Statistical Methods for the Discovery of Co-operative Transcription Factors: the Co-bind Code Revised

Giovanni Lavorgna
Biotech. Dept.
Hospital San Raffaele
Milan
lavorgna.giovanni@hsr.it

Alessandro Marongiu
ENEA
Informatic Unit
Portici,
Ylichron s.r.l, Rome
alessandro.marongiu@portici.enea.it

Simone Melchionna
Albatel SpA – Rome
and
INFM University "La Sapienza",
Physics Dept
Rome
simone.melchionna@roma1.infn.it

Paolo Palazzari
ENEA
Computing and Modeling Unit, Rome,
Ylichron s.r.l., Rome
palazzari@casaccia.enea.it

Vittorio Rosato
ENEA
Computing and Modeling Unit
Rome
Ylichron s.r.l., Rome
rosato@casaccia.enea.it

Paolo Verrecchia
Albatel SpA
Rome
pverrecchia@hotmail.com

Abstract

Discovering co-operative Transcription Factors (TF's) within the genome is a computationally challenging problem, tackled through Monte Carlo-like analysis by the Co-Bind code, developed at the Department of Genetics of the St. Louis Washington University. Due to its statistical nature, Co-Bind is characterized by very long execution times, order of days on current high-end workstations, and could benefit from parallelization and a wise optimization, performed at both the algorithmic and coding levels.

This work presents the results achieved by parallelizing Co-Bind and optimising the parallel code and shows that, on a 16-processor architecture, a speed-up greater than two orders of magnitude is achieved with respect to the serial version released by the code's authors.

1. Introduction

Genes having similar expression profiles are generally considered to have transcription mechanisms activated by similar transcription factors (TF). The discovery of TF sites from a collection of DNA sequences is a major task in bioinformatics. TF's are often present in an unknown mutated form (typically a string of length 8-12 base pairs with 2-3 mutations) and in unknown positions along the sequence. There exist several methods to locate TF's, usually grouped into two distinct classes: greedy methods to enumerate all potential recurrent strings within different DNA sequences, and non deterministic methods which preferentially sample strings having a high degree of recurrence. Both strategies have their pros and cons. Given the exceedingly large number of recurrent candidates, exact methods are based on the introduction of different heuristics to prune the search space, either by searching only for a subset of all the possible patterns or by imposing restrictions on the locations of mismatches [4][6][7]; usually these methods work well for short patterns and for a limited number of mismatches.

On the contrary, statistical methods are based on the well known Gibbs sampling strategy [2][3][8], a closely related Monte Carlo scheme, well suited to locate states bound by conditional probabilities. These methods are particularly well suited to identify recurrent patterns which are present in the sequence set (the positive set) but not in the background (i.e. the negative set, a collection of DNA sequences not necessarily containing the binding sites, usually constituted by the entire genome). Therefore, one can distinguish a "true positive" solution (a TF present only in the positive set) from a "false positive" solution,

the latter class containing the vast majority of recurrent candidates.

TF binding sites are often organized in functional groups which can operate in a co-operative fashion by simultaneous interaction of two closely situated target sites. The main difficulty in detecting pairs of TF's relies on the fact that the relative position of co-operative TF's is generally unknown. Therefore, the search of potential pairs of TF's is an intensive computational task which cannot be done by exact enumeration. Furthermore, two sites might have individually low signal-to-noise ratio, while the simultaneous occurrence of both signals greatly enhances their statistical significance.

A successful method to locate pairs of putative TF's has been developed by Stormo and collaborators [1], and the software Co-Bind represents a practical implementation of the underlying theory. Co-Bind works on a statistical basis, relying on an intensive Gibbs sampling search starting from different initial guesses.

As we will show in the following, Co-Bind relies on the estimate of the statistical weight of strings on both the positive and the negative set, as achieved by a Monte Carlo-like algorithm, therefore the software is characterized by a very high computing time, requiring up to days to perform analysis of biologically relevant data. As a consequence, optimization and parallelization of the code are mandatory.

The present work will firstly recall the theory at the bases of the Co-Bind code and then present the actions we have performed to speed-up the calculation.

2. The Co-Bind method

The essential ingredient of Co-Bind is the definition of a weight matrix [5] $\omega_{k,b}$ where k=0,...,l-1 (l being the length of the pattern) and b ranges over all four DNA bases (A,C,G,T). This matrix will contain the consensus of each letter as found along the recurrent pattern as the calculation proceeds. In total, Co-Bind operates with two weight matrices, one for each binding site. Furthermore, it is assumed that the binding energy for a TF molecule to a particular site is given by the following potential energy:

$$H_{i,j} = \sum_{k=0}^{l-1} \sum_{b=0}^{3} \omega_{k,b}^{\alpha} \cdot x_{k,b}^{j} \qquad (1)$$

where α =1,2 and $x_{k,b}^{j}$ is defined as

$$x_{k,b}^{j} = \begin{cases} 0 & if\ s(j+k) \neq b \\ 1 & if\ s(j+k) = b \end{cases}$$

being $s(j+k)$ the nucleotide at position $j+k$ in sequence s ($s(j)$ is the starting nucleotide of the DNA pattern at offset j along the DNA sequence). In total, the software handles p sequences ($i=1,...,p$) each containing a pair of TF's.

It is further assumed that the binding probability of a single TF follows the Boltzmann distribution, $\propto P_j e^{-H_j}$, where P_j is the likelihood of binding the pattern in the background, so that one can write the partition function for the joint occurrence of the two patterns in both the positive and negative sets

$$Y^C = \sum_{i=1}^{p} \sum_{j,j'} P_j P_{j'} e^{-(H_{i,j}+H_{i,j'})} \qquad (2)$$

The marginal probability that both factors simultaneously occupy their respective binding sites in the positive set, at offsets j and j' of sequence i, can be written as

$$b_i^C = \frac{e^{-(H_{i,j}+H_{i,j'})}}{Y^C} \qquad (3)$$

Here the numerator is calculated over the positive set, while the denominator over the background set. Next, the method optimizes the marginal probability with respect to the recurring representative patterns. Therefore, the objective function, defined as

$$U_C = -\frac{1}{p} \sum_{i=1}^{p} \left(H_{i,j} + H_{i,j'}\right) - \boldsymbol{ln}\left(Y^C\right) \qquad (4)$$

is maximized with respect to the weight matrices via a steepest descent algorithm

$$\left[\omega_{k,b}^{\alpha}\right]^{(n+1)} = \left[\omega_{k,b}^{\alpha}\right]^{(n)} + \eta \left[\frac{\partial U_C}{\partial \omega_{k,b}^{\alpha}}\right]^{(n)} \qquad (5)$$

In essence, Co-Bind estimates separately the partition function YC for the background and subsequently optimizes with respect to the weights matrices. Given the parametric dependence of the partition function on the weight matrices, this implies the iterative repetition of these two computational steps.

The pseudo code of the algorithm is the following:

input:
- positive set constituted by p sequences each containing the two TF's
- background set

output:
- optimized weight matrices, representing the two TF's

```
for each initial condition do
{
  for i=0 to Max_i
  {
      estimate the partition function Y^C from the
      background;
```

compute the scoring terms $e^{-\left(H_{i,j}+H_{i,j'}\right)}$ and derivatives w.r.t. $\omega_{k,b}^{\alpha}$;

sample random sites over the positive set (Gibbs sampling);

compute object function U_C and gradients $\frac{\partial U_C}{\partial \omega_{k,b}^{\alpha}}$

do steepest descent through the $\left[\omega_{k,b}^{\alpha}\right]^{(n+1)} = \left[\omega_{k,b}^{\alpha}\right]^{(n)} + \eta\left[\frac{\partial U_C}{\partial \omega_{k,b}^{\alpha}}\right]^{(n)}$

depending on the U_C value, **retain** the best $\omega_{k,b}^{\alpha}$ values

}

}

In realistic biological cases, Co-Bind is able to detect the TF's in around 30% of the cases. This is probably due to the fact that the steepest descent method optimization may get trapped in local minima. Therefore, extensive runs over the initial conditions must be performed in order to increase the search performances. Typically, a few hundreds iterations are required to converge with the steepest descent, whilst a few thousands runs over the initial conditions are a typical figure.

3. Optimization of Co-Bind

Co-Bind is a C++ code mostly written with an object-oriented paradigm. Most of the calculation is made on 64-bit floating point variables. The code does not require a substantial amount of memory and runs well on workstations equipped with standard RAM size.

Given the searching procedure, it is clear that Co-Bind is prone to parallelization via trivial splitting of tasks. In fact, one can distribute across different CPUs the iterations of the external **for each** statement in the pseudo-code previously described. The initial conditions are randomly generated for each processor by using different seeds. We have therefore implemented a parallel version of Co-Bind based on the MPI message passing library. The embarrassing parallel scheme allows to practically achieving the ideal linear speed-up over the conventional single processor calculations. On our system, constituted by 8 dual-node Xeon machines clocked at 3.06 GHz and connected through a Gigabit Ethernet switch, we obtained a speed-up S=15.1.

Next, we have considered the optimization of Co-Bind on the single-processor section of the code. The most significant part of Co-Bind is embodied by the class **perceptron**, which estimates the partition function (denominator of eq. (3)), computes the scoring terms (numerator of eq. (3)) and subsequently applies the steepest descent procedure (eq. (5)).

The estimate of the partition function can be done in two different ways, by enumeration of all possible states or by sampling a fixed but large number N_{sample} of random sites along the background. We have chosen to consider the second route, which is by far the most convenient one, and realized that, by profiling the code performances, this is a CPU-intensive part of the calculation. In a typical test case made of

- 30 sequence sets (10 for the positive and 20 for the negative set),
- the search for a 15-letters long pattern
- $N_{sample}=10^4$,

the estimate of the partition function can require about 40 % of the total CPU time. Furthermore, the computation of the scoring over the positive set requires another 40% of the CPU time. Therefore, we have considered to optimize both these sections.

As previously noticed, the class **perceptron** has a C++ programming style with a deep nesting of functions and methods. This is particularly true for computing the scoring terms over the background (eq. (3)), which turns out to require many clock cycles. Therefore, we have added a new class containing a modified version of the **perceptron** class, renamed **perceptron_inlined**. The idea is to rewrite part of the code by major in-lining of its most-exploited functions and methods. To this aim, we have partially sacrificed protection and hiding of variables, previously accessed mostly via methods, by duplicating part of them in the new class. In this way, it is a straightforward task to optimize the code performances via a procedural programming style. Besides the use of in-lined functions, the procedural style allows to re-shape and ameliorate the access to memory and avoiding cache miss which are the typical bottleneck in Monte Carlo-like calculations. The simple optimization of adopting a procedural programming style for the computation of the scoring and the partition function, while leaving nearly unchanged the size of the executable code, allowed us to reduce the overall computing time – referred to the typical test case reported above – from 16.85 to 11.47 sec/search cycle, corresponding to a speed-up S = 1.47.

Secondly, we have noticed that a 64-bit precision is not, strictly speaking, needed for the problem at hand. Given the non deterministic nature of the calculation, and the fact that the final result, represented by the optimized weight matrices, is required with a reduced number of digits, we have considered to work with 32-bit floating point precision in the architectures which do not support in HW 64-bit computations. It is worth noticing that this

intervention does not allow to compare the results of the modified code with the unmodified one in a strict sense, i.e. by comparing numbers up to the machine round-off precision. In fact, the optimization procedure tends to diverge, as the iteration proceeds, due to small perturbations in the weight matrices. However, we have always checked the correctness of results in a statistical way, by comparing the patterns extracted by the code, and the relative consensus ranking, for a large number of runs over different initial conditions. In fact, the number of steepest descent searches should be large enough to robustly sample the maxima of the object function. In our typical test case 200 iterations appear to be sufficient. By fixing this number of searches, we have verified that Co-Bind always provided satisfactory results in its various versions where we obtained the same 3 top scoring patterns. The usage of 32 bit precision on machines with 32 bit architectures, given for granted the previous optimizations, allowed us to reduce the overall computing time from 11.47 to 6.52 sec/search cycle, corresponding to a speed-up S = 1.76.

Thirdly, we intervened at the algorithmic level by modifying the way the calculation of the scoring term $e^{-H_{i,j}}$ is made. In fact, it is clear from the previous section that a large number of CPU cycles is required by the recurrent calculation of exponentials. We start noticing that

$$e^{-H_{i,j}} = e^{-\sum_{k=0}^{l-1}\sum_{b=0}^{3}\omega_{k,b}\cdot x_{k,b}^{j}} = \prod_{k=0}^{l-1}\prod_{b=0}^{3} e^{-\omega_{k,b}\Delta_{k,b}^{j}} \quad (6)$$

where $\Delta_{k,b}^{j}$ is defined as

$$\Delta_{k,b}^{j} = \begin{cases} 1 & if\ x_{k,b}^{j} = 0 \\ e^{\omega_{k,b}} & if\ x_{k,b}^{j} = 1 \end{cases}$$

From (6) it is clear that it is sufficient to store the values $e^{-\omega_{k,b}}$ (computed once) to avoid multiple evaluations of the exponential, both in the estimate of the partition function and in the scoring of the positive set. In the former case such an evaluation is repeated $N_{sample} \times l$ times, whilst the latter case requires a substantial number of evaluations, depending on the convergence of the Gibbs sampling procedure. The described method allows us to substitute the expensive computation of exponential functions with an access to a small matrix ($l \times 4$ float) held in the L1 cache.

A further advantage of the proposed computation scheme concerns the numerical error propagation, due to the finite arithmetic precision. Let us analyze the error propagation for the computation of $e^{-H_{i,j}}$ through the leftmost and rightmost parts of expression (6). This is best seen in the following. Let us define $f_j = e^{-\omega_j}$. The former computation is $x(\omega) = e^{-\sum_{j=0}^{l-1}\omega_j}$, the second is $y(f) = \prod_{j=0}^{l-1} f_j$. Let us indicate with $\delta\alpha$ the error in the numerical representation of α. The propagation of statistically independent errors on x and y is given by

$$\begin{aligned} \delta x &= \sum_{i=0}^{l-1}\frac{\partial x(\omega)}{\partial \omega_i}\delta\omega_i = \\ &= \sum_{i=0}^{l-1}\frac{\partial\, e^{-\sum_{j=0}^{l-1}\omega_j}}{\partial \omega_i}\delta\omega_i = \\ &= -x(\omega)\sum_{i=0}^{l-1}\delta\omega_i \end{aligned} \quad (7)$$

$$\begin{aligned} \delta y &= \sum_{i=0}^{l-1}\frac{\partial y(f)}{\partial f_i}\delta f_i = \\ &= \sum_{i=0}^{l-1}\frac{\partial\prod_{j=0}^{l-1} f_j}{\partial f_i}\delta f_i = \\ &= y(f)\sum_{i=0}^{l-1}\frac{\delta f_i}{f_i} \end{aligned} \quad (8)$$

As $x(\omega) = y(f)$, they represent the same quantity $e^{-H_{i,j}}$, $\delta y < \delta x$ since $\frac{\delta f_i}{f_i} < \delta\omega_i$.

The adoption of a computation scheme based on the rightmost part of expression (6), while maintaining all previous optimizations, allowed us to reduce the overall computing time from 6.52 to 2.15 sec/search cycle, corresponding to a speed-up S = 3.03.

The performances of the presented optimizations on the serial code, for the test case previously described, are summarized in Table 1. Computing time refers to the addition of new optimizations, starting from top (no optimization present) to bottom (all optimizations included). Putting together parallelization and the described optimizations, we achieved a speed-up, on a 8

dual-node Xeon system, with respect to the original serial code as downloaded from the web site [9], S = 118.

Table.1: performances of Co-Bind in its various versions

	Computing Time [sec/search cycle]	Cumulative Speed-up
original code	16.85	1.00
+ procedural version	11.47	1.47
+ 32 bit	6.52	2.59
+ exp tabled	2.15	7.85

4. Implementation of Co-Bind on a special hardware platform

In order to investigating the suitability of DSP devices to sustain this type of computation, we have implemented the Co-Bind code, described in the previous section, on a platform constituted by a 4 DSPs co-processing board attached – through the PCI bus – to a bi-processor (Xeon) computing system..

The co-processing board is the BittWare Tiger-PCI [10], equipped with four Analog Devices TigerSHARC ADSP-TS101TS digital signal processors [11]. The architecture of the board is shown in Figure 1.

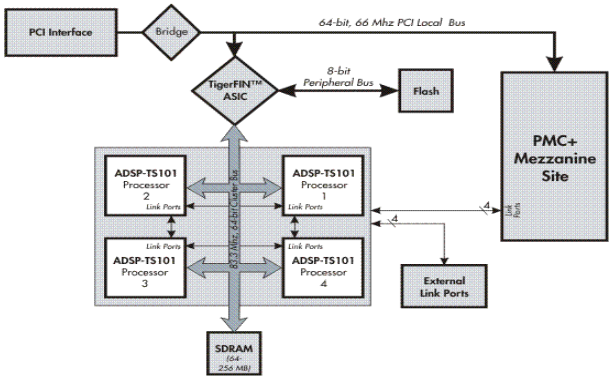

Figure 1. Architecture of the BittWare Tiger-PCI board, picture taken on the web site www.bittware.com.

Each TigerSHARC ADSP-TS101S processor has 6 Mbit of SRAM memory, organized in one 2 Mbit bank of program memory and two 2 Mbit banks of data memory. ADSP-TS101S has a Static Superscalar architecture [11]: in fact, the processor's core of each ADSP-TS101S can execute simultaneously from one to four 32-bit instructions encoded in a Very Long Instruction Word (VLIW) instruction line controlling the two DSP compute blocks (Figure 2). Each compute block contains one ALU, one multiplier, one 64-bit shifter and a 32-word register file. Three independent 128-bit wide internal data busses, each connected to one of three 2M bit memory banks, enable quad word data, instruction and I/O accesses. A functional block diagram of the TigerSHARC ADSP-TS101S DSP processor is shown in Figure 2.

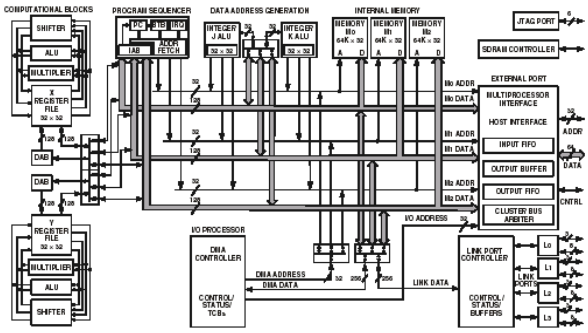

Figure 2. Functional block diagram of the TigerSHARC ADSP-TS101S DSP processor.

We decided to execute on the DSPs the most demanding section of the code (the estimate of the partition function). The remaining code runs on the dual-processor Xeon host machine. Data are transferred from host to DSP and vice versa through a DMA channel.

As previously described, the code of Co-Bind is composed by two nested cycles. Each iteration of the inner loop of Co-Bind, using a single Xeon processor and a single DSP, can be split in three successive parts: *host1*, *dsp* and *host2* respectively (Figure 3). In the *host1* part the host machine reads, from the host memory, the data to be transferred to the DSP. In the *dsp* part the DSP enables the DMA Controller to execute the DMA transfer from the host to its internal data memory, computes the partition function and activates the DMA Controller to execute the DMA transfer from its internal data memory to the host machine. In the *host2* part the host performs the remaining operations, using the partition function value computed by the DSP, and updates the results in memory (they will be used in the next iteration).

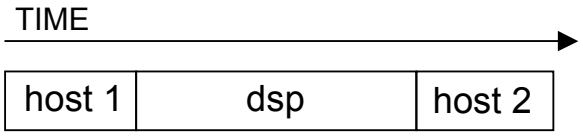

Figure 3. Block diagram of a single iteration of the inner loop of Co-Bind, running on the host-DSP platform.

In previous example we used only two resources, one Xeon processor and one DSP, while 2 Xeon and 4 DSPs are available in the considered platform.

In order to use all the available computational resources, we implemented a multi-DSP version of the Co-Bind code. We divided the entire workload of a single

node in more concurrent processes, for instance 10. In this case, two processes will work exclusively on the Xeon processors, while the remaining eight processes will execute their workload on a single processor of the host machine and on the available DSPs of the board. A typical Gantt chart representing such a parallel execution is shown in Fig. 4.

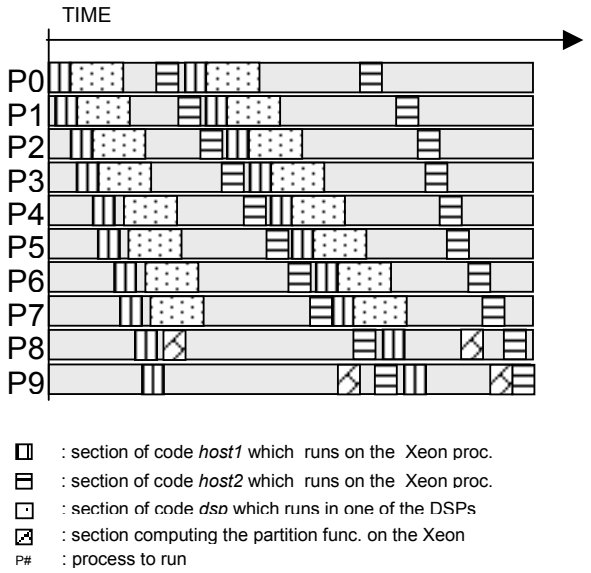

Figure 4. Gantt diagram of the execution of 20 iterations of the inner loop of Co-Bind in ten concurrent processes

The performances of the presented implementation of the code on hybrid Xeon-DSP platform, for the test case previously described, are summarized in Table 2. Each row of the table corresponds to a different number of processes. Computing time refers to different numbers of concurrent processes running on the Xeon processors and on the DSPs. In all the tests the workload is the same, being the constant number of Co-Bind inner iterations divided among a different number of processes (ranging such a number from 1, running in the Xeon processor, up to 18, in the case of 10 processes running on the 2 Xeon processors and 8 on the 4 DSPs).

5. Conclusions

We have investigated the possibility of optimising a software which is a popular tool to discover co-operative Transcription Factors in multiple DNA sequences. This type of task is rather common in bioinformatics and, given the extensive CPU-time required for such calculations, the optimisation of a workhorse such as Co-Bind represents a fruitful advance for the interested community.

We have considered three levels of optimisation via: 1) parallelization over the search procedure, 2) rewriting of the most CPU-intensive parts in terms of a procedural scheme, and 3) ameliorating the numerical way to compute some core quantities. The undertaken routes proved very successful and a considerable speed-up of the code, larger than two orders of magnitude, was achieved on a standard 16-node Xeon cluster.

A parallel version of Co-Bind has also been implemented on a hybrid Xeon/DSP parallel system.

Table.2: performances of Co-Bind

Number of processes on the host	Number of processes on the DSP	Computing Time [sec/search cycle]	Cumulative Speed-up
1	0	2.15	1.00
1	1	3.52	0.61
2	0	1.14	1.89
8	8	1.83	1.17
6	4	1.12	1.92
10	8	0.85	2.52

6. Acknowledgments

The described activities have been performed within the project "GeneFun", funded by Italian Ministry of Education, University and Research, under the framework of D.M. 20/10/2000, Progetti Strategici, Legge 449/97.

7. References

[1] Guhathakurta D. and Stormo G.D., "Identifying target sites for cooperatively binding factors", Bioinformatics, vol. 17, n. 7, 608 (2001)

[2] Lawrence CE et Al. "Detecting subtle sequence signals: a Gibbs sampling strategy for multiple alignment", Science, vol. 262, 208. Oct 8, 1993

[3] Liu J.S., "Monte Carlo strategies in scientific computing", Springer Verlag NY, 2001

[4] Pavesi G., Mauri G., Pesole G., "An algorithm for finding signals of unknown length in DNA sequences", Bioinformatics, vol. 17, suppl. 1, S207, 2001

[5] Stormo G.D., "DNA binding sites: representation and discovery", Bioinformatics, vol. 16, 16, 2000

[6] Tompa M., "An exact method for finding short motifs in sequences, with applications to the ribosome binding site problem", Proc. 7th Intl. Conf. on Intelligent Systems for Molecular Biology, 1999

[7] Wolferstetter F.et Al. "Identification of functional elements in unaligned nucleic acids sequences by a novel tuple search algorithm", Computer applications in biosciences, vol. 12, 71, 1996

[8] Workman C.T., Stormo G.D., "ANN-Spec: a method for discovering transcription factor binding sites with improved specificity", Pac. Symp. Biocomput. vol. 5, 464, 2000

[9] http://ural.wustl.edu/~dg/co-bind.html

[10] http://www.bittware.com

[11] Data Sheets of Analog Devices TigerSHARC ADSP-TS101S