Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 561 – 570

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

Streaming Breakpoint Graph Analytics for Accelerating and Parallelizing the Computation of DCJ Median of Three Genomes

Zhaoming Yin[1], Jijun Tang[2], Stephen W. Schaeffer[3], David A. Bader[1,*]

1, School of Compuational Science and Engineering, Georgia Institute of Technology, Atlanta, GA 30332, USA
2,Deptartment of Computer Science and Engineering, University of South Carolina, Columbia, SC 29208, USA
3,The Huck Institutes of Life Sciences, Pennsylvania State University, State College, PA 16802, USA

Abstract

The problem of finding the median of three genomes is the key process in building the most parsimonious phylogenetic trees from genome rearrangement data. The median problem using Double-Cut-and-Join (DCJ) distance is NP-hard and the best exact algorithm is based on a branch-and-bound best-first search strategy to explore sub-graph patterns in Multiple BreakPoint Graph (MBG). In this paper, by taking advantage of the "streaming" property of MBG, we introduce the "footprint-based" data structure to reduce the space requirement of a single search nodes from $O(v^2)$ to $O(v)$; minimize the redundant computation in counting cycles/paths to update bounds, which leads to dramatically decrease of workload of a single search node. Additional heuristic of branching strategy is introduced to help reducing the searching space. Last but not least, the introduction of a multi-thread shared memory parallel algorithm with two load balancing strategies bring in additional benefit by distributing search work efficiently among different processors. We conduct extensive experiments on simulated datasets and our results show significant improvement on all datasets. And we test our DCJ median algorithm with GASTS, a state of the art software phylogenetic tree construction package. On the real high resolution Drosophila data set, our exact algorithm run as fast as the heuristic algorithm and help construct a better phylogenetic tree.

Keywords: Genome Rearrangement, Double-cut-and-joining Median, Parallel Programming

1. Introduction

Inferring phylogenies (evolutionary history) of given species is a fundamental problem in computational biology [20]. Although DNA sequence data is still the dominant source of data, building phylogenies from

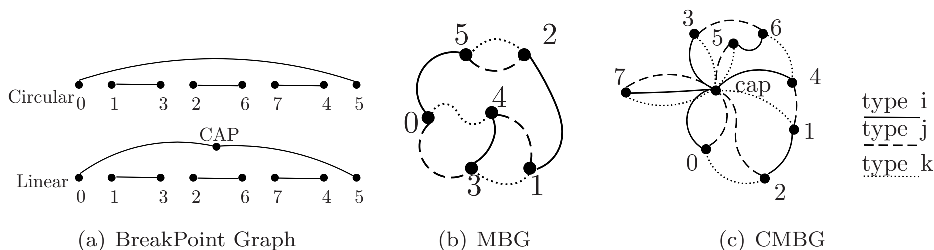
[1]This Research was sponsored in part by the NSF PetaApps Grants OCI-0904461 (Bader), OCI-0904179 (Tang), OCI-0904166 (Schaeffer) and NSF OCI-1214504 (Bader).

* *Corresponding author, URL:* `http://www.cc.gatech.edu/~bader/` (David A. Bader)

1877-0509 © 2013 The Authors. Published by Elsevier B.V. Open access under CC BY-NC-ND license.
Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.220

higher-level changes such as genome rearrangements has gained increasing interests from the field because of the larger number of states (4 for nucleotide sequences versus n-1 gene adjacencies). The problem of finding median of three genomes based on genome rearrangement, is the building block for constructing a phylogenetic tree under the maximum parsimony criteria [8, 14]. Given three genomes, it asks for a "median" genome which has the minimum accumulated genome rearrangement operations (distance) between these three genomes. There are different methods to solve the median problem using different measurement of distance metrics, such as break-point [16], reversal [3], translocation [1] and double-cut-and-join(DCJ) [23]. Double-cut-and-join (DCJ) operation can unify the operation of reversal, translocation, fusion, and fission with only two moves on a Breakpoint graphs, which makes it the most studied operation in the last few years. The DCJ median problem has proven to be NP-hard and APX-hard [13] and several exact algorithms have been implemented to solve the DCJ median problems on both circular chromosomes [4, 5] and linear chromosomes [6, 7]. These methods are fast when genomes are similar, but as the search space grows to prohibitively large, it may take months if not years for them to finish when the genomes are distant.

Emerging social network research has introduced the method of streaming graph analysis [11, 18], which deals with how to quickly maintain information on a temporally or spatially changing graph without traversing the whole graph. Because of the combinatoric nature of genome rearrangement analysis, various types of graphs are introduced to represent genome sequences, for example the overlap graph [9], breakpoint graph (BPG), [17] and cycle graph [15], of which the breakpoint graph is studied the most. Many algorithms explore the changing properties of these graphs, which can also be viewed as graph streaming. In this paper, we picked the *DCJ median* algorithm which deals with *BPG* as an example, to show how streaming graph analysis methods can help design genome rearrangement algorithms and achieve significant improvement on both time and space needed to complete the analysis.



(a) BreakPoint Graph (b) MBG (c) CMBG

Figure 1: Examples of Breakpoint Graphs.

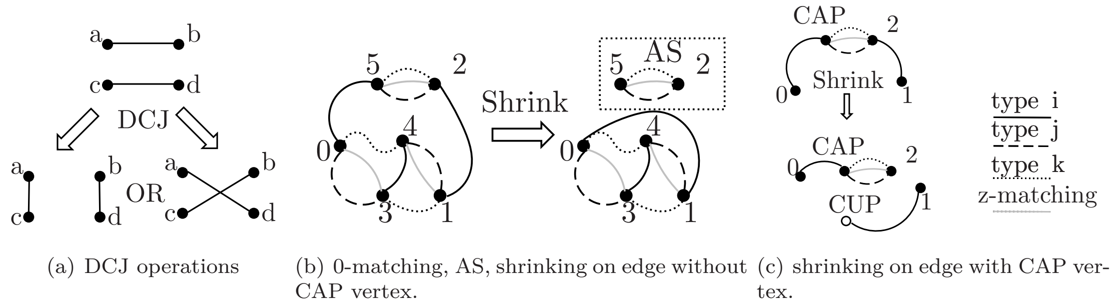
2. Preliminaries

Breakpoint Graphs (BPG): given a genome which contains g non-duplicate genes, each gene is marked with a signed number $1 \leq |i| \leq g$. A breakpoint graph $BPG = (V, E)$, such that V is consists of $2g$ vertices, each gene i is represented by a pair of vertices head (marked $|i| * 2 - 2$) and tail(marked $|i| * 2 - 1$), and these vertices are ordered following the gene sequence: $head(i)$ is ordered in front of $tail(i)$ if i is positive, otherwise, $tail(i)$ is put in front of $head(i)$. E consists of g edges and if two genes i and j are adjacent to each other in the gene order(..., i, j....), an edge will connect their adjacent vertices. When dealing with end points, if the chromosome that the genome contains is circular, two end points of the breakpoint graph will be connected by an single edge, if the chromosome is linear, each of two end points will connect by an edge to a separate vertex called a CAP vertex. Figure 1(a) shows the BPG for gene sequence (1,-2,4,3) under circular and linear cases.

Multiple Breakpoint Graph (MBG): Given multiple genomes which have the same set of non-duplicated genes, we can define a MBG by using different type of edges to represent different genomes. If genomes

consist only of circular chromosomes, the constructed graph is a MBG. Figure 1(b) shows the MBG for three gene orders ((1,2,3);(1,3,2);(1,-2,-3)). It's easy to notice that MBG is 3-regular graph.

Capped Multiple Breakpoint Graph (CMBG): If genomes consist of single or multiple linear chromosomes, the constructed graph is $CMBG$. Figure 1(c) shows the CMBG for three gene orders that each are consisted of two chromosomes ((1,2;3,4);(1,3;2,4);(-1,2;-4,3)) (';' indicates the end of a chromosome). Other than CAP vertex, every vertex in $CMBG$ has degree of 3. If not specified, we use BPG to represent all these three classes of graphs.



(a) DCJ operations (b) 0-matching, AS, shrinking on edge without CAP vertex. (c) shrinking on edge with CAP vertex.

Figure 2: Examples of operations on BPG.

Double-Cut-and-Join(DCJ): *DCJ* is the operation that cuts two edges of the *BPG* then rejoins its four end vertices. Figure 2(a) shows an example of *DCJ* operations. Given two genomes, the *DCJ* distance is the minimum number of *DCJ* operations to transform one genome into another. Given three genomes, the *DCJ* median is the genome which has the least accumulate distance between the other three genomes, which can be mapped to *BPG* by a maximal matching that represented by black edges called *0-matchings*.

Adequate Subgraphs (ASs): *ASs* defined in [4, 6] are such graphs that if a *BPG* is partitioned into two parts, one part is an *AS*, the other part is the rest of the graph, there will be no *0-matching* edges connecting these two parts. *ASs* can be used to parition the *BPG* and help to find the *DCJ median* without losing accuracy.

Edge Shrinking: *ASs* can be bridged out from the *BPG* by edge shrinking operations. When an *AS* is selected to partition the graph, its possible *0-matching* edges are selected at first. Then, if one *0-matching* edge does not contain *CAP* vertex, edges of the same type that incident to the same *0-matching* edge will be joined into a single edge; If the *0-matching* edge contains a *CAP* vertex, the edge incident to the non-*CAP* vertex will be connected to a *CUP* vertex. Figure 2(b) shows examples of *0-matching*, *AS*, and edge shrinking on the edge that does not contain *CAP* vertex. Figure 2(c) shows an example of edge shrinking on edge that contains a *CAP* vertex.

Edge Expansion: Edge expansion is the reverse operation of edge shrinking.

$$DCJ(i,j) = ij_c + i_u i_u + j_u j_u + \frac{i_a j_u + i_u j_a + i_a j_a + i_u j_u}{2} + min(i_a i_a, i_a i_u) + min(j_a j_a, j_a j_u) \quad (1)$$

Cycle: In a *BPG*, start with a vertex, and visit the edges with type i and j repeatedly, if this tour comes back to the start vertex, all vertices and edges contained in this tour are in an *i-j cycle* which in abbreviation ij_c.

Path: In a *CMBG*, start with a *CAP* or *CUP* vertex, and visit edges with type i and j repeatedly, and this tour will come back to the CAP or CUP vertex, all vertices and edges contained in this tour are in a

path, we can name different types of paths by types of their first and last edge and the start/end vertex. For example, *i-cap-j-cup path* defines a path that start from *CAP* vertex with an type-i edge and end until a *CUP* vertex with a type-j edge, we can abbreviated this path as $i_a j_u$. Every vertex in an ij_c or in a path has degree of 2. In *BPG*, the *DCJ* distance between two genomes marked by i and type-j edges, is calculated by Formula 1.

Best-first-search based Branch and Bound Algorithm (A* BnB): The best algorithm to solve the DCJ median problem is an *A* BnB* algorithm. When searching on a search node represented by a *BPG* with v vertices, **first** it traverses the *BPG* to detect *ASs*, which is a classic subgraph isomorphism problem [12]. Since this step only detects small ASs of limited patterns, special algorithms have been developed which all have time/space complexity of $O(v)$; **second**, if there are *ASs* found, one or two child search nodes will be expanded by shrinking the *0-matching* edges in *ASs*, otherwise there will be $v - 1$ child search nodes expanded. At the very beginning of this search process, there usually are a lot of *ASs* in *BPG*, but after a few steps, *BPG* will be altered to a state that for the first time there is no *ASs* exists in it, we call it *BPG kernel*, and the number of vertices in *BPG kernel* is marked as κ. When the searching process reaches *BPG kernel*, it's hard to generate new *ASs* by shrinking "assumed" *0-matching* edges, and the following search processes have a branching factor dicided by κ. This step has time/space complexity of $O(v)$; **third**, each *BPG* of the expanded child search node will be traversed to get cycle/path number for updating the upper and lower bounds. In this step, if *ASs* are detected, the time/space complexity is $O(v)$. In contrast, if there is no *ASs* detected, the time/space complexity become $O(v^2)$.

3. Streaming Breakpoint Graph Analysis Methods

One very important point to notice is that, κ determines the performance of this *A* BnB* algorithm. To begin with, the algorithm's branching factor is decided by κ. In addition, if there is no *ASs* in a search node, the third step in the *A* BnB* will have complexity of $O(v^2)$, which makes the complexity for processing this search node $O(v^2)$. Last but not least, if κ is large there will be a huge number of *BPGs* generated for child search nodes, which makes this algorithm additionally bound by I/O. However, most of the operations on *BPG* is minor, which only involves change on one edge, by utilizing the properties of streaming *BPG*, it is possible to reduce both time and space requirement for processing a search node.

3.1. Compressed Data Structures for Memory and I/O Efficiency

In the current *DCJ* median algorithm, every time when there are new child search nodes expanded, new *BPGs* will be allocated and pushed into a search list. As a result, huge amount of memory allocation and I/O can not be avoided. Because each *BPG* of a child search node is generated from shrinking edges of the *BPG* of the root search node following several steps, we can store these edges as "footprints" instead of *BPGs* themselves. Here, because we only consider search nodes from the *BPG kernel* stage (because processing node before this stage is fast), every search node could only store footprints after the *BPG kernel* stage. In other words, there is only one *BPG* in the memory, when finished processing search node a with footprint *f-a*, and continue searching from another search node b with footprint *f-b*, edges of *BPG* in *f-a* are expanded then edges in *f-b* are shrunk to switch from node a to node b. In addition, if there is no *AS* found in the parent search node, the expanded $v - 1$ child search nodes will all share the same footprint of their parent. Under such circumstance, the cost of storing a child search node can be additionally contracted to only 1 edge and a pointer to the footprint of their parent search node.

3.2. A Fast Method to Update Cycle/Path Numbers

In the current *DCJ* median algorithm, every vertex needs to be visited once to estimate cycle/path numbers for bounding. However, when the *BPG* of a child search node is generated by just shrinking one edge from the *BPG* of its parent node, we have the following observations (see Figure 3 for example.) to help

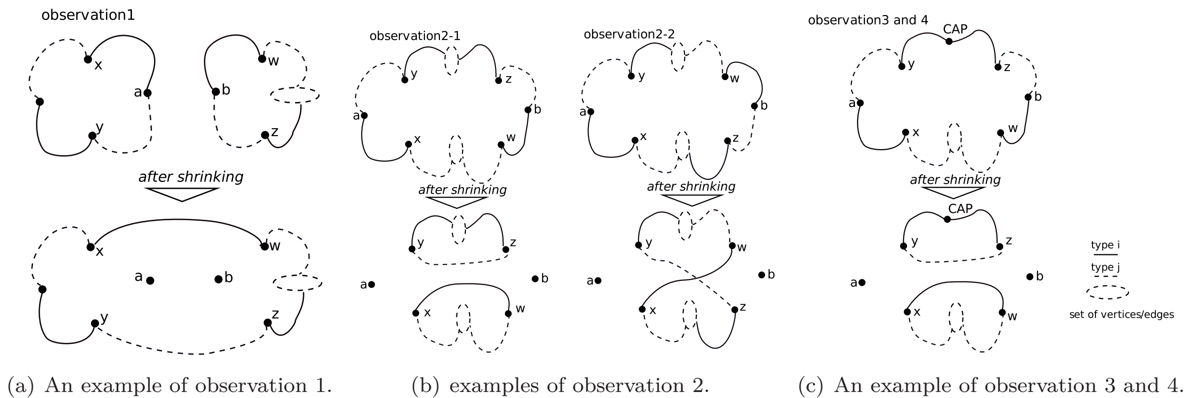
design quick method to update cycle/path numbers. Suppose shrinking one *0-matching* edge connecting two vertices a and b (a and b are not neighbors, because if they are neighbors after shrinking, they will be bridged out from the *BPG* and there will be no change in the cycle number), the type-i and type-j edge of a is connected to vertex x and y, the type-i and type-j edge of b is connected to vertex w and z.

Observation 1. *If a and b are in different connected components (ij_c or* path*), and at least one vertex is in a ij_c, after shrinking, the number of ij_c will decrease by 1.*

Observation 2. *If a and b are in the same cycle, after shrinking, the number of ij_c could be changed by either increase 1, or stay the same).*

Observation 3. *If a and b are in the same path, after shrinking, the number of cycles will be increase by 1 if and only if x and w are in the same adjacent vertices set after the component is separated by a and b.*

Observation 4. *If at least one of a and b are in a path, after shrinking, the number of path stays the same.*



(a) An example of observation 1. (b) examples of observation 2. (c) An example of observation 3 and 4.

Figure 3: Examples of different observations when only one 0-matching is shrunk.

update_cycle_fast(a, b)
if a and b in different cycle **then**
  cycle number decrease 1;
else $\{a$ or b is in a cycle$\}$
  x=neighbor[a][i] ;
  w=neighbor[b][i] ;
  if x and w are in the same adjacency vertices set **then**
    cycle number increase 1;
  end if
end if

update_path_type_fast(a, b)
if a and b in different path **then**
  update path types;
else $\{a$ or b is in a path$\}$
  x=neighbor[a][i] ;
  w=neighbor[b][i] ;
  if x and w are in the same adjacency vertices set **then**
    cycle number increase 1;
    update path types;
  end if
end if

Based on Observation 1, 2 and 3, we can update the cycle number in $O(1)$ time if only one *0-matching* edge is shrunk. Based on Observation 4, it's possible to visit only the connected component of affected vertices to determine the change of path type. The summarized fast algorithm for updating cycle/path number is shown in *update_cycle_fast*(a, b) and *update_path_fast*(a, b) separately.

3.3. Heuristics for Reducing Search Space

In a recent paper by Rajan et. al, [22], they designed approximation algorithm to compute DCJ median, though DCJ median problem is *APX-hard*, the algorithm achieved a good approximation rate for reverse median problem [3] (using DCJ median algorithm to approximate reverse median problem). This method

can be served to provide initial tighter lower bound, even though it might not be helpful for reducing search space, it can be useful in reducing space requirement, because those branched search node whose upper bound is smaller than the global maximal lower bound will be discarded, and the initialization of a tighter global lower bound can increase this threshold. In this paper, we introduce a branching strategy, which is helpful to quickly exhaust search nodes with the maximal upper bound and approach lower bound faster. In [5], the author state that the upper bound is $upperBound = c + \frac{3}{n} + \frac{c_{1,2}+c_{1,3}+c2,3}{2}$ (actually, this formula is just for circular chromosomes, but we can extent it to linear chromosomes). Suppose vertex V_k is in ijc_k and we define *cycle rank sum of* V_k as $R(V_k) = \sum_{(i,j)\in(1,2,3),i\neq j} |ijc_k|$.

Proposition 1. *If there is no* AS *detected, and we select the vertex* V_k *in* BPG *such that* $R(V_k) = min(R(V_i))(V_i \in BPG)$*, and shrink this vertex with other vertices in* BPG *to generate* $|2v - 1|$ *intermediate child BPGs, the number of child BPGs with upper bound value that is equal to the global maximal upper bound is minimized.*

Based on Observation 1, we know that if two vertices are in different component (cycle), after shrinkage the total cycle number will decrease by 1, since vertex V_k with $R(V_k) = min(R(V_i))(V_i \in BPG)$ has the least number of vertices that share component(cycle) with it, then shrink this vertex with all other vertices will cause the maximal number of cycles to merge, and the upper bound will decreased.

3.4. Shared Memory Parallel Algorithm

Since there are a lot of articles about parallel branch and bound algorithms [10], we will not dive into detail about the framework of the parallel algorithm. Here we discuss two load balancing strategies that we use in our shared memory multi-thread parallel algorithm. The first strategy is, when a thread has finished its work, it will check the intermediate files of other threads with the maximal upper bound value, if such files exits, it can “steal” the tasks of other threads by just renaming the files to its own, this strategy has very little overhead of synchronization. The second strategy is, when there is no files left of other threads, instead of “stealing” other threads’ work, this thread just kill itself, and notify one of other threads with maximal expected work to do to fork their jobs to a new thread to continue searching.

4. Experimental Results

We conduct our experimental tests of this method on a machine using the linux operating system with 16 Gb of memory and an Intel(R) Xeon(R) CPU E5530 16 core processor, each core has 2.4GHz of speed. Because the existing DCJ algorithms (both for circular and linear chromosomes) are implemented using JAVA [4, 6, 7], we also implement our algorithm using JAVA, all of the source code are compiled with JDK1.7 with -O option.

4.1. Time/space usage

In [4, 6, 7], the authors analyzed the performance of the original *DCJ* median algorithm with a simplified model, i.e. they only allowed reversal as the event and assumed that the *DCJ* distance between each of the three species is the same. In this paper, we generate simulated data using a model which is biologically more accurate. First we generate phylogenetic model trees of three genomes by using a birth-death model [24]. Then, based on the model tree, we execute *DCJ* operations on each genome for some random times, then we select the generated data of a specific κ, we collect 10 data samples for each κ. For the test of time/space complexity, we run the program to process only 10k search nodes then return. In the experiment, we have a parameter called *thresh*, which stands for the threshold for the number of graphs stored in the memory, if the number exceeds this threshold, these graphs will be stored back to the disk. We have tested the threshold for 2k and 10k nodes (k=1000).

Figure 4: Time complexity comparison for our streaming algorithm (_s) and the original algorithm (_o).

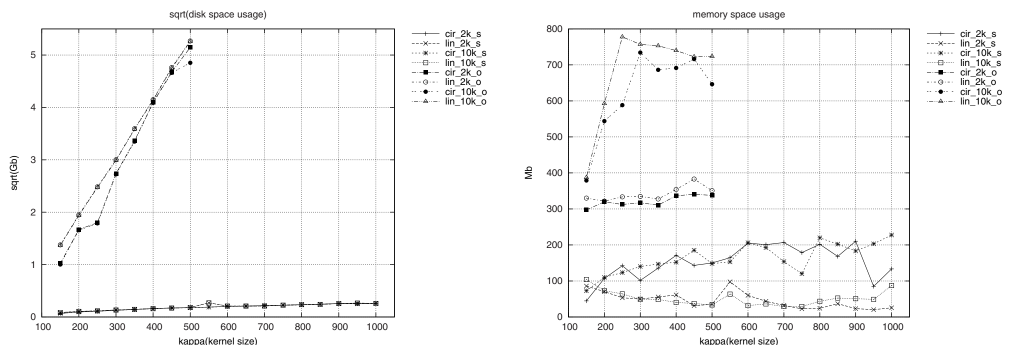


Figure 5: Space complexity comparison for our streaming algorithm (_s) and the original algorithm (_o).

Table 1: The running time and search space for circular chromosomes

Search space for Circular Chromosome

κ	method	finished	avg time (seconds)	avg search space
80	optimized	10/10	6.42	159784
	non-optimized	8/10	>336.61	> 9861818
90	optimized	10/10	27.66	669608
	non-optimized	6/10	N/A	N/A
100	optimized	10/10	850.33	17146636
	non-optimized	1/10	N/A	N/A

Search space for linear Chromosome

κ	method	finished	avg time (seconds)	avg space
80	optimized	10/10	44.77	442855
	non-optimized	10/10	6576.07	71782336
90	optimized	10/10	201.48	1729328
	non-optimized	2/10	N/A	N/A
100	optimized	10/10	24236.91	138420207
	non-optimized	0/10	N/A	N/A

In Figure 4, combined with the total time, we also recorded four types of time: time to detect adequate subgraphs, time to store and retrieve graphs, time for I/O and time to update lower/upper bounds. We can see from the figure that for the original algorithm, under all of cases, the time grows quadratic (since the time for the original algorithm to run larger data ($\kappa > 500$) is so large, we do not include these results), and for our stream algorithm, the time grows linearly. This is also true for the separate time of I/O, memory access and bound update. Another thing is, when the threshold is increased from 2k to 10k, our stream algorithm time reduces a lot on both circular and linear chromosome cases, but as for the original algorithm, it even takes more time. In addition, we have also noticed the reduce of time for detecting ASs, this could due to our code optimization, but also the improvement of memory locality, since there is only one graph stored in the memory. Last but not least, we can see that in the original algorithm, it spends more time on bound update than *ASs* detection, but our streaming algorithm reduce the bound update time to less than *ASs* detection time.

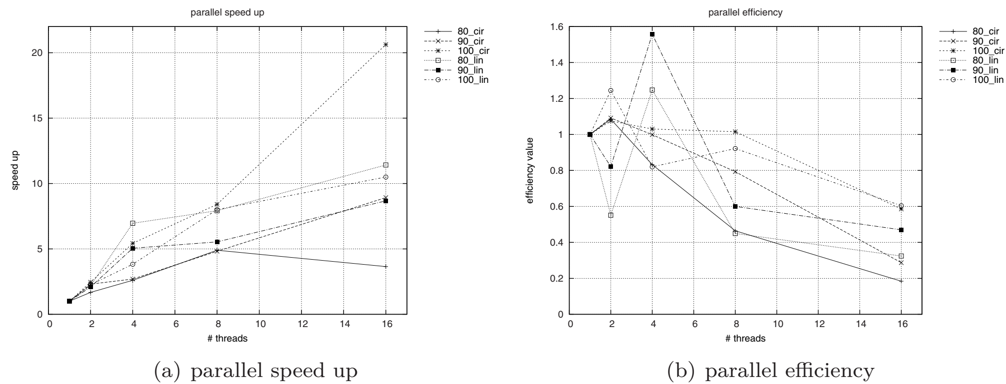
Figure 5 shows the space usage for the original algorithm and our streaming algorithm, it's very clear that our algorithm takes way less storage(memory and disk) than the original algorithm. One thing to notice is, for every graph in the original algorithm, it stores the according median genome for that graph, and the space usage is $O(g)$, we disable this storage to make the comparison fair, because our "footprint" based data structure and make sure that we don't need to keep the median genome information.

4.2. Problem space for complete search

In our previous discussion, we state that the performance of *DCJ* algorithm is directly dependent on *BPG kernel* size κ, and the figure shows that both the time/space complexity have been reduced, which makes our improvements well justified. However, the improvement of the time/space complexity can only introduce at most two orders of magnitude of speedup, we can additionally reduce the search space to gain more performance by using heuristics. In Table 1, it shows the comparison between methods that performs the heuristic of upper/lower bound (optimized) and which does not use heuristics (non-optimized). For a given κ, if the non-optimized algorithm can not finish the search by searching the maximal number of search nodes of optimized algorithm, we marked it as "unfinished". From the table, it's very clear that our heuristic reduced search space for both circular and linear chromosomes by a factor of nearly two.

4.3. Parallel speedup

We implement our parallel algorithm by using JAVA threads, and we perform the experiment on the same data set of the previous section, the speedup is averaged over 10 cases. We can see from Figure 6(a) that our algorithm achieved very good parallel speed up, especially on large data search space problems. For



(a) parallel speed up (b) parallel efficiency

Figure 6: Parallel speed up.

Table 2: The experiment result for phylogenetic tree construction

dataset	num genes	max κ	median κ	Heuristic tree length	Exact tree length	Heuristic time (seconds)	Serial time (seconds)	Parallel time (seconds)
dros-5	9738	172	12	4395	4320	102.8	1449	394.5
dros-12	7332	234	60	5305	5244	547.1	7055	1933

the circular chromosome, when $\kappa = 80$, the algorithm scales well up to 8 threads, and when $\kappa = 90$ we can achieve speed up close to 10 when using 16 threads. Surprisingly, we can see even super-linear speed-ups for some cases such when $\kappa = 100$ and thread number is 16.The observation of super-linear speed up is due to the reduce of the search space when multiple thread is running. As for linear chromosomes, because the data we generated has larger search space and they are more evenly distributed, we can see that for all three kernel sizes, the algorithm scales well.

Figure 6(b) shows the normalized efficiency for our parallel program which is calculated by $\frac{T_s}{T_p} \times \frac{W_p}{W_s}$ of which T_s/T_p is the serial/parallel execution time and W_s/W_p is the number of total processed nodes (total work) for serial/parallel program. In general, when increasing the number of threads, the parallel efficiency is reducing. There might be multiple reasons: 1) there is large overhead for Java thread scheduling, 2) the memory allocated is increasing with the number of threads growing, 3) there are overhead for synchronization when threads are doing load balancing.

4.4. Experiment on Real Drosophila data

We plugged in our algorithm into the state of art phylogenetic tree construction package GASTS [7] which uses a DCJ median heuristic [22], and conduct the experiment on the real Drosophila data set [2, 21], we designed two data sets with the number of species increased. One with 5 species (which are Dmel, Dere, Dana, Dpse and Dwil), we choose these 5 species because they have a long diameter in the phylogenetic relations. Another with all of 12 species of Drosophila. We deleted all of insertion, deletion and duplications to make each species has the same gene content, and the 5 species data set has more genes than 12 species data set because as more divergent taxa are included the number of shared orthologous genes decreases. (see Table 2). The experimental results are shown in Table 2, we do not use a model tree in our experiment. We can see that during the process of constructing the phylogenetic tree, most of the median problems are easy to solve (with a small κ). However, there are a few median problems that is extremly hard to solve, which has a very large κ. For these problems, we set the program's threshold to search up to 1 million node then return the local minimum. On both data sets, our DCJ median plug-in helps to find a better phylogenetic tree which has smaller accumulated tree length, and comparing to the heuristics based median solver our serial code is just about 10 times slower, by utilizing the parallel algorithm, this gap reduces to about 4.

5. Conclusion and Future Work

Our optimized algorithm specialized in reducing redundant storage and computation by taking advantages of streaming properties of *BPG*, achieves many folds of acceleration. With the introduction of heuristic, and parallel computing method, many real-size datasets can now be solved within a limit number of search nodes. Additional benifits might be explored to further improve our algorithm. For example, Zhang et al. [25] introduced a mixtrue framework to accelerate the DCJ median computation, combined with their method, our streaming algorithm might run even faster. Compeau [19] proposed the breakpoint graph analysis based *DCJ-Indel* distance model, it's possible to apply our algorithm to compute the *DCJ-Indel* median efficiently.

References

[1] A. Bergeron, J. Mixtacki and J. Stoye, *On sorting by Translocations*, J Comput Biol, 2006 Mar,13(2) pp. 567-78.

[2] Bhutkar, A., S. W. Schaeffer, S. Russo, M. Xu, T. F. Smith et al., *Chromosomal rearrangement inferred from comparisons of twelve Drosophila genomes*, Genetics 179: 2008 1657-1680.

[3] A. Caprara, *The Reversal Median Problem*, INFORMS Journal on Computing, 2003, 15(1), pp. 93-113.

[4] A. W. Xu and D. Sankoff, *Decompositions of multiple breakpoint graphs and rapid exact solutions to the median problem*, WABI, 2008, pp. 25-37.

[5] A. W. Xu, *A Fast and Exact Algorithm for the Median of three Problem: a Graph Decomposition Approach*, J. Comput Biol, 2009, 16(10), pp. 1-13.

[6] A. W. Xu, *DCJ Median Problems on Linear Multichromosomal Genomes: Graph Representation and Fast Exact Solutions*, F.D. Ciccarelli and I. Miklos (Eds.): Proceedings of RECOMB Comparative Genomcis, LNBI 5817, 7083. 2009.

[7] A. W. Xu and B. M. Moret, *GASTS: Parsimony Scoring under Rearrangements*, WABI, 2011, pp. 351-363.

[8] B. M. Moret , J. Tang , L. Wang and T. Warnow, *Steps Toward Accurate Reconstructions of Phylogenies from Gene-Order Data*, J. Compt. Syst, 2002, Vol 65, pp. 508–525.

[9] D. A. Bader, B. M Moret and M. Yan, *A Linear-time Algorithm for Computing Inversion Distance between Signed Permutations with an Experimental Study*, J Comput Biol, 2001, 8(5), pp. 483-91.

[10] D .A. Bader, W .E. Hart, C .A. Phillips *Parallel Algorithm Design for Branch and Bound* Tutorials on Emerging Methodologies and Applications in Operations Research International Series in Operations Research & Management Science Volume 76, pp 5.1-5.44, 2005

[11] D. Ediger, J. Riedy, H. Meyerhenke, and D.A. Bader, *Tracking Structure of Streaming Social Networks*, 5th Workshop on Multithreaded Architectures and Applications (MTAAP), Anchorage, AK, May 20, 2011.

[12] D. Ullmann, R Julian, *An algorithm for subgraph isomorphism*, Journal of the ACM, 1976, 23 (1), pp. 31-42.

[13] E. Tannier, C. Zheng, D. Sankoff, *Multichromosomal genome median and halving problems*, WABI; Vol. 5251 LLNCS, 2008, pp. 1-13.

[14] G. Bourque, P. A. Pevzner, *Genome-scale evolution: reconstructing gene orders in the ancestral species*, Genome Res, 2002 Jan, Vol. 12, No. 1, pp. 26-36.

[15] G. Fertin, A. Labarre, I. Rusu, E. Tannier and S. Vialette, *Combinatorics of Genome Rearrangements*, First sd., The MIT press, Cambridge, MA, 2009.

[16] I. Pe'er , R. Shamir, *The median problems for breakpoints are NP-complete*, Elec. Colloq. on Comput. Complexity, 1998, Vol 71.

[17] M. A. Alekseyev and P. A. Pevzner, *Breakpoint Graphs and Ancestral Genome Reconstructions*, Genome Res, 2009, May 19(5), pp. 943-57.

[18] O. Green, R. McColl and D. Bader, *A Fast Algorithm For Incremental Betweenness Centrality*, 4th ASE/IEEE International Conference on Social Computing, 2012.

[19] P.E. C. Compeau *A Simplified View of DCJ-Indel Distance*, Algorithms in Bioinformatics Lecture Notes in Computer Science Volume 7534, 2012, pp 365-377

[20] P. A. Pevzner, *Computational Molecular Biology: An Algorithmic Approach* First sd., The MIT press, Cambridge, MA, 2000.

[21] S. Shaeffer, S. W., A. Bhutkar, B. F. Mcallister, M. Matsuda, L. M. Matzkin et al., *Polytene chromosomal maps of 11 Drosophila species: The order of genomic scaffolds inferred from genetic and physical maps*, Genetics 179: 2008 1601-1655.

[22] V. Rajan, A. W. Xu , Y. Lin , K. M. Swenson and B. M.E. Moret *Heuristics for the Inversion Median Problem*, Proc. 8th Asia-Pacific Bioinformatics Conf. APBC 10.

[23] S. Yancopoulos, O. Attie, R. Friedberg, *Efficient sorting of genomic permutations by translocation, inversion and block interchange*, Bioinform, 21, 2005 3340-C3346.

[24] Y. Lin, V. Rajan and B .M .E. Moret, *Fast and accurate phylogenetic reconstruction from high-resolution whole-genome data and a novel robustness estimator*, RECOMB-CG, 2010, pp. 137-148.

[25] Y. Zhang, F. Hu and J. Tang, *A Mixture Framework for Inferring Ancestral Gene Orders*, APBC 2012, in BMC Bioinformatics 2012, 13(Suppl 1):S7