ABSTRACT

HIGH PERFORMANCE GENE PREDICTION FOR SMALL RNA

Gene prediction is one of the most important and alluring problems in computational biology. Meanwhile, it is one of the most time-consuming tasks. To achieve efficiency in gene prediction, it's desired to build high-performance version tools. Infernal, which we aim to accelerate in this project, is a popularly used tool to predict small RNAs in prokaryotic genomic sequences. Infernal aims to accurately detect homologs by modeling sequences of secondary structure based RNA families. However, it becomes very expensive when it comes to searching speed. Also the performance varies depending on both inputs, the small RNA family and target database sequence. In this thesis, we have developed highly scalable task parallel CMSEARCH program, which is a part of Internal tool, using Pthread and OpenMP, which can also be extended with Infernal’s data parallelization feature. In addition, we have also developed pipelined parallel computation model to further accelerate the tool. This pipelined parallelism exploits further performance gains with a load balancing strategy to dynamically assign the number of threads to each pipelined stage.

In our practice, the proposed acceleration schemes, i.e., task parallel and task parallel with pipelined approaches, showed approximately 20% and 50% of performance gain, respectively.

Linqiang Ouyang
May 2016
HIGH PERFORMANCE GENE PREDICTION FOR SMALL RNA

by

Linqiang Ouyang

A thesis
submitted in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science
in the College of Science and Mathematics
California State University, Fresno
May 2016
APPROVED

For the Department of Computer Science:

We, the undersigned, certify that the thesis of the following student meets the required standards of scholarship, format, and style of the university and the student’s graduate degree program for the awarding of the master's degree.

________________________________________
Linqiang Ouyang
Thesis Author

________________________________________
Jin Park (Chair)  Computer Science

________________________________________
Ming Li  Computer Science

________________________________________
Shih-Hsi Liu  Computer Science

For the University Graduate Committee:

________________________________________
Dean, Division of Graduate Studies
AUTHORIZATION FOR REPRODUCTION
OF MASTER’S THESIS

I grant permission for the reproduction of this thesis in part or in its entirety without further authorization from me, on the condition that the person or agency requesting reproduction absorbs the cost and provides proper acknowledgment of authorship.

X Permission to reproduce this thesis in part or in its entirety must be obtained from me.

Signature of thesis author: ________________________________
ACKNOWLEDGMENTS

I want to thank my parents, my wife and my newborn son in supporting my study in Fresno State both financially and mentally. And I am sincerely grateful of all the committee members, Dr. Jin Park, Dr. Ming Li and Dr. Alex Liu for helping me with my research and thesis work. Especially I want to thank my academic advisor Dr. Jin Park for patiently and greatly guiding me to finish my research in my thesis work. With his guidance, I learnt a lot in Bioinformatics and Parallel Computing areas which will benefit me in my whole future career.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>7</td>
</tr>
<tr>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>BACKGROUND</td>
<td>4</td>
</tr>
<tr>
<td>Covariance Model</td>
<td>5</td>
</tr>
<tr>
<td>Profile Hidden Markov Model</td>
<td>12</td>
</tr>
<tr>
<td>Infernal CMSEARCH Pipeline</td>
<td>17</td>
</tr>
<tr>
<td>PROPOSED HIGH PERFORMANCE APPROACH</td>
<td>21</td>
</tr>
<tr>
<td>Shared-Memory Based Acceleration</td>
<td>22</td>
</tr>
<tr>
<td>Exploiting Pipeline Parallelism</td>
<td>25</td>
</tr>
<tr>
<td>Predicting CMSEARCH Execution Time with Multiple Linear Regression model</td>
<td>30</td>
</tr>
<tr>
<td>PRACTICE AND EXPERIMENTAL RESULTS</td>
<td>32</td>
</tr>
<tr>
<td>Shared-Memory based Task Parallel Results</td>
<td>32</td>
</tr>
<tr>
<td>Task Pipelining Results</td>
<td>32</td>
</tr>
<tr>
<td>Task Pipelining with Data Parallelism</td>
<td>34</td>
</tr>
<tr>
<td>Infernal Pipeline Stage 1 Prediction Results with Linear Regression</td>
<td>34</td>
</tr>
<tr>
<td>CONCLUSION AND DISCUSSION</td>
<td>36</td>
</tr>
<tr>
<td>Future Work</td>
<td>36</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>38</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1</td>
<td>CM has seven types of states and production rules</td>
<td>5</td>
</tr>
<tr>
<td>Table 2</td>
<td>Nodes and states in a CM</td>
<td>7</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>1</td>
<td>An Example RNA Family and Corresponding CM</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>CM data stored in CM file</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>A toy Hidden Markov Model</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>The structure of profile HMM</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>profile HMM data stored in CM file</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>Infernal Pipeline Stages</td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>Elapsed time of CMSEARCH pipeline stages with sample data</td>
<td>23</td>
</tr>
<tr>
<td>8</td>
<td>Task parallel implementation</td>
<td>23</td>
</tr>
<tr>
<td>9</td>
<td>CPU pipeline</td>
<td>26</td>
</tr>
<tr>
<td>10</td>
<td>Infernal CMSEARCH pipeline implementation</td>
<td>26</td>
</tr>
<tr>
<td>11</td>
<td>Initial pipelining design</td>
<td>27</td>
</tr>
<tr>
<td>12</td>
<td>Task Pipelining + Task Parallel design</td>
<td>28</td>
</tr>
<tr>
<td>13</td>
<td>Shared-Memory based Task Parallelism on CMSEARCH</td>
<td>33</td>
</tr>
<tr>
<td>14</td>
<td>Task Pipelining and Task Parallel performance comparison</td>
<td>33</td>
</tr>
<tr>
<td>15</td>
<td>Task Pipelining and Data Parallel performance comparison</td>
<td>34</td>
</tr>
<tr>
<td>16</td>
<td>Predicted stage 1 execution time and actual time comparison</td>
<td>35</td>
</tr>
</tbody>
</table>
INTRODUCTION

RNA structure can be described in terms of its primary (sequence), secondary (hairpins, bulges and internal loops), tertiary (A-minor motif, 3-way junction, pseudoknot, etc.) and quaternary structure (super-molecular organization) [32]. Lots of algorithms for homologous gene finding and prediction, such as BLAST [2] and Smith-Waterman algorithms [23], look for regions of similarity between the target sequence and possible candidate matches similarity search based on primary structure [31]. However, many interesting RNAs conserve a secondary structure of base-pairing interactions more than they conserve their sequence and RNA secondary structure [1]. This makes RNA sequence analysis more complicated and difficult than protein or DNA sequence analysis [14]. Infernal ("INFERence of RNA ALignment") is a powerful tool for searching DNA sequence databases for RNA structure and sequence similarities based on RNA secondary structure [9]. Infernal builds a profile from a structurally annotated multiple sequence alignment of an RNA family with a position-specific scoring system for substitutions, insertions, and deletions. Positions in the profile that are basepaired in the consensus secondary structure of the alignment are modeled as dependent on one another, allowing Infernal’s scoring system to consider the secondary structure, in addition to the primary sequence, of the family being modeled [12, 29]. Infernal profiles are probabilistic models called “covariance models” [3, 4, 26], a specialized type of stochastic context-free grammar (SCFG) [33].

Infernal is able to accurately detect more homologs compared to other alignment tools but it comes at more significant computational cost because it models the sequence structure. And also because biology data sequences are generally very large, the slow seraching speed has been a serious problem in
Infernal's program. The homolog search logic, CMSEARCH, is implemented in a filter pipeline in the current version (Infernal 1.1), which is used to improve the searching performance. It is claimed that the current Infernal is over 100 times faster than previous version. However, our research reveals that depending on the input files, RNA family files and target database sequence files, the performance of Infernal CMSEARCH can be significantly different. To address this problem, Infernal provides data parallel support which divides large database sequence files into smaller pieces and run in parallel by taking advantage of multi-threading and multi-processors technologies. But Infernal doesn't provide high performance task parallel way of CMSERACH. Plus, Infernal's data parallel strategy is the preliminary data parallelism by evenly dividing target database sequences without considering load balancing problem. By that, we mean, depending on the input files, longer CMSEARCH jobs should be assigned more threads or processors than shorted ones. The key to this solution is to predict the execution time of CMSEARCH jobs actually launching them. This challenging problem, forever, is stated explicitly in Infernal's User Guide[12, 29] that because it is hard, Infernal doesn't even try to make any prediction like that.

In this thesis, we propose and developed highly scalable task parallel and task pipelining algorithm that is compatible with Infernal's data parallel approach. We also propose to predict the running time of CMSEARCH jobs before they start using Multiple Linear Regression in order to achieve load balancing. This task is still ongoing but we've got very positive results doing very preliminary forecast. We've tested our program on bacteria RNA families from Rfam database.

In the rest of this thesis, I will first talk about more background information and related work regarding Infernal CMSEARCH. Next, I will explain our high
performance approach in details. After that, the practice and experiment results are given before I conclude the thesis work.
BACKGROUND

Infernal is first implemented by Sean R. Eddy in 2002. Before Eddy developed Infernal, some fast sequence homology search algorithms had already appeared such as BLAST and FASTA, but both are only search algorithms based on RNA primary structure for similarity between linear sequences. Eddy was working on finding homology from Group I Introns that don't share much similarity in their linear sequences but do have similar secondary structure. So algorithms like BLAST didn't work well for him. Inspired by Krogh/Haussler's paper for formalizing linear sequence alignments [10], Eddy implemented an HMM-liked structural RNA model, which later evolves to the CM used in Infernal. CM is a special type of Stochastic context-free grammars (SCFGs) [33] that reflects the base-pairing consensus structure of an RNA.

Infernal's CMSEARCH takes a CM file and a target database sequence file as its input. CM files are converted by Infernal from multiple alignment sequences that represents RNA families [9, 36]. A CM file contains two probabilistic models, CM model and its equivalent profile Hidden Markov Model (profile HMM). Profile HMM [profile hmm] derives from HMM [15] and is extended to a three dimensional model instead of linear model to accommodate insertion and deletion between homologous sequences. Infernal's CMSEARCH filter pipeline is based on both models and can be divided into profile HMM stages and CM stages. CMSEARCH reads data from both input files and passed into the pipeline to be processed with combinations of probabilistic model based algorithms including Viterbi algorithm [3, 15], Forward-Backward algorithm [15], CYK algorithm [15] and Inside algorithm[15]. All the algorithms are dynamic programming algorithms based on either profile HMM or CM. In this section, we will talk about Covariance
Model, profile Hidden Markov Model, CMSEARCH pipeline used in Infernal and related work.

**Covariance Model**

Infernal builds a CM based on a file containing multiple alignment sequences of one RNA family. The RNA family sequences are converted to a guide tree first before creating the covariance model. As we stated before, a CM is a specific stochastic context free grammar (SCFG), as shown in Table 1.

**Table 1. CM has seven types of states and production rules**

<table>
<thead>
<tr>
<th>State Type</th>
<th>Description</th>
<th>Production</th>
<th>Emission</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>(pair emitting)</td>
<td>$P \rightarrow aYb$</td>
<td>$e_v(a,b)$</td>
<td>$t_v(Y)$</td>
</tr>
<tr>
<td>L</td>
<td>(left emitting)</td>
<td>$L \rightarrow aY$</td>
<td>$e_v(a)$</td>
<td>$t_v(Y)$</td>
</tr>
<tr>
<td>R</td>
<td>(right emitting)</td>
<td>$R \rightarrow Ya$</td>
<td>$e_v(a)$</td>
<td>$t_v(Y)$</td>
</tr>
<tr>
<td>B</td>
<td>(bifurcation)</td>
<td>$B \rightarrow SS$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>(delete)</td>
<td>$D \rightarrow Y$</td>
<td>1</td>
<td>$t_v(Y)$</td>
</tr>
<tr>
<td>S</td>
<td>(start)</td>
<td>$S \rightarrow Y$</td>
<td>1</td>
<td>$t_v(Y)$</td>
</tr>
<tr>
<td>E</td>
<td>(end)</td>
<td>$E \rightarrow \varepsilon$</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

A SCFG consists of the following elements [12, 29]:

- M different non-terminals (here called states).
- K different terminal symbols.
- A number of production rules of the form: $V$, where can be any string of nonterminal and/or terminal symbols, including (as a special case) the empty string.
- Each production rule is associated with a probability, such that the sum of the production probabilities for any given nonterminal $V$ is equal to 1.
Figure 1 shows the conversion from consensus alignment sequences that represents RNA secondary structure, to guide tree and then to CM. The base pairs in the consensus structure are converted to MATP nodes; the unpaired residues are represented by either MATL or MATR nodes; BIF nodes always have two sub-trees that starts with BEGL and BEGR; Only one ROOT node is used as the head of the tree; There might be multiple END nodes in a guide tree.

**From Guide Tree to Covariance Model**

As in primary sequence alignment procedure, a CM also has to deal with insertions and deletions. Therefore, every node in a CM has different number of states depending the node type (Table 2).
Table 2. Nodes and states in a CM

<table>
<thead>
<tr>
<th>Node</th>
<th>States</th>
<th>Total # of states</th>
<th># of split states</th>
<th># of insert states</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATP</td>
<td>[MP ML MR D] IL IR</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>MATL</td>
<td>[ML D] IL</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>MATR</td>
<td>[MR D] IR</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>BIF</td>
<td>[B]</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ROOT</td>
<td>[S] IL IR</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>BEGL</td>
<td>[S]</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>BEGR</td>
<td>[S] IL</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>END</td>
<td>[E]</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The states are grouped into two different parts: 1-4 states in brackets and 0-2 states out of brackets. Basically the states out of brackets are insertion states and all other states are in brackets, such as match and deletion states.

State transition rules apply according to the following rules:

1. B states always make transitions to the S states of BEGL and BEGR nodes.
2. For other nodes, all states in brackets may transit to the insertion states of the same node or to every state in brackets of the next node.
3. IL states can do self-transitions, and transit to IR state in the same node or possibly transit to every state in brackets of the next node.
4. IR states can do self-transitions and transit to every state in brackets of the next node.

The final CM becomes a list of states connected in a directed graph (see Figure 1), which is similar with HMM. Each state in a CM has both emission and transition probabilities, and each state can have multiple parent and child states.

Figure 2 gives us the CM model data stored in a CM file.
A CM file consists of one or more CMs and associated filter HMMs. Each CM is immediately followed by its filter HMM, this is mandatory. Each CM starts with a format version identifier (here, INFERNAL1/a) and ends with // on a line by itself.

All emission and transition model parameters are stored as log-odds scores in bits with three digits of precision to the right of the decimal point, rounded. The special case of a score of infinity, corresponding to an impossible emission or transition, is stored as ‘*’.

The CM format is described in more detail below:
CM Header Section

The header section is parsed line by line in a tag/value format. Each line type is either mandatory or optional as indicated. I will only give the description of the fields from the header section that are very useful in our research. All details can be found in [12, 29].

**INFERNAL1/a** Unique identifier for the save file format version; the /a means that this is INFERNAL1 CM file format version a. When INFERNAL changes its save file format, the revision code advances.

**NAME <s>** Model name; <s> is a single word containing no spaces or tabs. The name is normally picked up from the #=GF ID line from a Stockholm alignment file. If this is not present, the name is created from the name of the alignment file by removing any file type suffix. For example, an otherwise nameless CM built from the alignment file tRNA.sto would be named tRNA. Mandatory.

**STATES <d>** Number of states; <d>, a positive nonzero integer, is the number of states in the model. Mandatory.

**CLEN <d>** Consensus model length; <d>, a positive nonzero integer, is the number of consensus positions in the model, which equals the number of MATL nodes plus the number of MATR nodes plus two times the number of MATP nodes. Mandatory.

**W <d>** Window length; <d>, a positive nonzero integer, is the length in residues of the maximum expected size of a hit to this model. This is calculated based on the transition probabilities of the model. Mandatory.
**ALPH <s>**

Symbol alphabet type. Currently this will necessarily be RNA for RNA sequence analysis models. The symbol alphabet size K is set to 4 and the symbol alphabet to “ACGU”. Mandatory.

**CM Flags the start of the main model section. Mandatory.**

**CM Main Model Section**

The model section consists of two types of lines: node lines and state lines. Each node line is immediately followed by one or more state lines, one each for each state within the node.

Each node line begins with 45 spaces, and includes ten fields.

The first field is always a [ character. The second is the node type, one of ROOT, MATP, MATL, MATR, BIF, BEGL, BEGR, or END.

The next field is the index of the node in the model, greater than or equal to 0. Node indices are not always in increasing order, e.g. node 200 may come on a line before node 100. The fourth field is always a ] character.

The next two fields are the MAP annotation for this node. If MAP was yes in the header and the node is a MATP (match pair) node, then these fields will both be positive integers, representing the alignment column indices for the left and right halves of this match pair state, respectively.

The next two fields are the CONS consensus residue(s) for this node. If CONS was yes in the header and the node is a MATP (match pair) node, then these fields will both be characters, the consensus residues for the left and right halves of this match pair state, respectively.

The final two fields are the RF annotation for this node. If RF was yes in the header and the node is a MATP (match pair) node, then these fields
will both be characters, the reference annotation character for the left and right halves of this match pair state, respectively.

Each node line is followed by 1 to 6 state lines depending on the node type. ROOT, MATL, and MATR node lines are followed by 3 state lines. BIF, BEGL, and END nodes are followed by 1 state line. BEGLR node lines have 2 state lines after them, and MATP node lines are followed by 6 state lines.

The number of fields on a state line is variable depending on the state type and the number of possible transitions from the state.

The first field is the state type, either “MP”, “ML”, “MR”, “IL”, “IR”, “D”, “B”, “S”, or “E”.

The next field is the state index; these are in increasing order starting with 0 (i.e. lower numbered states always occur earlier in the file than higher numbered ones). The next field is the index of the highest numbered “parent” state for the current state, where state a is a parent of state b if state a can transition to state b.

The next field is the number of parent states for the current state. A set of parent states are always contiguously numbered. For example, if state a is the highest numbered parent state of b and b has 3 parent states, then a − 2, a − 1, and a are the three parent states of b.

The next field is the index of the lowest numbered “child” state for the current state, where state c is a child of state b if b can transition to state c.

The next field is the number of child states for the current state. A set of child states are always contiguously numbered. For example, if state c is the lowest numbered parent state of b and b has 3 parent states, then c, c + 1, and c + 2 are the three child states of b. As a special case, for “B” (bifurcation) states this field is the state index of the “BEGR S” state to which the “B” state necessarily transitions with probability 1.0.
The next four fields \(<n1>\), \(<n2>\), \(<n3>\), and \(<n4>\) are query dependent band values for the current state. By default, they are not used.

After the four QDB values, the next set of fields are log-odds bit scores for possible transitions out of this state to all child states of the current states. The number of child states is given earlier on the line as the sixth field. It varies depending on the state type and the node type of the next node in the model. As a special case, “B” (bifurcation) states have zero transition score fields, they necessarily transition to their child “BEGL S” and “BEGR S” states with a probability of 1.0 (score of 0 bits).

After the transition scores are the emission scores. “MP” state lines have 16 emission log-odds bit scores. All other types of emitting states (“ML”, “MR”, “IL”, “IR”) will have four emission scores. All other types of states will have no emission scores. For “MP” states, the sixteen scores are for the sixteen possible nondegenerate RNA basepairs: “AA”, “AC”, “AG”, “AU”, “CA”, “CC”, “CG”, “CU”, “GA”, “GC”, “GG”, “GU”, “UA”, “UC”, “UG”, “UU”, in that order. For the other emitting states, the four scores are for “A”, “C”, “G”, and “U”, in that order. Finally, the last line of the format is the “//” record separator.

Profile Hidden Markov Model

Hidden Markov Model (HMM) [3] is a statistical model that consists of sets of states that emit items and transit over to other states. Figure 3 gives us a simple HMM that consists of two nodes. An HMM is usually an infinite linear sequence in which at each position it can emit items/symbols and transit from one state to other states at the next position. One characteristics of HMM is only the symbol sequence is visible and the state sequence is hidden. Another important property of
HMM is at each position of the model, the probability of a state only depends on the previous position.

\[ P_{jk} = P(\pi_i = k | \pi_{i-1} = j) \]

where \( P_{jk} \) is the transition probability from \( j \) to \( k \) at position \( i \), \( \pi_i \) is the \( i \)th state in the HMM path.

Figure 3. A toy Hidden Markov Model

HMMs were introduced to computational biology in the late 1980s [19], and for use as profile models from early 1990s [18]. Profile HMMs merge the ideas HMMs with Gribskov’s profiles [16]. Given an alignment of sequence family members, in this case, RNA Family, a profile HMM can be built for that family. In profile HMMs, each node represents a consensus position of the multiple alignments. Every node has three states: a match state, an insert state and a delete state. Both match states and insert states emit residues but insert states generate residues in between two consensus positions and allow self-transitions. Delete states don’t emit residues at consensus positions and transit over to next consensus positions. The structure of the model is shown in Figure 4. The bottom line of states is called the main states, because they model the columns of the
alignment. The second row of diamond shaped states are called insert states and are used to model highly variable regions in the alignment. The top line of circular states is called delete states. These are a different type of state, called a silent or null state. They do not match any residues, and they are there merely to make it possible to jump over one or more columns in the alignment, i.e., to model the situation when just a few of the sequences have a ‘–’ in the multiple alignment at a position.

Figure 4. The structure of profile HMM.

Profile HMM structure in CM file

In a CM file, the profile HMM of the corresponding RNA family is also generated and is used by CMSEARCH pipeline. Figure 5 shows the structure of profile HMM data stored in a CM file [12, 29].

Just like CM data section in CM file, profile HMM data has header section and main section. All HMM probability parameters are all stored as negative log probabilities with five digits of precision to the right of the decimal point, rounded. For example, a probability of 0.25 is stored as −log 0.25 = 1.38629. The special case of a zero probability is stored as ‘∗’. Again, I will only go through the fields that have been used in our research.
**Figure 5.** profile HMM data stored in CM file

### HMM Header Section

**HMMER3/f**

Unique identifier for the save file format version; the /b means that this is HMMER3 HMM file format version b. When HMMER3 changes its save file format, the revision code advances. This way, parsers may easily remain backwards compatible. The remainder of the line after the HMMER3/b tag is free text that is ignored by the parser. HMMER currently writes its version number and release date in brackets here, e.g. [3.0b2 | June 2009] in this example. Mandatory.

**NAME <s>**

Model name; <s> is a single word containing no spaces or tabs. The name is normally picked up from the #=GF ID line from a Stockholm alignment file. If this is not present, the name is created from the name of the alignment file by removing any file type suffix. For example, an otherwise nameless HMM built from the alignment file rrm.slx would be named rrm. Mandatory.

**LENG <d>**

Model length; <d>, a positive nonzero integer, is the number of match states in the model. Mandatory.

**COMPO <f>*K**

The first line in the main model section may be an optional line starting with COMPO: these are the model’s overall average match state emission probabilities, which are used as a background residue composition in the “filter null” model. The K fields on this line are log probabilities for each residue in the appropriate biosequence alphabet’s order. Optional.
**HMM Main Model Section**

The first two lines in the main model section are atypical. They contain information for the core model’s BEGIN node. This is stored as model node 0, and match state 0 is treated as the BEGIN state. The begin state is mute, so there are no match emission probabilities. The first line is the insert 0 emissions. The second line contains the transitions from the begin state and insert state 0.

The remainder of the model has three lines per node, for M nodes (where M is the number of match states, as given by the LENG line). These three lines are (K is the alphabet size in residues):

- **Match emission line** The first field is the node number (1 . . . M). The parser verifies this number as a consistency check (it expects the nodes to come in order). The next K numbers for match emissions, one per symbol, in alphabetic order.
  
  The next field is the MAP annotation for this node. If MAP was yes in the header, then this is an integer, representing the alignment column index for this match state (1..alen); otherwise, this field is ‘-’.
  
  The next field is the CONS consensus residue for this node. If CONS was yes in the header, then this is a single character, representing the consensus residue annotation for this match state; otherwise, this field is ‘-’.
  
  The next field is the RF annotation for this node. If RF was yes in the header, then this is a single character, representing the reference annotation for this match state; otherwise, this field is ‘-’.
  
  The next field is the CS annotation for this node. If CS was yes, then this is a single character, representing the consensus structure at this match state; otherwise this field is ‘-’.

- **Insert emission line** The K fields on this line are the insert emission scores, one per symbol, in alphabetic order.

- **State transition line** The seven fields on this line are the transitions for node k, in the order shown by
the transition header line: \( M_k \rightarrow M_{k+1}, M_k \rightarrow I_k, M_k \rightarrow D_{k+1}, I_k \rightarrow M_{k+1}, I_k \rightarrow I_k, \)
\( D_k \rightarrow M_{k+1} \) and \( D_k \rightarrow D_{k+1} \). For transitions from the final node \( M \), match state \( M + 1 \) is interpreted as the END state \( E \), and there is no delete state \( M + 1 \); therefore the final \( M \rightarrow D \) and \( M \rightarrow D \) transitions are always \(*\) (zero probability), and the final \( M \rightarrow D \) transition is always \( 0.0 \) (probability 1.0).

Finally, the last line of the format is the “/” record separator.

Infernal CMSEARCH Pipeline

The core logic of Infernal CMSEARCH is implemented in a filter pipeline (see Figure 6). The pipeline contains profile HMM filter stages and CM stages. Profile HMM model is used as filters first because it's computationally less expensive than CM model. The target database sequences are split into smaller subsequences and passed to each stage. In each stage, different algorithms will run the subsequences against either profile HMM and CM to compute the scores. If the score hit certain threshold value, the corresponding subsequences are passed to the next stage. Otherwise, they are skipped. Each earlier stage is designed to be more efficient than the later stages so this filter pipeline can accelerate the searching speed [12, 29].

The first stage in profile HMM filter stages is called Local Scanning SSV (Single Segment Viterbi) filter. Viterbi algorithm[15] is a dynamic programming (DP) algorithm that calculates the maximum likelihood of a certain sequence generated by a HMM. Here Viterbi algorithm is used to find local un-gapped alignments with high scores against the profile HMM model. After running the SSV algorithm against the target database sequence, we get multiple un-gapped subsequences. Overlapping subsequences are merged together and then split into windows with length \( L \), where \( L \) is twice of the window length defined in CM model.
Windows that survive SSV filter are passed to the next stage, Local Viterbi filter, in which although we still apply Viterbi algorithm, gapped alignment score is calculated.

If the score passed the set threshold value, the window is passed to BiasedComposition filter. The biased Composition filter is used to reduce false positive due to biased composition sequences. For example, a sequence can't be highly rich in any of the four nucleotide amino acids "A", "C", "G" and "U". A two-state HMM is constructed to detect the composition bias and the sequence is rescored using HMM Forward algorithm if required. This composition bias filter score is also used in the next two stages in the pipeline: Local Forward and Global Forward stages.
Windows whose scores passed Biased Composition filter are passed to the next stage. Now the full Forward algorithm is applied with the profile HMM with a local mode in the model, which means the alignments can start and stop at any state of the model. In Forward algorithm, all possible alignments are summed up to calculate the score. This is also essentially the difference between Forward algorithm and Viterbi algorithm since in Viterbi algorithm only the alignment with the maximum likelihood is selected.

The next stage is Global Forward filter, and by the name we know the surviving windows are now aligned to profile HMM with global mode, which is the windows should always start at the first position and end at the last position of the model. But it's still local against the sequence windows.

The next stage is Global Backward parser, which is quite similar with Global Forward parser. In Forward algorithm, the summed likelihood of all prefixes in the target sequence is calculated, while in Backward algorithm, the likelihood of all suffixes are summed up. Based on the results from doing Forward and Backward parsers, the posterior probability of alignments starts and ends at sequences positions are calculated so that sequence windows are further divides into smaller subsequences called envelops, which are defined to have significant probability mass of a match in the target sequences with regard to the profile HMM. This completes the profile HMM filter stages.

The next stage is called HMM banded CM CYK filter and is the most time consuming part in the program for long time CMSEARCH jobs based on my study and experiments. In this stage, a banded three dimensional dynamic programming algorithm, CYK algorithm, is implemented. The algorithm starts with the End state of CM model defined in the CM file and recursively calculates the log probability of the most likely CM sub-tree rooted at the current state $v$ that
generates the length d subsequence $X_j - d + 1 \ldots X_j$ that ends at position j of target sequence $x[3, 4, 7, 14, 26]$.

Envelops that survive this stage are passed to the last stage of the pipeline, HMM banded CM inside filter. CM inside algorithm is the CM analog of HMM Forward algorithm with all possible alignments being summed up to determine the probability of the subsequences. The hits that passed Inside filter is added to the final hit list.
PROPOSED HIGH PERFORMANCE APPROACH

Although Infernal's newly introduced filter pipeline has significantly improved the performance of Infernal CMSEARCH, the computation of CMSEARCH program is still considerably expensive. In original Infernal serial way of CMSEARCH implementation, target sequence data is processed sequence by sequence. For example, one database sequence file may contain more than one sequences. A block size of 100,000 residues is defined when reading from the target database sequence file. If a sequence is a lot longer than the default block size, it is broken to several blocks and passed into CMSEARCH pipeline stages one by one. If a sequence size is less than 100,000, the entire sequence is processed by CMSEARCH pipeline without reading any residues from next sequence. Because some sequences can be very short, e.g. thousands of residues, if we implement our multi-threading version, there will be more thread creation, destroying and communications, which will introduce more overhead. In order to minimize the overhead introduced by multi-threading, we firstly enlarged the data size by making sure every time a block is read from the sequence file, we read at least a full block, which is 100,000 residues long, except the last block. In this case, even one sequence is significantly shorter than block size, we keep reading the next sequence from the input sequence file until the block is full or reaching the end of the sequence file.

Infernal has applied data parallel method to CMSEARCH to improve the performance by using multi-threading or multiple nodes in a cluster system. With this approach, the large sequence files are divided into smaller pieces and each will launch a CMSEARCH pipeline independently either in a separate thread or computer.
**Shared-Memory Based Acceleration**

Infernal's CMSEARCH filter pipeline is one type of preprocessing and filtering idea. Each stage in the pipeline is designed to run more efficient than later ones. We tested over 500 randomly chosen input combinations of CM files and target database sequence files and measured the execution time of every single stage of CMSEARCH pipeline respectively. Figure 7 gives us an illustration of execution time of all stages of CMSEARCH pipeline, and only 35 results are shown here for demonstration purposes. Observed that the third stage of CMSEARCH pipeline takes up the most time especially for long CMSEARCH jobs, we propose shared-memory based task parallel CMSEARCH program by assigning multiple threads to the third stage of CMSEARCH pipeline in order to accelerate CMSEARCH pipeline performance. We developed shared-memory version of task parallel using Pthread and OpenMP separately. This approach also lays the foundation to develop the optimized task pipelining version. Because the significance of the third stage in CMSEARCH pipeline, we implemented the task parallel strategy by allocating 2 threads to the third stage and keep all other stages unchanged (see Figure 8). The implementation steps are explained in details below.

However, our research on CMSEARCH filter pipeline suggests that the pipeline can be accelerated internally through multi-threading technologies running in a multiple core system. This type of task parallel will make the pipeline itself be executed more efficiently. We developed shared-memory way of task parallel to improve CMSEARCH pipeline speed with different multi-threading techniques. Because CMSEARCH is implemented in a pipeline, we also developed task pipelining version to speed up CMSEARCH program by running all pipeline stages in parallel.
Figure 7. Elapsed time of CMSEARCH pipeline stages with sample data

Figure 8. Task parallel implementation
Task Parallel Implementation with Pthread

Step 1. Create a function that wraps up function pli_cyk_env_filter called pli_stage_3, in which there's only one argument with type (void *).

Step 2. In CMSEARCH pipeline, after stage 2 is completed, create two Pthreads that will both run pli_stage_3 at the same time.

Step 3. Compute the specific positions each thread should start and end in the input sequences. One thread will process the first half of the block of sequences output from stage 2 and the other one will process the second half. The parameters passed to pli_stage_3 should involve the point to the actual block of sequences as well as the start and end positions that each thread will process. That is the thread that will process the first half of the sequences will get start and end positions as the first position and the middle position of the block. The other thread will have start position at the position right after the middle position and the end position at the last residue of the sequences.

Step 4. After both finishes CYK algorithm, combine the result from two threads and join them.

Step 5. Repeat Step 3 until all sequences are processed.

Task Parallel Implementation with OpenMP

Step 1. In the beginning of pipeline stage 3, before running pli_cyk_env_filter, call function omp_set_num_threads and set the number of threads to two.

Step 2. Use keywords #pragma omp parallel to wrap up the part that calls pli_cyk_env_filter.

Step 3. Inside the wrapped code block, distinguish the thread id by calling
method omp_get_thread_num. If the result is 0, run pli_cyk_env_filter on the first half of the input block, otherwise run it on the second half of the input block.

Step 4. Repeat Step 3 until all sequences are processed.

Exploiting Pipeline Parallelism

Inspired by the instruction level pipelining design in modern CPU, we developed Task Pipelining using multi-threading technologies. In CPU pipeline, there are usually five different stages: Instruction Fetch stage, Instruction Decode stage, Instruction Execute stage, Memory stage and Write-Back stage (see Figure 9). The hardware level pipelining runs the five stages in parallel on different instructions at the same clock(s). For example, when instruction i is done with Instruction Decode stage, it will be passed to Instruction Execute stage while instruction i+1 enters Instruction Retrieve stage. With this design, CPU throughput is greatly improved. In Infernal’s CMSEARCH pipeline implementation, the eight stages we described in previous section is grouped into four logical stages and written in functions: pli_p7_filter, pli_p7_env_def, pli_cyk_env_filter, pli_final_stage (see Figure 10). Likewise, we propose Task Pipelining way to execute CMSEARCH pipeline stages in parallel by assigning separate threads to each individual stage. Task Pipelining becomes a chained Producer-Consumer problem in that each stage is a producer for the next stage except the final stage while each stage except the first stage is a consumer of the previous stage. Our initial approach was assigning one thread for each of the four functions in CMSEARCH program (see Figure 11). However, because the third stage sometimes could be remarkably more time-consuming than the other two, it becomes the bottleneck in our design. The details of the experimental results are shown in the next section.
Figure 9. CPU pipeline

Figure 10. Infernal CMSEARCH pipeline implementation
In the very beginning of Task Pipelining development, we assigned one thread to each of the stages in CMSEARCH pipeline. With this approach we failed to achieve good performance improvement and for long jobs, they even got slower. This means applying task pipelining directly on CMSEARCH pipeline introduced more overhead instead of improving the efficiency. The reason for that is that the execution time of different stages in CMSEARCH pipeline is not equally distributed, especially for longer jobs. The key to achieve optimal performance gain in a pipeline is to make sure all stages perform equally or similarly fast.

Therefore, we combined the two most time significant stages into one, making the original four stages pipeline into three stages. Then we aim to apply multi-threading on the second stage of our new pipeline. If we can make the execution time of the second stage small enough, we can achieve great efficiency in pipelining task parallelism. This design of Task Parallel plus Task Pipelining is shown in Figure 12.

The implementation of our Task Pipelining paradigm is explained in the following subsection, after Figure 12.
Task Pipelining Implementation with Pthread

**Step 1.** Create a wrap-up function for each function in the revised CMSEARCH pipeline respectively. Namely, pli_stage_1 for pli_p7_filter, pli_stage_2 for pli_p7_env_def and pli_cyk_env_filter and pli_stage_3 for pli_final_stage.

**Step 2.** Before entering the pipeline, create three threads for the three wrap-up functions respectively using Pthread_Create.

**Step 3.** In pli_stage_2, use Pthread_Create to create a wrap-up function, namely, pli_stage_2_task_parallel, used to call pli_p7_env_def and pli_cyk_env_filter on different dataset.

**Step 4.** After pli_stage_1 is done, in pli_stage_2, compute the start and end positions of the input data block and pass to the two threads that will run pli_stage_2_task_parallel function respectively. One thread should execute on the first half of the data block, and the other one should execute on the second half.

**Step 5.** Combine the results from the two threads and join them.
Step 6. Repeat Step 4 until all sequences are processed.

Task Pipelining Implementation with OpenMP

This approach is implemented in a similar way. Because OpenMP automatically handles thread creation and joining, it is easier to implement than Pthread version.

Here are the implementation steps in details.

Step 1. Create a wrap-up function for each function in the revised CMSEARCH pipeline respectively. Namely, pli_stage_1 for pli_p7_filter, pli_stage_2 for pli_p7_env_def and pli_cyk_env_filter and pli_stage_3 for pli_final_stage.

Step 2. Call omp_set_num_threads function and set the number of threads to three.

Step 3. Call omp_get_thread_num function to get the thread rank id. If the rank id is 0, prepare data and enter the pipeline; If the rank id is 1, call pli_stage_1; If the rank id is 2, call pli_stage_2; Otherwise, call pli_stage_3.

Step 4. In pli_stage_2, call omp_set_num_threads to set the number of threads to two; Use keyword #omp pragma parallel to wrap up the code block that calls pli_p7_env_def and pli_cyk_env_filter.

Step 5. In the OpenMP wrapped code block, call omp_get_thread_num to get the thread rank id. If the rank id is 0, execute the rest of the program on the first half of the input sequence block; Otherwise, execute the rest of the program on the second half of the input sequence block.

Step 6. Repeat Step 4 until all sequences are processed.
Task Pipelining Implementation with Pthread and OpenMP

In this hybrid approach, we create three Pthreads for the pipeline stages and in pli_stage_2 we use OpenMP to do the Task Parallel logic. The implementation steps can be derived from the previous two approaches.

Predicting CMSEARCH Execution Time with Multiple Linear Regression model

So far we described our CMSEARCH Task Pipelining implementation by statically assigning two threads for the second stage. However, depending on the input CM files and target database files, the ratio of the execution time of the second stage, which is the most time-consuming part of the pipeline, over the other two stages vary a lot. If we assign the same number of threads to pipeline stage 2, it's not guaranteed that we can always get the optimal performance gains. For example, if the stage 2 takes ten times longer than the other two stages, allocating five threads will most likely yield better performance than two threads. If stage 2 has similar execution time with the other two, it will be better not allocate more threads because it will only introduce more overhead. That means, our Task Pipelining approach can be further improved if we can dynamically determine the optimal number of threads we should assign to the most time-consuming stage in the pipeline, before running CMSEARCH jobs.

Based on Infernal CMSEARCH logic we have explained in the previous section, the performance of each stage highly depends on the output of previous stage except the very first one. Any changes in either CM file or target sequence file may lead to complete different output in each stage of the pipeline. We started with building regression model for the first stage of the pipeline, namely, pli_p7_filter in the implementation. We observed that the pli_p7_filter highly relies on the length of profile HMM, the length of consensus sequence in CM, the
window length defined in CM file and the size of target sequences. These four factors can all be retrieved by simply reading the CM file and target sequence file. The tool we used to build the linear regression model is Scikit-learn, a machine learning library in Python.

Below are the steps of building and testing linear regression model.

**Step 1.** Run Infernal's original serial version CMSEARCH pipeline on 500 combinations of CM files and database sequence files and compute the execution time of each stage.

**Step 2.** Write a Python program to read CM file and target sequence file to get the length of profile HMM, the length of consensus sequence in CM, the window length defined in CM file and the size of target sequences.

**Step 3.** In Python, create a list, namely training_x, in which every item is a list that stores the values of the four variables we read from Step 2.

**Step 4.** In Python, create a list, namely training_y, in which every item is the execution time of CMSEARCH pipeline stage 1. This list should match the training_x in the same order.

**Step 5.** In Python, call scikit-learn.linear_model.linear_regression to create a linear regression model, namely rgm and then call rgm.fit function on training_x, training_y.
PRACTICE AND EXPERIMENTAL RESULTS

Our practice was done on a server, PowerEdge T620 system, which is installed with 16 cores Intel Xeon E5-2670 processor with 2.60GHz and 32 GB Memory. The database was selected from Human Microbiome Project (HMP) and we divided the HMP database file into smaller pieces. Each database piece is approximately 120 MB large. The RNA families we were testing on are bacteria families. For the high performance model, we tested OpenMP, Pthread and OpenMP plus Pthread paradigms.

In this section, we will show our experiment results of our Shared Memory Based Task Parallel and Task Pipelining approaches that have remarkably improved CMSEARCH performance.

Shared-Memory based Task Parallel Results

Figure 13 gives us the performance comparison between Pthread and OpenMP approaches. We can see the two methods has both achieved similar performance gains by improving CMSEARCH pipeline performance by around 10 to 40 percent depending on the input files.

Task Pipelining Results

Figure 14 shows the performance comparison between the three Task Pipelining version, the Task Parallel itself and the original CMSEARCH version. We can see that with different multi-threading techniques, we have achieved similar performance gains with Task Pipelining. The performance improvement Task Pipelining itself is also remarkable.
Figure 13. Shared-Memory based Task Parallelism on CMSEARCH

Figure 14. Task Pipelining and Task Parallel performance comparison
Task Pipelining with Data Parallelism

Our Task Pipelining version of CMSEARCH is highly scalable as it can achieve the same performance gain working with Infernal's original Data Parallel version. We tested the scalability by launching two pipelines at the same time. This is accomplished by allocating two threads using Infernal's Data Parallel strategy and in each thread, we execute the new pipeline we developed, instead of calling the original Infernal pipeline functions. Figure 15 shows that the two pipelines CMSEARCH is able to improve the performance by about 50 percent compared to Infernal's original Data Parallel version with two threads. This approach can be extended to as many threads as possible.

![Task Pipelining and Data Parallel performance comparison](image)

Figure 15. Task Pipelining and Data Parallel performance comparison

Infernal Pipeline Stage 1 Prediction Results with Linear Regression

After the linear regression model is created, we collected our test dataset of randomly chosen 50 combinations of CM files and database sequences. The R-Squared score we've got is over 0.8 (the maximum score is 1, which means the
exact match between predict value and the actual value). Figure 16 shows the comparison between our predicted running time of pipeline stage 1 and the actual time.

Figure 16. Predicted stage 1 execution time and actual time comparison

By the time of this writing, we're still testing and modifying our regression model. In addition, we also need to do prediction on the second stage of our new pipeline. This is more difficult because of the reasons we've explained.
CONCLUSION AND DISCUSSION

We presented high performance computation models for accelerating a popularly used gene prediction tool, Infernal CMSEARCH, with task parallel and pipelined approaches. Since CMSEARCH is very sophisticated software with complex implementation, analysis the code and developing high performance version took considerable amount of time. From the original CMSEARCH pipeline, our task parallel approach attacked the most time consuming part, searching covariance model with 3D dynamic programming, and the pipelined parallel approach implemented a software-pipeline among those stages of operations used in the tool. To achieve the optimum performance improvement from the pipelined version, we are developing a method of dynamically determining the number of threads to assign to each stage of the pipeline. Currently, we are working on building an optimal regression model to predict the execution time of the most time-consuming stage of the pipeline. We expect that we could make a sophisticated heuristic methodology in the near future.

In our practice, we achieved significant performance gain with varying multi-threading and pipelined technologies. We achieved approximately 20% and 50% performance improvement on Infernal CMSEARCH with our task parallel and task parallel plus pipelined approaches, respectively. We also tested and demonstrated that our proposed approaches are scalable (on varying number of cores in the system) and flexible for the hybrid usage with trivial task parallel approaches.

Future Work

Currently, we are working on developing a MapReduce version of our approaches targeting cloud systems. For the load balancing on the nodes in the
cloud system, we are working on developing a sophisticated execution time prediction methodology based on the covariance models of small RNAs.
REFERENCES


Fresno State

Non-Exclusive Distribution License
(to archive your thesis/dissertation electronically via the library’s eCollections database)

By submitting this license, you (the author or copyright holder) grant to Fresno State Digital Scholar the non-exclusive right to reproduce, translate (as defined in the next paragraph), and/or distribute your submission (including the abstract) worldwide in print and electronic format and in any medium, including but not limited to audio or video.

You agree that Fresno State may, without changing the content, translate the submission to any medium or format for the purpose of preservation.

You also agree that the submission is your original work, and that you have the right to grant the rights contained in this license. You also represent that your submission does not, to the best of your knowledge, infringe upon anyone’s copyright.

If the submission reproduces material for which you do not hold copyright and that would not be considered fair use outside the copyright law, you represent that you have obtained the unrestricted permission of the copyright owner to grant Fresno State the rights required by this license, and that such third-party material is clearly identified and acknowledged within the text or content of the submission.

If the submission is based upon work that has been sponsored or supported by an agency or organization other than Fresno State, you represent that you have fulfilled any right of review or other obligations required by such contract or agreement.

Fresno State will clearly identify your name as the author or owner of the submission and will not make any alteration, other than as allowed by this license, to your submission. By typing your name and date in the fields below, you indicate your agreement to the terms of this distribution license.

Embargo options (fill box with an X).

[ ] Make my thesis or dissertation available to eCollections immediately upon submission.

[ ] Embargo my thesis or dissertation for a period of 2 years from date of graduation.

[ ] Embargo my thesis or dissertation for a period of 5 years from date of graduation.

Type full name as it appears on submission

[ ] Linqiang Ouyang

Date

Apr. 25, 2016