Efficient Parallelization Techniques for Exact Algorithms in the Planted Motif Problem

Samuel J. Van Pelt

1 2013 NSF Bio-Grid REU Research Fellow
University of Connecticut, Storrs, CT 06269
2 Colorado Mesa University, Grand Junction, CO 81501
samuel.van.pelt@uconn.edu

Abstract. Meaningful Patterns, or Motifs, often exist in large volumes of biological data. They are theorized to be highly significant elements in protein and DNA and are important for understanding the function of particular sequences. Identifying these motifs can help researchers to accurately target human disease with therapeutic drugs and understand gene function more precisely. There have been several identified problems, but this project looks primarily at parallel optimizations for the planted \((l, d)\)-motif problem. Planted motifs are sequences of length \(l\) that have a hamming distance\(^3\) of exactly \(d\) from at least one sub-sequence in each of the input strings (input strings are DNA sequences of a consistent length). Being able to quickly identify these motifs is difficult due to the nature of the problem. The number of comparisons we need to make grows exponentially with \(d\). It quickly becomes impractical to attempt sequential computation of the motif search as \(d\) grows. This project demonstrates the practical application of parallel techniques using OpenMP and shows the degree to which performance can be improved through loop and block-level parallelization. It also discusses how these techniques can be applied to other similar problems in the field of bioinformatics.

Keywords: Planted Motif, Parallelization, NP-Hard, Exact Algorithms, Sequence Motif Search.

1 Introduction

Large volumes of biological data are available to researchers because of multiple genome projects. The problem of processing this data to search for useful information is one for which many approaches have been described. One of these approaches is motif search. This approach, like many in bioinformatics, is sequential and can benefit from parallelization. This paper presents techniques which are applicable to this problem and others which may benefit from parallelization.

\[^3\] The Hamming distance between two words is the number of characters where they differ.
These problems benefit from parallelization so much because they are proven to be NP-hard [1]. That is, they cannot be computed in polynomial time. This means that the time it takes to process data grows exponentially. Because of this restriction sequential algorithms fail to adequately address the time complexity. By splitting difficult tasks across $p$ processors time can be reduced by a factor of $p$.

1.1 Background

Motif search is the problem of identifying meaningful patterns within a set of sequences. The motifs are much smaller than the sequences in which they are found and can therefore be difficult to identify[2]. One has to look through large volumes of data to identify small significant patterns. Several variants of motif search problems have been identified, but this paper will focus on methods as they apply to the Planted Motif Problem:

The Planted Motif Problem

Given $t$ input sequences comprised of biological data of length $n$ and two integers $l$ and $d$, it is a problem to identify a motif (or motifs) $M$ of length $l$. It is given that each input sequence contains a variant of $M$. A variant of $M$ is defined as a sequence of length $l$ at a hamming distance of exactly $d$ from $M$.

Significance in Bioinformatics

This problem is important due to the fact that these “planted” motifs are sites of biological significance. Being able to identify these can help researches to accurately target human disease with therapeutic drugs and understand gene function more precisely. Planted Motifs are one of several identified motif types in biological sequences. They are of interest to this project due to the difficult of the problem computationally.

Several algorithms exist to solve this problem[2]. This paper focuses on Planted Motif Search 1 (PMS1) [3] to demonstrate techniques for efficient parallelization. Newer exact algorithms have been described for in sequence and parallel motif search [1], but this paper seeks to describe general techniques that can be applied outside the scope of motif search. PMS1 provides ample opportunity for parallelization.

Planted Motif Search

The naive approach to planted motif search has been implemented for purpose of demonstrating the efficiency of PMS1. This approach is to generate\(^4\) all possible $l$-mers\(^5\) for a given length $l$ and compare each of these

\(^4\) For this implementation, strings are represented as integers and generation can be completed in $2^{l+2}$ time by simply iterating through all numbers $0 < i < 2^{l+2}$

\(^5\) An l-mer is a sequence of length l. Motifs are l-mers that satisfy the Hamming Distance requirement
to every sub-string of each input sequence and retaining the ones that appear in ever input sequence\cite{3}.

The Algorithm PMS1 Rajasekaran, Balla, and Huang (2005) describe in detail PMS1, the algorithm for finding motifs in large volumes of biological data\cite{3}. The method is to first generate all l-mers from each of the t input sequences. This can be done in $O(tn)$ time. Then, from each of these sequences, given a particular input sequence, generate all possible l-mers that would satisfy the hamming distance requirement. This results in:

\begin{equation}
O(n \binom{l}{d} \sum |d|)
\end{equation}

“candidate” motifs. Sort these motifs, eliminate the erroneous duplicate results, and merge these with the candidate motif lists from the remaining $t-1$ sequences.

This algorithm is the base for a series of exact algorithms that perform motif search effectively\cite{3, 6, 1}. These algorithms have accurate results faster than competing algorithms. It provides an expandable base case for the implementation of parallelization.

OpenMP

OpenMP is an industry leading API for shared-memory parallel programming \cite{4, 5}. Modern computers are built for performance, and the increased proliferation of multi-core or multi-processor machines has made it apparent that efficient techniques for harnessing all of these resources are needed. Compilers have been optimized \cite{5} to make use of these multi-threaded environments, but it has been shown that programmer intervention is needed to make best use of them. The lack of interdependent data in many scenarios allows multiple instructions to be passed to the processor at the same time and achieve the same results.

OpenMP allows for scalable shared memory programming. It is important to avoid tuning a program to work with only a specific number of cores \cite{9} because this disallows the software to run on a wide array of machines and theoretically limits its utility.

In high performance computing, the speed of processors increased very quickly for a long period of time. However, higher clock speeds led to physical barriers that limited the increase in speed. The change that occurred was that successive generations of processors had more cores instead of higher clock speeds. This meant that the clear increase in performance on pre-existing programs disappeared. It became necessary to change the way programs are made. They need to be able to scale with the number of cores, not just the clock speed \cite{9}. OpenMP is a good solution to this problem.
1.2 Utility

The methods described in this paper will be applicable both to sequential implementations of algorithms for the planted motif problem, and also other problems in bioinformatics. This problem, like many in the field, suffers from being NP-Hard and therefore has notoriously long run-times.

Planted Motif Search

PMS1, PMS2, and PMS3 are all sequential algorithms [3]. While strictly parallel algorithms have been described for this problem recently [1], the changes are not general enough to be applied to other implementations. This paper uses PMS1 to provide a base case for improved performance with a parallelization tool that can be used by any researcher, in any lab, anywhere.

WINNOWER [10] and MITRA [11] are two such algorithms for motif search that suffer from being sequential in nature [7]. These and other algorithms in planted motif search can benefit from the techniques this paper describes.

Bioinformatics

Many problems in bioinformatics are NP-Hard or NP-Complete, meaning they are not able to be computed in polynomial time [12–14]. These problems all suffer from long run-times and could therefore theoretically benefit from parallelization. The techniques this paper describes can be applied to a wide array of problems assuming some “needle in a haystack” [2] like element, that is that there are large volumes of data that aren’t interdependent. The ease of use of OpenMP for these types of problems means that most researchers can benefit from the existence of multi-core machines and multi-core programming.

2 Methodology

The code for this paper was written in C++. First, an efficient sequential implementation of PMS1 was built, as described by Petersen, and Felekey (2013) [15], and then the techniques this project found were applied to that implementation. We do not compare run-times of this implementation of Planted Motif Search to run-times of other algorithms or other implementations because the significant portion of this research is the self-contained improvement because of parallel techniques, not competing with other implementations.

---

6 Long being exponentially long.

7 Problems where there is a lot of interdependence of data can also benefit from parallelization, it just requires greater effort on the part of the programmer.
The output of these implementations were compared against the naive approach to Planted Motif Search and, for the modified version of the program that detects motifs where hamming distance may be less than the input $d$, the output of PMS5 [6]\(^8\). They were run on the same randomly generated data-sets where $t = 20$ and $n = 600$. That is, 20 input sequences of length 600.

### 2.1 Loop-level Parallelism

**Planted Motif Search**

*Generation, Sort, and Elimination of Duplicates* In areas of the planted motif search where there is no interdependence of data, it is trivial to optimally parallelize the execution of these tasks. Because there is no interdependence of data it is possible to implement loop-level parallelism which is scalable across any number of processors with OpenMP.

The advantage of loop-level parallelism is that it is *easy* to implement. Any one in any lab can take tasks where there isn’t interdependent data and push these tasks across all their processors. The following\(^9\) code demonstrates the changes to a sequential implementation necessary to effectively parallelize it.

```c
#pragma omp parallel for
for (int i = 0; i < fN; i++)
{
    #pragma omp parallel for
    for (int j = 0; j <= n-1; j++)
    {
        generate_motif_candidates(sequences[i].substr(j, l),
                                   c_lmer[i]);
    }
    counting_sort(c_lmer[i],1);
    eliminate_duplicates(c_lmer[i]);
}
```

The sequential implementation does the following task:

```
for all t sequences do
    for all n sub-strings do
        Generate all motif candidates and store them in an array at c_lmer[i].
    end for
    Sort the generated l-mers.
    Eliminate the duplicate l-mers.
end for
```

\(^8\) It should be noted PMS5 misses a small number of motifs, but this implementation does not

\(^9\) Sections of this code were removed for brevity. The full text is located in the appendices.
To parallelize this task it is simply necessary to call OpenMP’s “Parallel For” loop function defined by `#pragma omp parallel for`.

This technique can be applied to all algorithms derived from PMS1 very clearly. PMS2 and PMS3 use a modified version [3] and this loop level technique can be implemented in almost exactly the same way. This can be expanded to other bioinformatic problems that share the lack of interdependent data.

**Considerations** Loop-level parallelism, while incredibly powerful and useful, has drawbacks. For one, in the shown case, the memory usage in this section is approximately $t$ times the sequential usage. This is because instead of unwinding these loops (as we do sequentially) to only have two vectors of data at a time, we have all $t$ undergoing processing at once. The majority of the speed up in runtime happens in this loop-level section of code, but the block-level parallelism that follows uses only $p$ times the memory, instead of $t$.

### 2.2 Block-level Parallelism

**Intersection** The intersection phase of this algorithm is prime example of the intricacies of block-level parallelization. It requires interaction between different sets of l-mers and therefore requires specific action on the part of the programmer. All the work cannot be passed to the compiler.

It is demonstrable that block-level parallelism is more memory efficient, but it suffers from being more difficult to implement. One of the main advantages of OpenMP is that it allows anyone to implement efficient, scalable, parallel programming. This section shows a method for parallelizing the intersection of candidate motif sets. This portion requires block level parallelism because data between loop iterations requires interaction. The data is interdependent.

The method for parallelizing sections of code which require data interaction is to unwind interdependent loops. Separate statements of the loop that aren’t dependent on one another and execute those statements in parallel. The algorithm we use to achieve this in intersection is as follows:

```plaintext
for the number of cores $p$ do
    Divide the array of arrays of candidates into $p$ parts.
    for the number of elements in each part do
        Intersect each array with the first array in that part.
    end for
end for
for the number of cores do
    Intersect the $p$ intersection arrays with the first array. This is the array of motifs.
end for
```
if there were remainder neighborhoods then
    intersect them sequentially with the first neighborhood,
end if

This method of accessing non-interdependent sections of data and acting on
them as they were acted upon sequentially achieves a rudimentary parallelization
without compromising the integrity of the program. It is fairly efficient.

An Alternative Method

It is possible, for this particular case, to achieve a more efficient parallelization.
Ideally, each core would handle the intersection of two arbitrary sequence at a
time until all sequences had been intersected once, and then they would handle
intersection the resultant $t/2$ neighborhoods in the same manner and recursively
continue until there were only one neighborhood remaining.

3 Experimental Results

Run times are averages of five (5) runs each on five (5) randomly generated
data-sets. There isn’t much fluctuation of run time due to the deterministic
nature of PMS1. The greatest length of time is spent sorting the generated lists
of “candidate” motifs, which there is a predetermined number of based on $l$, $d$,
n, and $t$, not the contents of the sequences.

3.1 Improvement on Sequential Run-times

In all test cases with both synthesized and real data10 the parallel implementa-
tion of PMS1 performed markedly better than the sequential implementation.
All results (including the run times on real biological data) are in Appendix B.

---

10 In this algorithm, real data and synthesized data will have essentially the same run-
time. It is a deterministic algorithm and looks at the same number of possible motifs
regardless of the nature of the data.
The run times are significantly different given an non-optimized compiler (see experimental details).

![Run Times of PMS1 when \(d = 4\)](chart)

An efficient parallel implementation of this sequential algorithm results in a significant performance boost. In the second graph, the sequential implementation hits 890 seconds (almost 15 minutes) whereas the parallel implementation stays under 270 seconds (4.5 minutes). The ratio of sequential run-times to parallel run-times is 3.3, graphed below.

![Ratio of Sequential to Parallel Run-Time \(d = 4\)](chart)

Since it is theoretically possible to divide all the tasks equally across all 4 cores the ideal ratio of sequential to parallel run-times is 4. This means that this implementation is approximately 83% efficient, in this case.

### 3.2 Effectiveness Across All Tested Cases of \(d\) in an Optimized Environment

**Biological Data** The real biological that the tests in this section were run on are from Homo Sapiens chromosomes taken from the LASAGNA search tool [16]. The implementation of PMS1 in this paper would work on biological data taken from any species.
The case where $d = 4$ is examined above in a non-optimized environment because of the fact that it takes the most time and the effectiveness of parallelization is most apparent. Data was acquired for the lesser $d$ cases in an optimized environment (where tests should be performed) and are shown below in two separate graphs, one for sequential time and one for parallel time using the real biological data:

Sequential Run Times (log)

Parallel Run Times (log)

The parallel run times shown above are faster than the sequential ones.

The improvement in an optimized environment does not reach the level of the non-optimized environment. The ratio of sequential to parallel run-times is
generally between 1.5 and 2.5 for the quad core machine. These improvements (while highly beneficial) do not meet the theoretical limit that exists at the number of cores. That is shown below:

Ratio of Sequential to Parallel Run-Time

![Graph showing the ratio of sequential to parallel run-time for different values of d.]

3.3 Details

This section provides details on the implementation of PMS1 using OpenMP.

These tests were run on a 4-core machine with a 2.8GHz processor and 8GB of RAM. As in the works leading up to this one, they were run on data where \( t = 20 \) and \( n = 600 \). The input sequences were generated randomly. The implementations were also tested on real biological data from [16].

It should be noted that the C++ code was compiled with GCC using the optimization flag “-O3”. In this environment, overall performance for both implementations was much better but the relative performance of the parallel implementation was less significant. Compiling without the “-O3” flag resulted in a consistent improvement from sequential to parallel by a factor of the number of cores the test was run on. For example, on a 2-core, 1.7GHz machine, the (9, 2) challenge case on [16] data ran in 1.24 seconds sequentially and 0.63 seconds in parallel. On a similar specifications 4-core machine the same test was completed in 0.33 seconds. The improvement in a non-optimized environment scales very clearly with the number of cores. The performance boost in an optimized environment is significantly less. This is likely because optimized compilers often perform some form of automatic loop unwinding, allowing statements to be executed more in parallel without programmer intervention. The non-optimized run times are shown in Appendix B, Table 2.

\[11\] On average from 5 runs of the test
Experimental results where $l > 16$ utilize the $\text{uint64}_t$ datatype instead of $\text{uint32}_t$ datatype and assume a word-length of 64 on the testing machine. Due to the bitwise nature of many operations performed in this implementation and the fact that the input strings and motifs are treated as integers with each character taking two bits, ($G = \text{“00”}$, $C = \text{“01”}$, $T = \text{“10”}$, $A = \text{“11”}$) as described by Rajasekaran, Balla, and Huang (2005)[3], the maximum size of $l$ is limited by the word-length $W$ of the machine to $W/2$.

4 Conclusions

Modern computers are built around the idea that tasks can be done in parallel. It would be difficult to walk into a store today and purchase a computer that didn’t have a multi-core processor. It is essential that researchers begin using techniques for parallelization in a lot of situations because they can benefit from it. It is no longer the case that only high-performance computers can benefit from these techniques. Even modern game consoles and cell phones are multi-cored. By implementing the techniques described in this paper many people could benefit. Parallelism is a clear path toward more efficient computing.

The performance boost in the NP-Hard problem of Planted Motif Search was significant. There was up to a 70% reduction in run time on a quad-core machine. This goes to show that techniques using OpenMP (and likely other parallelization APIs) greatly improve the run-times of resource-intensive tasks like Planted Motif Search.

4.1 Future Work

Since the publication of PMS1 in 2005 [3], descriptions of strictly parallel algorithms have been released [1]. It would be beneficial to, instead of describing techniques for which a sequential implementation of a sequential algorithm may be parallelized, describe efficient techniques for implementing algorithms which have been designed to take advantage of the fact that tasks can be done in parallel. While it is useful to be able to parallelize algorithms that weren’t designed for this purpose (given the fact that a large number of sequential algorithms already exist), parallel algorithms are a clear path toward more efficient computing.

Acknowledgments

This research in the Summer 2013 Bio-Grid REU program was supported in part by the NSF grant OCI-1156837.

References


8. N. T. Tran. High Performance Multi-Threaded Programming Using OpenMP.


Appendix

A Source

A.1 PMS1 Sequential

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <cmath>
#include <cstring>
#include <omp.h>
#include <stdint.h>

using namespace std;

const int W = 32;

int main()
{
    // Init config file reading
    ifstream inf("config.dat");
    ifstream infileFile;
    string infilen, temp;

    // Init timing variables
    double start, stop;
    double t = 0.0;

    // Declare/Init variables
    int l, d, n, count = 0;
    vector<string> sequences;
    vector<uint32_t> candidate_neighborhood,
        secondary_neighborhood, sorting_neighborhood;
    uint32_t n_size = 0;

    inf >> l >> d >> infilen;
    if(l>16)
    {
        cout << "'l'(ength) exceeds the word-length of this implementation";
    
```
return 5;
}

// Read input sequences
inFile.open(inFile.c_str());
while(!inFile.eof() && (inFile >> temp))
{
    sequences.push_back(temp);
}
inFile.close();

// Init neighborhood memory allocation info
n = sequences[0].length();
n_size = (n - 1 + 1)*(NCR(1,d))*(pow(3.0,d));

// Starts timing.
start = omp_get_wtime();

candidate_neighborhood.resize(n_size);
sorting_neighborhood.resize(n_size);

for (int i = 0; i < sequences.size(); i++)
{
    secondary_neighborhood.resize(n_size);
    // Generate neighborhoods
    count = 0;
    for (int j = 0; j <= n-1; j++)
    {
        generate_motif_candidates(string_to_int(sequences[i].substr(j,1),1),
                                  secondary_neighborhood,d,-1,1,count);
    }
    // Sort
    counting_sort(sorting_neighborhood,
                  secondary_neighborhood,1);
    // Eliminate duplicates
    eliminate_duplicates(secondary_neighborhood);
    if(i==0)
    {
        candidate_neighborhood.resize(secondary_neighborhood.size());
        for(int j = 0; j < secondary_neighborhood.size(); j ++)
            candidate_neighborhood[j]=secondary_neighborhood[j];
        continue;
    }
    intersect(secondary_neighborhood,candidate_neighborhood);
}
stop = omp_get_wtime();
ofstream outf("motifs.txt");
for (int i = 0; i < candidate_neighborhood.size(); i++)
{
    outf << int_to_string(candidate_neighborhood[i], l) << endl;
}
outf.close();
t = (double) (stop - start);
cout << "Motifs found: " << candidate_neighborhood.size() << endl;
cout << "Time Taken: " << t << " seconds" << endl;
return 0;

A.2 PMS1 Parallel

/* Samuel Van Pelt */
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <cmath>
#include <cstring>
#include <omp.h>
#include <stdlib.h>
using namespace std;
const int NUM_THREADS = 4;
const int W = 32;
int main()
{
    ifstream inf("config.dat");
    ifstream infileFile;
    int l, d;
    string infile, temp;
    double start, stop;
    double t = 0.0;
    vector<string> sequences;
    uint32_t n_size = 0;
    int fN, n;
```cpp
inf >> l >> d >> infile;
infFile.open(infile.c_str());
while(!infFile.eof() && (infFile >> temp))
{
    sequences.push_back(temp);
}
infFile.close();

n = sequences[0].length();
n_size = (n-l+1)*(NCR(1,d))*(pow((double)3,d));
fN = sequences.size();
vector<uint32_t> c_lmer[fN];

// Starts timing.
start = omp_get_wtime();

#pragma omp parallel for
for (int i = 0; i < fN; i++)
{
    c_lmer[i].resize(n_size);
}

#pragma omp parallel for
for (int i = 0; i < fN; i++)
{
    int count = 0;
    #pragma omp parallel for
    for (int j = 0; j <= n-1; j++)
    {
        generate_motif_candidates(string_to_int(sequences[i].substr(j,l)),l,c_lmer[i],d,l,l,count);
    }
    counting_sort(c_lmer[i],1);
    eliminate_duplicates(c_lmer[i]);
}

int remainder = fN % NUM_THREADS;
int sequences_per_thread = floor(double(fN/NUM_THREADS));
vector<uint32_t> intersection_neighborhood[NUM_THREADS];

// Block Level Intersection
#pragma omp parallel for
for (int i = 0; i < NUM_THREADS; i++)
{
    intersection_neighborhood[i] = c_lmer[i*sequences_per_thread];
    #pragma omp parallel for
```
```c
for(int j = (i+1)*(sequences_per_thread)-1; j > i*sequences_per_thread; j--)
{
    intersect(clmer[j],intersection_neighborhood[i]);
}

for(int i = 1; i < NUM_THREADS; i++)
    intersect(intersection_neighborhood[i], intersection_neighborhood[0]);
if (remainder != 0)
{
    for(int i = fN-1; i >= (fN - remainder); i--)
    {
        intersect(clmer[i],intersection_neighborhood[0]);
    }
}
stop = omp_get_wtime();
ofstream outf("motifs.txt");
for (int i = 0; i < intersection_neighborhood[0].size(); i ++)
{
    outf << int_to_string(intersection_neighborhood[0][i],1) << endl;
}
outf.close();
t = (double) (stop-start);
cout << "Motifs found: " << intersection_neighborhood[0].size() << endl;
cout << "Time Taken: " << t << " seconds." << endl;
return 0;
```

### A.3 Functions

```c
// Accepts a sequence represented as an integer (o_int) and
// converts the character at pos to the next character. That
// is G->C,T->A.
void increment_char(uint32_t &o_int, short int pos)
{
    uint32_t temp = o_int;
    temp <<= (W -(2*(pos+1))) ;
    temp >>= (W - 2);
    o_int += ((uint32_t)(((-4)*(temp==3) + 1)) << (2*pos));
}
```

// Accepts a sequence represented as an integer (o_int) and
// returns the value at pos
inline uint32_t get_char(uint32_t o_int, short int pos)
{
    o_int <<= (W - (2*(pos+1)));
    o_int >>= (W - 2);
    return o_int;
}

// Accepts a character c and converts it to the corresponding integer (0, 1, 2, or 3)
inline int char_to_int(char c)
{
    char x = ~c;
    char y = x;
    x >>= 2;
    y = y & 2;
    x = x & 1;
    x += y;
    return x;
}

// Accepts a string (o_string) and converts it to an uint32_t
uint32_t string_to_int(string o_string, int l)
{
    uint32_t o_int = 0;
    for(int i = 0; i < l; i++)
    {
        o_int += ((uint32_t)char_to_int(o_string[i]) << (2*(l-i-1)));
    }
    return o_int;
}

// Accepts an int (o_int) and converts it to a string of length l
string int_to_string(uint32_t o_int, short int l)
{
    string sequence = "";
    for (int i = 0; i < l; i++)
    {
        sequence += 'G';
    }
    for (int i = l-1; i >=0; i--)
    {
        if(o_int != 0)
        {
            if (o_int % 2 == 0)
            {
                if (((o_int = o_int/2) % 2) != 0)
                {
                    sequence[i] = 'T';
                }
            }
        }
    }
    return sequence;
}
if (o_int = o_int /2) % 2 == 0)
    
    sequence[i] = 'C';
    
    else
    
    sequence[i] = 'A';
    

    o_int = o_int /2;
}

return sequence;

// Generates all the possible motif candidates from a given
// string represented as an integer
void generate_motif_candidates(uint32_t c, vector<uint32_t>&n,
    int d, int pos, int length, int &count)
{
    if (d==0){ n[count++] = c; return;}
    for (int j=0; j < 4-1; j++)//4 is the size of the alphabet
    {
        for (int i=pos-1; i >= d-1; i--)
        {
            uint32_t temp = c;
            for (int k = 0; k < j; k++)
            {
                increment_char(temp,i);
            }
            if(d>1) generate_motif_candidates(temp,n,d
-1,i,length,count);
            else n[count++] = temp;
    }
    // call gen here for d<=hammingdistance, don’t forget to
    // modify nSize
    //generate_motif_candidates(c,n,d-1,-1,length,count);
    return;
}

// Sorts a vector of integers
void counting_sort(vector<uint32_t>&n, int l)
{

```cpp
vector<uint32_t> temp_n;
int s = n.size();
temp_n.resize(s);
int bucket[4];
for (int i = 0; i < 1; i++)
{
    memset(bucket, 0, 4 * sizeof(int));
    for (int j = 0; j < s; j++)
        bucket[get_char(n[j], i)]++;
    for (int j = 1; j < 4; j++)
        bucket[j] += bucket[j - 1];
    for (int j = s - 1; j >= 0; j--)
        temp_n[--bucket[get_char(n[j], i)]] = n[j];
    for (int j = 0; j < s; j++)
        n[j] = temp_n[j];
}

void eliminate_duplicates(vector<uint32_t> &n)
{
    n.erase(unique(n.begin(), n.end()), n.end());
}

void intersect(vector<uint32_t> &n, vector<uint32_t> &destination)
{
    vector<uint32_t> temp;
    vector<uint32_t>::iterator it;
    temp.resize(destination.size(), 0);
    it = set_intersection(n.begin(), n.end(), destination.begin(),
                         destination.end(), temp.begin());
    temp.resize(it - temp.begin());
    destination.resize(temp.size());
    for (int i = 0; i < temp.size(); i++)
        destination[i] = temp[i];
}

// n choose r
long long NCR(int n, int r)
{
    if (r > n / 2) r = n - r;
    long long ans = 1;
    int i;
    for (i = 1; i <= r; i++)
    {
        ans *= n - r + i;
    }
```
B Run Times

Table 1. Sequential vs. Parallel Run Times

<table>
<thead>
<tr>
<th>length</th>
<th>dist</th>
<th>Seq. Time</th>
<th>Par. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>1</td>
<td>0.0195</td>
<td>0.0071</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0.0225</td>
<td>0.0083</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0.0261</td>
<td>0.0096</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>0.0299</td>
<td>0.0109</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>0.0341</td>
<td>0.0125</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>0.0388</td>
<td>0.0138</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>0.0437</td>
<td>0.0523</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>0.2162</td>
<td>0.0685</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>0.2862</td>
<td>0.0905</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>0.3696</td>
<td>0.1165</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>0.4693</td>
<td>0.1542</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>0.5876</td>
<td>0.2201</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>0.7284</td>
<td>0.3423</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>0.8943</td>
<td>0.5229</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>1.8742</td>
<td>1.1150</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>3.0205</td>
<td>1.8397</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>4.5178</td>
<td>2.7963</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>6.4148</td>
<td>4.0426</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>8.8579</td>
<td>5.6447</td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>11.9893</td>
<td>7.6711</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>15.8922</td>
<td>10.2094</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>8.7024</td>
<td>5.3312</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>15.9799</td>
<td>9.8716</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>27.5267</td>
<td>17.1408</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>44.9601</td>
<td>27.9755</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>69.0533</td>
<td>43.5442</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>101.9410</td>
<td>62.6757</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>146.0530</td>
<td>92.0538</td>
</tr>
</tbody>
</table>
Table 2. Sequential vs. Parallel Run Times (non-optimized)

<table>
<thead>
<tr>
<th>length</th>
<th>dist</th>
<th>Seq. Time</th>
<th>Par. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>1</td>
<td>0.1454</td>
<td>0.0456</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0.1717</td>
<td>0.0674</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0.2022</td>
<td>0.0640</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>0.2362</td>
<td>0.0811</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>0.2730</td>
<td>0.0845</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>0.3130</td>
<td>0.0961</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>0.3546</td>
<td>0.1095</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>1.6988</td>
<td>0.5036</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>2.3202</td>
<td>0.6854</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>3.0642</td>
<td>0.9043</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>3.9129</td>
<td>1.1637</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>4.8829</td>
<td>1.4724</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>6.0642</td>
<td>1.8316</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>7.4110</td>
<td>2.2424</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>11.6757</td>
<td>3.4698</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>18.4981</td>
<td>5.4995</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>27.7777</td>
<td>8.2761</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>39.896</td>
<td>11.8560</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>55.4615</td>
<td>16.4322</td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>75.0236</td>
<td>22.2002</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>98.5143</td>
<td>29.4195</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>51.9539</td>
<td>15.4018</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>95.1593</td>
<td>28.2511</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>163.976</td>
<td>48.7285</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>267.673</td>
<td>79.6906</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>416.166</td>
<td>128.233</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>621.22</td>
<td>184.308</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>898.559</td>
<td>266.037</td>
</tr>
<tr>
<td>length</td>
<td>dist</td>
<td>Seq. Time</td>
<td>Par. Time</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td>-----------</td>
<td>-----------</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0.0183</td>
<td>0.0072</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0.0197</td>
<td>0.0082</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0.0225</td>
<td>0.0097</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>0.0258</td>
<td>0.0112</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>0.0294</td>
<td>0.0128</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>0.0333</td>
<td>0.0144</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>0.0375</td>
<td>0.0151</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>0.1858</td>
<td>0.0726</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>0.2441</td>
<td>0.1025</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>0.3135</td>
<td>0.1575</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>0.3995</td>
<td>0.2452</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>0.5405</td>
<td>0.3407</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>0.6731</td>
<td>0.4514</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>0.8622</td>
<td>0.5693</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>1.6013</td>
<td>0.9948</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>2.5849</td>
<td>1.6242</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>3.8845</td>
<td>2.4613</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>5.4718</td>
<td>3.5294</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>7.5269</td>
<td>4.9043</td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>10.1410</td>
<td>6.6012</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>13.4492</td>
<td>8.6132</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>7.3207</td>
<td>4.5378</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>13.3692</td>
<td>8.3202</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>23.0535</td>
<td>14.1650</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>38.2737</td>
<td>23.8045</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>58.8774</td>
<td>37.1959</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>86.3091</td>
<td>55.2827</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>124.0200</td>
<td>78.5180</td>
</tr>
</tbody>
</table>