Algorithms for clustering expressed sequence tags: the `wcd` tool

Scott Hazelhurst

University of the Witwatersrand, Johannesburg

ABSTRACT

Understanding which genes are active, and when and why, is an important question for molecular biology. Expressed Sequence Tags (ESTs) are a technology used to explore the transcriptome (a record of this gene activity). ESTs are short fragments of DNA created in the laboratory from mRNA extracted from a cell. The key computational step in their processing is *clustering*: putting all ESTs associated from the same RNA together. Accurate clustering is quadratic in time in average EST length and number of ESTs, which makes naïve algorithms infeasible for real data sets. The `wcd` EST clustering system is an open source clustering system that provides efficient implementations of key distance measures, heuristics for speeding up clustering, a pre-clustering booster based on suffix arrays, as well as parallelised implementations based on MPI and Pthreads. This paper presents the underlying algorithms in `wcd`. The code is available from `http://code.google.com/p/wcdest`.

KEYWORDS: clustering, expressed sequence tags, strings, heuristics, d^2, suffix arrays.
CR Categories: G2.2 – graph algorithms; J.3 – biology and genetics; I.5.3 Clustering.

1 INTRODUCTION

Expressed Sequence Tags (ESTs) are short fragments of DNA that can be used to study the *transcriptome*, i.e., which genes are active, and when they are active. The EST clustering problem is computationally straightforward – the mathematical definition is:

Definition 1 (EST Clustering) *Represent each EST as a vertex in a graph, placing edges between two vertices if the ESTs share regions of similarity. The clusters are the graph's connected components.*

The conceptual simplicity of the computational problem and associated algorithms hides the shortcomings of quadratic algorithms with realistic data set sizes, at least on single processors. Determining which pairs of vertices have edges between them requires a quadratic number of comparisons in the worst case, and each comparison is quadratic in the size of the underlying fragments. A program which takes 5 minutes on a set of 10 000 sequences would take almost 4 weeks with the 875 000 wheat ESTs. Bearing in mind contamination and problems with the data, real data sets often have to be processed several times, and so quadratic algorithms can be impractical.

This paper presents the `wcd` EST clustering system: `wcd` (pronounced *wicked*) is a modular clustering system that provides

- efficient implementations of the basic quadratic algorithm;
- heuristics for speeding up testing of relationships between vertices;
- a suffix array based pre-clustering algorithm that can speed up clustering;
- parallelisation for both shared memory and distributed memory architectures based on Pthreads and MPI.

The paper is organised as follows: Section 2 gives biological background and motivation. Section 3 discusses how we model sequence similarity and compute whether two sequences have overlapping regions. In particular, it presents an efficient implementation of the d^2 distance function. Section 4 discusses heuristics that can speed up the determination of overlap. Section 5 explores the use of suffix arrays to make the clustering process subquadratic. Section 6 discusses the parallelisation of `wcd` for both shared memory and distributed memory architectures. Section 7 presents related work and provides performance results. Finally, Section 8 concludes.

Email: Scott Hazelhurst `Scott.Hazelhurst@wits.ac.za`

2 BACKGROUND

Only a brief description of the underlying genetics is given here: for more details aimed at a computer science audience see [1]. Genetic information of living organisms is stored in a chain of deoxyribonucleic acid (DNA) molecules. There are four possible molecules (also called nucleotides or bases), adenine, cytosine, guanine and thymine, denoted A, C, G, T. We can thus represent the entire genetic information as a string over the alphabet $\{A, C, G, T\}$.

In eukaryotes – relatively advanced organisms such as yeast and humans – the genetic information has a complex organisation. The genome is divided into chromosomes, each of which contains *genes*, which have the actual genetic information. Genes comprise a small proportion of the DNA. Genes are not stored contiguously: typically a gene is broken into a set of *exons* interspersed with non-coding regions called *introns*.

When active, genes are used by the cell to create proteins. A copy of the genetic information on each exon in the DNA is *transcribed* (copied) into a strand of RNA. The RNAs from each strand are then joined, splicing out the introns. The new *messenger* RNA (mRNA) is then transported out of the cell nucleus into the ribosome where it is *translated* into protein, which is made from amino acids. The above description is a gross simplification – see [2] for a good overview.

EST production: To determine what mRNA (and hence protein) is produced, the mRNA in the cell can be extracted. These RNA strands provide snapshots of cell activity. By taking RNA from cells in different types of tissue, or at different stages in an organism's life cycle, or in both healthy and diseased tissue, we can start to understand the *transcriptome*: what genes are active, where and when; what functions genes have; or the molecular causes of disease.

Unfortunately, it is not easy to sequence (determine the composition of) RNA directly, and so the process is indirect and complex. The RNA is reverse-transcribed into DNA (the product is called cDNA). The cDNAs are copied several times and fragmented. Since we have multiple copies and the fragmentation is done randomly, typically there will be many fragments that overlap. These overlaps can be used to reconstruct the original sequence.

These fragments are the expressed sequence tags and are on average between 300 and 500 nucleotides in length, and so short enough to be sequenced. Once the ESTs are sequenced, the underlying RNA can be reconstructed. The problem is akin to being given the pieces of a jig-saw puzzle and reconstructing the underlying picture. There are many complications. Sequencing ESTs is an error prone process – see [3] for a fuller description. There is often missing or duplicate data. And areas of low complexity (in an information theoretic sense) may confound the process.

The EST clustering problem is the step before layout: rather than the input to the process above being one RNA, it is typically many RNA sequences – some copies of RNA from the same source, but also RNA generated from many genes. So we have a number of jig-saw puzzles all mixed up. Before reconstructing them, we work out which pieces belong to which puzzle.

The biological EST clustering problem is:

Definition 2 *Given a set of ESTs, partition the set into clusters so that each cluster contains exactly the ESTs from the same gene or RNA.*

Definition 1 models Definition 2. ESTs that overlap come from the same gene product and so vertices in our graph that have edges between them should be clustered together; and since there is a transitive relationship, all vertices that are in the same component model all the ESTs which are related by overlapping. Due to errors in the data, it may not be possible to solve the problem exactly. Moreover, we have several models to choose from: how much overlap is required? how do we measure similarity? what degree of similarity is required for an overlap? There are plausible candidates for all of these and the quality of the results depends on what choices are made. A recent review of EST clustering techniques appears in [4].

A straight-forward algorithm for performing clustering is shown below.

```
procedure Cluster($\mathcal{S}$) :
  foreach $s \in \mathcal{S}$ :
      cluster_of[$s$] $\longleftarrow$ $\{s\}$
  foreach $s, t \in \mathcal{S}$ :
      if (overlap($s, t$))
        merge(cluster_of[$s$], cluster_of[$t$])
```

This algorithm is quadratic in the number of sequences and average length of the sequences (the `overlap` function is typically quadratic). The basic `wcd` code looks like this, though an optimisation of using the union-find data structure eliminates unnecessary checks. (Also, the `overlap` function must check s against t and the reverse complement of t.)

3 SEQUENCE SIMILARITY

Given two strings x and y, finding an overlapping region means finding a substring x' of x and a substring y' of y of a suitable length such that the distance between x' and y' is below some threshold. The length of the sub-strings is known as the *window size*. The choice of window size, threshold and distance function is up to the person doing the clustering.

There are many different distance functions, some metrics, some not. There are two distance functions that are commonly used in EST clustering, edit distance and d^2 (pronounced d2), and these are both implemented in `wcd`. See [3, 5] for a broader discussion.

Edit distance is the number of edit operations (substitutions, insertions, deletions) required to transform one sequence into another. It is possible to weight different operations with different penalties. `wcd` implements a memory-efficient dynamic programming algorithm to perform local alignment (Smith-Waterman) [6]. This will not be discussed further in this paper, which will focus on the alternative, and `wcd`'s default, the d^2 distance function.

3.1 The d^2 distance function

The d^2 distance function is based upon the word composition of the sequences. *Notation:* If x is a string, then $|x|$ is its length. For two strings $y = y_1y_2 \ldots y_k$ and $x = x_1x_2 \ldots x_n$, y is a substring of x, denoted $y \sqsubseteq x$, if there is a position i such that $y = x_ix_{i+1} \ldots x_{i+k-1}$. If w is a string with $|w| \leq |x|$, $c_x(w)$ is the number of times w occurs in x. This definition allows overlapping occurrences of w. A k-word in a sequence is a substring of length k.

Definition 3 (Basic d^2 distance function)

$$d^2_k(x, y) \stackrel{\text{def}}{=} \sum_{|w|=k} (c_x(w) - c_y(w))^2 \qquad (4)$$

In the most general form, the d^2 score between two sequences x and y is defined by $d^2(x, y) \stackrel{\text{def}}{=} \sum_{k=l}^{L} d^2_k(x, y)$. However, experimental evidence has shown that fixing k is satisfactory, and commonly d^2_6 is used (we only look at words of length 6). Words can be weighted, but this is typically not done in EST clustering. In the rest of the paper, for simplicity we refer to d^2_6 just as d^2. Using only the initialisation phase of the algorithm presented below, the (global) d^2 distance function can be computed in *linear* time, which is significantly better than edit distance which requires quadratic time.

In EST clustering, the goal is not to determine whether two strings are similar, but whether two strings have sub-strings (windows) that are similar. For a fixed window length r, given two strings, x and y, we consider all pairs of windows of length r drawn from x and y, compute the d^2 score between these windows, and choose the minimum score over all pairs of windows: see Equation 5.

$$\widehat{d^2}(x, y) = \min\{d^2(u, v) : u \sqsubseteq x, v \sqsubseteq y, |u| = |v| = r\} \qquad (5)$$

If the $\widehat{d^2}$ score between two sequences is below some threshold, we say that the two sequences have an overlap. A consequence of using the *min* operator is that $\widehat{d^2}$ does not have the triangle inequality.

In describing and analysing the algorithms that follow, there are six parameters of interest (1) n, the number of sequences being clustered (the typical ranges $10^3 \ldots 10^6$); (2) m, the average length of the sequences (typical ranges $300 \ldots 500$ but can be as large as 10^4); (3) k, word length used for d^2 (typical range 6. . . 8); (4) r, the size of the windows (typically 100); (5) θ, the threshold d^2-value for two sequences to be declared similar; and (6) b, the size of our alphabet (typical value is 4).

A naïve algorithm to implement the distance function is very expensive. The cost of implementing Equation 4 is $\Theta(b^k)$, and there are $\Theta(m^2)$ windows, so the cost of implementing Equation 5 directly is $\Theta(b^k m^2)$. The algorithm presented here computes $\widehat{d^2}$ in $\Theta(m^2)$ time.

3.2 An incremental algorithm

A key observation in improving the d^2 algorithm is to make it incremental.[1] Suppose $u \sqsubseteq x, v \sqsubseteq y$. Let the words of length k be $w_0, \ldots, w_{b^k-1}$. Let $\mathcal{B} = \{0, \ldots, b^k - 1\}$ (we use this set to index all the possible words of length k).

[1] I would like to thank Zsuzsanna Lipták for explaining this approach to me, which I believe is implemented in the `d2_cluster` program. [7] gives on overview of its biological effectiveness, but I have been unable to find any published description of the algorithmics.

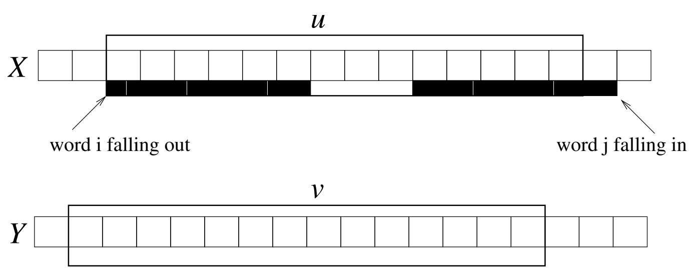

Figure 1: Effect of shifting a window up

If we know $d^2(u,v)$ and 'shift' u up x to get u', then we can compute $d^2(u',v)$ cheaply from $d^2(u,v)$. Suppose when u shifts one step to the right that word w_i falls out of the window and the word w_j falls into the window u' (see Figure 1). If $w_i = w_j$, then $d^2(u,v) = d^2(u',v)$. Otherwise, if $w_i \neq w_j$, we get $d^2(u',v) = 2(\delta_j - \delta_i + 1) + d^2(u,v)$, where $\delta_i \overset{\text{def}}{=} c_u(w_i) - c_v(w_i)$. See Figure 2 for the derivation. This means that we can compute $\delta(u',v)$ from $\delta(u,v)$ with a few additions and a shift (multiply by 2).

In a similar way, we can compute the d^2 function in an incremental way if the v-window changes. If we know $d^2(u,v)$ and v' is the next window in the string y, then $d^2(u,v') = 2(\delta_i - \delta_j + 1) + d^2(u,v)$.

The basic algorithm is shown in Figure 3. The length of the windows is `window_len`, and we refer to the window by the index of the left-most position. The counts of the word frequencies in each current window need not be kept — an array `delta` stores the difference in the counts. `delta[w]` is the difference in the number of times the word w appears in the x window and the y window. When the y window moves one step to the right, and a new word w moves into the window, we decrement `delta[w]` because we think of it as decreasing the difference in counts (of course if it's already negative, it becomes more negative). Similarly, if w moves out of the window, we increment `delta[w]`.

The algorithm works as follows. For each window in x, we compute the d^2 score of that window and the first window in y. The y-window moves rightwards, incrementally computing the d^2 score.

3.3 Novel implementations

The incremental implementation still computes the d^2 score of the initial windows explicitly for each iteration of the outer loop. This initialisation is expensive ($\Theta(b^k)$), and requires multiplications so the constant factors are large. As the window sizes are relatively large ($20-30\%$ of the sequence size on average), the initialisation costs are a significant portion of the overall cost.

`wcd` has two novel features to improve performance: (1) the number of times initialisation is done is reduced; and (2) the cost of initialisation is reduced. Together these two features change the cost of initialisation to $O(m + b^k/n^2)$ amortised time. For typical values of b, k, n, $b^k/n \ll 1$, thus the algorithm is $O(m)$ in practice.

3.3.1 Reducing the number of initialisations

The basic algorithm of Figure 3 performs initialisation once for *each* window of the first sequence. However, as shown below, it need only be done once for each pair of sequences. In the code of Figure 3, once the first iteration of the j loop has terminated, the y-window is the right-most one in y. If we now move the x-window one step to the right and reset the y-window to the leftmost window, we can no longer use the previous information and must recompute the initial d^2 score.

Our algorithm uses a zig-zag approach, leaving the y-window at the extreme right. The x-window moves one step to the right, and the *score* value is updated incrementally. Then, the y window moves *leftwards* one step at a time, incrementally updating the d^2 score, until it gets to the extreme left of the y. Then we update the x-window, and reverse the process so the basic algorithm is:

```
while more x windows to see
   scan all y windows rightwards
   update the x window
   scan all y windows leftwards
   update the x window
```

3.3.2 Reducing the cost of initialisation

This improvement is not as important as the first, but is still interesting. The direct way initialising is to build up the *delta* table for the first pair of windows, and then to sum the squares of all the entries. This is $O(b^k + r)$. However, initialisation can be computed more cheaply inductively.

$$
\begin{aligned}
d^2(u',v) =& \sum_{\ell\in\mathcal{B}}(c_{u'}(w_\ell)-c_v(w_\ell))^2 \\
& \text{[By the definition of } d^2.] \\
=&(c_{u'}(w_i)-c_v(w_i))^2+(c_{u'}(w_j)-c_v(w_j))^2+\sum_{\ell\in\mathcal{B}-\{i,j\}}(c_u(w_\ell)-c_v(w_\ell))^2 \\
& \text{[Since for } \ell\neq i,j, c_{u'}(w_\ell)=c_u(w_\ell).] \\
=&(c_u(w_i)-1-c_v(w_i))^2+(c_u(w_j)+1-c_v(w_j))^2+\sum_{\ell\in\mathcal{B}-\{i,j\}}(c_u(w_\ell)-c_v(w_\ell))^2 \\
& \text{[By definition of } u,\ u'\text{. Now multiplying out, rearranging and factoring:]} \\
=&(c_u(w_i)-c_v(w_i))^2-2(c_u(w_i)-c_v(w_i))+1\ +\ (c_u(w_j)-c_v(w_j))^2+2(c_u(w_j)-c_v(w_j))+1 \\
&+\sum_{\ell\in\mathcal{B}-\{i,j\}}(c_u(w_\ell)-c_v(w_\ell))^2 \\
=&2((c_u(w_j)-c_v(w_j))-(c_u(w_i)-c_v(w_i))+1)+\sum_{\ell\in\mathcal{B}}(c_u(w_\ell)-c_v(w_\ell))^2 \\
=&2((c_u(w_j)-c_v(w_j))-(c_u(w_i)-c_v(w_i))+1)+d^2(u,v) \\
=&2(\delta_j-\delta_i+1)+d^2(u,v) \text{ where } \delta_i\overset{\text{def}}{=}c_u(w_i)-c_v(w_i)
\end{aligned}
$$

Figure 2: Derivation of $d^2(u',v)$

- If $r=k$ (window length equals word length) the d^2 score is easy to compute: it is 0 if both windows contain the same word, and 2 otherwise.
- Suppose we have two windows u,v of size r and have computed $d^2(u,v)$. Let u' (v') be the window obtained by enlarging u (v) in x (y) by one character to the right. New words, w_i and w_j, appear in u' and v' that don't appear in u and v. Both u' and v' are of size $r+1$.
- If $w_i=w_j$, then $d^2(u',v')=d^2(u,v)$. Otherwise, by similar reasoning to that of Section 3.2, $d^2(u',v')=2(\delta_i-\delta_j+1)+d^2(u,v)$ [8].

This means that initialisation can be done by computing the d^2 score of a pair of windows that each contain exactly one word, and then growing the windows until they are the required size. This cost is $O(r)$ which is also $O(m)$.

There is still initialisation that is $O(b^k)$ if done in the straightforward way. The *delta* table must still be initialised to all zeroes, and there are $O(b^k)$ entries. This can be avoided with a little trouble (see [9, Ex. 12.1.4, p. 221]), but the following three factors together militated against this approach, in favour of a simpler approach, which though asymptotically not as good, will lead to better performance in real data sets: (1) the simpler algorithm has much smaller constant factors; (2) the desire to make access to the *delta* table as quick as possible; and (3) the d^2 computation is called very many times in a typical run during clustering (in fact $O(n^2)$ times).

`wcd` zeroes the *delta* table once, before the d^2 algorithm is ever called; this is $O(b^k)$. The d^2 function is then called $O(n^2)$ times. An auxiliary array stores all the words seen in a particular call of d^2. Once the d^2 score has been computed, we need just clear the entries in the *delta* table that were used by the call of d^2. This is $O(m)$. Thus the total work done on doing d^2 initialisation calls is $O(b^k+n^2m)$. Hence the amortised work done per initialisation is $O(b^k/n^2+m)$. Since for real values of b,k and n, $b^k/n^2\ll 1$, the cost of the algorithm is $O(m^2)$ in practice.

4 HEURISTICS

Computing d^2 is $O(m^2)$ and with $O(n^2)$ comparisons for an all-versus-all clustering — this is not practical for real data. For this reason, `wcd` 0.3 uses two heuristics to avoid doing comparisons where possible. These heuristics have been based upon empirical studies to show that they are sound (though there is a need for theoretical work). The basic idea of a heuristic is that be-

```
function compute_D2(x, y)
  low_score = maxint
  foreach i in range(0, num_windows(x))
     score = ComputeInitWindowPair(i, 0)
     low_score = min(score, low_score)
     foreach j in range(0, num_windows(y) − 1)
        new_word = y[j + window_length]
        old_word = y[j]
        score = score − 2 ∗ delta[new_word] + 1;
        delta[new_word]--;
        score = score + 2 ∗ delta[old_word] + 1;
        delta[old_word]++;
        low_score = min(score, low_score)
```

Figure 3: Basic Incremental Algorithm

fore computing $d^2(x,y)$, we compute a heuristic function $h(x,y)$. If $h(x,y)$ is greater than some threshold, then we are guaranteed (empirically) that $d^2(x,y) \not\leq \theta$.

Common 6-word heuristic: An obvious heuristic is that the two sequences must share at least a certain number of common words. This can be checked in linear time by building up a table. The question is how long the words should be, and how many words are required before proceeding to a full check. The longer the words, the greater the threshold, the cheaper clustering will be, but the more likely errors are to be made. Since, d^2 uses 6-words as building blocks, the initial experimentation focused on 6-words, but these prove to be too crude.

For two regions to have a d^2 threshold of 40, theoretically they must share at least 54 common 6-words. Experimentation with a sample of the SANBI Benchmark 10000 set showed that the minimum number of 6-words shared by two sequences which contain windows that meet a d^2 threshold of 40 is 74. Of all the pairs of sequences (roughly 500M) that need to be compared, 704k pairs had a common 6-word threshold greater than 74 (about 1.6% of the pairs).

***t*/*v*-heuristic:** The 6-word heuristic ignores locality. Similarity can be over-rated by having either (1) lots of common words so close that they overlap (2) or having relatively few common words spread evenly through long sequences. This will particularly be a problem in sets that contain full-length mRNAs where common words are likely to occur by chance. What we want are a reasonable number of common words spread over a window of size approximately 100. The idea of the t/v-heuristic is to require the common words be found reasonably close to each other but not too close to each other. The rule of this heuristic is, given a threshold t, when comparing sequences i and j:

- Consider all v-words appearing in i;
- At least t of these v-words must appear in j without overlap within 100 base pairs of each other.
- If there at least t v-words, the heuristic passes and then `wcd` tests the d^2 score; if not, the pair is not considered further.

If we compare two sequences using this heuristic and fix a v, the smallest t for which the t/v-heuristic passes is known as the pair's t/v-score. `wcd` uses by default the 5/10-heuristic, so the $t/10$ score must be at least 5 (the user can use other parameters). To make this linear time, it is not symmetric – there are no restrictions about where the common words appear in i, only in j.

Sample heuristic: The 5/10-heuristic improves the performance significantly. However, although the test is linear, there are high constant factors. The u/v-sample heuristic says:

- Consider all the v-words in sequence i.
- Consider every 16-th v-word in sequence j.
- At least u of these words must be common. If so, apply the t/v-heuristic.

The default sample heuristic in `wcd` is the 2/10-sample heuristic. The choice of 16 as the sampling frequency is determined by the representation of data. `wcd` represents the sequences internally in compressed format (2 bits per base=4 bases per byte). In the core of the d^2 algorithm, as the zig-zag algorithm progresses, we have to

	d^2	5/10	Sample
Percent pass	0.013	0.32	0.29
Time taken by `wcd`	7100	2108	231

Table 1: Performance of heuristics on the SANBI Benchmark 10000. The d^2 column shows what happens if the heuristics are not used; the 5/10 column shows the effect of using the 5/10 heuristic; the sample column shows the effect of using both heuristics.

extract out the bases one by one by doing bit-level operations. This can be done without compromising efficiency provided we are processing all the bases in a sequence. However, if we stride through the sequence, it is only possible to extract out every 8-th character efficiently (or every 16th, 32nd etc). Experimentation showed that 16 was a good choice for stride.

The 2/10-sample heuristic is very effective and conservative. Preliminary experimentation has shown that other parameters can lead to significant improvement with minimal impact on quality. For example, the 4/8-sample heuristic can lead to a 2-3 fold improvement in performance.

Empirical analysis: Empirical analysis of the SANBI Benchmark 10000 EST set shows that of the possible 49 995k possible pairwise comparisons:

- the d^2 test passes 6464 times,
- the 2/10-sample heuristic passes 161 108 times,
- the 5/10-heuristic passes 143 243 times. Table 1 shows the performance of `wcd` on the benchmark. The first row shows what percentage of the comparisons pass the given test. The second row shows the costs of doing the clustering using (i) only d^2, (ii) the 5/10-heuristic and only using d^2 on those that pass and (iii) the sample heuristic and then using the 5/10 and d^2.

Running `wcd` on this data set shows that of the total number of 499 995 000 possible comparisons:

- The total number of times 2/10-sample heuristic called 483 715 670 (the difference is due to the use of the union-find and also a few under-length sequences are ignored);
- The sample heuristic succeeds 156 823 times, and calls the t/v-heuristic for these cases;
- The t/v-heuristic succeeds 83 202 times, and calls d^2 for these cases;
- The d^2 function succeeds 2155 times.

Correctness of the heuristics The danger of the heuristics is that too aggressive parameters for the heuristics will lead to incorrect clustering. Our experimentation showed that the use of the 5/10-heuristic and the 2/10-sample heuristic lost no accuracy in the clustering. More specifically: the default parameters of `wcd` (5/10) gave no loss of accuracy; using the 7/10-heuristic lost no accuracy; using the 8/10-heuristic affected about 10 sequences; and using the 10/10 heuristic affected about 50 sequences.

This was tested on the Glossina MG test set too. The set was too large to do an exhaustive comparison. A random sample of pairs of sequences was chosen so that 580k comparisons were made in the forward direction and 580k comparisons of the first sequence to the reverse complement of the second. Note that the default d^2 threshold is 40. When we graph d^2 score versus $t/10$-score, the line that cuts the plane is roughly

$$t/10\text{-score} = 10 - d^2/12.$$

Table 2 shows this more precisely: it shows for all the pairs of sequences with the given $t/10$ score, the lowest d^2 obtained by those pairs. In summary, the default $t/10$-score of 5 is conservative. The table shows that a more aggressive strategy is acceptable. A more detailed view of the data shows: of the 52k pairs with a t/10-score of 4, *only 1* had a d^2 score less than 70. Only 46 had a d^2 score less than 80; of the 61k sequences with a t/10-score of 5, *only 1* has a d^2 score less than 60, 36 less than 65, and 229 (0.37%) less than 70; of the 42k pairs with a t/10 score of 6, only 12 had a d^2 score less than 50; and for no occurrence of the 2/10-sample heuristic failing was there a d^2 score of 60 or less.

Similar results hold for the sample heuristic as shown in Table 3. This table shows the effect of using different sample heuristics, using 2/10-sample as a base, since this does not reduce accuracy. The time column shows how long the clustering takes as a ratio of the time taken for the default; the Jaccard index (JI) [10] and Matthews correlation co-efficient (CC) [11] are commonly used measures[2] of the differences between clusters. To put this into perspective, if we used a

[2]Note, the specificity in all cases is 1 since ultimately `wcd` will only cluster sequences together if the d^2 distance is under the threshold. Since the specificity is 1, the Jaccard index is equal to sensitivity. The 4/8-sample heuristic is faster than the 4/10-sample heuristic which is not expected since the 4/10 heuristic is more aggressive. The most likely explanation for this is cache performance. Implementing the sample heuristic requires a table of size 4^v. For $v = 8$, the table would fit easily in the cache of the computer used,

t/10-score	0	1	2	3	4	5	6	7	8
smallest d^2 score	128	114	102	88	64	58	46	30	24

Table 2: t/10 score versus d^2 score

d^2 threshold of 45 rather than 40 (which is an equally plausible threshold to use), then the Jaccard index of comparing the clusterings produced with 40 and 45 is 0.84. The effect of changing the threshold slightly has a *far* more profound effect on the quality of the clustering than changing the quality of the heuristics.

Sample Heuristic	Time	JI	CC
2/10	1.00	1.0000	1.0000
3/10	0.65	0.9995	0.9997
4/10	0.57	0.9720	0.9859
3/9	0.42	0.9995	0.9998
4/9	0.30	0.9759	0.9879
4/8	0.31	0.9784	0.9892
6/8	0.16	0.9364	0.9677

Table 3: Effect of the use of different sample heuristics using the 2/10-sample heuristic as the base case

Other relevant results are shown in Table 4 and Figure 4 which shows the distribution of d^2 scores between pairs of sequences. These results show that the default parameters for `wcd` are conservative and allow the user considerable latitude to experiment.

d^2 score	Histogram
0-9	3577
10-19	1187
20-29	799
30-39	780
40-49	861
50-59	1249
60-69	2353
70-79	4738
80-89	9534
90-99	17700
100-109	29406

Table 4: Partial histogram of d^2 scores

for $v = 10$, not. This shows that algorithm analysis has to take into account architectural concerns. Here, the cache performance outweighs the advantage of a more aggressive heuristic.

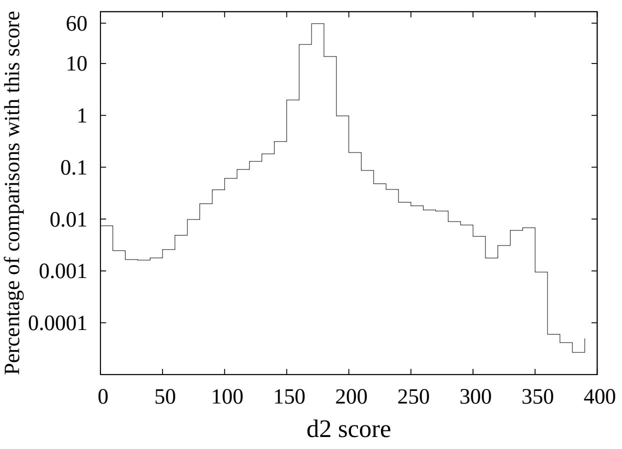

Figure 4: Histogram of SANBI Benchmark 100000 d^2 scores (log scale): x-intervals of size 10. For each interval, the percentage of all pairwise comparisons which yield d^2 scores in this interval.

5 A SUFFIX ARRAY BOOSTER

The success of the heuristics shows that a common word heuristic can be very successful in improving the performance of the clustering. The heuristics tackled the problem that comparing individual pairs of sequences is $O(m^2)$. This section describes the *reclustering* feature of `wcd` and builds on the common word heuristic to tackle the problem that doing pairwise comparisons requires $O(n^2)$ comparisons.

The heuristic used is that we cluster two sequences if they share a single common word of length k. Our algorithm becomes:

```
foreach $s \in \mathcal{S}$ : cluster_of[$s$] $\longleftarrow \{s\}$
    words $\longleftarrow$ list of unique $k$-words in $\mathcal{S}$
    foreach $w$ in words :
            currseqs $\leftarrow$ sequences that contain $w$
            merge(cluster_of[currseqs$_1$], ...,
                    cluster_of[currseqs$_r$])
```

The choice of k is crucial: too short a k and sequences are clustered together that should not be; too large, then too conservative the clustering. Obviously for two sequences to have windows that are close together as measured by d^2, they must share a common word of some minimum length. Unfortunately, the theoretical minimum length is well under 20, which leads to very poor clustering. Our empirical testing shows that 25-30 is a safe range to use.

The efficiency of the algorithm relies on finding the list of unique k-words and the sequences that contain them. `wcd` uses a suffix array [12] representation of the data file to keep the list of words and to cluster efficiently using the algorithm above. `wcd` does not construct the suffix array itself. `wcd` is distributed with a Python script that prepares an input sequence file for the `sary1.2.0` package [13], which produces the suffix array. The suffix array produced is 8 times bigger than the input sequence file (the suffix array requires 4 bytes per byte of input and we need to index the file and the reverse complement of the file). For large data files this can easily be in the multi-gigabyte range. This is not a problem from a RAM perspective since `wcd` processes the suffix array linearly, on-the-fly and so only 2 words of the array need to be kept in memory at a time. A more serious limitation is that `sary` is designed for 32-bit machines which makes 2G a hard limit for `sary`. We have produced scripts that can take a larger file, split it, run `sary` separately and then merge the resulting files. However, this increases the burden on the user and is difficult to make transparent without introducing errors.

The goal of using the suffix array approach is not to produce a final clustering but to produce a pre-cluster or super-cluster. If k is small enough, real clusters are all sub-sets of super-clusters. If k is big enough, super-clusters will be significantly smaller than the entire set. Since clustering is quadratic, even if the biggest cluster is 40% of entire set, speed-up can be significant.

The output can then be re-input into `wcd` using the *recluster* algorithm to produce a finer clustering.

A secondary use of the suffix array clustering is for exploring data which may contain vectors or repeats which produce very large super-clusters. This is often an indication of problems with the data and using this mode allows problems to be found quickly.

Experimental results: Performance results of some sample data sets is shown in Table 5. The data was trimmed using `trimest` to remove poly-A tails and repeats of longer than 13 bases were removed.

These results indicate that the suffix array can provide a significant speed up with minimal impact on quality of results. By changing the parameters, more aggressive clustering can be found.

However, the suffix array approach does not always work, especially when the data has repeats, does not have its poly-A tail trimmed and most especially when the low complexity regions are not removed. This results in the initial clustering done using the suffix array producing very large 'superclusters', and so the reclustering takes almost as long as doing clustering from scratch. But even in this case it may be worthwhile using the suffix array clustering. It is cheap to do, and often the fact that there are large super-clusters indicates that there may be problems with the data. For example, we have been able to use this to good effect with a large metagenomic dataset: the initial suffix array clustering with a word length of 100 obtained a very large super-cluster; we could then remove the sequences from this super-cluster and then cluster the remaining sequences, which produced some useful results. This reduced turn-around time from days to hours.

6 PARALLELISATION AND PERFORMANCE

This section discusses the parallelisation of `wcd` and provides performance results.

6.1 MPI Parallelisation for clusters of workstations

EST clustering is an obvious candidate for parallelisation. With a quadratic amount of calculation for a linear amount of communication, it is possible to parallelise easily.

There have been several `wcd`-based parallelisation approaches. The first (and still provided in `wcd`) assumes an NFS-style shared disk between a cluster of workstations. This is crude but can work well, and was used in the clustering of large mammalian EST sets.

Ranchod parallelised `wcd` on a cluster of workstations [14]. His approach used a master-slave architecture with a dynamic workload allocation mechanism using UDP as the communication mechanism. A heartbeat protocol was used to detect nodes that died. Data sets of between 10k and 300k ESTs were tested with good results.

Although Ranchod's approach was successful, it was decided to re-implement the parallelisation using MPI. The primary reason for this is to make it more portable and easier to install, manage and use. It also has secondary benefits of being potentially more extensible, and able to run across different networks. A master-slave architecture was adopted. The master distributes the sequences to

Step	SANBI 10000	Public Cotton	Drosophila
Building a suffix array	75	128	598
Clustering using suffix array	12	68	202
Reclustering	75	751	1002
Total	162	947	1802
Normal sequential	197	2157	15861

Table 5: Performance of suffix array booster

the slaves, which statically determine their workload. The slaves independently cluster a sub-set of the sequences and when they complete their tasks, they linearise their local union-find structures and send it back to the master which merges the structures.

The limitation of `wcd` 0.3 is that work allocation is done statically and that the master is idle for most of the time. Static work allocation works reasonably well in a homogeneous environment and has not been a priority to extend; for a heterogeneous environment, a dynamic work allocation scheme is highly desirable. The restriction that the master only does the coordination is not a serious issue since it can be scheduled to run on the same processor as a slave. The key metric is the efficiency of parallelisation (speed-up divided by number of processors). Ideally this should be close to 1.

Performance results. Some performance results are shown below; results on other data sets and architectures are reported in [15], and are broadly in line with the figures shown here. Table 6 and Figure 5 show the time taken and efficiency of clustering four data sets using MPI in terms of number of processors used. The four data sets are:

- C01, C05 and C10 are three artificially generated EST sets. By generating three artificial sets with similar properties we can explore the impact of size alone on time taken. C10 is a set of 125k ESTs totalling 60.9M in total; C05 is half the size of C10; C01 is one-tenth the size of C10.
- A32: This is a set of 76846 *Arabidopsis thaliana* ESTs with approximately 32M of sequence data.

The experiments ran `wcd` 3.2 on the iQudu cluster of the Centre for High Performance Computing (160 nodes, each with 16G of RAM and 4 2.6GHz AMD 2218/stepping 2 CPUs). Table 6 shows the time[3] in seconds for each data set with different numbers of slaves. Figure 5 shows the efficiency of the parallelisation (bear in mind for C01 that at 31 slaves, the total time taken is 9s).

#	1	7	15	31	63	127	255
C01	230	35	18	9	6	5	12
C05	5980	916	441	214	110	59	40
C10	24703	3615	1701	832	420	218	126
A32	7246	1204	585	286	147	78	53

Table 6: Computational cost of parallelising using MPI. The columns show the number of slaves used. The rows are labelled with the data sets used. The time is shown in seconds.

6.2 Pthreads parallelisation

A pthreads-based parallelisation has also been done, suitable for shared memory architectures. In the current implementation[4], work allocation is done using a dynamic scheduler with slaves requesting work when they no longer have tasks to do. The work allocation process attempts to allocate chunks of work to slaves so that slaves are not accessing the same memory.

With dual and quad-core processors becoming common-place, this mode of parallelisation will be important for even low-end systems. Multicore and manycore architectures will become more popular in the future and more traditional SMP architectures available at the high-end.

The main testing of this was done on an IBM 7040-681 SMP Server with 32 1.9GHz Power4+ CPUs, 128G of RAM and an Intel dual quad-core processor (2.33GHz E5345, 4M of L2 cache per processor). The results generally are respectable though the scale-up is not as good as hoped for,

[3]On a dedicated machine or cluster, there is little variation in the time taken to cluster data sets. On a production cluster running several jobs under the control of a job submission system like LoadLeveler, there is often variation of 30% due to contention of resources. The figures shown here are generated by running the same job several times and choosing the *minimum* time for each job. This is fair because this is the best indication of performance on a dedicated system.

[4]The coding for version 0.3 of `wcd` was largely done by Richard Starfield.

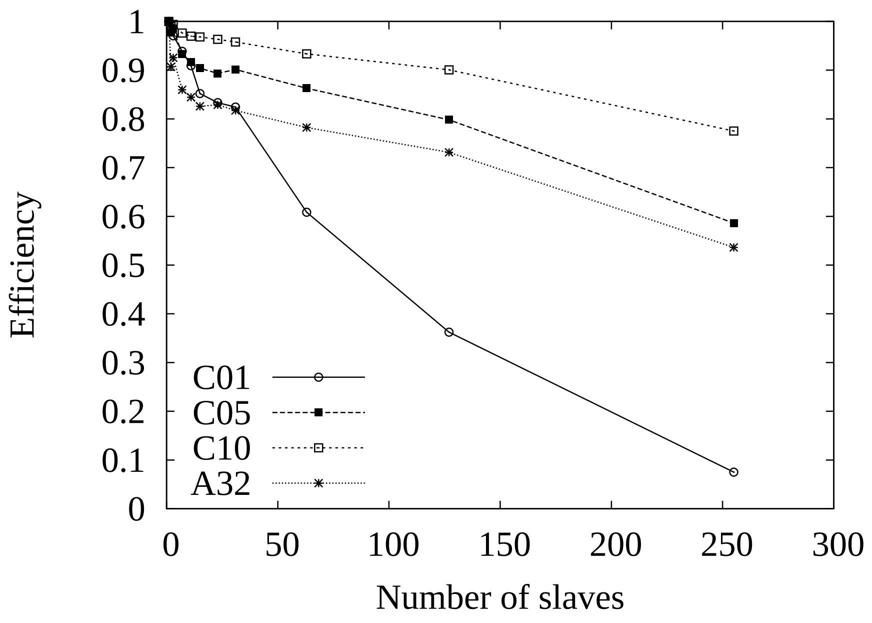

Figure 5: Efficiency of parallelisation on the CHPC iQudu cluster.

and this is worth further research. Note the architecture has an impact on scaling. Synchronisation costs can be improved, but the major reason for poor scaling appears to be cache contention. Table 7 gives the performance on the C05 data set, which are broadly in line with the results on other data sets.

6.3 Memory

`wcd` has a very efficient internal representation of data (four bases per byte). In addition, each sequence requires 64 bytes. The auxiliary data structures are large (tens of megabytes), but are independent of input file size. This means that for large data files, the internal RAM used is less than the data file size. For example, a 375M data file of 686k ESTs has a resident set size of 165M (total virtual memory of 289M). Moreover, all dynamic memory allocation is done shortly after initialisation. The memory usage means `wcd` is suitable for clustering large data sets on a small number of machines.

Under Pthreads, the main data structures are shared by the threads, with some auxiliary data structures being duplicated. In MPI mode, each MPI process has its own copy of the data structures.

7 RELATED WORK

As EST clustering is an important application with different variations, there are a number of clustering tools. Comparing results based on published results is difficult for two reasons. First, comparing computational efficiency is difficult for the common reason that different architectures and systems have been used, confounding the effect of different algorithms and data structures. Raw chip speed is an extremely poor proxy for architecture performance. Preliminary experimentation with `wcd` showed almost an order of magnitude difference in performance on chips with similar chip speed, probably due to different cache size [15].

A more fundamental problem is that the different systems actually produce different biological results. The way in which Definition 1 models Definition 2 leaves a lot for the algorithm developer to play with. This is a tremendous weakness of the papers in the literature (including this paper). For example, as described above, changing `wcd`'s heuristic parameters slightly changes the performance by over a factor of 4 and produces only a slightly different clustering (Jaccard index 0.98). So, by making very small changes to the biological output, we can change the performance dramatically. These differences can easily be hidden by changing other parameters. By manipulating the parameters discussed (as well as others such as only sampling windows), we can probably improve the performance of `wcd` by an order of magnitude and change the quality of the clustering by less than making relatively small changes in the clustering threshold. A final practical difficulty in comparison is that with the exception

Number of CPUs	1	2	3	4	5	6	7	8	9	10
7040-681										
Time (s)	7543	3580	2792	2041	1912	1411	1419	1211	1140	1121
Efficiency	1.00	1.05	0.90	0.92	0.80	0.89	0.75	0.78	0.73	0.67
$2 \times$ 4-core E5345										
Time (s)	3136	1731	1314	1053	1064	1046	1063	1078	–	–
Efficiency	1.00	0.91	0.80	0.74	0.59	0.50	0.42	0.36		

Table 7: Cost of running wcd 0.3 on the C05 data set with different number of processors

of the SANBI 10000 set, there are few publicly available benchmarks, so different researchers test their programs on different data. Much of this data is in public EST databases, but these grow dramatically. For these reasons, a proper exploration beyond the scope of this paper, and is part of our ongoing research which we hope to publish in a companion paper.

Some tools primarily aim at supervised or seeded clustering. For organisms such as humans for which the entire genome is known, clustering can be done by approximate matching of the ESTs against the genome. TIGR is a good example of such a system [16, 17]. Other biological data can be used. For example, Phrap – an assembly tool – uses clone-linking to improve the clustering process [18]. If we know that two ESTs come from the same clone, we pair them immediately. If the number of source mRNAs is known then a modified k-means clustering can be done either in clustering or assembly [19]. However this is not usually the case, and chimeric data can confound the outcome.

Clustering can also be accomplished by sequence assembly programs such as Phrap, the TIGR assembler [20] and CAP3 [21] though this is usually at a significant computational cost overhead. There are also EST clustering algorithms based on hash tables [22] and finite state machines [23].

The d^2 distance function was originally proposed by Torney *et al.* and validated biologically for EST clustering by Hide *et al.* [24]. The `d2_cluster` program, originally written by Burke *et al.*, also performs d^2 clustering. However, it is a proprietary system and the details of the algorithm implementation are not published. Carpenter *et al.* parallelised the algorithm on an SMP architecture using an SGI Origins 2000 making comparison difficult [25].

The suffix-array booster: There are other algorithms based on suffix trees rather than suffix arrays. Suffix trees admit much more sophisticated algorithms. Kalyanaraman *et al.* [11] implemented the PaCE tool, which builds a generalised suffix tree to identify possible matches (based on a single common word heuristic) and then uses a dynamic programming based algorithm to validate matches. This is parallelised on a distributed architecture, and they achieve very good scaling. They benchmarked the quality of the their output to known clusterings which produces a clustering “close” to that of the quality of CAP3. These results partially contradict our results reported above, which showed that the single word heuristic is not a good heuristic by itself. The memory overhead is large, 80 bytes per byte of input on a 32-bit machine, more on a 64-bit machine. For clustering the 17M Public Cotton Data set (a small set) on a single machine, PaCE required over 2G of RAM.

Another good example of the suffix-array based approaches is that of Malde *et al.* [26]. A direct comparison with the algorithm of Malde et al. is difficult. Their *xsact* algorithm is much more sophisticated and is intended to produce a final clustering. Although their work is “promising” (their description) the quality of the clustering produced was not top quality – our goal is to very closely approach the quality of d^2 clustering. Performance-wise, it is difficult to assess too since their published results were on a different architecture and only for small data sets. As a direct comparison on our Intel E5355 server with 4G of RAM, `wcd` 3.2 clustered the Benchmark 10000 (a very small set) in 93s using less than 2% of memory. *xsact* took 37s but used 37% of memory. The biological quality of the clustering was significantly poorer than `wcd`. Moreover, with larger files, like the Public Cotton set, *xsact* was not able to run due to memory limitations and is not practical for large input.

On the other hand, `wcd` has a memory usage of 64 bytes per sequence plus 0.25 bytes per byte of input (about two orders of magnitude smaller than PaCE). This means that memory is not a bottleneck. We have successfully clustered an EST set of 800M on such a machine (even if it takes a few days with only one processor).

Suffix array based approaches *are* promising but they need more work. At the moment they are a useful tool in the repertoire, but currently parallelisation appears to be a better, if brute force approach. As memory becomes larger, suffix tree based approaches may become feasible though sensible cache behaviour will remain a challenge. One of the advantages of parallelisation is that it allows RAM on many systems to be used [11].

8 CONCLUSIONS

This paper has presented the `wcd` EST clustering system. The key contributions have been:

- the development and presentation of an efficient algorithm for computing the d^2 distance function;
- new heuristics for efficient screening of comparisons and an experimental evaluation of their effectiveness;
- a suffix-array based booster which has modest RAM requirements;
- parallelisation for different memory architectures

The algorithm and heuristics presented here have been implemented. The program source and documentation are available under an open source licence (`http://code.google.com/p/wcdest`).

Future research

The most pressing need for future research are better techniques for assessing the biological efficacy of the clustering. It is clear that seemingly minor changes to parameters used can change the cost of clustering by an order of magnitude or more. More benchmarks are needed and a qualitative analysis of the effect of different errors needs to be made. Without this, it will be difficult to compare different approaches or evaluate new techniques.

The parallelisation techniques work well but there is room for improving them all significantly and unifying the way in which parallelisation is done.

Other approaches currently being investigated for `wcd` are new heuristics based on common word frequency, as well as index-based approaches based on the work of Giladi [27].

Acknowledgements This research received financial support from the National Research Foundation (GUN2053410) and the National Bioinformatics Network. Part of this work was done during a visit to SANBI. Winston Hide and Zsuzsanna Lipták provided significant help and suggestions at various stages of the development of `wcd`. I thank Anton Bergheim, Pravesh Ranchod and Richard Starfield for their help and contributions. Stephen Levitt and the anonymous reviewers gave useful feedback. Peter van Heusden integrated `wcd` into StackPACK. The following users of `wcd` provided data and feedback and tolerance: Winston Hide, Tania Hide, Janet Kelso, Mario Jonas, Gwen Koning (SANBI); Alison Murray and Joe Grzymski (Desert Research Institute); Chris Maher (Cold Spring Harbor/Stanford); Mao Xizeng (Peking University); Julian Schwerdt (Adelaide) and Dave Edwards (Primary Industries Research Victoria, Australia). Some of the experimental work was carried out on equipment of the South African National Bioinformatics Institute and the SA Centre for High Performance Computing (thanks to Jeff Chen and Jeremy Main). Section 3 and small parts of sections 2 and 4 appeared in SAICSIT 2004 as a short paper.

REFERENCES

[1] J. Cohen. "Bioinformatics — an introduction for computer scientists". *ACM Computing Surveys*, vol. 36, no. 2, pp. 122–158, 2004. ISSN 0360-0300.

[2] M. Gerstein, C. Bruce, J. Rozowsky, D. Zheng, J. Du, J. Korbel, O. Emanuelsson, Z. Zhang, S. Weissman and M. Snyder. "What is a gene, post-ENCODE?: History and updated definition". *Genome Research*, vol. 17, pp. 669–681, Jun. 2007.

[3] J. Zimmermann, Z. Lipták and S. Hazelhurst. "A Method for Evaluating the Quality of String Dissimilarity Measures and Clustering Algorithms for EST Clustering". In *Proceedings of the Fourth IEEE Symposium on Bioinformatics and Bioengineering (BIBE' 04)*, pp. 301–309. IEEE Computer Society Press, May 2004.

[4] S. Nagaraj, R. Gasser and S. Ranganathan. "A hitchhiker's guide to expressed sequence tag (EST) analysis". *Briefings in Bioinformatics*, vol. 8, no. 1, pp. 6–21, 2007.

[5] S. Vinga and J. Almeida. "Alignment-free sequence comparison – a review". *Bioinformatics*, vol. 19, no. 3, pp. 513–524, 2003. doi: 10.1093/bioinformatics/btg005.

[6] D. Gusfield. *Algorithms on strings, trees, and sequences.* Cambridge University Press, Cambridge, United Kingdom, 1997.

[7] J. Burke, D. Davison and W. Hide. "D2_cluster: A Validated Method for Clustering EST and Full-length cDNA Sequences". *Genome Research*, vol. 9, no. 11, pp. 1135–1142, 1999.

[8] S. Hazelhurst. "An efficient implementation of the d^2 distance function for EST clustering: preliminary investigations". In *SAICSIT '04: Proceedings of the 2004 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, ACM International Conference Series, pp. 229–233. 2004.

[9] T. Cormen, C. Leiserson and R. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[10] M. Halkidi, Y. Batistakis and M. Vazirgiannis. "Clustering Algorithms and Validity Measures". In ssdbm 2001: *Thirteenth International Conference on Scientific and Statistical Database Management*, p. 0003. 2001.

[11] A. Kalyanaraman, S. Aluru, V.Brendel and S. Kothari. "Space and Time Efficient Parallel Algorithms and Software for EST Clustering". *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 12, pp. 1209–1220, Dec. 2003.

[12] U. Manber and G. Myers. "Suffix arrays: a new method for on-line string searches". *SIAM Journal on Computing*, vol. 22, no. 5, Oct. 1993.

[13] S. Takabayashi. "Sary: a suffix array library and tools", 2005. `http://sary.sourceforge.net/`, Last accessed 30 April 2007.

[14] P. Ranchod. "Parallelisation of EST Clustering". MSc Dissertation, School of Computer Science, University of the Witwatersrand, 2005.

[15] S. Hazelhurst. "Computational Performance Benchmarking of the wcd EST Clustering System". Technical Report TR-Wits-CS-2007-1, School of Computer Science, University of the Witwatersrand, May 2007.

[16] F. Liang, I. Holt, G. Pertea, S. Karamycheva, S. Salzberg and J. Quackenbush. "An optimized protocol for analysis of EST sequences". *Nucleic Acids Research*, vol. 26, no. 18, pp. 3657–3665, 2000.

[17] G. Pertea, X. Huang, F. Liang, V. Antonescu, R. Sultana, S. Karamycheva, Y. Lee, J. White, F. Cheung, B. Parvizi, J. Tsai and J. Quackenbush. "TIGR Gene Indices clustering tools (TGICL): a software system for fast clustering of large EST datasets". *Bioinformatics*, vol. 19, no. 5, pp. 651–652, 2003.

[18] P. Green. "Documentation for Phrap and Cross_match", 1999. `http://www.phrap.org`, Last accessed 24 April 2007.

[19] H. Otu and K. Sayood. "A divide-and-conquer approach to fragment assembly". *Bioinformatics*, vol. 19, no. 1, pp. 22–29, 2003.

[20] G. Sutton, O. White, M.D.Adams and A. Kerlavage. "TIGR Assembler: A New Tool for Assembling Large Shotgun Sequencing Projects. Genome Science and Technology". *Genome Science and Technology*, vol. 1, pp. 9–18, 1995.

[21] X. Huang and A. Madan. "CAP3: A DNA sequence assembly program". *Genome Reseach*, vol. 9, no. 9, pp. 868–77, Sep. 1999.

[22] R. Mudhireddy, F. Ercal and R. Frank. "Parallel hash-based EST clustering algorithm for gene sequencing". *DNA Cell Biology*, vol. 23, no. 10, pp. 615–23, Oct. 2004.

[23] G. Slater. *Algorithms for the Analysis of Expressed Sequence Tags.* Ph.D. thesis, University of Cambridge, 2000.

[24] W. Hide, J. Burke and D.Davison. "Biological Evaluation of d^2, an algorithm for high-performance sequence comparison". *Journal of Computational Biology*, vol. 1, pp. 199–215, 1994.

[25] J. Carpenter, A. Christoffels, Y. Weinbach and W. Hide. "Assessment of the Parallelisation approach of d2-cluster for High-Performance Sequence Clustering". *Journal of Computational Chemistry*, vol. 23, pp. 1–3, 2002.

[26] K. Malde, E. Coward and I. Jonassen. "Fast sequence clustering using a suffix array algorithm". *Bioinformatics*, vol. 19, pp. 1221–1226, 2003.

[27] E. Giladi, M. Walker, J. Wang and W. Volkmuth. "SST: an algorithm for finding near-exact sequence matches in time proportional to the logarithm of the database size". *Bioinformatics*, vol. 18, no. 6, pp. 873–877, 2002.