Optimal Distributed Multiple Sequence Alignment Using Conformal Computing Methods

Author:
Helal, Manal; El-Gindy, Hossam; Gaeta, Bruno; Mullin, Lenore

Publication details:
Proceedings of the International Conference on High Performance Computing, Networking and Communication Systems (HPCNCS- 07
pp. 120-127
978-0-9727412-5-5 (ISBN)

Event details:
International Conference on High Performance Computing, Networking and Communication Systems (HPCNCS- 07)
Orlando, USA

Publication Date:
2007

DOI:
https://doi.org/10.26190/unsworks/368

Optimal Distributed Multiple Sequence Alignment Using Conformal Computing Methods

Abstract

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) The exponential growth in time and memory requirements were found to be compensated by exponential parallelism, using the proposed partitioning scheme, and optimizing the communication cost..

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 Needleman and Wunch algorithm, and for local alignment using the Smith and Waterman algorithm. The idea, as described in [Gusfield 1997] for 2 sequences, 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 ith and jth position in the matrix, and sub is the scoring function that reads the value from the scoring matrix used, and g is the gap penalty value.

Using the Dynamic Programming algorithm described above to align more than two sequences will require computational steps and memory space that increases exponentially with the number of sequences to be analyzed. 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 the rest of the matrix using the recurrence executes in O (nm). The trace back executes in O (n+m). If sequences have the 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: management of dependencies and selection of a suitable partitioning method. A candidate solution is provided by Mathematics of Arrays (MoA) provided in Conformal Computing methods.

The solution needs to work invariant of the number of sequences used to avoid rebuilding the program for every new dataset. This means the retrieval of neighbors function needs to be scalable and not static and is defined as an index transformation. 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 need to be decremented to retrieve this neighbor.

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

2. Conformal Computing Methods

Conformal Computing [1] as described in [Mullin / Raynolds - 2005] 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-dimensional 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 many 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 $\Psi\rho\ \xi \uparrow\downarrow = <2\ 3\ 6>$

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

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

$<0\ 2> \Psi \xi = <13\ 14\ 15\ 16\ 17\ 18>$

$<0\ 1\ 3> \Psi \xi = 10$

$+red <1\ 1> \downarrow (<1> \uparrow \xi) = <22\ 24\ 26\ 28\ 30>$

Figure 1: *MOA Example*

Figure 1 describes an example of how to use MoA constructs, and its representation in memory. The nested function in the bottom, takes (↑) 1 from the first dimension like in <0> Ψ ξ as shown 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 needs to be created for partitioning, dependency analysis, and scheduling over processors, and 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, a receiving thread that buffers all received packets from the master or from other slaves, and a sending thread to send dependency to the waiting slaves' processors.

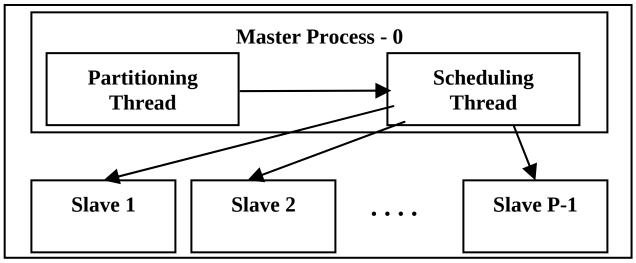


Figure 2: Solution Abstract Design

4. Dependency Analysis

To be able to parallelize the score computation, the dependency between the scoring of elements (cells) needs to be understood to communicate the required scores

---

[1] The name Conformal Computing © is protected. Copyright 2003, the Research Foundation of State University of New York, University at Albany.

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 3.

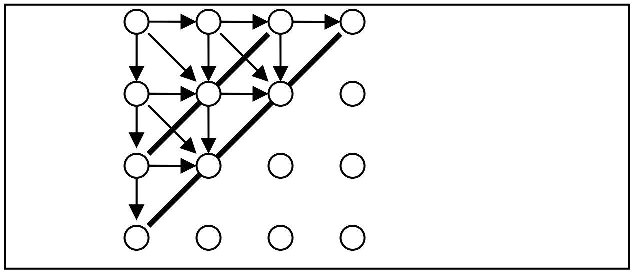


Figure 3: *MSA Pair Wise Traditional Dependency*

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

Generalizing the problem to multiple dimensions requires retrieving the dependency invariant of dimension. In K-dimensions, each internal cell has 2^k-1 lower border 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:

$$(2\ 2\ ) \uparrow ((-1)+(2\ 4))\ \downarrow S)$$

This is a nested function, where the drop section gets executed, and the take function gets executed on the results. This function 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 to K-Dimension, the neighboring function becomes:

$$<2_0\ 2_1\ 2_2 ...\ 2_k> \uparrow (((-1)+<i_0\ \ i_1\ i_2\ i_3\ ...\ i_k>) \downarrow S)$$

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

5. Partitioning Scheme

Having understood the dependency invariant of dimension and shape, we can follow the same scheme to partition the alignment tensor to maximize parallelism, in a 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 neighboring 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 $< p_0\ \ p_1\ p_2\ p_3\ ...\ p_k >$, where p is the partitioning size chosen. Then at each higher border corner cell of this partition, the get higher border function is called to retrieve the next neighboring partition from this corner, and adding them to the next wave. This traversal method is based on the following generalized MoA equation:

$$<p_0\ \ p_1\ p_2\ p_3\ ...\ p_k> \uparrow (((+1)+<i_0\ \ i_1\ i_2\ i_3\ ...\ i_k>) \downarrow S)$$

That is, 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 start with the cell at zero index, and get its higher neighboring partitions for the next wave, and then for all partitions in the next wave, we get all higher border partitions for the following wave, creating breadth-first traversal method till the whole tensor is covered. Figure 4 shows the communication pattern between the respective threads in both master and slave processes responsible for the partitioning.

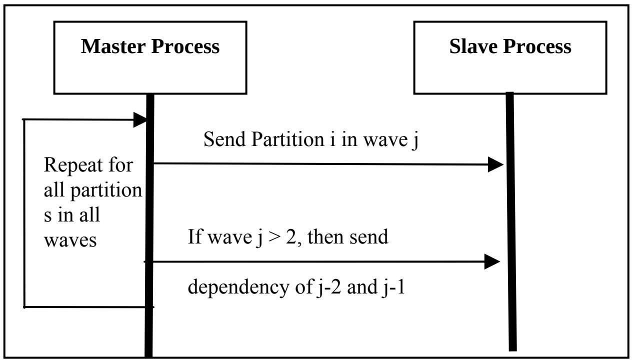

Master Process
Slave Process
Send Partition i in wave j
Repeat for all partition s in all waves
If wave j > 2, then send
dependency of j-2 and j-1

Figure 4: *Partitioning Thread in Master & Receiving Thread in Slave*

In 2-D MSA dependency takes the form of small squares around the previously finished wave. The dependency changes as shown in figure 5.

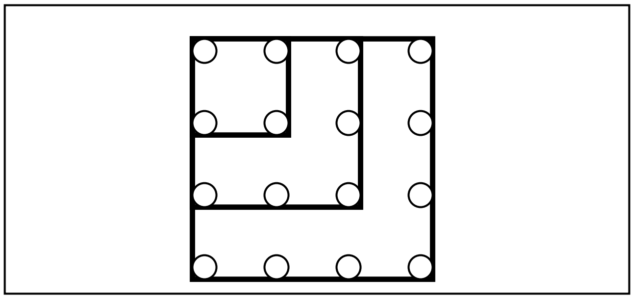


Figure 5: *MoA 2D MSA Waves Partitions*

This makes the parallelism increase from one wave to another, and not dependent on a fixed dimension distribution. In 3-D MSA, dependency takes the shape of enclosed cubes, with inner cubes being scored before the outer ones. As shown in figure 5, 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. Later waves will contain higher neighbors partitions of the partitions in the previous wave, minus the ones previously partitioned (neighbors to other partitions that were traversed before). The overlapping edge cells in each partition need to be communicated between processors.

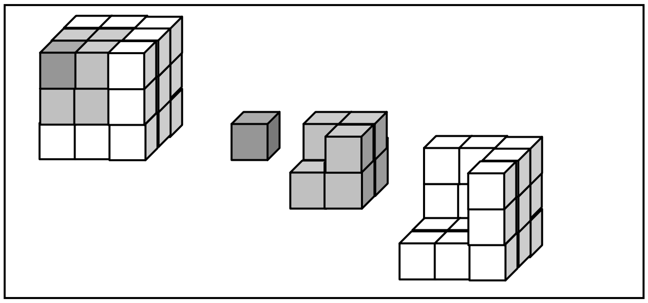


Figure 6: *3D MoA MSA Waves Partitions for shape <3 3 3>*

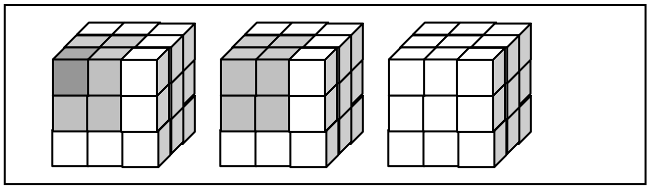


Figure 7: *4D MoA MSA Waves Partitions for shape <3 3 3 3>*

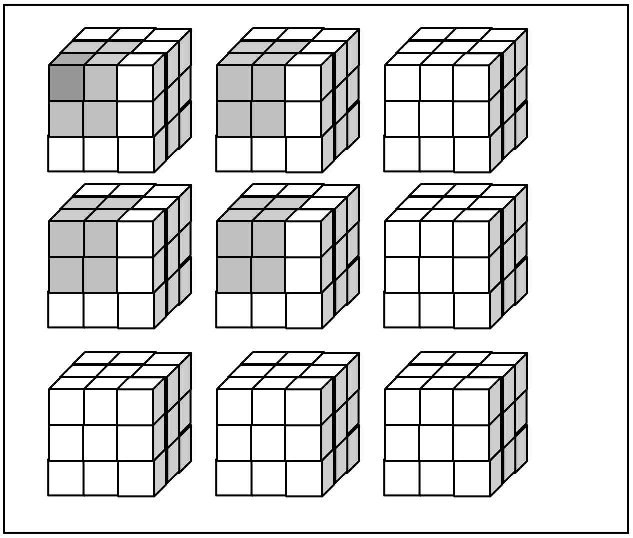


Figure 8: *5D MoA MSA Waves Partitions for shape <3 3 3 3 3>*

As shown in figures 6, 7 and 8, the number of partitions that can be scored at one wave increase exponentially with the increase in dimension. However, the communication dependency between the partitions increases as well, and optimization on the communication vs. computation is required on the choice of the partition size. Similarly distribution over processors and achieving data locality as much as possible will affect the performance significantly.

6. Distributed Scoring

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

$$S(i_0\ \ i_1\ i_2\ i_3 \dots i_k) = \max \quad \begin{array}{l} G_1 + TS\ (G_1) \\ G_2 + TS\ (G_2) \\ : \\ G_2k_{-1} + TS\ (G_2k_{-1}) \end{array}$$

Where:
$TS\ (_{Gi}) = (sub(d_j, d_k)$ for each pair j, k in G) +($gS * (K-D)$)

G_i: Neighbor i of current cell, up to 2^k-1 neighbors
D: No of decremented indices to get this particular neighbor
TS: Temporary Score function assigned to each neighbor based on how many multidimensional indices were decremented to get to this neighbor
gS: gap Score Value * (K-D): multiply the gap Score Value with number of indices that remained the same (were not decremented to get this neighbor), retrieved by Total Dimensions K (Sequences) – D.

We iterate through the partitions received by each processor. At each cell, and retrieve the lower border neighboring cells scores, using the function described above. These neighbors might be local (in the same partition, or in another partition computed by the same processor), or remote (in another processor), or a lower border cell on the whole un-partitioned scoring tensor. In the first two cases, we retrieve the score, and compute TS based on how many indices got decremented in the multidimensional index to retrieve this neighbor. If the neighbor is remote, the processor computation thread waits to receive the score from the remote processor. If the neighbor is a 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. Figure 9 shows the 2^k-1 lower border cell neighbors that are required to score a cell. After scoring this cell, another 2^k-1 cells can retrieve one of their required scores. Both lower indexed neighbor's cells and higher, can be local or remote.

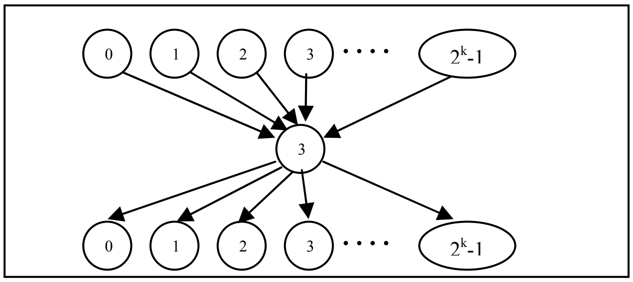
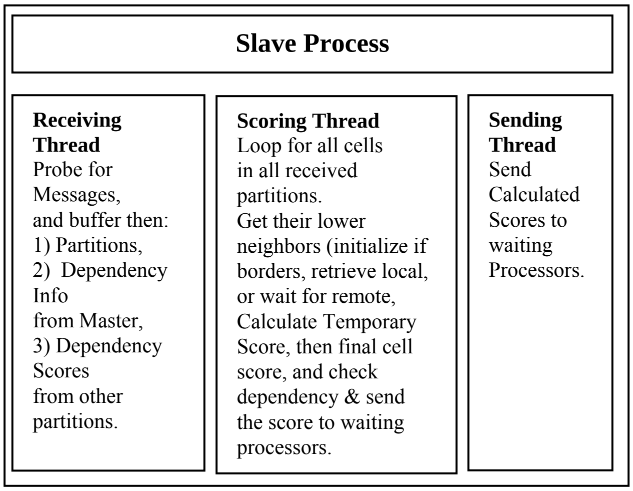


Figure 9: *ND MoA MSA Dependency*


Slave Process

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 processors.

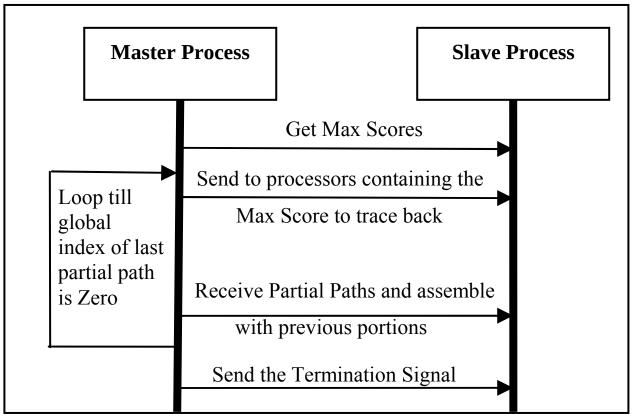
Sending Thread
Send Calculated Scores to waiting Processors.

Figure 10: *Slave Process Threads*

Figure 10 shows the design of the slave process, with its main threads and functionality.

7. Distributed Trace Back

Once the partitions have been fully scored and scores stored in each slave processor's local disk space, the distributed trace back program starts. Again a master/slave approach is followed. The master process retrieves the highest scoring higher border cell from all higher edge partitions in all processors, and sends to the processor with the highest score to trace back through its partitions. If the trace back done by the slave reaches the lower border edge of the current partition it is working on, it checks if the next adjacent partition was previously scored by the same processor, and available in its local memory. If so, it loads the adjacent partition and resumes the trace back from it. This process continues, till the next adjacent partition is not local. Then, the slave process reports to the master process with the last cell index, the partial path found so far among all its adjacent local partitions, and which processor contain the adjacent required partition. The master sends to the next processor containing the last cell index reported from the previous processor, to resume the trace back and repeat the same process. The process iterates like that until there are no more partitions in any of the processors. The master then assembles all received partitions, forms the optimal full path and reports it. Figure 11 illustrates the distributed trace back process.



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

Figure 11: *Distributed Trace Back Design*

8. Scheduling Scheme

Since we are not following a fixed dimension distribution scheme, there is no fixed row or column distribution method as was described before in literature for the pair-wise MSA. Three methods of scheduling are considered, each with positives and negatives. The first two 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 differs 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, which is optimized to increase locality and decrease data communications. Bag of tasks scheduling is based on adding processors to a queue using push and pop. Starting with a queue containing all slave processors, a processor is retrieved to be assigned, and after it finishes computation, it returns to the scheduler, to receive another assignment. The advantage of this method is that each processor can finish in its own time. The disadvantage is that the 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 by sending all partitions and their dependency. Once done, the scheduler can serve as slave itself, to avoid idleness. The advantage is that the master process will be better optimized. However, the disadvantage is that there is no consideration for dependency and 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

Initial testing was carried out using small hand written data sets, using different machines. Test sequence numbers and lengths were increased incrementally until it reached that of reference 1 in Balibase. Balibase is a database of hand-written reference sequence alignments 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].

Initial experiments were carried out on a single processor machine (Intel Pentium M Processor 740 – 1.73 Ghz, 1 GB RAM, 70 GB HDD), with simulated distributed processes using the mpich library version 2-1.0.4-rc1. Then, scalability testing was carried out on an SGI Altix 3700 Bx2 cluster with 1928 1.6Ghz Itanium2 processors L1 cache, 16 Kbytes (D) + 16 Kbytes (I). Cache line 64bytes, L2 cache: 256 Kbytes. Cache line 128 bytes, L3 cache: 6 Mbytes. Cache line 128 bytes, Total memory is 5.6Tbytes, interconnect with 3.2 GBytes/s bidirectional bandwidth per link and < 2us MPI latency, and 30 TBytes of global storage. The operating system is based on SUSE SLES9 Linux with an enhanced Linux 2.6 kernel and SGI, Total peak speed of over 11Tflops. It is ranked 26th in June 2005 Top500 list. The distributed messaging library in SGI is Intel MPI Library.[2]

Initial results are summarized in table 1. M1 is the single machine, and M2 is the SGI Altix as described above. The testing is continuing to identify the optimization chances based on varying the partitioning size, number of processors, and better scheduling techniques. Partition size of 3 is used on all tests, except the last, a partition size of 20 was used.

P	K	L	M1 CPU	M2 CPU	M2 E-Time	P Mem	V Mem
3	3	4, 3, 2	00:00.12	00:00:00	00:00:03	15	122
3	3	7, 8, 9	00:03:77	00:00:03	00:00:07	48	281
4	5	6, 5,4,3,2	01:00.15	00:00:08	00:00:05	62	355
4	6	7, 6, 5,4,3,2	01:19.36	00:00:10	00:00:05	76	429
3	3	90,80, 85	39:30.34	02:09:43	00:44:25	371	606

Table 1: *Initial Results: column "P" is the number of processes created, "K" is the number of sequences aligned, and "L" is their lengths, and the "M1 CPU" is the CPU time in first machine in minutes:seconds format, then "M2 CPU"& M2 E-Time are CPU & Elapsed Time in second machine, and the"P Mem" and "V Mem" are Physical and Virtual Memory used in Mega Bytes in both machines.*

10. Conclusion

---

[2] This work was supported by an award under the Merit Allocation Scheme on the National Facility of the Australian Partnership for Advanced Computing.
(http://nf.apac.edu.au/facilities/ac/hardware.php)

We have applied conformal computing methods in order to parallelize the alignment of multiple sequences. The method does not reduce the complexity of the problem, which is still growing exponentially with the data size. However, conformal computing provides a method for computing MSA invariant of dimension and shape, and dividing the complexity into chunks that can be distributed over processors. The scalability of the parallelism is found to be growing exponentially as well. Our approach 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. 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. Other high dimensional scientific computation problems can also benefit from these methods

Currently, the work is focused on optimizing the communication and computation costs by enhancing the dependency based scheduling. Future work includes the reduction of search space without loosing optimality. Also, the program can be modified to return more than one optimal paths, or sub-optimal paths. This can be achieved without much penalty as it impacts only the trace back process, which is the least computationally demanding. Moreover, the program can be modified to generate distributed local (rather than global) alignments, which would only require minor changes.

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 is implemented in standard C and can be easily 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. Our presentation assumes 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], the correspondence between the scoring tensor elements and processor elements can be established to achieve the best partitioning and scheduling required. Further optimization on the memory hierarchy levels can be achieved on each processor, by mapping the memory hierarchies as an extra dimension, and partition required elements to be in the fastest memory level, in order to avoid frequent context-switching.

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.

J. Raynolds and L. Mullin, “Applications of conformal computing Techniques to Problems in computational physics: the FFT”, Computer Physics communications 170(2005)1-10, 2005

L. Mullin, “A Uniform way of reasoning about array based computation in radar”, Digital Signal Processing Elsevier Publishers, September 2005

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.