well as a gap penalty model that determines the penalty associate with a gap sequence. In the linear gap penalty model, the penalty for a gap sequence of length $k > 0$ is kg, where g is some constant while in the affine model this penalty is $g_{open} + (k-1) * g_{ext}$. The affine model more accurately reflects the fact that opening a gap is more expensive than extending one. Two versions of sequence alignment–global and local–are of interest. In global alignment the entire A sequence is to be aligned with the entire B sequence while in local alignment we wish to find a substring of A and B that have the highest alignment score. The alphabet for DNA, RNA, and protein sequences is, respectively, {A, T, G, C}, {A, U, G, C}, and {A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y}.

Figure 1 illustrates these concepts using the DNA sequences $A[1:8]$ = {AGTACGCA} and $B[1:5]$ = {TATGC}. The symbol '_' denotes the gap character. The alignment of Figure 1(a) is a global alignment and that of Figure 1(b) is a local one. To score the alignments, we have used the linear penalty model with $g = -2$ and the scores for pairs of aligned characters, which are taken from BLOSUM62 [1], are $c(T,T) = 5$, $c(A,A) = 4$, $c(C,C) = 9$, $c(G,G) = 6$, and $c(C,T) = -1$. The score for the shown global alignment is 17 while that for the shown local alignment is 23. If we were using an affine penalty model with $g_{open} = -4$ and $g_{ext} = -2$, then the penalty for each of the gaps in positions 1 and 8 of the global alignment would be -4 and the overall score for the global alignment would be 13.

Needleman and Wunsch(NW) [2] proposed an $O(mn)$ time algorithm for global alignment using the linear gap model. This algorithm requires $O(n)$ space when only the score of the best alignment is to be determined and $O(mn)$ space when the best alignment is also to be determined. Smith and Waterman(SW) [3] modified the Needleman-Wunsch algorithm so as to determine the best local alignment. Gotoh [4] proposed a dynamic programming algorithm for sequence alignment using an affine gap penalty model. The asymptotic complexity of the SW and Gotoh algorithms is the same as that of the Needlman-Wunsch algorithm.



Global alignment Local alignment

Fig. 1. Example alignments using the linear gap penalty model

When mn is large and a best alignment is sought, the space, $O(mn)$, required by the algorithms of NW, SW and Gotoh [2], [3], [4] exceeds what is available on most computers. The best alignment for these large instances can be found using sequence alignment algorithms derived from Hirschberg's linear space divide-and-conquer algorithm [5] for the longest common subsequence problem. Myers and Miller [6] develop a linear space $O(mn)$ time version of Hirschberg's algorithm for global sequence alignment using an affine gap penalty model and Huang, Hardison, and Miller [7] do this for local alignment.

In an effort to speed sequence alignment, fast sequence-alignment heuristics have been developed. BLAST, FASTA, and Sim2 [8], [9], [10] are a few examples of software systems that employ sequence-alignment heuristics. Another direction of research, also aimed at speeding sequence alignment, has been the development of parallel algorithms. Parallel algorithms for sequence alignment may be found in [11], [12], [13], [14], [15], [16], [17], [18], [19], for example.

In this paper, we focus on reducing the cache misses that occur in the computation of the score of the best alignment as well as in determining the best alignment. Although we explicitly consider only the linear gap penalty model, our methods readily extend to the affine gap penalty model. Our interest in cache misses stems from two observations–(1)

the time required to service a last-level-cache (LLC) miss is typically 2 to 3 orders of magnitude more than the time for an arithmetic operation and (2) the energy required to fetch data from main memory is typically between 60 to 600 times that needed when the data is on chip. As a result of observation (1), cache misses dominate the overall run time of applications for which the hardware/software cache prefetch modules on the target computer are ineffective in predicting future cache misses. The effectiveness of hardware/software cache prefetch mechanisms varies with application, computer, and compiler. So, if we are writing code that is to be used on a variety of computer platforms, it is desirable to write cache-efficient code rather than to rely exclusively on the cache prefetching of the target platform. Even when the hardware/software prefetch mechanism of the target platform is very effective in hiding memory latency, observation (2) implies excessive energy use when there are many cache misses.

Our cache efficient scoring algorithm took up to 37% less time and up to 37% less energy when run on a single node of the IBM Blue Gene/Q while our cache efficient adaptation of the Myers and Miller alignment algorithm reduced run time by up to 43% and reduced energy consumption by up to 45%, relative to a classical implementation of this algorithm.

The rest of the paper is organized in the following way. In Section II, we describe our cache model and provide experimental results demonstrating the variability in performance of hardware/software cache prefetch mechanisms with changes in application and target computational platform. Our cache-efficient algorithms for scoring and alignment are developed and analyzed in Section III and experimental results presented in Section IV. Finally, Section V presented conclusion remarks.

II. Cache Model

For simplicity in analysis, we assume a single cache comprised of s lines of size w words (a word is large enough to hold a piece of data, typically 4 bytes) each. So, the total cache capacity is sw words. The main memory is partitioned into blocks also of size w words each. When the program needs to read a word that is not in the cache, a cache miss occurs. To service this cache miss, the block of main memory that includes the needed word is fetched and stored in a cache line, which is selected using the LRU (least recently used) rule. Until this block of main memory is evicted from this cache line, its words may be read without additional cache misses. We assume the cache is write back with write allocate. Write allocate means that when the program needs to write a word of data, a write miss occurs if the block corresponding to the main memory is not currently in cache. To service the write miss, the corresponding block of main memory is fetched and stored in a cache line. Write back means that the word is written to the appropriate cache line only. A cache line with changed content is written back to the main memory when it is about to be overwritten by a new block from main memory.

Rather than directly assess the number of read and write misses incurred by an algorithm, we shall count the number of read and write accesses to main memory. Every read and write miss makes a read access. A read and write miss also makes a write access when the data in the replacement cache line is written to main memory.

Hardware and software cache prefetch mechanisms are often deployed to hide the latency involved in servicing a cache miss. These mechanisms may, for example, attempt to learn the memory access pattern of the current application and then predict the future need for blocks of main memory. The predicted blocks are brought into cache before the program actually tries to read/write from/into those blocks thereby avoiding (or reducing) the delay involved in servicing a cache miss.

III. Cache Efficient Algorithms

A. Scoring Algorithms

1) Needleman-Wunsch and Smith-Waterman Algorithms: Let H_{ij} be the score of the best global alignment for $A[1:i]$ and $B[1:j]$. We wish to determine H_{mn}. Needleman and Wunsch [2] derived the following dynamic programming equations for H using the linear gap penalty model. These equations may be used to compute H_{mn}.

$$H_{i,0} = -i * g \; H_{0,j} = -j * g, \; 0 \le i \le m, \; 0 \le j \le n \quad (1)$$

When $i > 0$ and $j > 0$,

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + c(a_i, b_j) \\ H_{i,j-1} + c(\_, b_j) = H_{i,j-1} - g \\ H_{i-1,j} + c(a_i, \_) = H_{i-1,j} - g \end{cases} \quad (2)$$

where $c(a_i, b_j)$ is the match score between characters a_i and b_j and g is the gap penalty.

For local alignment, H_{ij} denotes the score of the best local alignment for $A[1:i]$ and $B[1:j]$. The Smith and Waterman [3] equations for local alignment using the linear gap penalty model are:

$$H_{i,0} = 0, \; H_{0,j} = 0, \; 0 \le i \le m, \; 0 \le j \le n \quad (3)$$

When $i > 0$ and $j > 0$,

$$H_{i,j} = \max \begin{cases} 0 \\ H_{i-1,j-1} + c(a_i, b_j) \\ H_{i,j-1} + c(\_, b_j) = H_{i,j-1} - g \\ H_{i-1,j} + c(a_i, \_) = H_{i-1,j} - g \end{cases} \quad (4)$$

Several authors ([5], [6], for example) have observed that the score of the best local alignment may be determined using a single array $H[0:n]$ as in Figure 2.

The scoring algorithm for the Needleman and Wunsch algorithm is similar. It is easy to see that the time complexity of the algorithm of Figure 2 is $O(mn)$ and its space complexity is $O(n)$.

For the (data) cache miss analysis, we focus on read and write misses of the array H and ignore misses due to the reads of the sequences A and B as well as of the scoring matrix c (notice that there are no write misses for A, B, and c). Figure 3 shows the memory access pattern for H by algorithm $Score$. The first row denotes the initialization of H and subsequent rows denote access for different value of i (i.e., different iterations of the `for i` loop). Although the

```
Algorithm Score(A[1:m], B[1:n])
  for j = 0 to n do
    H[j] = 0; //Initialize row 0
  end
  for i = 1 to m do // Compute row i
    diag = 0;
    for j = 1 to n do
      nextdiag = H[j];
      H[j] = max(0, diag + c(A[i],B[j]),
                    H[j-1] - g, H[j] - g);
      diag = nextdiag;
    end
  end
  return H[n];
end Algorithm
```

Fig. 2. Smith-Waterman scoring algorithm

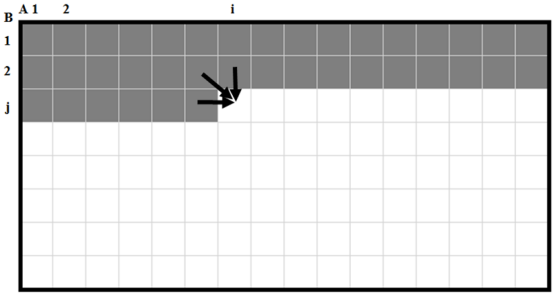


Fig. 3. Memory access pattern for $Score$ algorithm (Figure 2)

computation of H_{ij} is done using a single one-dimensional array $H[]$, following the i'th iteration of the `for i` loop, $H[j] = H_{ij}$. In each iteration of this loop, the elements of $H[]$ are accessed left-to-right. During the initialization loop, H is brought into cache in blocks of size w. Assume that n is sufficiently large so that $H[]$ does not entirely fit into cache. Hence, at some value of j, the cache capacity is reached and further progress of the initialization loop causes the least recently used blocks of $H[]$ (i.e., blocks from left to right) to be evicted from cache. The evicted blocks are written to main memory as they have been updated. So, the initialization loop results in n/w read accesses and (approximately) n/w write accesses (the number of write accesses is actually $n/w - s$). Since the left part of $H[]$ has been evicted from cache by the time we start the computation for row $i > 0$ (see Figure 3), each iteration of the `for i` loop also results in n/w read accesses and approximately n/w write accesses. So, the total number of read accesses is $(m + 1)n/w \approx mn/w$ and the number of write accesses is also $\approx mn/w$. The number of read and write accesses is $\approx 2mn/w$, when n is large.

We note, however, that when n is sufficiently small that $H[]$ fits into cache, the number of read accesses is n/w (all occur in the initialization loop) and there are no write accesses. In practice, especially in the case of local alignment involving a long sequence, one of the two sequences A and B is small enough to fit in cache while the other may not fit in cache. So, in these cases, it is desirable to ensure that A is the longer sequence and B is the shorter one so that H fits in cache.

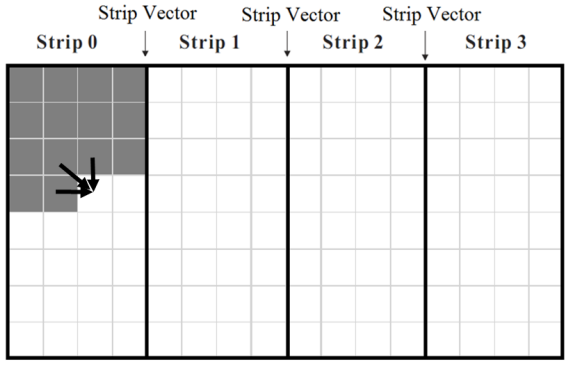


Fig. 4. Memory access pattern for $Strip$ algorithm (Figure 5)

This is accomplished using Algorithm $Swap$, which swaps A and B when $|A| < |B|$ and then runs Algorithm $Score$. When $H[1 : m]$ fits into cache and $H[1 : n]$ does not, Algorithm $Score$ incurs $O(mn/w)$ read/write accesses while Algorithm $Swap$ incurs $O(m/w)$ read/write accesses; when $m < n$ and $H[1 : n]$ fits into cache the number of read/write accesses is $O(m/w)$ for Algorithm $Swap$ and $O(n/w)$ for Algorithm $Score$; both algorithms incur approximately the same number, $\approx 2mn/w$, of read/write accesses for other values of m and n.

2) Strip Algorithm: When neither $H[1 : m]$ nor $H[1 : n]$ fits into cache, accesses to main memory may be reduced by computing H_{ij} by strips of width q such that q consecutive elements of $H[]$ fit into cache. Specifically, we partition $H[1 : n]$ into n/q strips of size q (except possibly the last strip whose size may be smaller than q) as in Figure 4. First, all H_{ij} in strip 0 are computed, then those in strip 1, and so on. When computing the values in a strip, we need those in the rightmost column of the preceding strip. So, we save these rightmost values in a one-dimensional array $previous[0 : m]$. The algorithm is given in Figure 5. We note that sequence alignment by strips has been considered before. For example, it is used by Li et al. [12] in their GPU algorithm. Their use differs from ours in that they compute the strips in pipeline fashion with each strip assigned to a different pipeline stage in round robin fashion and within a strip the computation is done by anti-diagonals in parallel. On the other hand, we do not pipeline the computation among strips and within a strip our computation is by rows.

It is easy to see that the time complexity of Algorithm $Strip$ is $O(mn)$ and that its space complexity is $O(m + n)$. For the cache misses, we focus on those resulting from the reads and writes of $H[]$ and $previous[]$. The initialization of $previous$ results in m/w read accesses and approximately the same number of write accesses. The computation of each strip makes the following accesses to main memory:

1) q/w read accesses for the appropriate set of q entries of H for the current strip and q/w write accesses for the cache lines whose data are replaced by these H values. The write accesses are, however, not made for the first strip.
2) m/w read accesses for $previous$ and m/w write accesses. The number of write accesses is less by s for the last strip.

```
Algorithm Strip(A[1:m], B[1:n])
  for j = 1 to m do
    previous[j] = 0 //leftmost strip
  end
  for t = 1 to n/q do //assume q divides n
    for j =t*q to t*q+q-1 do
      H[j] = 0 //Initialize first row
    end
    for i = 1 to m do //rows of strip
      diag = previous[i-1];
      H[j] = previous[i];
      for j = t*q to t*q+q-1 do
        nextdiag = H[j];
        H[j] = max( diag + c(A[i], B[j]),
                    H[j-1] - g, H[j] - g );
        diag = nextdiag;
      end
      previous[i] = H[j-1];
    end
  end
  return H[n]
end Algorithm
```

Fig. 5. Scoring algorithm by strips

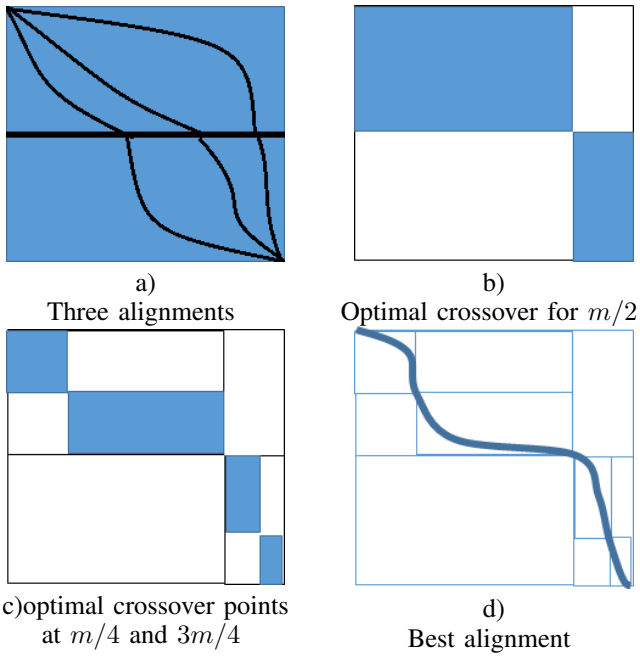


Fig. 6. An alignment as crossover points

So, the overall number of read accesses is $m/w + (q/w + m/w) * n/q = m/w + n/w + mn/(wq)$ and the number of write accesses is approximately the same as this. So, the total number of main memory accesses is $\approx 2mn/(wq)$ when m and n are large.

B. Alignment Algorithms

In this section we examine algorithms that compute the alignment that results in the best score rather than just the best score. While in Section III-A we explicitly considered local alignment and remarked that the results readily extend to global alignment, in this section we explicitly consider global alignment and remark that the methods extend to local alignment.

1) Myers and Miller's Algorithm: When aligning very long sequences, the $O(mn)$ space requirement of the full-matrix algorithm (NW [2]) exceeds the available memory on most computers. For these instances, we need a more memory efficient alignment algorithm. Myers and Miller [6] have adapted Hirschberg's linear space algorithm for the longest common subsequence problem to find the best global alignment in linear space. Its time complexity is $O(mn)$. However, this linear space adaptation performs about twice as many operations as does the full-matrix algorithm. Driga et al. [11] have developed a hybrid algorithm, FastLSA, whose memory requirement adapts to the amount of memory available on the target computing platform. We focus here on the adaptation of Myers and Miller [6].

It is easy to see that an optimal (global) alignment is comprised of an optimal alignment of $A[1 : m/2]$ and $B[1 : j]$ and an optimal alignment of $A[m : m/2 + 1]$ ($A[m : i]$ is the reverse of $A[i : m]$) and $B[n : j + 1]$ for some j, $1 \leq j \leq n$. The value of j for which this is true is called the *optimal crossover point*. Myers and Miller's linear space algorithm for alignment determines the optimal alignment by first determining the optimal crossover point k and then recursively aligning $A[1 : m/2]$ and $B[1 : k]$ as well as $A[m : m/2 + 1]$ and $B[n : k + 1]$. Equivalently, an optimal alignment of $A[1 : m]$ and $B[1 : n]$ is an optimal alignment of $A[1 : m/2]$ and $B[1 : k]$ concatenated with the reverse of an optimal alignment of $A[m : m/2 + 1]$ and $B[n : k + 1]$. Hence, an optimal alignment is comprised of a sequence of optimal crossover points. This is depicted visually in Figure 6. Figure 6(a) shows alignments using 3 possible crossover points at row $m/2$ of H. Figure 6(b) shows the partitioning of the alignment problem into 2 smaller alignment problems (shaded rectangles) using the optimal crossover point (meeting point of the 2 shaded rectangles) at row $m/2$. Figure 6(c) shows the partitioning of each of the 2 subproblems of Figure 6(b) using the optimal crossover points for these subproblems (note that these crossovers take place at rows $m/4$ and $3m/4$, respectively). Figure 6(d) shows the constructed optimal alignment, which is presently comprised of the 3 determined optimal crossover points.

The Myers and Miller algorithm, MM, uses a modified version of the linear space scoring algorithm $Score$ (Figure 2) to obtain the scores for the best alignments of $A[1 : i]$ and $B[1 : j]$, $1 \leq i \leq m/2$, $1 \leq j \leq n$ as well as for the best alignments of $A[m : i]$ and $B[m : j]$, $m/2 < i \leq m$, $1 \leq j \leq n$. This modified version $MScore$ differs from $Score$ only in that $MScore$ returns the entire array H rather than just $H[n]$. Using the returned H arrays for the forward and reverse alignments, the optimal crossover point for the best alignment is computed by evaluating all possible crossover points. Once the optimal crossover point is known, two recursive calls to MM are made to optimally align the top and bottom halves of A.

In each level of recursion, the number of main memory accesses is dominated by those made in the calls to $MScore$.

From the analysis for $Score$ (Section III-A), it follows that when n is large, the number of accesses to main memory is $\approx 2mn/w(1+1/2+1/4+\cdots \approx 4mn/w$.

2) Swapped and Striped Myers and Miller Algorithm: Let $MSwap$ be algorithm $Swap$ with the use of $Score$ replaced $MScore$ and let $MStrip$ be algorithm $Strip$ (Figure 5) modified to return the entire H array rather than just $H[n]$. $MMSwap$ is obtained from MM by replacing the use of $MScore$ with $MSwap$. Our striped Myers and Miller algorithm ($MMStrip$) replaces the use of $MScore$ in MM with a test that causes $MSwap$ to be used in place of $MScore$ when one (or both) of m and n is sufficiently small; otherwise, $MStrip$ is used.

From the analysis for $Strip$ (Section III-A2), it follows that when n and m are large, the number of accesses to main memory is $\approx 2mn/(wq)(1+1/2+1/4+\cdots \approx 4mn/(wq)$.

IV. Experimental Results

A. Experimental Platform and Test Data

We implemented the algorithms of Section III in C and measured their relative performance on a single node of the IBM Blue Gene/Q. Each node is a 1.33GHz 64-bit PowerPC A2 with 32MB LLC cache. The MonEQ software [20], [21] was used to measure power usage every half second. We chose the Blue Gene/Q for our experiments because of the availability of an energy monitor. For our experiments, we ran each algorithm on every node of the Blue Gene/Q and divided the total energy consumed by the number of nodes.

All codes were compiled using the gcc compiler with the O2 option. For test data, we used randomly generated protein sequences as well as real protein sequences obtained from the Globin Gene Server[22] and DNA/RNA/protein sequences from the National Center for Biotechnology Information (NCBI) database [23]. We used the BLOSUM62[1] scoring matrix for all our experiments. The results for our randomly generated protein sequences were similar to those for the sequences used from the two databases [22] and [23]. So, we present only those for the latter data sets here.

B. Scoring Algorithms

Table I gives the run times, in seconds, and energy, in joules, consumed by $Score$ and $Swap$ for instances with $|A| << |B|$. The run time reduction resulting from $Swap$ ranged from 9.4% to 9.5% and the energy reduction ranged from 11.1% to 21.0%.

Figures 7 and 8 graph the run time and energy consumption of $Score$, $Swap$ and $Strip$ for 5 data sets. The values of $(|A|, |B|)$ for these, left to right, are $(97,634, 94,647)$, $(104,267, 103,004)$, $(200,000, 200,000)$, $(200,000, 398,273)$, and $(392,981, 398,273)$. On these data sets, $Strip$ was up to 37% faster than $Swap$ and consumed up to 37% less energy.

C. Alignment Algorithms

Figure 9 graphs the run time of algorithms MM, $MMSwap$ and $MMStrip$ and Figure 10 graphs the energy consumed by each algorithm for the first 4 data sets used in Figure 7. $MMSwap$ reduced run time by between 19% and 33% relative to MM and the energy reduction ranged between 23% and 38%. $MMStrip$ reduced run time, relative to $MMSwap$, by 15% to 20% and reduced energy consumption by between 11% and 17%. Also, $MMStrip$ delivers a speedup, relative to MM, between 35% and 43% and the energy reduction is between 35% and 45%.

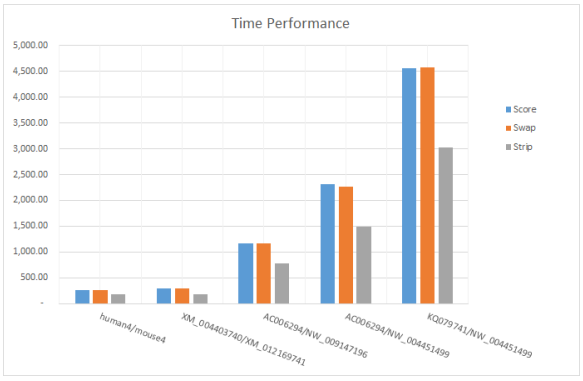


Fig. 7. Run time of scoring algorithms

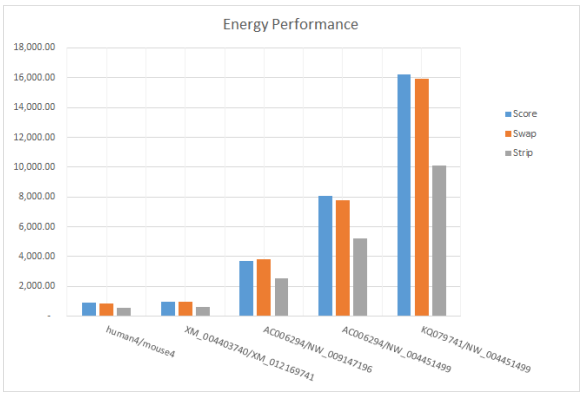


Fig. 8. Energy consumed by scoring algorithms

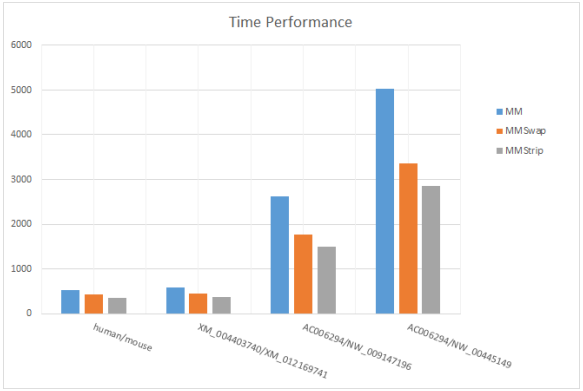


Fig. 9. Run time of alignment algorithms

B	$\|B\|$	$Score$		$Swap$		$\Delta\%$	
		Time(s)	Energy(J)	Time(s)	Energy(J)	Time(s)	Energy(J)
$KQ079794$	$1,083,068$	28.97	103.77	26.24	84.38	9.4	18.7
$GL897058$	$5,000,004$	133.74	454.43	121.09	404.15	9.5	11.1
$NW\_012187090$	$10,009,425$	267.73	943.57	242.41	791.52	9.5	16.1
$JH798151$	$50,600,503$	1353.40	5,015.51	1225.42	3960.18	9.5	21.0

A is the DNA sequence $U51865$ whose size is 968.

TABLE I. PERFORMANCE OF $Score$ AND $Swap$

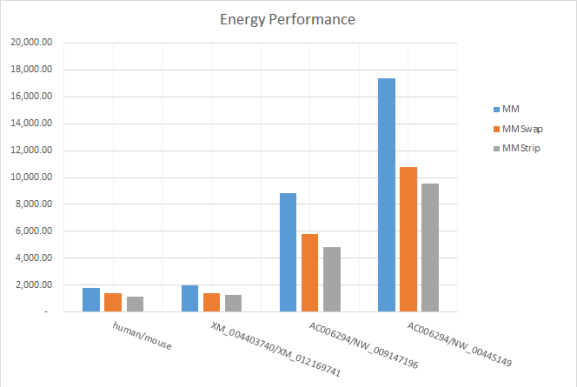


Fig. 10. Energy consumed by alignment algorithms

V. CONCLUSION

We have developed cache efficient algorithms for scoring and alignment of sequences. These cache efficient algorithms were empirically evaluated on a single node of the IBM Blue Gene/Q. This platform was selected because of the availability of an energy monitor. On our test platform, reducing cache misses translated into a significant reduction in both run time and energy consumption.

ACKNOWLEDGMENT

We are grateful to Argonne Labs for providing access to their IBM Blue Gene/Q computer and to Dr. Tania Banerjee for assistance with the energy measurement tools on this computer. This work was supported, in part, by the National Institutes of Health and the National Science Foundation under awards R01-LM010101 and NSF 1447711 .

REFERENCES

[1] S. Henikoff and J. G. Henikoff, "Amino acid substitution matrices from protein blocks," *Proc Natl Acad Sci U S A*, vol. 89, pp. 10915–10919, 1992.

[2] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, pp. 443–453, 1970.

[3] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, pp. 195–197, 1981.

[4] O. Gotoh, "An improved algorithm for matching biological sequences," *Journal of Molecular Biology*, vol. 162, pp. 705–708, 1982.

[5] D. S. Hirschberg, "A linear space algorithm for computing longest common subsequences," *Communications of the ACM*, vol. 18, pp. 341–343, 1975.

[6] E. Myers and W. Miller, "Optimal alignments in linear space," *Computer Applications in the Biosciences(CABIOS)*, vol. 4, pp. 11–17, 1988.

[7] X. Huang, R. Hardison, and W. Miller, "A space-efficient algorithm for local similarities," *Comput Appl Biosci*, vol. 6, p. 373381, 1990.

[8] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, pp. 403–410, 1990.

[9] W. Pearson and D. Lipman, "Improved tools for biological sequence comparison," *Proceedings of the National Academy of Sciences USA*, vol. 85, pp. 2444–2448, 1988.

[10] K. Chao, J. Zhang, J. Ostell, and W. Miller, "A local alignment tool for very long dna sequences," *Comput Appl Biosci*, vol. 11, pp. 147–153, 1995.

[11] A. Driga, P. Lu, J. Schaeffer, D. Szafron, K. Charter, and I. Parsons, "Fastlsa: a fast, linear-space, parallel and sequential algorithm for sequence alignment," *Algorithmica*, vol. 45, p. 337375, 2006.

[12] J. Li, S. Ranka, and S. Sahni, "Pairwise sequence alignment for very long sequences on gpus," *IEEE 2nd International Conference on Computational Advances in Bio and Medical Sciences (ICCABS)*, 2012.

[13] E. O. Sandes and A. C. M. A. Melo, "Smith-waterman alignment of huge sequences with gpu in linear space," *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pp. 1199–1211, 2011.

[14] S. Aluru and N. Jammula, "A review of hardware acceleration for computational genomics," *IEEE Design and Test*, vol. 31, pp. 19–30, 2014.

[15] S. Rajko and S. Aluru, "Space and time optimal parallel sequence alignments," *IEEETPDS:IEEE Transactions on Parallel and Distributed Systems*, vol. 15, 2004.

[16] A. Khajeh-Saeed, S. Poole, and J. B. Perot, "Acceleration of the smith-waterman algorithm using single and multiple graphics processors," *Journal of Computational Physics*, p. 42474258, 2010.

[17] S. Kurtz, A. Phillippy, A. L. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. L. Salzberg, "Versatile and open software for comparing large genomes," *Genome Biol*, vol. 5, 2004.

[18] T. Almeida and N. Roma, "A parallel programming framework for multi-core dna sequence alignment," *Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International Conference on*, pp. 907 – 912, 2010.

[19] K. Hamidouche, F. M. Mendonca, J. Falcou, A. C. M. A. Melo, and D. Etiemble, "Parallel smith-waterman comparison on multicore and manycore computing platforms with bsp++," *International Journal of Parallel Programming*, vol. 41, pp. 1110–136, 2013.

[20] S. Wallace, V. Vishwanath, S. Coghlan, J. Tramm, L. Zhiling, and M. Papkay, "Application power profiling on ibm blue gene/q," *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, 2013.

[21] S. Wallace, V. Vishwanath, S. Coghlan, L. Zhiling, and M. Papkay, "Measuring power consumption on ibm blue gene/q," *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2013 IEEE 27th International*, 2013.

[22] "Globin gene server," http://globin.cse.psu.edu/globin/html/pip/examples.html.

[23] "Ncbi database," http://www.ncbi.nlm.nih.gov/gquery.