Genome alignments using MPI-LAGAN

Ruinan Zhang
Computer Science
University of Minnesota
Minneapolis, MN 55414
rzhang@cs.umn.edu

Huzefa Rangwala
Computer Science
George Mason University
Fairfax, VA 22030
rangwala@cs.gmu.edu

George Karypis
Computer Science
University of Minnesota
Minneapolis, MN 55414
karypis@cs.umn.edu

Abstract

We develop a parallel algorithm for a widely used whole genome alignment method called LAGAN. We use the MPI-based protocol to develop parallel solutions for two phases of the algorithm which take up a significant portion of the total runtime, and also have a high memory requirement. The serial LAGAN program uses CHAOS to quickly determine initial anchor or seeds, which are extended using a sparse dynamic programming based longest-increasing subsequence method. Our work involves parallelizing the CHAOS and LIS phases of the algorithm using a one-dimensional block cyclic partitioning of the computation. This leads to development of an efficient algorithm that utilizes the processors in a balanced way. We also ensure minimum time spent in communication or transfer of information across processors.

We also report experimental evaluation of our parallel implementation using pairs of human contigs of varying lengths. We discuss and illustrate the challenges faced in parallelizing a sparse dynamic programming formulation as in this work, and show equivalent to theoretical speedups for our parallelized phases of the LAGAN algorithm.

1 Introduction

Alignment algorithms play a crucial rule in analyzing whole genomes, identifying similar and conserved regions between pairs of genomes, leading to annotation of genomes with site-specific properties and functions.

Over the years several genome alignment algorithms like BLAT [4], MUMMER [5], and LAGAN [2] have been developed. These algorithms quickly and accurately align genome pairs by fast extraction of short exact or inexact matches using different indexing techniques.

In this work we parallelize the anchor-based LAGAN [2] to align large genomic sequences using a large number of processors available on supercomputers and clusters. The serial LAGAN program uses CHAOS [1] to quickly determine the initial anchor points which are extended using a sparse dynamic programming based longest-increasing subsequence method (LIS). Our work involves parallelizing the CHAOS step by dividing the workload row-wise across processors, and using a block cyclic partition based technique for implementing the parallel LIS algorithm. The parallel implementation is done using the MPI protocol, and the parallel implementation will be referred to as MPI-LAGAN.

We perform a set of empirical studies to evaluate the speedup and computational efficiency of our parallel implementation. Our parallel CHAOS implementation produces slightly different anchor points without loss in accuracy of the final global alignment, with an improved runtime and efficient use of memory. The parallel implementation allows us to achieve theoretical speedup, with good load balancing and processor utilization. [1]

2 Serial LAGAN Algorithm

LAGAN [2] is an anchor-based pairwise global alignment algorithm, and can be described using three steps: (i) the sequence pairs are analyzed to identify an initial set of high-quality local alignment that will restrict the search space for the final global alignment, (ii) LAGAN determines maximal-scoring ordered subsets of local alignment subsets (called *anchors*), and generates a rough global map of the alignment, and (iii) a dynamic-programming based algorithm is used to determine the optimal global alignment within a limited area of the global map. For pairs of successive anchors that are apart by a certain distance threshold, the first two steps are applied recursively to determine the rough global map between the subsequences, connecting the successive anchors.

3 Parallel LAGAN Algorithm

Even though the serial LAGAN algorithm is very efficient, it still requires a significant amount of time while aligning long genomic sequences. Table 1 shows the amount of time required by the various stages of LAGAN for aligning pairs of sequences of length 1.5, 3, and 6 million base-pairs (Mbp). In Table 1, we report the time required by CHAOS to find the anchor points, the time required by the sparse LIS algorithm to construct the rough global alignment, the time taken by all recursive calls to fill in the large gaps in the alignments (REC), and the time taken by the final global alignment restriction using the limited area dynamic programming (DP), around the rough global alignment, along with the memory required.

We see that the CHAOS phase of the algorithm takes the most amount of time followed by the dynamic-programming based global alignment. The time required by CHAOS grows at a faster rate than the length of the sequences. Moreover, LAGAN's overall memory requirements also increases sub-

[1] A more detailed version of this paper is available at `http://www.cs.umn.edu/research/technical_reports.php?page=report&report_id=08-019`

Table 1: Runtime and memory requirements of the serial LAGAN algorithm.

Sequence	CHAOS	LIS	REC	DP	Memory
1.5Mbp	49s	0.44s	4.63s	26s	242MB
3.0Mbp	175s	1.70s	11.38s	52s	504MB
6.0Mbp	853s	9.00s	33.00s	101s	1.05GB

The sequence pairs generated were contigs from human DNA sequencing project. These runs were computed on an Intel Pentium 3.0 GHz 64-bit dual core processor with 2GB memory.

stantially. Within the LAGAN algorithm the memory complexity is determined by the CHAOS algorithm and corresponds to the number of high quality anchors generated and the memory required for the *T-trie* data structure. The runtime of the LIS algorithm is dependent on the number of anchors generated and provides interesting challenges for developing a parallel algorithm. The remaining stages (recursive anchoring and final global alignment) have lower memory requirements as they do not need to maintain all the high-quality local alignments discovered by CHAOS, and have straight forward parallel implementations [3].

3.1 Parallel CHAOS

Given a pair of sequences X and Y, our parallel CHAOS formulation decomposes the work among the p processors in one dimensional fashion. Specifically, sequence X is split into p equal length segments $(X_0, \ldots, X_{p-1})$ with segment X_i assigned to processor P_i, and sequence Y is distributed to each processor. Each processor P_i then identifies the anchor points between sequence Y and subsequence X_i. Each processor builds the T-trie data structure for sequence Y, and finds the seeds using the walking procedure for subsequence X_i.

This implementation leads to a row-wise decomposition of the *seed matrix* identification between sequences X (along the rows) and Y (along the columns), and as such decomposes the work only along one of the dimensions. This decomposition is done to align with our one-dimensional decomposition used for the parallel LIS algorithm, described in Section 3.2 and also shown in Figure 1.

This parallel formulation of CHAOS does not perform any inter-processor communication during seed identification and extension steps. As a result, the final set of anchors that it finds may be somewhat different from those found by the serial CHAOS algorithm. The CHAOS algorithm is a heuristic to quickly filter and reduce the search space. Even if some of the boundary anchors (across processors) are missed by our parallel implementation, we find these differences to be very small. As such, in our empirical evaluation we find that the final alignments produced by the parallel and serial LAGAN implementation are identical.

The differences in the generated anchors between the serial and parallel CHAOS versions are due to the following two reasons. Firstly, during seed discovery, each processor P_i attempts to find seeds using words or subsequences of length k. Hence, the algorithm does not find seeds involving the last $k-1$ positions of its segment of Y_i and sequence X. Since the length of each segment Y_i is very large, usually greater than 100Kbp, whereas k is less than or equal to 10, the number of potential seeds that are ignored is extremely small. Secondly, our algorithm does not extend the anchors beyond processor boundaries. In most cases, this will not result in the loss of any anchors, as the anchors will be split among the two processors and each processor will discover a segment of it. The split anchor will then be combined in the next step of the algorithm. The only times in which some alignment information will be lost is when the portion of the anchor that is on one processor is very small, and hence gets filtered out.

Note, as part of our research we also implement a parallel version of CHAOS that faithfully parallelizes the serial version of the CHAOS algorithm. However, due to its high communication cost we decided to use the above no communication model.

We also use a block cyclic one-dimensional decomposition. This is primarily done to align the computation with the parallel LIS algorithm (discussed in Section 3.2). This avoids any transfer time associated with the redistribution of the anchors generated by CHAOS to processors involved in the LIS phase of LAGAN. However, if the partition sizes controlled by the number of block cycles and processors is set too small, then the differences in the anchors generated between the serial and parallel CHAOS may lead to significant changes in the final resulting alignment. In our experimental evaluation, we do not see this to be a problem because the lengths of sequences and subsequences within each partition are greater than the size of the seed (k) by a fair margin.

3.1.1 Alternate Considerations

We also considered alternate approaches to the parallel implementation of CHAOS discussed above. We could have potentially decomposed the seed matrix using a two-dimensional decomposition. In this case, the processors will be arranged in a $\sqrt{p} \times \sqrt{p}$ two-dimensional grid, both X and Y will be split into $\sqrt{p}$ segments, and each processor $P_{i,j}$ will be responsible for finding the seeds involving the pair of segments X_i and Y_j. However, CHAOS uses a normalization step that requires information from all seeds in sequence Y for a particular position of sequence X, and can easily be found using the row-wise decomposition. Removing this restriction would have required a substantial re-engineering of CHAOS's heuristics that we decided not to pursue for the current work.

3.2 Parallel LIS

We developed a block cyclic one-dimensional partition based parallel formulation [3] for solving the serial sparse LIS problem.

There exists an ordered dependency in the serial LIS algorithm that limits the concurrency that can be achieved by the parallel LIS algorithm. We partition the computation along sequence X into pb rows. In this setting each processor P_i is responsible for computing entries for all rows $pq + i$ for $q = 0 \ldots b - 1$, where p and b denote the number of processors and cycles, respectively. The cyclic distribution eliminates idling time of processors, and ensures that most of the

processors are active throughout the execution of the algorithm. Such an approach leads to better load balancing and good utilization of the processors.

All the processors or row blocks contain the anchors generated across the entire length of the sequence X and subsequence from Y. These anchors are generated by the parallel CHAOS in the same row partition. The parallel CHAOS explained above is also run in a block cyclic fashion. The parallel LIS algorithm follows the computation in a diagonal fashion as shown in Figure 1. For each position i of X, certain maximal scoring elements may get updated with anchors having start/end indices equal to X_i as in the serial LIS algorithm. This information after updated is transferred to the next row block. The problem with pursuing such a parallel model implies that more time will be spent in communication or transfer of information rather than computation. This is because for every position of X or start/end points of anchors there will be transfer of information.

To ensure that the communication between processors is done at a coarse-grain fashion, each of the pb rows of the matrix are conceptually partitioned into pb blocks as well. The overall computation is performed into $2pb - 1$ steps, one for each diagonal of the $pb \times pb$ block-matrix, and communication between the processors occurs once each processor has finished processing all the blocks of the current diagonal. This leads to a theoretical speedup of $(pb \times pb)/(2pb - 1)$. The overall decomposition and computational flow is illustrated in Figure 1 for $p = 4$ and $b = 2$.

The chaining algorithm for the parallel LIS algorithm remains the same as the serial LIS algorithm. Each computation block updates the maximal scoring lists stored in the current processor, transfers information about anchors that could be potentially chained to anchors present in the next processor. The information transfer is from top to bottom. The chaining conditions for each processor remains the same as the serial LIS algorithm, where the start and end points of each anchors are checked before carrying out the update operations. Information about high scoring chains/anchors provided by preceding processor blocks allows the current processor block to extend chains across processor blocks. The final maximal scoring chain, and the maximal score is computed by the last processor block. Since, the parallel CHAOS follows the same row-wise (even block cyclic) decomposition as the parallel LIS algorithm no redistribution of anchors generated by CHAOS is needed.

3.3 Recursive Chaining and Anchoring

To compute the overall global alignment and find anchors and local alignments between the long gapped regions, we perform this step locally across each of the processors. Each processor locally determines new anchors within its segment to fill in the local gaps. The local alignments generated from these gaps by CHAOS are put together with the anchors from the previous step. Then a process of chaining the anchors across all the processors takes place to output the final rough global map. This process takes advantage of the fact that the starts of the local alignments generated within gaps must be

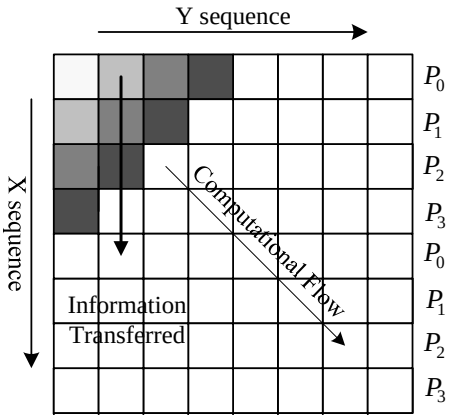

Figure 1: The cyclic decomposition and computational flow used by the parallel LIS algorithm.

greater than the end of the previous anchor present within another block. Hence the processor does not need to update previously generated anchors. The algorithm simply runs the serial chaining consecutively on the processor, passing information of the last anchor over to the next processor for chaining.

4 Results and Discussion

The performance of LAGAN is analyzed in terms of speedup and efficiency for different problem sizes (length of sequence pairs), number of processors, and number of blocks or cycles in cases of block-cyclic decomposition. We downloaded two contigs from human DNA sequencing project website for testing the program. Their sizes are $12.5Mbp$ and $13Mbp$. We generated shorter length sequences for testing, by truncating the sequences at lengths of $6Mbp$, $3Mbp$, and $1.5Mbp$.

All the empirical evaluations were performed on the IBM Bladecenter Linux Cluster at the Minnesota Supercomputing Institute, University of Minnesota. Each node of the cluster was a dual core 2.6GHz AMD Opteron processor with 2 GB of memory per processor.

4.1 Parallel CHAOS Performance

When the sequence size is doubled, the search space for the CHAOS algorithm increases four-fold and hence the run time for the CHAOS part on a single processor shows an approximate four fold increase.

Figure 2 shows the speedup for the CHAOS algorithm. We also plot the linear speedup line as reference. Using sequences of sizes $1.5Mbp$ each, we achieve speedups of 1.36, 2.3, 3.5, and 4.9 for 2, 4, 8, and 16 processors, respectively. For the $12Mbp$ sequences we achieve super linear speedup as seen in the Figure 2. The algorithm does not achieve linear speedups for the small sequences because of load imbalance and uneven distribution of work.

4.2 Parallel LIS Performance

Figure 3 presents the speedup achieved for the parallel LIS algorithm. When the number of cycles b are set to be one, we

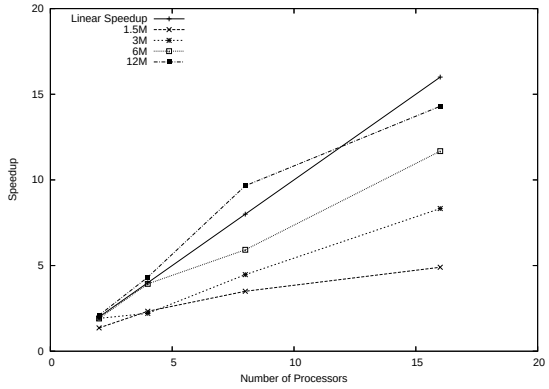

Figure 2: Speedup for Parallel CHAOS.

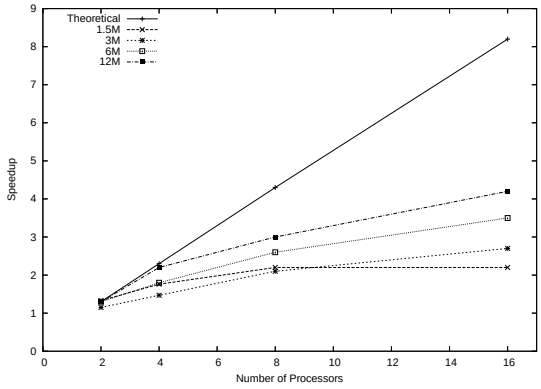

Figure 3: Speedup for Parallel LIS.

achieve a theoretical speedup of $p^2/(2p-1)$. This theoretical speedup is achieved provided the data is evenly distributed and all processors perform equal amount of work. However, in the experimental setting due to different anchors being handled by different processors, we cannot expect perfect load balancing. As the sequence size becomes larger the speedup approaches the theoretical speedup. This is largely because of increasing number of anchors and better load distribution for each processor.

4.2.1 Block-Cyclic Decomposition

To test the effectiveness of the block cyclic approach for the parallel LIS method we use the $1.5Mbp$ sized sequence pairs, but lower the threshold criterion for selection of seeds and anchors in the CHAOS algorithm to 15 rather than 20. This generates a larger number of anchors (4041358) to be chained by the LIS method. The run-time of the serial LIS program for this dataset is 7.23 seconds. Table 2 shows the speedups for varying number of processors and cycles for sequence pairs of sizes $1.5Mbp$ and $6.0Mbp$, respectively.

Parallel LIS algorithm allows the distribution of the local alignments among various processors with a limited amount of data transfer in between processors. We observe from Table 2 that the best speedups are obtained using two block cycles, and is dependent on the problem size. Increasing the number of cycles leads to an increase in the overall LIS runtime, due to increasing transfer times between the different processors at each step. Also, increasing the number of block cycles will cause the CHAOS algorithm to generate vastly different anchors.

Table 2: Parallel LIS Speedup with varying number of processors and cycles.

	1.5 Mbp				**6.0 Mbp**			
p/b	1	2	3	4	1	2	3	4
2	1.06	1.13	1.15	1.00	1.11	1.32	1.28	1.14
4	1.31	1.54	1.23	1.08	1.53	1.65	1.21	1.19
8	1.57	1.61	1.36	1.13	1.73	1.53	1.07	1.11
16	1.68	1.76	1.42	1.00	1.85	1.93	1.10	1.10

These speedup results are computed for the $1.5Mbp$ and $6.0Mbp$ sequences having 4041358 and 8988688 anchors generated by CHAOS using the parameters $\{k, c, F, EF\} = \{12, 0, 15, 0\}$, respectively.

5 Conclusions and Directions of Future Work

The MPI-LAGAN algorithm presented here has shown to be effective in reducing the run-times for both the CHAOS and LIS phases of the algorithm. The block-cyclic row-wise decomposition implemented in MPI-LAGAN reduced communication times and balanced loads between processors for achieving theoretical speedups. Though, our CHAOS algorithm generated slightly different anchors compared to the serial CHAOS algorithm, the final alignment generated was exactly the same as produced by the original LAGAN program. This was evaluated by matching the final alignment as well as the alignment scores of the original serial LAGAN and MPI-LAGAN.

Acknowledgements

This work was supported by IBM grant from IBM Life Sciences, Rochester, NSF EIA-9986042, ACI-0133464, IIS-0431135, NIH RLM008713A, the Digital Technology Center and the Minnesota Supercomputing Institute at the University of Minnesota.

References

[1] Michael Brudno, Michael Chapman, Berthold Gottgens, Serafim Batzogluo, and Burkhard Morgenstern. Fast and sensitive multiple alignment of large genomic sequences. *BMC Bioinformatics*, 4(66), 2003.

[2] Michael Brudno, Chuong B. Do, Gregory M. Cooper, Michael F. Kim, Eugene Davydov, NISC Comparative Sequencing Program, Eric D. Green, Arend Sidow, and Serafim Batzoglou. Lagan and multilagan: Efficient tools for large-scale multiple alignment of genomic dna. *Genome Research*, 13:721–731, 2003.

[3] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing: Design and Analysis of Algorithms, 2nd Edition*. Adison Wesley Publishing Company, Redwood City, CA, 2003.

[4] J. Kent. Blat–the blast-like alignment tool. *Genome Research*, 12(4):656–664, 2002.

[5] S. Kurtz, A. Phillippy, A. L. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. L. Salzberg. Versatile and open software for comparing large genomes. *Genome Biology*, 5(R12), 2004.