Parallelizing Basic Operations in Parallel Filtering Model of DNA Computations

Andrey Breslav

Department of Higher Methemtics, St. Petersburg State University of Information Technology, Mechanics and Optics, 49, Kronverskiy pr, St. Petersburg, 197101, Russia

Abstract

In this paper we propose an approach which increases productivity of DNA computations by performing laboratory operations in parallel. We consider algorithms implemented in *Parallel Filtering Model* proposed by M. Amos, A. Gibbons and D. Hodgson. Basic operation of this model can be implemented using parallel laboratory operations. We show that by using this approach time complexity of some algorithms may be increased from polynomial to logarithmic. We also examine amount of DNA strands used and number of laboratory equipment ("tubes") needed for proposed implementation.

Key words: Parallel Filtering Model, DNA computations, Parallel implementation, NP-complete problems, DNA computation complexity

1 Introduction

Since Leonard Adleman has published his work [1] in 1994 many additions, improvements and alternative approaches were developed [6,7,9,11,14]. Several works including [3,6,15] give good overview of the whole area.

Particularly Adleman's work is a root of the family of computational models working on the same principle: selecting result of the computation from a set of all possible variants (see for example works [2,5,12,16]). Paper [10] shows that these models are computationally equivalent. These models are suitable for solving NP-complete problems.

Email address: `abreslav@gmail.com` (Andrey Breslav).

This approach may be considered to be somewhat extensive since we do not construct the result but just filter out incorrect solutions. Hence algorithms for NP-complete problems in such models require exponential amount of DNA and do not scale to large problem sizes [8]. But there is no exact algorithm for NP-complete problem which would have not been exponential. Even probabilistic algorithms require exponential space or time [13]. Thus, we should not neglect even such "extensive" approaches.

Here we deal with the *Parallel Filtering Model* (*PFM*) which is proposed in paper [5]. It defines four basic operations:

- $\mathit{Remove}(T, \{s_1, \ldots, s_n\})$ removes form multiset T any string which contains at least one occurrence of some string s_i;
- $\mathit{Union}(\{T_1, \ldots, T_n\}, T)$ creates a multiset T which is a multiset union of U_i;
- $\mathit{Copy}(T, \{T_1, \ldots, T_n\})$ produces a number of copies, T_i, of T;
- $\mathit{Select}(T)$ selects an element of T uniformly at random, if T is empty then *empty* is returned.

Papers [3,4] describe *Remove*, *Union* and *Copy* as having linear time complexity ($O(n)$). Paper [10] proposes a *normal form* of molecular program in *PFM*, where operations (we will refer to them as *elementary steps*) work with minimum amount of data.

- $\mathit{Union}(T_1, T_2, T)$ produces a union T of two multisets T_1 and T_2. Another form of this operation is $\mathit{Union}(T', T)$ where T' is added to T – this can be implemented as $\mathit{Union}(T, T', T)$.
- $\mathit{Copy}(T, T_1, T_2)$ creates two copies of T (T itself is destroyed then because of the nature of laboratory implementation [5]). Another form of this operation is $\mathit{Copy}(T, T')$ where T' receives a copy of T and T remains existing – this can be implemented as $\mathit{Copy}(T, T, T')$.
- $\mathit{Remove}(T, s)$ (where s is a single string) removes all the strings having a substring s from T.

Paper [4] mentions implementations of basic operations using elementary steps implicitly, paper [10] defines them explicitly. These implementations are given below (*Copy* is slightly changed to handle the fact that source tube is destroyed by each elementary *Copy*, which is ignored by [10]).

```
procedure $\mathit{Copy}(T, \{T_1, ..., T_n\})$
 1: for $i \leftarrow 1$ to $n$ do
 2:     $\mathit{Copy}(T, T_{i+1})$
 3: end for
end
```

```
procedure $Union(\{T_1, ..., T_n\}, T)$
1: $Union(T_1, T_2, T)$
2: for $i \leftarrow 3$ to $n$ do
3:     $Union(T_i, T)$
4: end for
end
```

```
procedure $Remove(T, \{s_1, ..., s_n\})$
1: for $i \leftarrow 1$ to $n$ do
2:     $Remove(T, s_i)$
3: end for
end
```

Time complexity of each of these operations is $O(n)$ [4,10].

In [10] author uses these linear implementations as a foundation of complexity analysis of some algorithms for NP-complete problems which results in polynomial time estimations like $O(n^3)$ for Hamiltonian circuit problem.

In this paper we propose parallel implementations of basic *PFM* operations which lead to $O(log(n))$ time complexities for them. Using of such implementations for some algorithms described in [10] reduces their execution times from $O(n^3)$ to $O(log(n))$ parallel time.

In section 2 we describe proposed parallel implementation of basic operations. In section 3 we analyze changes in complexity of some algorithms proposed in [10]. Section 4 gives concluding notes.

2 Modified Parallel Filtering Model

The nature of basic *PFM* operations does not require linear implementation. Some steps can be performed in parallel which leads to lower time complexities for basic operations and thus for algorithms implemented in *PFM*.

2.1 Parallel Copy operation

Here we present parallel implementation of *Copy* operation. This implementation is based on a simple principle of going down the binary tree: the first tube is copied into two ones, each of those is copied to another two and so on.

```
procedure $Copy(T, \{T_1, ..., T_n\})$
 1: if $n = 1$ then
 2:     $T_1 \leftarrow T$
 3: else if $n = 2$ then
 4:     $Copy(T, T_1, T_2)$
 5: else
 6:     $Copy(T, T'_1, T'_2)$
 7:     in parallel do
 8:         $Copy(T'_1, \{T_1, \ldots, T_{\lceil n/2 \rceil}\})$
 9:         $Copy(T'_2, \{T_{\lceil n/2 \rceil + 1}, \ldots, T_n\})$
10:     end in parallel
11: end if
end
```

Lines 2 and 4 properly handle basic cases of *Copy* and form the recursion base.

Calls made on lines 8 and 9 reduce problem size to $n/2$. Assuming that *Copy* works properly for $\vec{T}$ of size $n/2$ we see that these two calls fill up vector of size n with copies of original multiset.

Each level of recursion reduces n to $n/2$, thus any call to *Copy* ends up in one of basic cases after $O(log(n))$ levels of recursion. As all the calls on the same level of recursion are executed in parallel (lines 8 and 9) the whole operation ends up in $O(log(n))$ parallel time.

2.2 Parallel Union operation

Parallel implementation of *Union* operation uses the same metaphor of binary tree but unlike *Copy* it goes *up* the tree – from leaves to the root: parent node receives a union of it's children.

```
procedure $Union(\{T_1, ..., T_n\}, T)$
 1: if n = 1 then
 2:     $T \leftarrow T_1$
 3:     return
 4: end if
 5: $m \leftarrow \lfloor n/2 \rfloor$
 6: for each $i = (1, 2, \ldots, m)$ in parallel do
 7:     $Union(T_{2i-1}, T_{2i}, T'_i)$
 8: end for
 9: if $2m < n$ then
10:     $Union(T_n, T'_{m+1})$
11:     $m \leftarrow m + 1$
12: end if
13: $Union(\{T'_1, \ldots, T'_m\}, T)$
end
```

Lines 5 to 12 produce intermediate tubes $T'_1, \ldots, T'_{\lceil n/2 \rceil}$ each of which is a union of two tubes T_k and T_{k+1}. Recursive call on line 13 reduces n to $n/2$, thus recursion leads to $n = 1$ where it ends up on line 3 having all the tubes united in T.

Since each level of recursion reduces n to $n/2$ and **for** loop on line 6 is executed in parallel and takes $O(1)$ parallel time, the whole operation takes $O(log(n))$ parallel time.

2.3 Intersect operation and it's parallel version

To provide parallel implementation of *Remove* operation we need an operation of multiset intersection.

"Elementary step" $Intersect(T_1, T_2, T)$ takes two tubes T_1 and T_2 and produces their intersection T, during this process T_1 and T_2 are destroyed. Paper [7] describes it as feasible laboratory operation. This operation does not need to be added to *PFM* basic operation set to make the model more strong, but will be used only inside *Remove* operation's implementation.

Here we provide parallel implementation of $Intersect(\{T_1, ..., T_n\}, T)$.

```
procedure $Intersect(\{T_1, ..., T_n\}, T)$
 1: if $n = 1$ then
 2:     $T \leftarrow T_1$
 3:     return
 4: end if
 5: $m \leftarrow \lfloor n/2 \rfloor$
 6: for each $i = (1, 2, \ldots, m)$ in parallel do
 7:     $Intersect(T_{2i-1}, T_{2i}, T'_i)$
 8: end for
 9: if $2m < n$ then
10:     $T'_{m+1} \leftarrow T_n$
11:     $m \leftarrow m + 1$
12: end if
13: $Intersect(\{T'_1, \ldots, T'_m\}, T)$
end
```

This procedure is very similar to *Union* and thus has the same time complexity $O(log(n))$.

2.4 Parallel Remove operation

Parallel implementation of *Remove* is a little bit tricky: we cannot simply filter the same tube T in parallel but we can copy it and filter individual copies. After that we need just to intersect those copies as multisets.

```
procedure $Remove(T, \{s_1, \ldots, s_n\})$
 1: $Copy(T, \{T_1, \ldots, T_n\})$
 2: for each $i = (1, 2, \ldots, n)$ in parallel do
 3:     $Remove(T_i, s_i)$
 4: end for
 5: $Intersect(\{T_1, \ldots, T_n\}, T)$
end
```

Line 1 creates n copies of T, this takes $O(log(n))$ parallel time as shown above. On lines 2 to 4 each copy T_i is filtered by a substring s_i, being run in parallel this takes $O(1)$ parallel time. Finally all the filtered multisets are intersected and produce result T which has no string having a substring from $\{s_1, \ldots, s_n\}$. *Intersect* takes $O(log(n))$ parallel time.

Thus, the whole operation *Remove* takes $O(log(n))$ parallel time.

3 Hamiltonian path and circuit

Paper [10] represents several algorithms for NP-complete problems implemented in *PFM*. Here we examine complexity of two of these algorithms – Hamiltonian path and circuit – according to proposed parallel implementations of basic operations.

3.1 Notation preliminaries

Here we provide brief overview of notation used in [10].

There is an alphabet

$$\Sigma_n = \{p_1, p_2, \dots, p_n, a_1, a_2, \dots, a_n\}$$

and a set

$$T_n = \{p_1 a_{i_1} p_2 a_{i_2} \dots p_n a_{i_n} | 1 \leq i_j \leq n (j = 1, 2, \dots, n)\}.$$

The indexed p symbols denote the *positions* within a word and the indexed a symbols denote integer values between 1 and n. Integers are always separated by position markers.

Many algorithms operate on initial multiset

$$P_n = \{p_1 a_{i_1} p_2 a_{i_2} \dots p_n a_{i_n} \in T_n | 1 \leq j, j' \leq n, j \neq j' \Rightarrow i_j \neq i_{j'}\}$$

which represents all permutations of n distinct things. This multiset can be generated using basic operations in polynomial time [5] but actually since wide range of algorithms uses it, it might be useful to generate many instances of P_n in advance and then use those instances without spending time on it's generation.

3.2 Hamiltonian path problem

For a graph $G = (V, E)$, where $V = \{v_1, \dots, v_n\}$, node sequences are represented by strings in P_n. The string $p_1 a_{i_1} p_2 a_{i_2} \dots p_n a_{i_n}$ represents the node sequence $v_{i_1}, v_{i_2}, \dots, v_{i_n}$. For example such a node sequence is a Hamiltonian path if and only if it is a path. That is, the consecutive nodes in the sequence must be connected by an edge in E.

Thus, the following function from [5] finds Hamiltonian path in a graph.

```
**function** $HamiltonianPath(V, E)$
1: $T \leftarrow P_n$
2: $Remove(T, \{a_j p_i a_{j'} | 2 \leq i \leq n, 1 \leq j, j' \leq n \text{ and } \{j, j'\} \notin E\})$
3: **return** $Select(T)$
**end**
```

Paper [10] shows that when non-parallel implementation of *Remove* is used, this algorithm's time complexity is $O(n^3)$, since this is cardinality of the set of strings by which T is being filtered. If parallel implementation is used, this complexity reduces to $O(log(n^3)) = O(log(n))$ – complexity of $Remove(T, \{s_1, \ldots, s_{n^3}\})$.

3.3 Hamiltonian circuit problem

Given an n-node undirected graph $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$ find a Hamiltonian circuit (a simple circuit that includes all vertices of G).

The following function is quoted from [10].

```
**function** $HamiltonianCircuit(V, E)$
1: $T \leftarrow P_n$
2: $Remove(T, \{a_j p_i a_{j'} | 2 \leq i \leq n, 1 \leq j, j' \leq n \text{ and } \{v_j, v_{j'}\} \notin E\})$
3: $Copy(T, \{T_1, \ldots, T_n\})$
4: **for each** $i = (1, 2, \ldots, n)$ **in parallel do**
5:     $Remove(T_i, \{p_1 a_j | 1 \leq j \leq n \text{ and } j \neq i\})$
6:     $Remove(T_i, \{p_n a_j | 1 \leq j \leq n \text{ and } \{v_i, v_j\} \notin E\})$
7: **end for**
8: $Union(\{T_1, \ldots, T_n\}, T)$
9: **return** $Select(T)$
**end**
```

Correctness of this algorithm is proved in [10]. Time complexity in non-parallel model is $O(n^3)$, the longest operation is line 2 since cardinality of set of subsrtings by which T is being filtered is $O(n^3)$.

Let us analyze time complexity of this algorithm using proposed parallel implementation of basic operations. Then line 2 takes $O(log(n^3)) = O(log(n))$ parallel time. So does the following line. Lines 5 and 6 take $O(log(n))$ parallel time each. So the whole **for** does since it is executed in parallel. Final *Union* takes $O(log(n))$ parallel time and *Select* is executed in constant time. Thus, time complexity of this algorithm with parallel implementations of basic operations is $O(log(n))$.

Now let us look at other aspects of the algorithm's complexity. Number of DNA strands used by the algorithm is $O(n!)$ in both parallel and non-parallel implementations.

Number of elementary steps performed by this algorithm is still $O(n^3)$.

We also can examine a metric that can be thought about as *number of tubes*. Maybe tubes are not a really critical resource but this metric allows to measure number of laboratory equipment (other than tubes) used to execute the algorithm.

How many tubes do we need for *HamiltonianCircuit* procedure? Each of $Copy(T, \{T_1, \ldots, T_n\}$, $Union(\{T_1, \ldots, T_n, T\}$ and $Intersect(\{T_1, \ldots, T_n, T\}$ need $O(n)$ tubes which form a binary tree with n leaves. $Remove(T, \{s_1, \ldots, s_n\})$ consists of *Copy* and *Intersect* and thus needs $O(n)$ tubes too.

A call to *Remove* on line 2 needs $O(n^3)$ tubes, other operations need $O(n)$ tubes each. Thus "tube-complexity" of *HamiltonianCircuit* procedure is $O(n^3)$.

However we can reduce amount of tubes needed but we must pay for this with execution time. We can restrict *Remove* implementation to use not more than M tubes. Let us call this new operation $Remove_M(T, S)$. Having only M tubes we can build a binary tree with $\lceil M/2 \rceil$ leaves, so we simply take first $\lceil M/2 \rceil$ items from set S and filter T by them. Then we take next $\lceil M/2 \rceil$ and do the same and so on.

```
procedure $Remove_M(T, \{s_1, \ldots, s_n\})$
 1: $count \leftarrow 0$
 2: while $count < n$ do
 3:     $Remove(T, \{s_{count+1}, \ldots, s_{min\{count+M,n\}}\})$
 4:     $count \leftarrow count + M$
 5: end while
end
```

This results in $O\left(\frac{n \cdot log(M)}{M}\right)$ parallel time. So if we restrict the algorithm to use $O(n)$ tubes, we will use $Remove_n$ instead of *Remove* on line 2 and this line will take $O(n^2 log(n))$ parallel time, which is still better when non-parallel version while using asymptotically the same number of tubes.

4 Conclusion

This paper describes parallel implementations of basic operations of *PFM* which reduce each operation's execution time to $O(log(n))$ and allows to run solutions for such problems as Hamiltonian path and circuit in $O(log(n))$ parallel time.

Such implementation for *Copy* and *Union* does not require any model extensions but for *Remove* we need *Intersect* operation which is anyway feasible and it's biolological foundations are rather similar to *Copy*'s.

This implementation does not asymptotically increase amount of used DNA. It also allows to keep number of used laboratory equipment ("tube-complexity") in certain bounds, which needs sacrifice of execution time but anyway keep it better than for non-parallel implementation.

References

[1] L. M. Adleman, Molecular computation of solutions to combinatorial problems, Science 266 (11) (1994) 1021–1024.

[2] L. M. Adleman, On constructing a molecular computer, R. Lipton and E. Baum (Eds.), DNA Based Computers, DIMACS: Series in Discrete Mathematics and Theoretical Computer Science (1996) 1–21.

[3] M. Amos, Theoretical and Experimental DNA Computation (Natural Computing Series), Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[4] M. Amos, A. Gibbons, P. E. Dunne, The complexity and viability of DNA computations, in: D. Lundh, B. Olsson, A. Narayanan (eds.), Biocomputing and Emergent Computation, World Scientific, 1997.

[5] M. Amos, A. Gibbons, D. Hodgson, Error-resistant implementation of DNA computations, in: Proceedings of the Second Annual Meeting on DNA Based Computers, 1996.
URL `citeseer.ist.psu.edu/26034.html`

[6] M. Amos, G. Paun, G. Rozenberg, A. Salomaa, Topics in the theory of DNA computing, Theoretical Computer Science 287 (1) (2002) 3–38.

[7] E. Dantsin, A. Wolpert, A robust DNA computation model that captures PSPACE, International Journal of Foundations of Computer Science 14 (5) (2003) 933–951.
URL `citeseer.ist.psu.edu/671121.html`

[8] J. Hartmanis, On the weight of computations, Bulletin of European Association For Theoretical Computer Science 55 (1995) 136–138.

[9] C. V. Henkel, T. Back, J. N. Kok, G. Rozenberg, H. P. Spaink, DNA computing of solutions to knapsack problems, Biosystems 88 (1-2) (2006) 156–162.

[10] I. Katsanyi, Solutions of some classical problems in various theoretical DNA computing models, in: Proceedings of MolCoNet2, the Second Annual Meeting of Molecular Computing Network, Vienna University of Technology, 2003.

[11] R. J. Lipton, Speeding up computations via molecular biology, Technical report.
URL `citeseer.ist.psu.edu/lipton94speeding.html`

[12] Q. Liu, Z. Guo, A. E. Condon, R. M. Corn, M. G. Lagally, L. M. Smith, A surface-based approach to DNA computation, in: Proceedings of the Second Annual Meeting on DNA Based Computers, held at Princeton University, June 10-12, 1996.
URL `citeseer.ist.psu.edu/liu96surfacebased.html`

[13] W. Liu, L. Gao, Q. Zhang, G. Xu, X. Zhu, X. Liu, J. Xu, A random walk DNA algorithm for the 3-SAT problem, Current Nanoscience 1 (2005) 85–90.

[14] G. Paun, Membrane Computing, Springer, 2002.

[15] G. Paun, G. Rozenberg, A. Salomaa, DNA Computing: New Computing Paradigms, Springer, 1998.

[16] G. Rozenberg, H. P. Spaink, DNA computing by blocking, Theoretical Computer Science 292 (3) (2003) 653–665.