Enhancing Parallel Recursive Brute Force Algorithm for Motif Finding

Marwa A. Radad
Faculty of Electronic Engineering Menoufia University
El-Gaish st.,Menouf, Menoufia, Egypt, Post code: 32952

Nawal A. El-fishawy
Faculty of Electronic Engineering
Menoufia University
El-Gaish st. , Menouf, Menoufia, Egypt, Post code: 32952

Hossam M. Faheem
Faculty of Computer and Information Sciences
Ain Shams University
El-Abbasiah, Cairo, Egypt

ABSTRACT
Motif search is a fundamental problem in bioinformatics. Its main application is locating transcription factor binding sites (TFBSs) in DNA sequences. Numerous algorithms have been proposed in the literature to solve this problem. The exact algorithms solve M(l,d) problem by reporting all l-length motifs M with at most d mutations. Recursive Brute Force (R-BF) algorithm is an exact algorithm that has solved M(l,d) problem in exponential time with d. Multicore implementations of R-BF have efficiently utilized computation resources of modern multicore architectures to achieve more advantageous operation than sequential one. In this paper, we explore an enhanced version of R-BF algorithm. The new algorithm is called R-BF2. R-BF2 enhances the running time of R-BF by memorizing more information about each node in search space. R-BF2 pays more than 40% memory space to achieve a speedup factor of 3. However, parallel implementations of R-BF2 keep the same scalability just like R-BF on multicore systems.

General Terms
Computational Biology, Pattern Recognition, Parallel Computing.

Keywords
Bioinformatics; Motif finding; Branch & Bound search; Parallel programming; OpenMP; MPI

1. INTRODUCTION
Activating a gene is mainly controlled by special proteins known as Transcription Factors(TFs). TFs bind to specific DNA patterns in a promoter region of a gene. TFs can switch a gene on or off with some existing cofactors. If we can identify the sites where transcription factors bind, we gain some insight into the regulation of genes. As a result, identifying transcription factor binding sites (TFBSs) is a very important task for decoding a genome. TFBSs are called motifs and finding these motifs computationally is an area of active research[1].

The motif search is an approximate pattern search problem in computational biology where a common pattern, albeit with a few mismatches, needs to be found from a set of DNA sequences. A precise definition of motif finding problem is given by [2] as follows:

Planted (l,d)-Motif Problem: Suppose there is a fixed but unknown nucleotide sequence M(the motif) of length l. Given t length-n sequences, each of which contains exactly one planted variant (binding site) of M, we want to determine M without knowing the positions of the planted variants. A variant is a length-l string derivable from M with maximum d point substitutions.

This problem becomes increasingly difficult as the number of allowed mutations grows relative to the length of the motif. Pevzner and Sze[2] presented the challenge problem (15,4), which determines particular values for the planted motif problem. The motif is of length-l=15, with allowed mutations d=4 and the number of sequences is t=20, each of size n=600. This problem is hard since the signal is too weak for applying probabilistic methods while exhaustive search is impractical since the motif is too long.

Two kinds of algorithms have been proposed in the literature for PMS: exact and approximate. Exact algorithms can guarantee finding the optimal solution. On the other hand, approximate algorithms employ local search techniques such as Expectation Maximization EM and Gibbs sampling. MEME[3] is the most popular implementation of EM method. Gibbs sampling technique is implemented firstly by Gibbs Sampler[4]. AlignACE[5], BioProspector[6] and GibbsDST[7] adopt Gibbs sampling approach in different ways.

While exact algorithms for the motif problem take longer to complete than approximate algorithms, they are preferable since they are guaranteed to report all the (l, d)-motifs. Some of these algorithms are based on enumeration methods such as Brute Force algorithm[8] which exhaustively search for all 4^l *l-mers* to find the motifs. While PMS series algorithms[9-14] extract *l-mers* from input sequences. Other algorithms use the suffix tree data structure like Weeder[15] and FLAME[16]. Mismatch tree is a novel data structure that was proposed by MITRA[17].

In this paper, we interest in an efficient exact algorithm that called Recursive Brute Force R-BF[18]. R-BF solves M(l,d) problem by examining only 4^{d+1} prefix patterns. R-BF extends recursively good prefixes and prunes others. Its main idea is keeping an occurrence list for each good prefix. Two parallel implementations of R-BF were proposed[18]. OMP-RBF based on OpenMP[19]. It lacked the scalability as a result of heap contention problem. Conversely, MPI-RBF that based on MPI[20] obtained high scalability.

We introduce in this paper R-BF2 algorithm that enhance the running time of R-BF. The big thought behind R-BF2 is: "keeping more information about a parent node to decrease its children processing time". To keep more information about a node, we must pay more memory price. We study this time-memory tradeoff on sequential and shared memory parallel implementations.

The rest of this paper is organized as follows: In Section 2 we describe the Recursive-Brute Force algorithm. Our advanced algorithm R-BF2 and its parallel implementations are explained in section 3. The performance of R-BF2 and R-BF

is compared experimentally in Section 4. Finally, Section 5 concludes our work and presents future work.

2. RECURSIVE BRUTE FORCE ALGORITHM R-BF

R-BF is a word-based enumeration method. It is an advanced version of the will known Brute Force algorithm. It traverses the search space in depth first order and uses branch and bound technique to prune the search space. In this section, we explain R-BF algorithm steps in details.

R-BF algorithm takes a set of ***t*** DNA sequences. The length of each sequence is n. In addition, it takes three input parameters; ***l***: the motif length, ***d***: the allowed mutations, ***q***: the Quorum that is the minimum number of input sequences where the motif must appear. The algorithm creates a prefix array of size (***d+1***). For each prefix it examines all the input sequences. If the prefix occurs in at least ***q*** sequences with hamming distance less than or equal to ***d***, it considers a good prefix. On the other side, bad prefixes are discarded. Discarding one prefix will save $4^{(l-d+1)}$ match operations. A good prefix is extended with the four basis (A,C,G,T), then the algorithm examines each child. This process is repeated until finding all existing motifs. Skipping bad prefixes also is allowed at any level of the search tree and it would save $4^{(l-d+i)}$ match operations, where ***i*** is the current level of the search tree. The key idea of the algorithm is to keep in mind the locations of good prefixes, so when go more deep it examines the parent's archive only, instead of examining all sequences each time. It creates an occurrence list for a good prefix that contains all positions of input sequences where the corresponding pattern matches it. The occurrence list will be a reference for the children of this prefix. Also, the occurrence list of a child will be a subset of it's parent's. R-BF recursively allocates occurrence list of each node, then frees this allocation when it return back. After catching a motif, the algorithm continues to search for other motifs in depth-first order.

For example: given ***t*** input **DNA** sequences, each of size ***n***. Find the motif of length ***l***=8, with allowed mutations ***d***=2. The motif should occur in at least ***q*** input sequences.(The different values of t, n and q do not affect the algorithm steps)

Firstly R-BF generates an array of ***l-mer***s prefixes. As illustrated in Figure 1, the array width= (d+1) = 2+1 = 3. Then, it performs the following steps:

1. For each prefix, it searches all sequences. The good prefix is the one that occurs in at least q sequences with at most 2 mutations. Skipping one prefix will save $4^{(8-2+1)}$=1024 match operations.
2. For good prefix, the algorithm saves an occurrence list that gives all the positions of input sequences where the corresponding pattern matches it.
3. R-BF goes more deep by extending the good prefix with the four basis (A,C,G,T).
4. R-BF examines the new children by searching the parent's archive only. The occurrence list of a good child will be a subset of its parent's.
5. Repeat these operations.

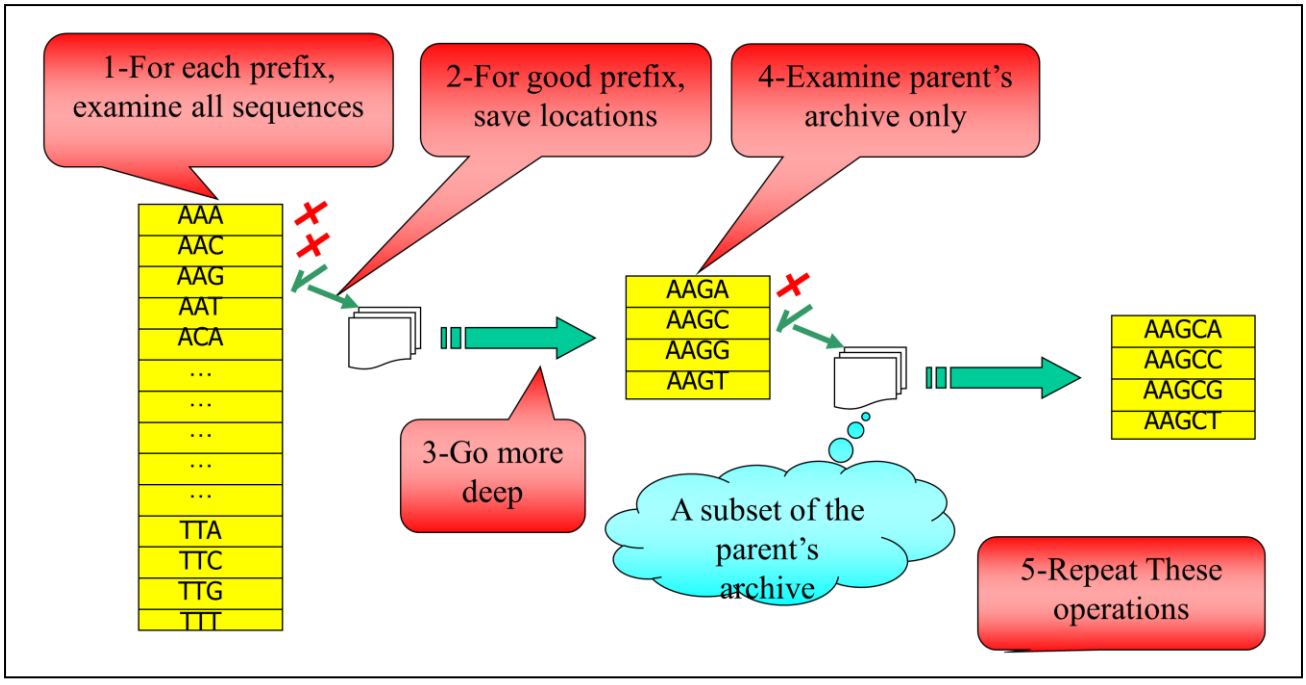

Fig 1: Recursive Brute Force steps

The complexity analysis of R-BF indicates that it does not enhance the worst case of the original Brute Force algorithm. However, the average case complexity reflexes the performance enhancement that achieved by the R-BF. Table 1 summarize the time/space complexity of R-BF as proved by [18].

Table 1. R-BF Complexity Analysis

	Time Complexity	Space Complexity
Best case	$O(cnt)$	$O(cnt)$
Average case	$nt4^{(d+1)} + 4nt * \sum_{j=1}^{l-(d+1)} E(d+j, d)\, P(d+j, d)$	$nt * \sum_{j=1}^{l-(d+1)} P(d+j, d) + nt*E(l,d)P(l,d)$
Worst case	$O(nt4^l)$	$O(nt4^l)$

The best case situation occurs when $d=0$, that is when R-BF searches for exact motif matching. Time/space complexity in this case is $O(cnt)$, where c is a constant. The average time complexity of the R-BF is exponential with the allowed mutations (d). Where $E(l, d)$ is the expected number of length-l strings with at least one d-variant in each sequence in t ,and $P(l,d)$ is the probability that a fixed l-*mer* occurs with up to d mutations at a given position of a random sequence. On the other hand, the worst case occurs when ($l=d+1$), that is incredible case because almost all the l-length patterns will be motifs. It is exactly the complexity of the original Brute Force $O(nt4^l)$.

2.1 Parallel R-BF

R-BF was parallelized according to SPMD (Single Program Multiple Data)[18]. The array of prefixes acts as a queue of tasks. In a parallel environment, each thread or process gets a prefix and all of them run simultaneously. When one worker finishes its task (returns back with or without a motif), it catches the next task from the queue (the next prefix). Two versions of R-BF were implemented on multicore shared-memory architecture. First, OMP-RBF is based on OpenMP[19]. In this version, OpenMP directives use compiler to automatically thread the loop. The details are hidden from the programmer. When a thread in OMP-RBF finishes its current job, it gets a new task (new prefix) from the prefixes array in despite of its id. Some threads may process more prefixes than others. There are no idle threads until finishing all the work. OMP-RBF suffers from a serious performance degradation due to the heap contention problem. The extensive (allocate/free) operations of different size memory blocks is the main cause of this problem. Different Solutions have been investigated to solve the heap contention problem[18].

Second, MPI-RBF is based on Message Passing Interface MPI[20]. MPI-RBF algorithm uses the process id to distribute the jobs among the running processes. Each process gets approximately the same number of prefixes. When a process finishes its quota early, it waits idle until the other processes finish their work. MPI-RBF outperforms OMP-RBF on multicore system. The high scalability of MPI-RBF is a result of its efficient handling of the data locality.

3. ENHANCED RECURSIVE BRUTE FORCE ALGORITHM R-BF2

R-BF is a simple algorithm that depends on the basic search algorithm Brute Force. As we shown before, it collects several ideas to enhance the running time of the search process. It is still easy to understand. In addition, it uses only simple data structures with clear computing operations. This makes R-BF a flexible algorithm. It is easy to add further improvement ideas.

In this section, we show an enhanced version of R-BF. We call this version of the algorithm : R-BF2. We illustrate that time-memory trade off is the master rule that control the algorithm enhancement. In other words, enhancing running time means paying more memory cost and vice versa.

The main idea behind R-BF2 is memorizing more information about a prefix node. The original R-BF creates an occurrence list for a good prefix that contains all positions of input sequences where the corresponding pattern matches it. The child of a good prefix will go to each position in the list and begin comparison operations character by character to decide if saving this location in its own occurrence list or discarding it. On the other hand, R-BF2 saves the matching positions plus the corresponding hamming distances in the occurrence list of a good prefix. The occurrence list is constructed as a two dimensional array. As we know, the child of a good prefix is equal to its parent plus a single character at the end of the string. In this case, The child of a good prefix will go to each position in the list and compare only its last character with its correspondence in the input sequences. If matched, the child prefix will save this location with the same hamming distance of the parent. On the other case, the child prefix adds one to the parent hamming distance. If the hamming distance is less than or equal to the allowed mutation, the child prefix will save this location with the new hamming distance. Else, it discards this location. We illustrate this idea in figure 2.

We illustrate the steps of R-BF2 based on the above example in figure 3. It is similar to R-BF at figure 1 except a simple change. At level1, R-BF2 calls match function for each prefix with all prefix-length substrings in the input sequences. Each match function does prefix-length comparison operations just like R-BF. Matching position means that the hamming distance between the occurrence and the motif is less than or equal to d. R-BF2 saves the matching positions and their hamming distances. In the extension levels, the match function does a single comparison operation in despite of the prefix length to decide the matching positions. This will save *[(prefix-length – 1) * parent_occurence _list _size]* comparison operations at each node.

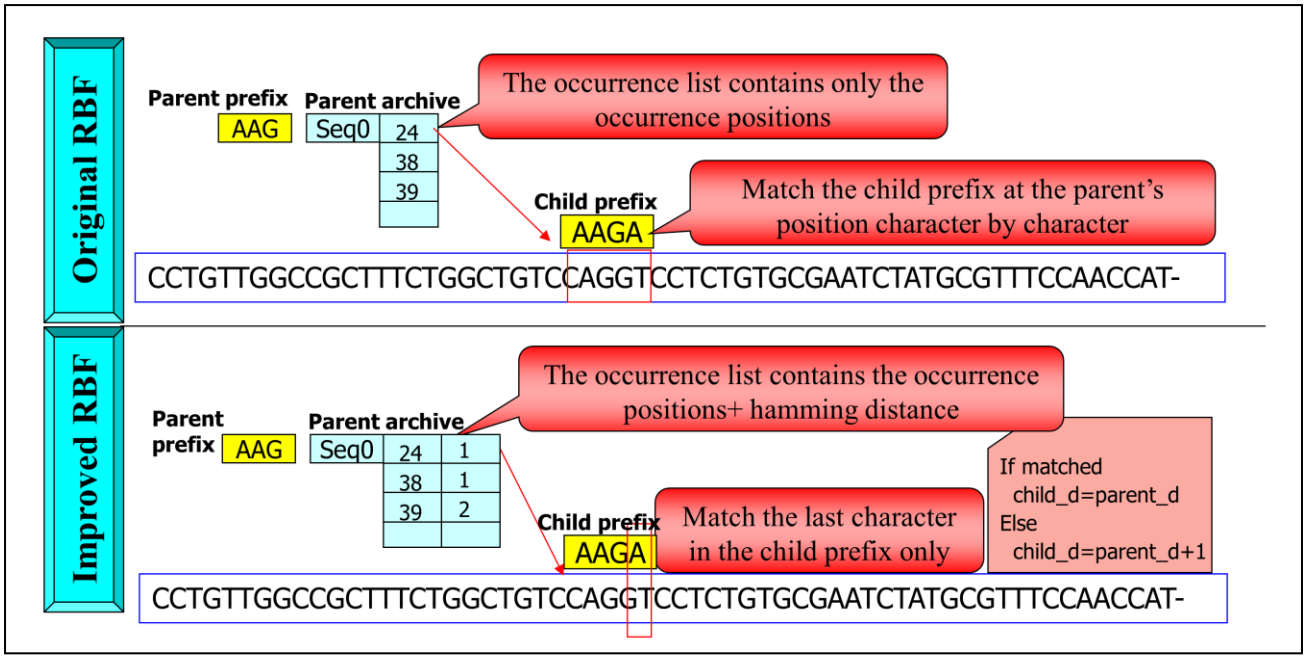

Fig 2: R-BF2 main idea

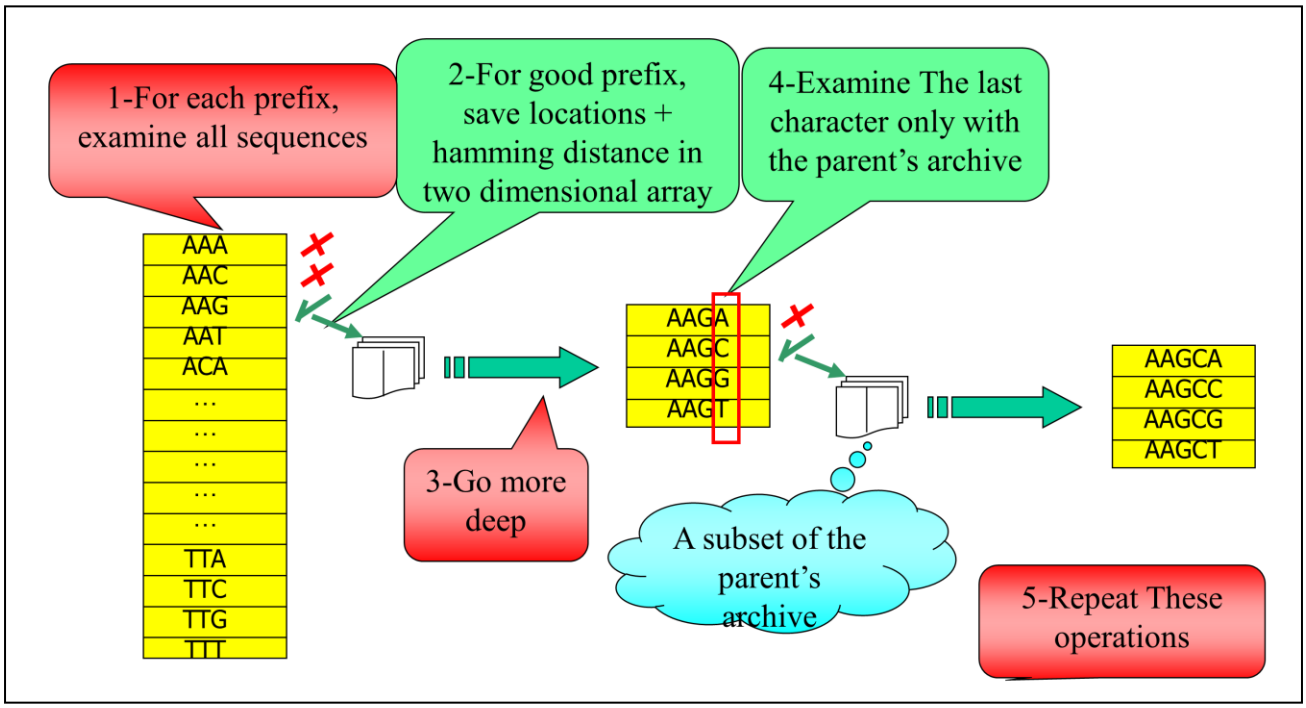

Fig 3: Enhanced Recursive Brute Force (R-BF2) steps

Memorizing a hamming distance archive for each prefix node will speed up the algorithm in practical experiments. However, R-BF time complexity is the same as that of R-BF. On the other hand, it doubles the required memory space. Because each prefix occurrence list array will be replaced by a two dimensional array with the same length. We study this time-memory tradeoff practically in section 4.

3.1 Parallel R-BF2

We parallelize R-BF2 such as parallel R-BF. Prefix array elements are distributed on the worker threads or processes. We implement two versions of parallel R-BF2 to compare them with their R-BF correspondence. First, we use OpenMP to implement OMP-RBF2. We expect that the heap contention problem will be worsen in OMP-RBF2 because OMP-RBF2 allocates and frees larger blocks of memory than OMP-RBF. In fact, heap contention problem is not affected by the size of allocated memory blocks. In stead, it is excited by the number of calling (allocate/free) functions and hence the number of allocated/freed objects. Fortunately, OMP-RBF2 allocates and frees the same number of occurrence lists with the same number of calling (allocate/free) functions such

as OMP-RBF. We conclude that OMP-RBF2 will outperform OMP-RBF in practical experiments.

Second, we implement MPI-RBF2 using MPICH2[21] library. While MPI paradigm is a distributed memory programming model, researchers tend to compare between MPI and OpenMP on a shared memory architecture[22,23]. We expect that memory-intensive characteristic of MPI-RBF2 will not affect its scalability on multicore systems. This because MPI-RBF2 has no communication among its processes.

4. EXPERIMENTAL EVALUATION

In this section, we evaluate our advanced algorithm R-BF2 for motif finding problem and compare it with its base algorithm R-BF. We indicate the time-memory tradeoff practically. We implement and compile both algorithms using MS VC++2008. Our evaluation experiments are done on synthetic data. Problem instances are generated according to the planted (l, d)-motif model. Problem instances are generated according to the planted ***(l, d)***-motif model. We followed up the (Fixed number of Mutations or FM) model described by [2]. In this approach, a randomly mutated pattern with exactly *d* substitutions is implanted in each sequence. Datasets are produced using rMotifGen tool [24] which provides an efficient and convenient method for creating random DNA or amino acid sequences with a variable number of motifs. For each set of parameters ***l*** and ***d***, we generate 10 test cases to get the average of the results.

4.1 R-BF2 Versus R-BF

These experiments are done on Intel Core2Duo, 2GHz, 2GB RAM machine. Table 2 shows the running time of the two algorithms. R-BF2 achieves 2.6 to 3.1 speedup. The speedup increases with the difficulty of the problem, because R-BF2 saves more comparison operations such as seen in table 3. R-BF2 performs only 0.125 comparison operations for small motif length with mutations d=3 and less than 0.1 comparison operations for long motifs with d=4.

Table 2. Comparison between R-BF and R-BF2 in terms of time needed to perform comparison operations in CPUsec

M(*l*,*d*)		R-BF	R-BF2	Speedup
l	*d*	T_{comp} *(sec)*	T_{comp} *(sec)*	
11	3	86.6	33.7	2.6
12	3	87.9	33.8	2.6
13	4	1025.3	361.6	2.8
14	4	1085.8	366.1	3
15	4	1136.2	368.9	3.1

Table 3. Comparison between R-BF and R-BF2 in terms of performed comparison operations.

M(*l*,*d*)		R-BF	R-BF2	*Ratio of comp. operations performed by R-BF2 %*
l	*d*	*Comp. Operations*	*Comp. Operations*	
11	3	2.044E+09	2.56E+08	0.125006
12	3	2.050E+09	2.57E+08	0.125223
13	4	2.533E+10	2.54E+09	0.100327
14	4	2.535E+10	2.55E+09	0.100724
15	4	2.537E+10	2.53E+09	0.099822

The memory space required by the two algorithms is reported in table 4. RB-F2 requires 32.5% more memory to solve (11,3). The ratio is increased with the difficulty of the problem to reach 43.7% to solve (15,4) problem. However, R-BF2 needs only 3.5MB to solve (15,4) problem in 6.1min instead of 19min by R-BF. In fact, R-BF2 behaves well and it is considered a step forward to enhance motif finding problem solutions.

Table 4. Comparison between R-BF and R-BF2 in terms of required memory space

M(*l*,*d*)		R-BF	R-BF2	*Ratio of Memory Required by R-BF2 %*
l	*d*	*Memory KB*	*Memory KB*	
11	3	2004	2656	1.325349
12	3	2272	3075	1.353433
13	4	2480	3400	1.370968
14	4	2388	3320	1.390285
15	4	2452	3524	1.437194

4.2 Parallel R-BF2

We implement two parallel versions of R-BF2 that called OMP-RBF2 and MPI-RBF2.To evaluate our parallel algorithms, we use a Dual, Quad -Core Intel(R) Xeon(R) CPU E5520 @ 2.27GHz 2.26 GHz (8-cores) machine, with hyperthreading enabled (enabling each core to run up to 2 threads for a total of 16threads). The system has 24GB of main memory and runs 64-bit Windows Server 2008 operating system.

4.2.1 OMP-RBF2 versus OMP-RBF2

In this experiment, we implement OMP-RBF2 using the standard malloc() function. We use the data set of (15,4) problem. Figure 4 shows a comparison between OMP-RBF2 and OMP-RBF in terms of execution time. Speedup and efficiency are illustrated in figure 5.

OMP-RBF2 is outstanding in terms of execution time. We can observe that OMP-RBF2 behaves with a similar curves like OMP-RBF in the relative speedup and efficiency, but OMP-RBF2 curves are slightly below OMP-RBF ones. The similarity between curves is a result of the similarity between the two algorithms in the allocate/free functions calling number. The small lake in the OMP-RBF2 curves is due to the large memory space that allocate/free functions deal with.

4.2.2 MPI-RBF2 Versus MPI-RBF2

We implement an MPI version of the enhanced algorithm RBF2. Then, we run MPI-RBF2 on the same machine with the same dataset. We report the execution time of MPI-RBF2 and compare its performance with MPI-RBF in fig. 6 and figure7.

We can see the execution time enhancement achieved by MPI-RBF2. In addition, It competes with MPI-RBF successfully in terms of speedup and efficiency. MPI-RBF2 scales well with the number of processes because each process has its own memory and its own memory manager.

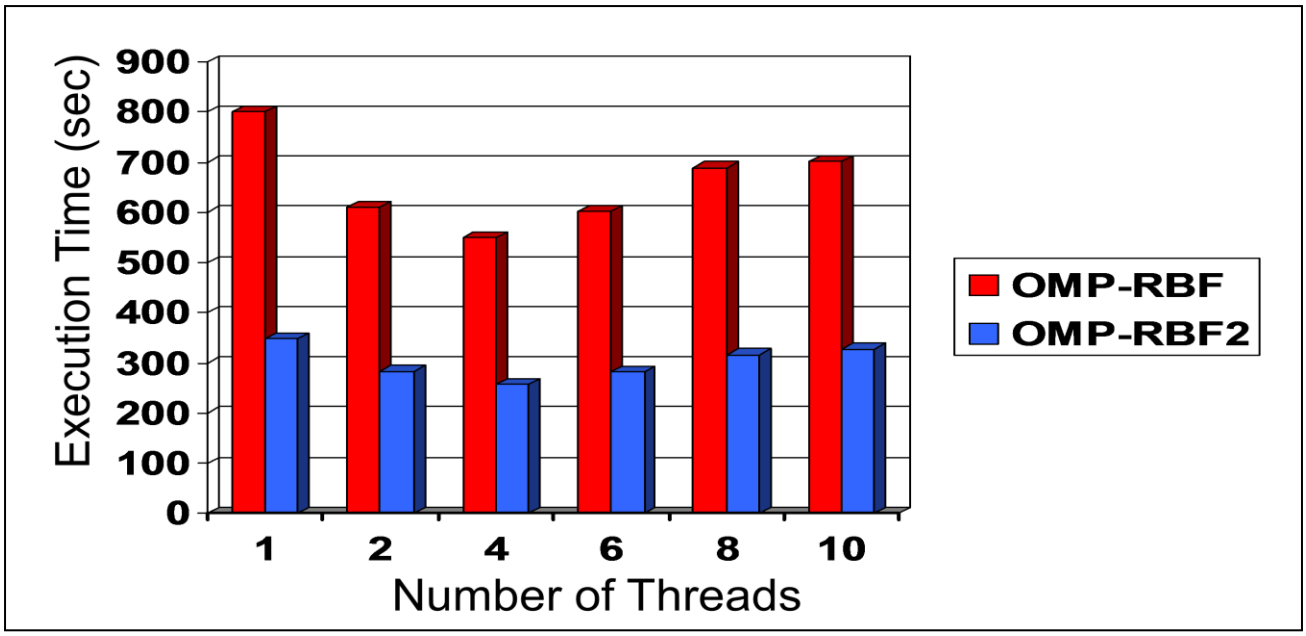

Fig 4: The execution time of OMP-RBF2 versus OMP-RBF

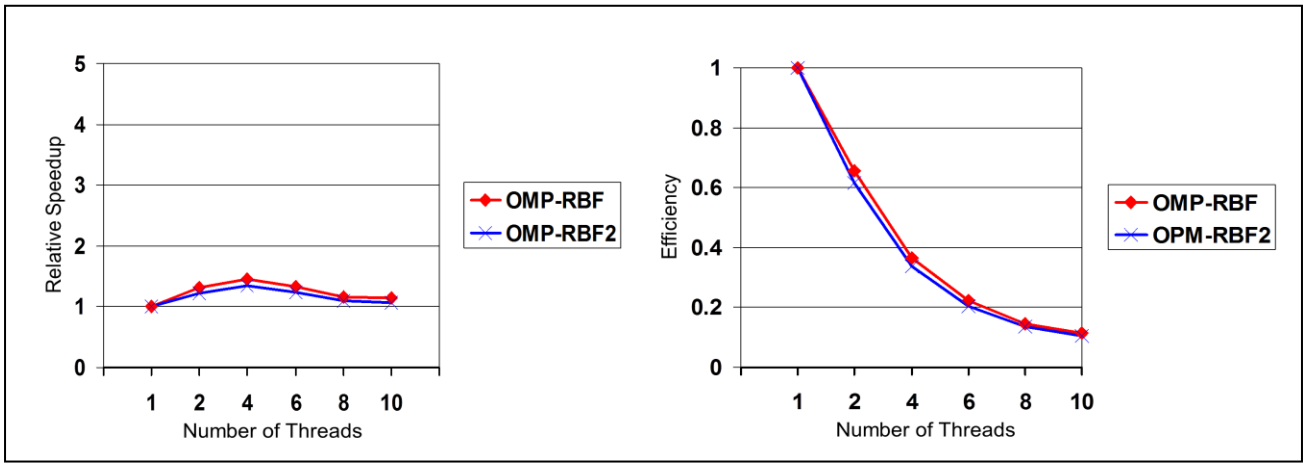

Fig 5: The speedup and efficiency of OMP-RBF2 & OMP-RBF algorithms

4.2.3 MPI Versus OMP

We combine between figure 4 and figure 6 to compare between MPI implementations and OMP ones. Figure 8 shows the success of MPI over OpenMP in both algorithms R-BF and R-BF2. Even the single thread OpenMP timing is slower than the single process MPI timing. This shows the overhead of creating threads in OpenMP. The efficient handling of the data locality boost the scalability of the MPI implementations. It is clear that MPI-RBF2 is the best algorithm.

5. CONCLUSION

Motif finding problem plays an important rule in bioinformatics. Although approximate algorithms are acceptable in some cases in practice, exact algorithms are preferable since they are guaranteed to find optimal solution. We focus in this paper on an efficient exact algorithm that called R-BF2. R-BF2 is an advanced version of R-BF. It enhances running time by memorizing more information about a prefix node. R-BF2 saves the matching positions plus the corresponding hamming distances in the occurrence list of a good prefix. R-BF2 performs less than 0.1 of the operations performed by R-BF, but it needs about 40% more memory. Parallel computing is a promising solution for such computationally intensive problems. Two parallel versions of R-BF2 are implemented. OMP-RBF2 uses OpenMP directives and MPI-RBF2 bases on MPICH2 library. We compare both algorithms with their R-BF correspondences. OMP-RBF2 outperforms OMP-RBF in running time. The speedup and efficiency of OMP-RBF2 are slightly below OMP-RBF curves. However, we emphasize on the fact that the excessive memory required by OMP-RBF2 does not worsen the heap contention negative effects. On the other hand, MPI-RBF behaves just like MPI-RBF in terms of speedup and efficiency. However, it is outstanding in terms of execution time.

We strongly believe that our algorithm R-BF2 is a step forward to enhance exact motif search methods. Parallel implementations are also promised. Modern high performance computing architectures like computer clusters and Grids could be more efficiently utilized to achieve more advantageous operation in this crucial domain.

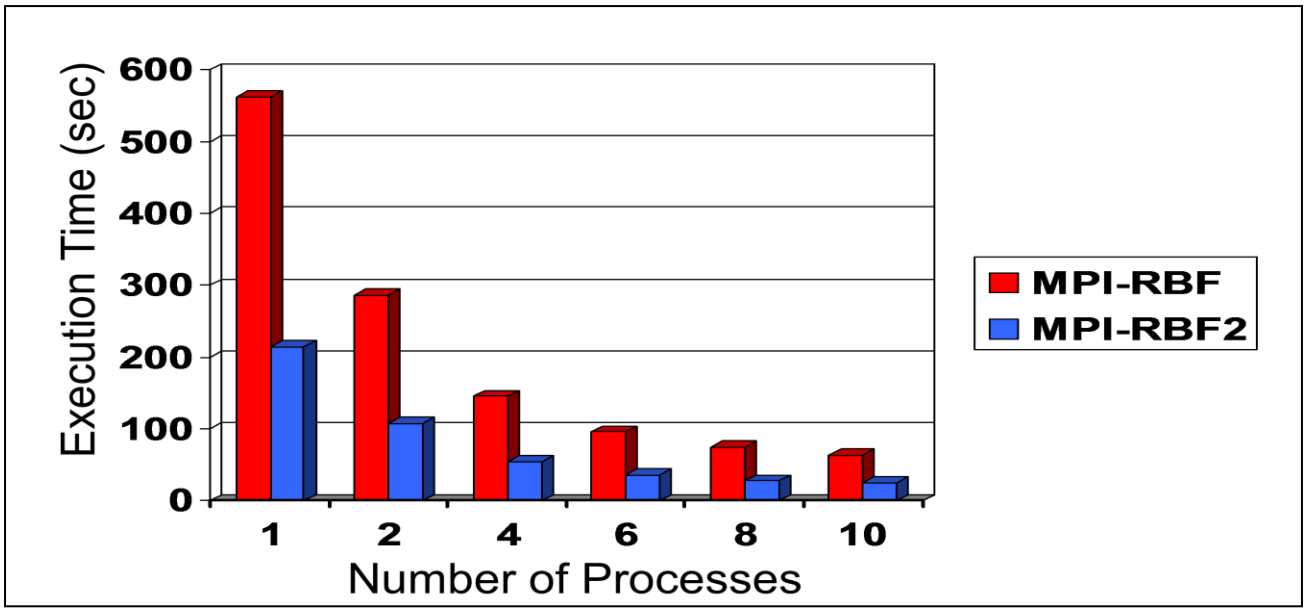

Fig 6: The execution time of MPI-RBF2 versus MPI-RB

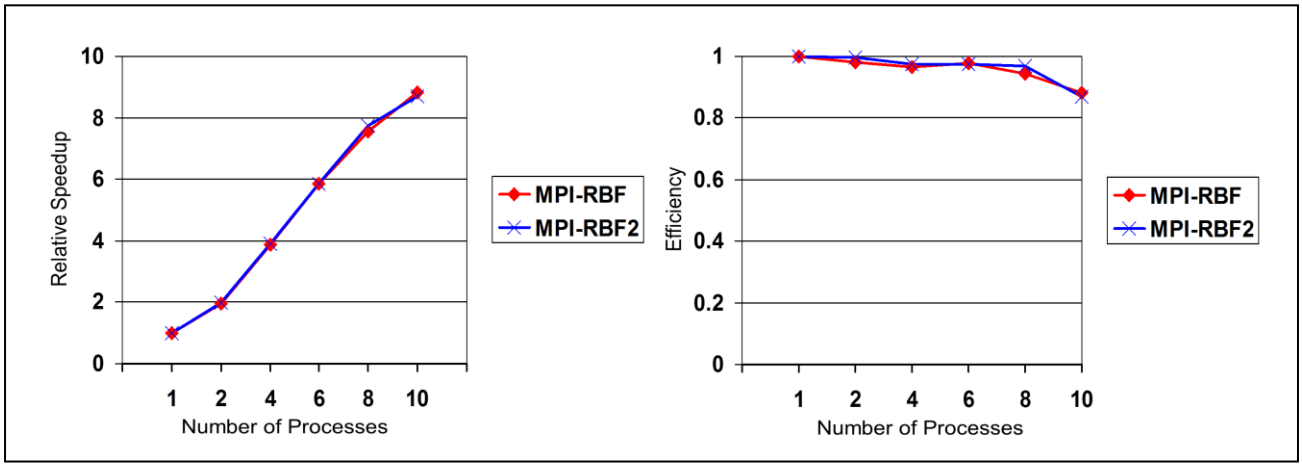

Fig 7: The speedup and efficiency of MPI-RBF2 & MPI-RBF algorithms

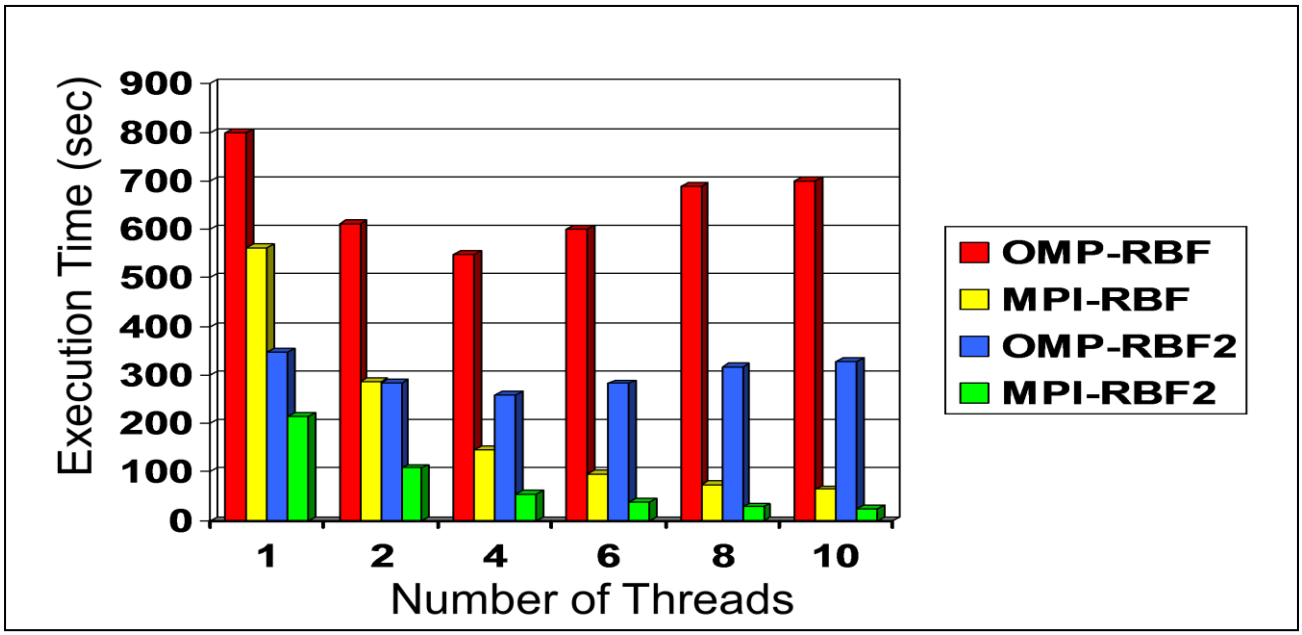

Fig 8: Compare MPI and OpenMP implementations of both algorithms R-BF and R-BF2

6. REFRENCES

[1] Gopal, S., Haake, A., Jones, R.P. and Tymann, P. 2009. Bioinformatics, A Computing Perspective, McGrawHill, International Edition.

[2] Pvzner, P. and Sze, S. H. 2000. Combinatorial approaches to finding subtle signals in DNA sequences. In Proceedings of the 8th International Conference on Intelligent Systems for Molecular Biology, 269–278.

[3] Bailey, T. L., Williams, N., Misleh, C., and Li, W. W. 2006. "MEME: discovering and analyzing DNA and protein motifs", Nucleic Acid Research, Vol. 34, 369–373.

[4] Lawrence, C. E., Altschul, S. F., Boguski, M. S., Liu, J. S., Neuwald, A. F., and Wootton, J. C. 1993. "Detecting subtle sequence signals: a Gibbs sampling strategy for multiple alignment", Science, Vol. 262, 208–214.

[5] Roth, F., Hughes, J., Estep, P., and Church, G. 1998. "Finding DNA regulatory motifs within unaligned noncoding sequences clustered by whole genome mRNA quantitation", Nature Biotechnology, Vol. 16, 939–945.

[6] Liu, X., Brutlag, D.L. and Liu, J.S. 2001. BioProspector: discovering conserved DNA motifs in upstream regulatory regions of co-expressed genes. In Proceedings of Pacific Symposium on Biocomputing, 127-138.

[7] Shida, K. 2006. Hybrid Gibbs-Sampling algorithm for challenging motif discovery: GibbsDST. In Proceedings of the 17th International Conference on Genome Informatics, 3-13.

[8] Jones, N. C. and Pevzner, P. A. 2004. An Introduction to Bioinformatics Algorithms. The MIT Press.

[9] Rajasekaran, S., Balla, S. and Huang, C-H. 2005. "Exact algorithms for planted motif challenge problems", APBC, 249-259.

[10] Rajasekaran, S., Balla, S., Huang, C-H, Thapar, V., Gryk, M., Maciejewski, M. and Schiller, M. 2005. "High-performance exact algorithms for motif search", Journal of Clinical Monitoring and Computing, Vol 19. 319–328.

[11] Rajasekaran, S. and Dinh, H. 2011. A speedup technique for (l, d) motif finding algorithms, BMC Research Notes, Vol. 4, No. 54, 1–7.

[12] Dinh, H., Rajasekaran, S., and Kundeti, V. 2011. "Pms5: an efficient exact algorithm for the (l,d) motif finding problem", BMC Bioinformatics, Vol. 12, No. 410.

[13] Bandyopadhyay, S., Sahni, S., and Rajasekaran, S. 2012. Pms6: A fast algorithm for motif discovery. In Second IEEE International Conference on Computational Advances in Bio and Medical Sciences (ICCABS).

[14] Dinh, H., Rajasekaran, S., Davila, J. J. 2012. "qPMS7: A Fast Algorithm for Finding (l, d)-Motifs in DNA and Protein Sequences", *PLoS ONE*, Vol. 7, No. 7, e41425.

[15] Pavesi, G., Mauri, G. and Pesole, G. 2001. "An algorithm for finding signals of unknown length in DNA sequences", ISMB (Supplement of Bioinformatics),Vol 17, No.1, 207-214.

[16] Floratou, A., Tata, S. and Patel, J. M. 2010. Efficient and accurate discovery of patterns in sequence datasets, In Proceedings of the IEEE 26th International Conference on Data Engineering – ICDE, 461-472.

[17] Eskin, E. and Pevzner, P.A. 2002. "Finding composite regulatory patterns in DNA sequences", Bioinformatics, Vol 18, No.1, 354-363.

[18] Radad M. A., El-Fishawy, N. A., Faheem, H. M. 2013. "Implementation of Recursive Brute Force for Solving Motif Finding Problem on Multi-core", International Journal of Systems Biology and Biomedical Technology (IJSBBT), 2013, in press.

[19] OpenMP: http://www.openmp.org (Last visited 14-12-2013).

[20] MPI Forum: http://www.mpi-forum.org (Last visited 14-12-2013)

[21] MPICH: http://www-unix.mcs.anl.gov/mpi/ mpich2 (Last visited 14-12-2013)

[22] Eadline, D. 2007. MPI on Multicore, an OpenMP Alternative, *Linux Magazine*, http://www.linux-mag.com/id/4608/(Last visited 14-12-2013).

[23] Mallón, D. A., Taboada, G. L., Teijeiro, Ca., Touriño, J., Fraguela, B. B., Gómez, A. , Doallo, R., Mouriño, J. C. 2009. Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures, In proceeding of the 16th Recent Advances in Parallel Virtual Machine and Message Passing Interface, PVM/MPI2009, Espoo, Finland, 174-184.

[24] Rouchka, E. C. and Hardin, C. T. 2007. "rMotifGen: random motif generator for DNA and protein sequences", BMC Bioinformatics, Vol. 8, 2007.