Optimal Distributed Multiple Sequence Alignment Using Conformal Computing Methods

1. Manal Helal, 2. Dr. Lenore R. Mullin, 1. Dr. Hossam El-Gindy, 1. Dr. Bruno Gaeta

1. Computer Science & Engineering Department, University of New South Wales, 2. Computer Science Department, University At Albany, New York

Abstract

The aim of this research is to investigate the potential of the Mathematics of Arrays (MoA) partitioning scheme in high dimensional scientific computational problems, such as Multiple Sequence Alignment (MSA) in Computational Biology. This work aims to deliver a unified partitioning scheme that works invariant of the dataset shape (dimension and lengths), and is portable among different high performance machines, cluster architectures, and potentially Grids.

Multiple sequence alignment (MSA) is a very common bioinformatics technique used in biological and medical research, to study the function, structure and evolution of genes and proteins. The algorithm for the optimal solution to the MSA problem is well-understood, but cannot be implemented even on high-performance computers since it cannot be easily distributed across multiple processors. We are redesigning the optimal MSA method to facilitate its deployment on supercomputers. This will allow high-performance and distributed computing platforms, which are becoming more prevalent in biological research, to be harnessed for the calculation of reference alignments for genes and protein sequences, and also for the identification of sequence regions in common in a group of sequences (multiple local sequence alignment).

1. Dynamic Programming MSA

MSA is solved optimally using the dynamic programming method. It is proven mathematically to produce the optimal global alignment using the Needlman and Wunch algorithm, and for local alignment using the Smith and Waterman algorithm. The idea, as described in [Gusfield 1997], is to start from the ends of both sequences and attempt to match all possible pairs of characters by following a scoring scheme for matches, mismatches and gaps, generating a matrix of numbers that represent all possible alignments. The optimal alignment can be found by tracing back, starting from the highest score on the bottom edges, and following the highest scores on the matrix. In the global alignment the recurrence used to fill in the scoring matrix is:

$$S_{ij} = MAX \begin{cases} S_{i-1,j-1} + sub(a_i, b_j) \\ S_{i-1,j} + g \\ S_{i,j-1} + g \end{cases}$$

where S is the scores matrix, a and b are the pairs being compared corresponding to the i^th and j^th position. in the matrix, and sub is the scoring function that reads the value from the scoring matrix used, and g is the gab penalty value.

Using the Dynamic Programming algorithm described above to align more than two sequences will require computational steps and memory space that is exponential with the number of sequences to be analysed. This creates a dimensionality problem, as the neighbors to be checked in the recurrence will grow $O(2^k-1)$, where k is the number of sequences or the dimensions, and makes the algorithm applicable only to a limited number of sequences. Filling a tensor of alignment scoring values will provide the alignment of combination of the sequences, and the internal values will be the alignment of all sequences together, without any bias to the order of the sequences.

Complexity Analysis

Given two sequences of lengths n and m, the matrix initialization executes in O(n+m), where n is the size of the first sequence, m is the size of the second sequence. Then, filling in rest of matrix using the recurrence executes in O(nm). The traceback executes in O(n+m). If sequences are same length, total time would be O(n^2). Dynamic programming is efficient since there are: $2n!/(n!)^2 = O(2^{2n})$ possible alignments. However, as we add more sequences it becomes exponential in data size as O (n^n), assuming n is the average length of all sequences.

The only way to decompose the complexity, is to distribute on HPCs or computer clusters. This direction will create two challenges, dependency problem and finding a suitable partitioning method. Mathematics of arrays provided in conformal computing methods were found to be a candidate solution, and hence investigated further.

The solution needed need to work invariant of the number of sequences used. This is to avoid rebuilding the program for every new dataset. This means the retrieval of neighbors function need to be scalable and is defined as index transformation, and not static. The assignments of temporary scores need to be generalized and aware of how many gap scores to add, based on the relative position of the neighbor to the current cell being computed, i.e. how many dimensions will decremented to retrieve this neighbor.

The partitioning method needed, require that elements in each part need to be aware of their positions in the global whole tensor at any time, and all neighbors locations can be identified, whether are local, remote, or border elements and can be initialized. This means, we need index mapping between whole and parts at any time.

2. Conformal Computing Methods

Conformal Computing as described in [Mullin / Raynolds - 2006] is a formalism based on an algebra of abstract data structures, A Mathematics of Arrays (MoA) and an array indexing calculus, the Psi-Calculus. The method allows the composition of a sequence of algebraic manipulations in terms of array shapes and abstract indexing. The approach works invariant of dimension and shape, and allows for partitioning an N-Dimension tensor based on a given MoA function. It is called conformal computing because the mathematics used to describe the problem is the same as that used to describe the details of the hardware. Thus at the end of a derivation the resulting final expression can simply be translated into portable, efficient code for implementation in hardware and/or software. MoA offers a set of constructs that help represent multidimensional arrays in memory in a linear concise and efficient way, with lots of useful properties, and applications. For a full listing of the MoA constructs, please refer to [Mullin–88].

MoA Example:

3D Tensor saves in memory as <1 2 3 .... 36>

$$\xi = \left(\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 & 17 & 18 \end{pmatrix} \quad \begin{pmatrix} 19 & 20 & 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 & 29 & 30 \\ 31 & 32 & 33 & 34 & 35 & 36 \end{pmatrix} \right)$$

of shape vector $\rho\xi = \langle 2\ 3\ 6 \rangle$

Psi-ψ Indexing function works as partitioning with partial indices, and for elements retrievals with full index.

$$\langle 1 \rangle\ \psi\ \xi = \begin{pmatrix} 19 & 20 & 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 & 29 & 30 \\ 31 & 32 & 33 & 34 & 35 & 36 \end{pmatrix}$$

$$\langle 0\ 2 \rangle\ \psi\ \xi = \langle 13\ 14\ 15\ 16\ 17\ 18 \rangle$$

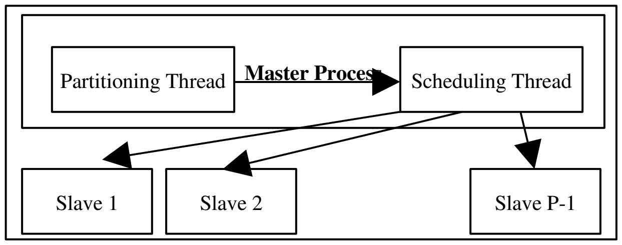
$$\langle 0\ 1\ 3 \rangle\ \psi\ \xi = 10$$

$$+\text{red} \langle 1\ 1 \rangle \text{ drop } (\langle 1 \rangle \text{ take } \xi) = \langle 22\ 24\ 26\ 28\ 30 \rangle$$

this function takes 1 from the first dimension in ξ, like <0> $\psi\ \xi$ above, then drops one row, and one column, then reduce the remaining by adding them on the first dimension.

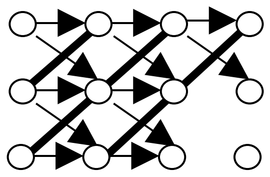
3. Solution Abstraction

The solution proposed is to redesign the dynamic programming algorithm using the MoA to generalize for K-Dimension, and to distribute the processing on HPC or computers cluster. A master process need to be created for the partitioning, dependency analysis, and scheduling over processors, managing the trace back processors over the distributed partitions. The rest of the available processors work as slave processes, receive partitions, and score them, receive dependency requirements, and trace back through the partitions. The master process has a partitioning thread, a dependency analysis thread, and a sending thread. The slave processes, contain a score computation thread, receiving thread that buffers all received packets from master or from other slaves, and a sending thread to send dependency to waiting slaves processors.



4. Dependency Analysis

To be able to parallelize the score computation, the dependency between the elements (cells) scoring need to be understood to communicate the required scores between processors. As analyzed in [Yap -1995] and [Chen-Schmidt 2005], the dependency to score each element in the scoring matrix for pair wise alignment, is based on retrieving the calculated score for the top, left, and left-up diagonal, creating a wave-front communication pattern as shown in figure.



So, if every processor takes a row, all can initialize the first element, and the first processor once finishes the second element, the second processor can do the second, and so forth. This will make parallelism increase to the middle diagonal, and then decrease as it approached the end of the scoring matrix.

In an attempt to generalize on that, to retrieve the dependency invariant of dimension, it will be the lower border cells generally. For K-dimensions, these are 2^k-1 cells. Using the MoA constructs, neighbors are retrieved by decrementing the multidimensional index in all possible combinations. For example, a 2D scoring matrix:

$$\begin{pmatrix} S_{0,0} & S_{1,0} & S_{2,0} & S_{3,0} \\ S_{0,1} & S_{1,1} & S_{2,1} & S_{3,1} \\ S_{0,2} & S_{1,2} & S_{2,2} & S_{3,2} \\ S_{0,3} & S_{1,3} & S_{2,3} & S_{3,3} \\ S_{0,4} & S_{1,4} & S_{2,4} & S_{3,4} \end{pmatrix}$$

neighbors for cell $S_{2,4}$ having multidimensional index vector as (2 4) are: $S_{1,3}$, $S_{2,3}$, $S_{1,4}$. and with MoA can be retrieved as:

(2 2) take ((-1)+(2 4)) drop S)

This is a nested function, where the drop section gets executed, and on the results, the take function gets executed. It drops the other lower indexed cells that are not of interest, by subtracting one from the current cell index to drop, and takes only 2 cells of each dimension to return the direct neighbors only. This will return a matrix with the points:

$$\begin{pmatrix} S_{1,3} & S_{2,3} \\ S_{1,4} & S_{2,4} \end{pmatrix}$$

generalizing on that to K-Dimension neighboring function, it becomes:

$<2_0\ 2_1\ 2_2 \ldots\ 2_k>$ Take (((-1)+$<i_0\ i_1\ i_2\ i_3 \ldots\ i_k>$) Drop S)

this will retrieve the elements required to compute the cell at the index represented by the i-vector above. We call this get lower border MoA function.

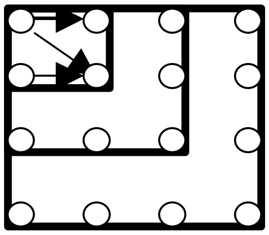
5. Partitioning Scheme

Having the dependency invariant of dimension & shape understood, we can follow the same scheme to partition the alignment tensor in order to maximize parallelism, in wave-front pattern. The MoA function created above, can be used iteratively, in a breadth-first traversal fashion, starting from i-vector containing zeros, for the first cell in the tensor, then on each retrieved partition, all higher order neighbors partitions can be retrieved to create the next diagonal wave. The first wave will be one partition starting at the zero-cell, and ending at $<p_0\ p_1\ p_2\ p_3 \ldots\ p_k>$ where p is the partitioning size chosen. Then at each higher border corner cell this partition, the get higher border function is called to retrieve the next wave partitions, This is function is based on the following equation:

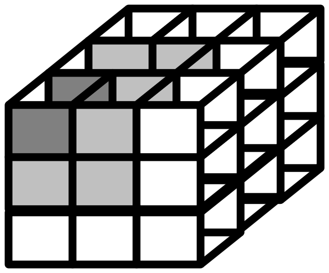
$<p_0\ p_1\ p_2\ p_3 \ldots\ p_k>$ Take (((+1)+$<i_0\ i_1\ i_2\ i_3 \ldots\ i_k>$) Drop S)

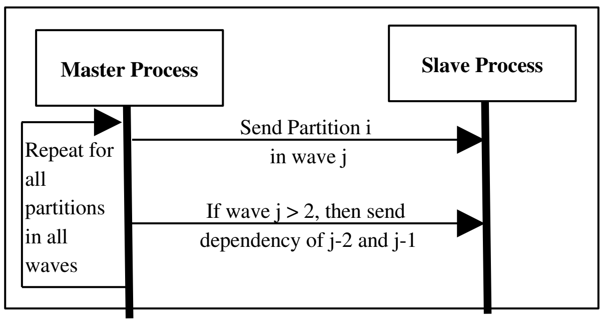
this means, we drop the higher indexed cells by adding one to the current cell index, then taking a partition of size p from the remaining tensor. We loop through all partitions created at one wave, for all their higher border corners, and create the partitions for the next wave, and on the next wave we do the same, till we reach the last higher order cell in the tensor.

In 2-D MSA dependency takes the form of small squares around the previously finished wave. The dependency changes to:



This makes the parallelism, keep increasing from one wave to another. In 3-D MSA, dependency takes the shape of enclosed cubes, with inner cubes being scored before the outer ones, like the first dark gray cube is scored first in one wave, and next wave contains the 2^k-1 neighboring cubes, colored in light gray, and then the white wave of cubes, and then later waves will contain remaining partitions minus the ones partitioned before (were neighbors to other partitions that was traversed before). The overlapping edge cells in each partition need to be communicated between processors.




Master Process
Slave Process
Repeat for all partitions in all waves
Send Partition i in wave j
If wave j > 2, then send dependency of j-2 and j-1

6. Distributed Scoring

The dynamic programming recurrence described above, is now generalized for K-dimension and arbitrary sequence lengths (shape), using the following recurrence:

$$S(i_1, i_2, i_3, \ldots, i_k) = \max \begin{pmatrix} G_1 + T_S(G_1) \\ G_2 + T_S(G_2) \\ : \\ G_{2^k-1} + TS(G_{2^k-1}) \end{pmatrix}$$

where:

TS ($_{Gi}$) = (sub(dj, dk) for each pair j, k in G) +(gS * (K-D))

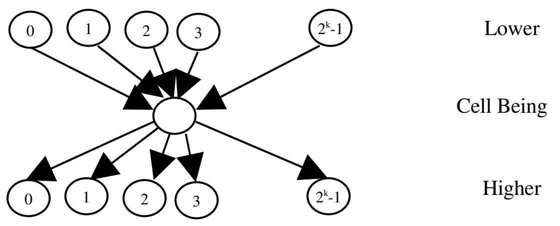
G_i: Neighbour i of current cell, up to 2^k-1 neighbours

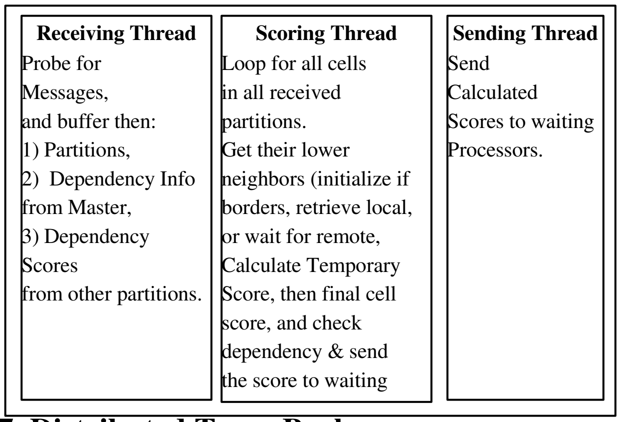
D: No of decremented indices to get this particular neighbour

TS: Temporary Score function assigned to each neighbour based on how many multidimensional indices were decremented to get to this neighbour

gS: gap Score Value * (K-D): multiply the gapScore Value with number of indices that remained the same (wasn't decremented to get this neighbour), retrieved by Total Dimensions K (Sequences) – D.

We iterate through the partition received by each processor, at each cell, we retrieve the lower border neighboring cells scores, used the function described above. These neighbors might be local (in the same partition, or another partition computed by the same processor), remote (in another processor), or a lower border cell on the whole unpartitioned scoring tensor. The first case, we retrieve the score, and compute TS based on hoe many indices got decremented in the multidimensional index to retrieve this neighbor. If remote, the processor computation thread, waits to receive the score from the remote processor, if lower border cell in the whole tensor, the score gets initialized to the gap score used multiplied by the values in the multi-dimensional index of the cell. The figure shows the 2^k-1 lower border cell neighbors that are required to score a cell, and after scoring this cell, there are 2^k-1 cells that can retrieve one of their required scores. Both lower indexed neighbors cells and higher indexed neighbors, can be local or remote.



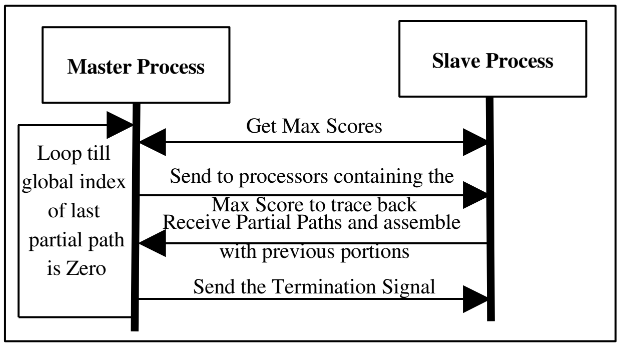

Receiving Thread
Probe for
Messages,
and buffer then:
1) Partitions,
2) Dependency Info
from Master,
3) Dependency
Scores
from other partitions.

Scoring Thread
Loop for all cells
in all received
partitions.
Get their lower
neighbors (initialize if
borders, retrieve local,
or wait for remote,
Calculate Temporary
Score, then final cell
score, and check
dependency & send
the score to waiting

Sending Thread
Send
Calculated
Scores to waiting
Processors.

7. Distributed Trace Back

After having the partitions fully scored in stored in each slave processors local disk space, the distributed trace back program starts. Again a master/slave approach is followed. The master processes retrieves the higher scoring higher border cell from all higher edge partitions in all processors, and send to the processor with the highest score to trace back through its partition. If the trace back done by the slave reaches the lower border edge of this partition, it checks if the next required partition is local to it, top resume on it, and then resume, till no more local partitions, it reports to the master process with that cell index, and the path partition found so far, and the master send to the next processor to resume the trace back and repeat the same process. The process iterates like that, till no more partitions in all processors, the master assemble all received partitions, and form the optimal full path and reports it.


Master Process
Slave Process
Loop till global index of last partial path is Zero
Get Max Scores
Send to processors containing the Max Score to trace back
Receive Partial Paths and assemble with previous portions
Send the Termination Signal

8. Scheduling Scheme

Three methods of scheduling are considered, each with positives and negatives. The first 2 are already implemented, and the third is in progress. First is the bag of tasks method. It is most suitable for heterogeneous systems, where each computing node differ in its computing power. The second method is round robin. It is currently used, because of the availability of clusters of homogeneous computing nodes. The third method is dependency based scheduling. It is optimized to increase locality and decrease data communications. Bag of tasks scheduling is based on adding processors to a queue, and pop up to be assigned, and after it finishes computation, it returns to the scheduler, and push it self to the queue again, to receive another assignment. The advantage of this method is that each processor can finish in its own time. The disadvantage is that scheduler might remain idle, waiting for processors to come back from an initial assignment in a previous wave. Round robin scheduling is based on getting the scheduler to finish partitioning all waves uniformly to all available processors, and send all partitions and the dependency, and once done, can serve as one slave itself, to avoid idleness. The advantage is that master process will be better optimized. However, the disadvantage is that there is no consideration for dependency & locality of data among the processors.

Dependency based scheduling optimizes the assignments to processors to increase dependency locality, to reduce communication time, and idleness due to waiting to receive required resources. The advantage is less communication overhead, and more data locality. Again, the disadvantage is the preprocessing overhead, to calculate the best assignment based on dependency.

9. Results

We started testing by hand written small data sets, using different machines. Then increased the sequences incrementally till we reached reference 1 in Balibase. Balibase is a hand-written reference sequence alignment as desired by biologists. It contains 142 reference alignments with over 1000 sequences. Of the 200,000 residues in the database, 58% are defined within the core blocks. The remaining 42% are in ambiguous regions, which cannot be reliably aligned. There are four hierarchical reference sets. Reference 1 provided the basis for construction of the following sets. Each of the main sets may be sub-divided into smaller groups, based on sequence length and percent similarity. [Balibase Website].

On a Single machine (Intel Pentium M Processor 740 – 1.73 Ghz, 1 GB RAM, 70 GB HDD), with simulated distributed processes, using mpich library version 2-1.0.4-rc1, the following table lists the results retrieved so far, where P is the number of processes used, K is the number of sequences aligned, and their lengths, and the CPU time in minutes:seconds format, and the Memory used in Mib.

P	**K**	**Lengths**	**CPU**	**Mem**
3	3	4, 3, 2	00:00.12	7.46
4	3	4, 3, 2	00:00.12	8.39
4	4	5,4,3,2	00:00.16	10.26
5	4	5,4,3,2	00:00.12	13.05
4	5	6, 5,4,3,2	00:00.12	10.26
4	6	7, 6, 5,4,3,2	00:01.13	27.97
3	3	7,8,9	00:00.34	7.46
3	3	90,80,85	00:00.37	

10. Conclusion

This method does not reduce the complexity of the problem; it is still growing exponentially with the data size. However, it provides a method to compute MSA invariant of dimension and shape, and divides the complexity into chunks that can be distributed over processors. This method provides automatic load balancing among processors, and better locality inside each single processor. The more powerful the machines used, the higher the upper-bound of the input data size. Further work on high dimensional scientific computation problems can benefit from these methods. Heuristics and further optimization can be applied to this implementation of the multiple sequence alignment, to reduce the search space to suit less powerful computing platforms.

Currently, the work is focused on optimizing the communication and computation costs, but enhancing the dependency based scheduling. More work can be done to reduce the search space without loosing optimality. Also, more optimal paths, or sub-optimal paths can be returned without much penalty. Moreover, distributed local alignments can be achieved as well, with minor changes. Currently we have only local alignments on the sequential MoA MSA implementation.

Portability is achieved by avoiding use of any proprietary libraries. Currently, standard C, and standard functions in the MPI standards are being used. The MoA library implemented in standard C, and can be reasily recompiled on any machine as required.

Further portability enhancement, would be to model the processors as an extra dimension to the alignment scoring tensor, and partition, by reshaping the tensor to divide itself over the processors automatically. This all for a simple one dimensional array of processors. For a hypercube or other topology of processors, the processors can be defined as another MoA tensor, and using the PSI correspondence theorem as described in [Mullin 1988], correspondence between the scoring tensor elements and processor elements, can be established to achieve the best scheduling required.

11. References

Manal Helal, “Mathematics of Arrays – The implementation and the application”, A Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science, Department of Computing Science, American University in Cairo, Fall 2001.

Lenore M. Mullin, “A Mathematics of Arrays”, Doctor of Philosophy Dissertation in Computer and

Information Science Completed at Syracuse University, Syracuse, NJ, December 1988.

Lenore M. Mullin, James E. Raynolds, “Density Matrix operations without matrix multiplication: Conformal Computing Techniques illustrated with a Quantum Computing example”, Under Review International Journal of Quantum Information (IJQI), 2006.

Dan Gusfield, “Algorithms on Strings, Trees, and Sequences”, Cambridge University Press, 1997

Chunxi Chen, Bertil Schmidt, “An Adaptive grid implementation of DNA sequence alignment”, Source, Future Generation Computer Systems archive Volume 21 , Issue 7, July 2005, pp: 988 - 1003 .

Tieng Kim Yap, “Parallel Computation In Biological Sequence Analysis”, A Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, Department of Computing Science, George Mason University, Spring 1995.