Parallelizing Tompa's Exact Algorithm for Finding Short Motifs in DNA

Christopher T. Mitchell[1], Jonathan Grochowski[1], Julian H. Dale[1], Nicolas B. Wilson[1], and Jens Mache[1]
[1]Department of Mathematics & Computer Science, Lewis & Clark College, Portland, Oregon, USA

Abstract—*Motif finding, the search for regulatory sequences in DNA, is a computationally expensive challenge in bioinformatics. This paper presents a pleasantly parallel version of Tompa's exact method for finding short motifs. We use a distributed-memory computer cluster and MPI to run our parallel algorithm and collect data. We vary motif length and allowed substitions. Results indicate good speedup and scalability.*

Keywords: cluster, parallel algorithm, bioinformatics, motif finding, performance evaluation, MPI

1. Introduction

A motif is a short sequence of DNA that has a specific function and appears multiple times throughout a genome. A motif could be many things, including a transcription factor binding site, or a ribosome binding site. Motifs are of interest to biologists because they often play important roles in the regulation of gene expression.

Finding a motif amongst a set of DNA sequences is not a trivial task. The motif may not appear in every single sequence, and instances of the motif may not be identical due to substitutions, insertions, and deletions. To find this motif, one not only needs to accommodate for inexact matches, but one must also devise a way to filter the true biological motifs from patterns that randomly occur within the sequences.

Our initial survey of DNA motif finding algorithms showed that the set of approaches to this problem is very diverse. Within this set, there are two general approaches to the problem [1]. The first approach uses a word-based algorithm that analyzes a string of nucleotides and counts and compares the frequency of specific k-mers (contiguous substrings of length k). Word-based algorithms rely on exhaustive enumeration and can guarantee an optimal result. The second approach involves using probabilistic models where the parameters are based on some form of statistical inference (maximum-likelihood, Bayesian) or weight matrix.

This paper will focus on an exhaustive word-based algorithm designed by Tompa [2]. Our goal was to decrease the run-time of this algorithm by parallelizing its execution. This paper will present our method of parallelization, performance results, and suggestions for continued work.

2. Parallelizing Tompa's algorithm

2.1 Tompa's word-based algorithm

Tompa developed a word-based algorithm that takes a set of DNA sequences and a k-mer length as its input, and outputs z-scores for all motifs (words) of length k. The algorithm was designed to overcome two weaknesses that he identified in more naive word-based motif finding approaches [2]. These naive algorithms (that simply count k-mer frequency or measure k-mer entropy) are vulnerable to improperly scoring motifs when either the background nucleotide distribution is not uniform or when pairs of motifs occur in largely different number of input sequences.

Tompa's approach was to score each k-mer with a z-score constructed from the observed and expected number of input sequences that have an occurrence of the given k-mer. The z-score for some k-mer s is given by

$$M_s = \frac{N_s - Np_s}{Np_s(1-p_s)}, \tag{1}$$

where N is the number of input sequences, N_s is the number of input sequences that contain an occurrence of s, and p_s is the probability of observing an occurrence of some s in a random sequence. The idea of the z-score and the technique for calculating p_s was the crux of Tompa's work, but not necessarily the most computationally expensive. It is worthwhile to impress that calculating N_s involves examining every input sequence in turn with respect to k-mer s.

2.2 Identifying opportunities for parallelization

To help us determine which portions of the algorithm would benefit most from parallelization, we profiled our sequential implementation of the algorithm (pseudocode in Figure 1). For our profiling, we ran our program with parameters that mimicked those defined by Tompa in the motivating computational problem in Section 1.1 of his paper: 4000 input sequences, each 20 nucleotides long, searching for 5-mer motifs [2].

The profiling revealed that the 64% of CPU time was spent in the routine that checks to see if an input sequence contains (or "matches") an occurrence of a k-mer (line 5). There were 4,096,000 calls to that matching function — more calls than were made to any other function. This number, while

```
1  for kmer in kmer_set:
2      p_s = ...
3      N_s = 0
4      for sequence in input_sequences:
5          if kmer occurs in sequence:
6          N_s += 1
7      M[kmer] = calc_z_score(N_s, p_s)
8
9  return top_kmers(M)
```

Fig. 1: Pseudocode for serial calculation of the z-scores

surprising, can be understood by realizing that for each k-mer, we check for matches against all sequences:

$$4^5 \ k\text{-mers} \times 4000 \text{ sequences} = 4,096,000 \text{ checks}. \quad (2)$$

Even though checking for a match is fast, by virtue of there being so many checks, counting N_s takes more time than any other step in the algorithm. The functions that calculate p_s (line 7), for example, are only called once per k-mer ($4^5 = 1024$ many times in this case) regardless of the number of input sequences.

The simplest way to break up the work of calculating N_s is to split the work up at the level of calculating M_s, since M_s is dependent on both N_s and p_s. Said differently, parallelizing the outer loop (line 1) of the algorithm will yield the easiest and most immediate gain. Only 0.3% of the program's execution time was spent outside of the loop.

2.3 Method of parallelization

Since the calculations in the step that we identified could be easily partitioned into separate (and independent) jobs that do not need communication, the task is well suited to run on a distributed computer like a Beowulf cluster with MPI. We assembled our own Beowulf cluster, called BeoPup, composed of 18 single-core AMD 64-bit Athlon processor nodes connected by a TCP/IP Ethernet switch. For parallelizing the motif finding algorithm on our BeoPup cluster, MPI was a natural choice because it is designed to enable parallel computing for interconnected machines that do not share memory, like our cluster. The Beowulf design of our cluster was well suited for the low communication requirements of our parallelization of Tompa's algorithm. To use MPI with our Python implementation, we settled on the mpi4py module [3] because of its active development.

To split the work of calculating the z-scores among multiple computers, we wrote a SPMD (single program, multiple data) type program that would calculate the z-scores for disjoint subsets of all of the k-mers, where the subset is dependent on which node in the cluster the program is running on. In MPI terminology, we used a "gather" at the end of the computations so that the z-scores each node calculated are sent to a master node (given rank zero in our example) that sorts and outputs the best scoring k-mers. The source code for this parallel implementation has been made available online [4].

3. Results

We ran our program with the same parameters given earlier on 1, 2, 4, 8, and 16 nodes. We recorded the shortest of five run-times for each condition in Table 1. Running the program on 16 nodes, for example, yielded a nearly 15 times speedup over the time it takes to run the program on just a single node. The efficiency for this 16 node case is given by the speedup divided by the number of nodes: $\frac{14.996}{16} = 0.937$. An efficiency of 1 would indicate a perfectly linear speedup, where the parallelized run-time is equal to the single-node run-time divided by the number of available nodes. The slightly decreasing efficiency can be explained by communication overheads and the redundant computation of the background nucleotide Markov model (used in finding p_s) that occurs on all nodes. A near perfectly linearly speedup of our algorithm holds until the number of available processors nears the granularity of the parallelization, which is given by the number of k-mers.

Table 1: The run-time of our parallelization of Tompa's algorithm descreases almost linearly with respect to the number of compute nodes.

Nodes	Speedup	Efficiency	Time (seconds)
1	1.000	1.000	37.227
2	1.971	0.985	16.873
4	3.894	0.974	8.539
8	7.706	0.963	4.315
16	14.996	0.937	2.217

Figure 3 shows run-time increasing exponentially when we increase the length of the motifs that we are searching for. Because of the design of our parallelization, increasing the problem size in this way will only increase the efficiency when running on many nodes. This is because the proportion of time spent calculating z-scores over the time spent in non-parallel code increases as k increases.

4. Discussion and Future Work

We have shown that the application of parallel techniques can greatly increase the speed of Tompa's motif-finding algorithm. Others have demonstrated success parallelizing widely-used motif finding algorithms; MEME is one such example that uses clusters of GPUs [5]. It reasons that other motif-finding algorithms could benefit from the application of parallel computing techniques and that the resultant speedup could make running precise (but expensive) algorithms a more practical prospect.

After the completion of the parallelization of Tompa's algorithm, we reached out to members of the bioinformatics community. So far, the improvements were met with a

```

for kmer in kmer_set:
    p_s = ...
    N_s = 0
    for sequence in input_sequences:
        if kmer occurs in sequence:
            N_s += 1
    M[kmer] = calc_z_score(N_s, p_s)




return top_kmers(M)
```

Serial version

```
kmer_set = get_kmers_for_rank(my_rank)
for kmer in kmer_set:
    p_s = ...
    N_s = 0
    for sequence in input_sequences:
        if kmer occurs in sequence:
            N_s += 1
    M[kmer] = calc_z_score(N_s, p_s)

M = gather_to_rank_0(M)

if my_rank is 0:
    return top_kmers(M)
```

Parallel version

Fig. 2: Pseudocode of parallelization

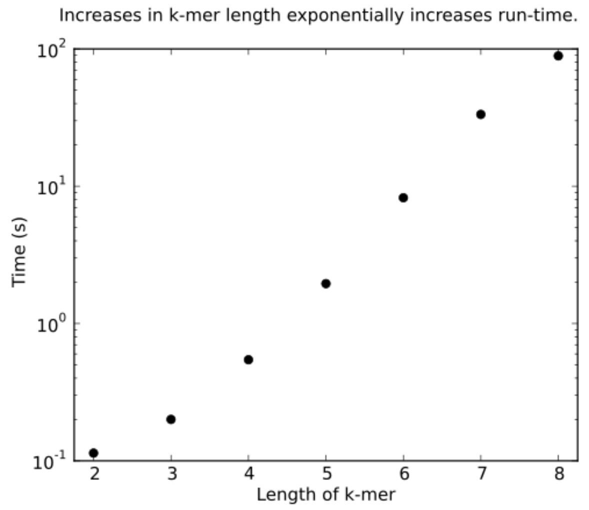

Fig. 3: Increases in k-mer length exponentially increase run-time.

variety of feedback as well as some constructive criticism to help form a plan for the future of our work with Tompa's algorithm. An insightful comment was provided by Dr. Jonathan Visick, a professor of Microbiology and Genetics at North Central College,

> With hundreds of new bacterial whole-genome sequences being completed each year, problems like identifying ribosome-binding sites are not going away: the sequences are not the same for different species, necessarily. Plus, an algorithm for finding a ribosome-binding site presumably would also be applicable to finding promoters in prokaryotes, transcription-factor binding sites in eukaryotes and various other sequence features. [6]

The next step for this algorithm probably entails a more collaborative effort from within the bioinformatics community. The feedback indicates that the importance of motif-finding is not dwindling and that their utility is greater now than ever before. Since Tompa's algorithm does not make any biological assumptions, it can be adapted to address some of the challenges mentioned by Dr. Visick.

Tompa's algorithm involves creating a distinct deterministic finite automaton (DFA) for each k-mer. This DFA is used to match sequences that have an occurrence (recall that an occurrence allows substitutions) of the related k-mer, and also to calculate p_s. As a possible extension to his work, Tompa suggests finding a way to accommodate longer k-mers and more substitutions. This is because the DFA creation algorithm that we implemented, while computationally fast for simple cases that allow only one substitution, becomes slow enough when it is modified to allow for more substitutions that the program's total run-time is markedly increased (Figure 4).

Profiling our code showed that DFA creation (instead of matching) takes up the majority of the run-time when we allow for three substitutions. Although Tompa suggests core modifications to his algorithm to allow for more substitutions, simply adding more compute nodes to decrease the number of k-mers that each node must inspect might be seen as a possible solution to the problem. For example, the many-thousand cored super computers ranked in the TOP500 list could easily tackle a problem with limited substitutions [7].

5. Conclusions

Motif finding is a computationally expensive task. This paper presented a parallel version of Tompa's exact method for finding short motifs [2]. Using a Beowulf cluster, we showed that our parallel version of the algorithm was able to significantly speed up the search for motifs. For example, when searching for a motif using 16 nodes, we achieved

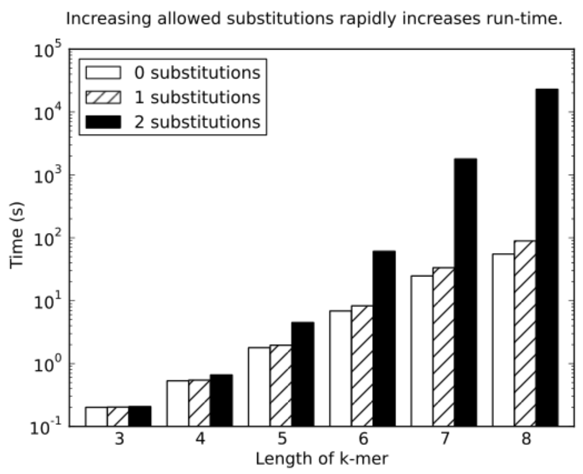

Fig. 4: For large k-mers, increasing the number of allowed substitutions significantly increase run-time.

a speedup of almost 15 times. Parallel motif-finding algorithms can enable searching for more complex motifs in reasonable amounts of time.

6. Acknowledgements

We would like to thank Dr. Jonathan Visick, Dr. Deborah Lycan and Dr. Adam A. Smith for discussions about biology and bioinformatics. This material is based upon work supported by the John S. Rogers Science Research Program at Lewis & Clark College and by the James F. and Marion L. Miller Foundation.

References

[1] M. Das and H.-K. Dai, “A survey of dna motif finding algorithms,” *BMC Bioinformatics*, vol. 8, no. Suppl 7, p. S21, 2007. [Online]. Available: http://www.biomedcentral.com/1471-2105/8/S7/S21

[2] M. Tompa, “An exact method for finding short motifs in sequences, with application to the ribosome binding site problem,” in *Intelligent Systems in Molecular Biology*, 1999, pp. 262–271.

[3] “MPI for Python,” http://mpi4py.scipy.org/.

[4] “Code : motif-finder,” https://code.launchpad.net/motif-finder, 2011.

[5] Y. Liu, B. Schmidt, and D. L. Maskell, “An ultrafast scalable many-core motif discovery algorithm for multiple gpus,” in *2011 IEEE International Parallel & Distributed Processing Symposium*.

[6] J. Visick, personal communication, 2011.

[7] “TOP500 List - November 2010,” http://top500.org/list/2010/11/100, 2010.