Cloud Technologies for Bioinformatics Applications

Jaliya Ekanayake, Thilina Gunarathne, and Judy Qiu

Abstract—Executing large number of independent jobs or jobs comprising of large number of tasks that perform minimal inter-task communication is a common requirement in many domains. Various technologies ranging from classic job schedulers to latest cloud technologies such as MapReduce can be used to execute these "many-tasks" in parallel. In this paper, we present our experience in applying two cloud technologies Apache Hadoop and Microsoft DryadLINQ to two bioinformatics applications with the above characteristics. The applications are a pairwise Alu sequence alignment application and an EST (Expressed Sequence Tag) sequence assembly program. First we compare the performance of these cloud technologies using the above case and also compare them with traditional MPI implementation in one application. Next we analyze the effect of inhomogeneous data on the scheduling mechanisms of the cloud technologies. Finally we present a comparison of performance of the cloud technologies under virtual and non-virtual hardware platforms.

Index Terms—Distributed Programming, Parallel Systems, Performance, Programming Paradigms.

1 Introduction

THERE is increasing interest in approaches to data analysis in scientific computing as essentially every field is seeing an exponential increase in the size of the data deluge. The data sizes imply that parallelism is essential to process the information in a timely fashion. This is generating justified interest in new runtimes and programming models that unlike traditional parallel models such as MPI, directly address the data-specific issues. Experience has shown that at least the initial (and often most time consuming) parts of data analysis are naturally data parallel and the processing can be made independent with perhaps some collective (reduction) operation. This structure has motivated the important MapReduce [1] paradigm and many follow-on extensions. Here we examine three technologies (Microsoft Dryad/DryadLINQ [2][3], Apache Hadoop [4] and MPI) on two different bioinformatics applications (EST [5][6] and Alu clustering [7][8]). Dryad is an implementation of extended MapReduce from Microsoft. All the applications are (as is often so in Biology) "doubly data parallel" (or "all pairs" [9]) as the basic computational unit is replicated over all pairs of data items from the same (in our cases) or different datasets. In the EST example, each parallel task executes the CAP3 program on an input data file independently of others and there is no "reduction" or "aggregation" necessary at the end of the computation. On the otherhand, in the Alu case, a global aggregation is necessary at the end of the independent computations to produce the resulting dissimilarity matrix. In this paper we evaluate the different technologies showing that they give similar performance despite the different programming models.

In section 2, we give a brief introduction to the two cloud technologies we used while the applications EST and Alu sequencing are discussed in section 3. Section 4 presents some performance results. Conclusions are given in section 7 after a discussion of the different programming models in section 5 and related work in section 6.

2 Cloud Technologies

2.1 Dryad/DryadLINQ

Dryad is a distributed execution engine for coarse grain data parallel applications. It combines the MapReduce programming style with dataflow graphs to solve the computation tasks. Dryad considers computation tasks as directed acyclic graph (DAG) where the *vertices* represent computation tasks and with the *edges* acting as communication channels over which the data flow from one vertex to another. The data is stored in (or partitioned to) local disks via the Windows shared directories and meta-data files and Dryad schedules the execution of vertices depending on the data locality. (Note: The academic release of Dryad only exposes the DryadLINQ API for programmers [3][10]. Therefore, all our implementations are written using DryadLINQ although it uses Dryad as the underlying runtime). Dryad also stores the output of vertices in local disks, and the other vertices which depend on these results, access them via the shared directories. This enables Dryad to re-execute failed vertices, a step which improves the fault tolerance of the programming model.

2.2 Apache Hadoop

Apache Hadoop has a similar architecture to Google's MapReduce[1] runtime. Hadoop accesses data via HDFS [4], which maps all the local disks of the compute nodes to a single file system hierarchy, allowing the data to be

- *Jaliya Ekanayake and Thilina Gunarathne are with School of Informatics and Computing and Pervasive Technology Institute of Indiana University, Bloomington, IN 47408. E-mail: jekanaya@cs.indiana.edu, tgunarat@indiana.edu.*
- *Judy Qiu is with Pervasive Technology Institute, Indiana University, Bloomington, IN 47408. E-mail: xqiu@indiana.edu.*

Manuscript received (insert date of submission if desired).

dispersed across all the data/computing nodes. HDFS also replicates the data on multiple nodes so that failures of nodes containing a portion of the data will not affect the computations which use that data. Hadoop schedules the MapReduce computation tasks depending on the data locality, improving the overall I/O bandwidth. The outputs of the *map* tasks are first stored in local disks until later, when the *reduce* tasks access them (pull) via HTTP connections. Although this approach simplifies the fault handling mechanism in Hadoop, it adds a significant communication overhead to the intermediate data transfers, especially for applications that produce small intermediate results frequently.

3 Applications

3.1 Alu Sequence Classification

The Alu clustering problem [8] is one of the most challenging problems for sequence clustering because Alus represent the largest repeat families in human genome. There are about 1 million copies of Alu sequences in human genome, in which most insertions can be found in other primates and only a small fraction (~ 7000) is human-specific. This indicates that the classification of Alu repeats can be deduced solely from the 1 million human Alu elements. Alu clustering can be viewed as a classical case study for the capacity of computational infrastructures because it is not only of great intrinsic biological interests, but also a problem of a scale that will remain as the upper limit of many other clustering problems in bioinformatics for the next few years, such as the automated protein family classification for a few millions of proteins predicted from large metagenomics projects. In our work we have examined Alu samples of 35339 and 50,000 sequences using the pipeline of figure 1.

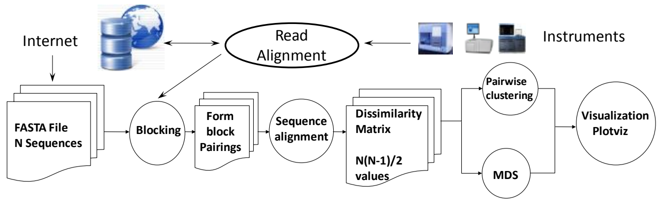


Fig. 1. Pipeline for analysis of sequence data.

3.1.1 Complete Alu Application

This application uses two highly parallel traditional MPI applications, i.e. MDS (Multi-Dimensional Scaling) and Pairwise (PW) Clustering algorithms described in Fox, Bae et al. [7]. The latter identifies sequence families as relatively isolated as seen for example in figure 2. MDS allows visualization by mapping the high dimension sequence data to three dimensions for visualization. MDS finds the best set of 3D vectors x(i) such that a weighted least squares sum of the difference between the sequence dissimilarity D(i,j) and the Euclidean distance |x(i) - x(j)| is minimized. This has a computational complexity of O(N^2) to find 3N unknowns for N sequences.

The PWClustering algorithm is an efficient MPI parallelization of a robust EM (Expectation Maximization) method using annealing (deterministic not Monte Carlo) originally developed by Ken Rose, Fox [14, 15] and others [16]. This improves over clustering methods like Kmeans which are sensitive to false minima. The original clustering work was based in a vector space (like Kmeans) where a cluster is defined by a vector as its center. However in a major advance 10 years ago [16], it was shown how one could use a vector free approach and operate with just the distances D(i,j). This method is clearly most natural for problems like Alu sequences where currently global sequence alignment (over all N sequences) is problematic but D(i,j) can be precisely calculated for each pair of sequenjces. PWClustering also has a time complexity of O(N^2) and in practice we find all three steps (Calculate D(i,j), MDS and PWClustering) take comparable times (a few hours for 50,000 sequences on 768 cores) although searching for a large number of clusters and refining the MDS can increase their execution time significantly. We have presented performance results for MDS and PWClustering elsewhere [7][12] and for large datasets the efficiencies are high (showing sometimes super linear speed up). For a node architecture reason, the initial distance calculation phase reported below has efficiencies of around 40-50% as the Smith Waterman computations are memory bandwidth limited. The more complex MDS and PWClustering algorithms show more computation per data access and higher efficiency.

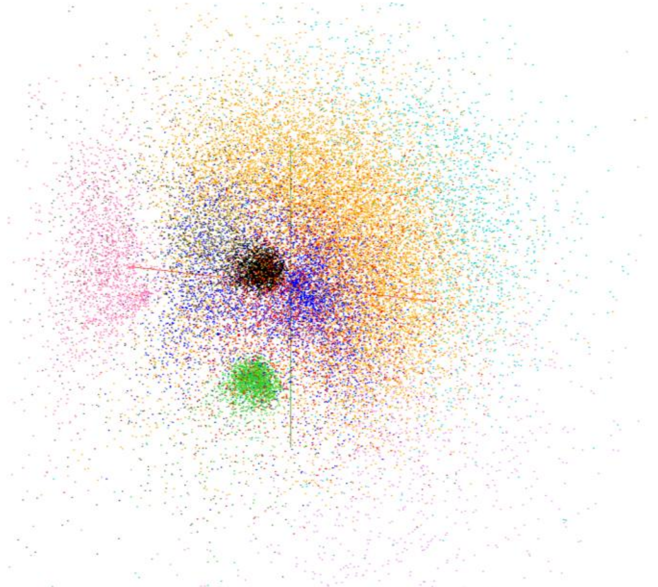


Fig. 2. Display of Alu clusters from MDS and clustering calculation from 35339 sequences using SW-G distances. The clusters corresponding to younger families AluYa, AluYb are particularly tight.

In the rest of the paper, we only discuss the initial dissimilarity computation although it is important that this links to later clustering and MDS stages as these require the output of the first stage in a format appropriate for the later MPI-based data mining stages. The MDS and PWClustering algorithms require a particular parallel decomposition where each of N processes (MPI processes, threads) has 1/N of sequences and for this subset {i} of sequences stores in memory D({i},j) for all sequences j and the subset {i} of sequences for which this node is responsible. This implies that we need D(i,j) and D(j,i) (which are equal) stored in different processors/disks. We design our initial calculation of D(i,j) so that efficiently we only calculated the independent set but the data was stored so that the later MPI jobs could efficiently access the data needed. We chose the simplest approach where the initial phase produced a single file holding the full set of D(i,j) stored by rows – all j for each successive value of i.

3.1.2 Smith Waterman Dissimilarities

We identified samples of the human and Chimpanzee Alu gene sequences using Repeatmasker [13] with Repbase Update [14]. We used open source version NAligner [11] of the Smith Waterman – Gotoh algorithm SW-G [15][16] modified to ensure low start up effects by each thread, processing large number (above a few hundred) of sequence calculations at a time. Memory bandwidth needed was reduced by storing data items in as few bytes as possible. In the following two sections, we discuss the initial phase of calculating distances D(i,j) for each pair of sequences so we can efficiently use MapReduce, DryadLINQ and MPI.

3.1.3 DryadLINQ Implementation

We developed a DryadLINQ application to perform the calculation of pairwise SW-G distances for a given set of genes by adopting a coarse grain task decomposition approach. This approach performs minimum inter-task communication and hence ameliorates the higher communication and synchronization costs of the parallel runtime. To clarify our algorithm, let's consider an example where N gene sequences produce a pairwise distance matrix of size NxN. We decompose the computation task by considering the resultant matrix and group the overall computation into a block matrix of size DxD where D is a multiple (>2) of the available computation nodes. Due to the symmetry of the distances D(i,j) and D(j,i) we only calculate the distances in the blocks of the upper triangle of the block matrix as shown in figure 3. (left).

Diagonal blocks are specially handled and calculated as full sub blocks. As the number of diagonal blocks is D and total number is D(D+1)/2, there is no significant compute overhead added. The blocks in the upper triangle are partitioned (assigned) to the available compute nodes and a DryadLINQ's "Apply" operation is used to execute a function to calculate (N/D)x(N/D) distances in each block, where d is defined as N/D. After computing the distances in each block, the function calculates the transpose matrix of the result matrix which corresponds to a block in the lower triangle, and writes both these matrices into two output files in the local file system. The names of these files and their block numbers are communicated back to the main program. The main program sorts the files based on their block numbers and performs another "Apply" operation to combine the files corresponding to rows in block matrix as shown in the figure 3 (right). The first step of this computation domintates the overall running time of the application and with the algorithm explained it clearly resembles the characteristics of a "many-task" problem.

3.1.4 Hadoop Implementation

We developed an Apache Hadoop version of the pairwise distance calculation program based on the JAligner[20] program, the java implementation of the NAligner code used in Dryad implementation. Similar to the other implementations, the computation is partitioned in to blocks based on the resultant distance matrix. Each of the blocks would get computed as a map task. The block size (D) can be specified via an argument to the program. The block size needs to be specified in such a way that there will be much more map tasks than the map task capacity of the system, so that the Apache Hadoop scheduling will happen as a pipeline of map tasks resulting in global load balancing inside the application. The input data is distributed to the worker nodes through the Hadoop distributed cache, which makes them available in the local disk of each compute node.

A load balanced task partitioning strategy according to the following rules is used to identify the blocks that need to be computed (dark grey) through map tasks as shown in the figure 4(left). In addition all the blocks in the diagonal (light grey) are computed. Even though the task partitioning mechanisms are different, both Dryad SW-G and Hadoop SW-G implementations end up with essentially identical set of computation blocks, if the same block size argument is given to both the programs.

When $\beta >= \alpha$, we calculate $D(\alpha,\beta)$ only if $\alpha+\beta$ is even,
When $\beta < \alpha$, we calculate $D(\alpha,\beta)$ only if $\alpha+\beta$ is odd.

The figure 4(right) depicts the run time behavior of the Hadoop SW-G program. In the given example the map task capacity of the system is "k" and the number of blocks is "N". The solid black lines represent the starting state, where "k" map tasks corresponding to "k" computation blocks will get scheduled in the compute nodes. The dashed black lines represent the state at time t1 , when 2 map tasks, m2 & m6, get completed and two map tasks from the pipeline get scheduled for the placeholders emptied by the completed map tasks. The gray dotted lines represent the future.

Map tasks use custom Hadoop writable objects as the map task output values to store the calculated pairwise distance matrices for the respective blocks. In addition, non-diagonal map tasks output the inverse of the distances matrix of the block as a separate output value. Hadoop uses local files and http transfers to transfer the map task output key value pairs to the reduce tasks.

The outputs of the map tasks are collected by the reduce tasks. Since the reduce tasks start collecting the outputs as soon as the first map task finishes and continue to do so while other map tasks are executing, the data transfers from the map tasks to reduce tasks do not present a significant performance overhead to the program. The program currently creates a single reduce task per each

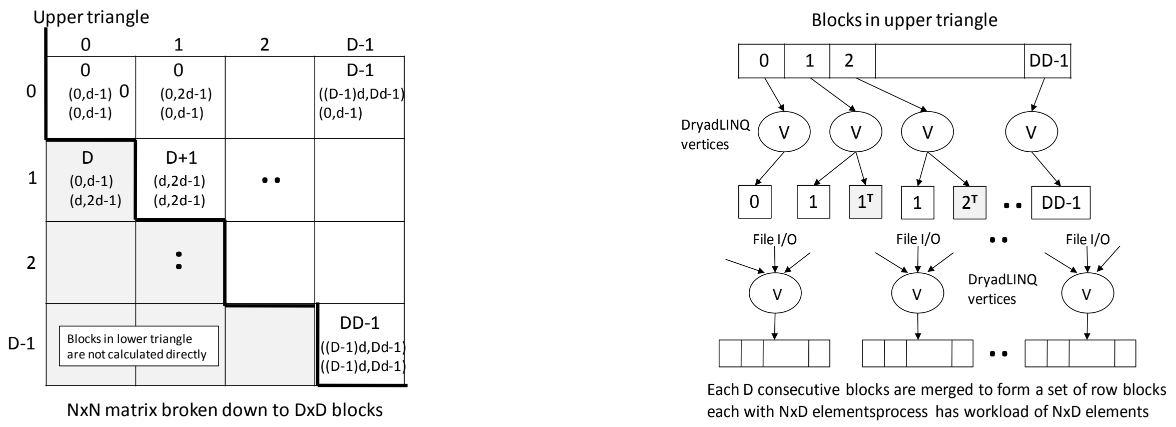

Upper triangle
0 1 2 D-1
(0,d-1) (0,d-1)
(0,2d-1) (0,d-1)
((D-1)d,Dd-1) (0,d-1)
D (0,d-1) (d,2d-1)
D+1 (d,2d-1) (d,2d-1)
DD-1 ((D-1)d,Dd-1) ((D-1)d,Dd-1)
Blocks in lower triangle are not calculated directly
NxN matrix broken down to DxD blocks
Blocks in upper triangle
DryadLINQ vertices
V
File I/O
Each D consecutive blocks are merged to form a set of row blocks each with NxD elementsprocess has workload of NxD elements

Fig. 3. Task decomposition (left) and the DryadLINQ vertex hierarchy (right) of the DryadLINQ implementation of SW-G pairwise distance calculation application.

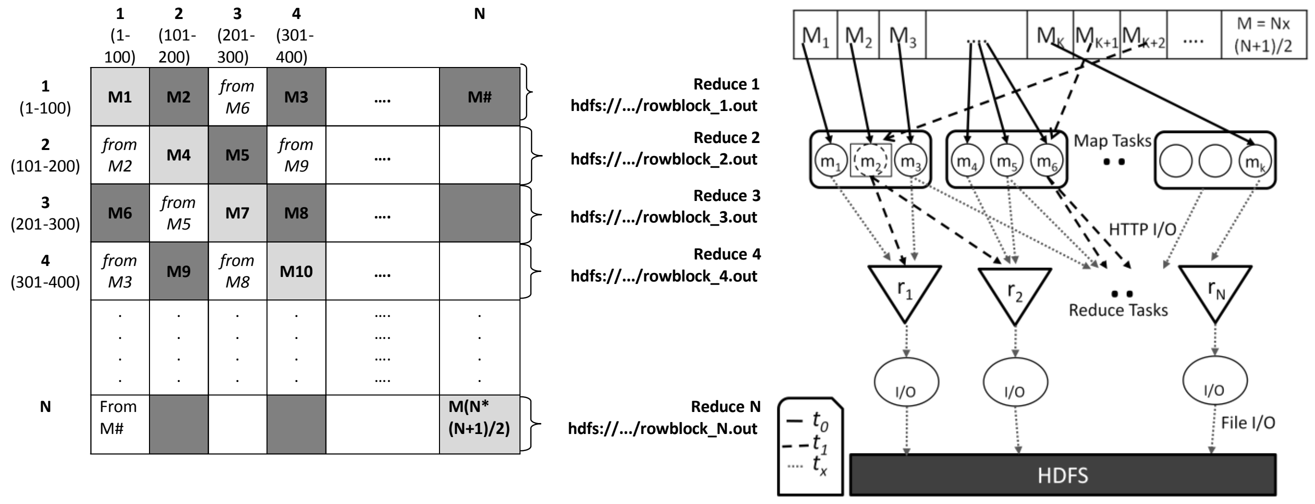

1 (1-100)
2 (101-200)
3 (201-300)
4 (301-400)
N
M1 M2 from M6 M3 M#
from M2 M4 M5 from M9
M6 from M5 M7 M8
from M3 M9 from M8 M10
From M# M(N*(N+1)/2)
Reduce 1 hdfs://.../rowblock_1.out
Reduce 2 hdfs://.../rowblock_2.out
Reduce 3 hdfs://.../rowblock_3.out
Reduce 4 hdfs://.../rowblock_4.out
Reduce N hdfs://.../rowblock_N.out
M1 M2 M3 Mk Mk+1 Mk+2 M = Nx(N+1)/2
Map Tasks
HTTP I/O
Reduce Tasks
I/O
File I/O
HDFS
t0 t1 tx

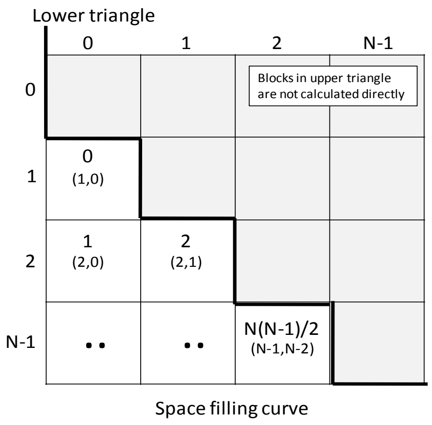
Fig. 4. Hadoop Implementation (left) Task (Map) decomposition and the reduce task data collection (right) Aapplication run time.

row block resulting in total of (no. of sequences/block size) Reduce tasks. Each reduce task accumulates the output distances for a row block and writes the collected output to a single file in Hadoop Distributed File System (HDFS). This results in N number of output files corresponding to each row block, similar to the output we produce in the Dryad version.

3.1.5 MPI Implementation

The MPI version of SW-G calculates pairwise distances using a set of either single or multi-threaded processes. For N gene sequences, we need to compute half of the values (in the lower triangular matrix), which is a total of M = N x (N-1) /2 distances. At a high level, computation tasks are evenly divided among P processes and execute in parallel. Namely, computation workload per process is M/P. At a low level, each computation task can be further divided into subgroups and run in T concurrent threads. Our implementation is designed for flexible use of shared memory multicore system and distributed memory clusters (tight to medium tight coupled communication technologies such threading and MPI). We provide options for any combinations of thread vs. process vs. node but in earlier papers [7][12], we have shown that threading is much slower than MPI for this class of problem. We have explored two different algorithms termed “Space Filling” and “Block Scattered”. In each case, we must calculate the independent distances and then build the full matrix exploiting symmetry of D(i,j).

The “Space Filling” MPI algorithm is shown in figure 5, where the data decomposition strategy runs a "space filling curve through lower triangular matrix" to produce equal numbers of pairs for each parallel unit such as process or thread. It is necessary to map indexes in each pairs group back to corresponding matrix coordinates (i, j) for constructing full matrix later on. We implemented a special function “PairEnumertator” as the convertor. We tried to limit runtime memory usage for performance optimization. This is done by writing a triple of i, j, and distance value of pairwise alignment to a stream writer and the system flashes accumulated results to a local file periodically. As the final stage, individual files are merged to form a full distance matrix. Next we describe the “Block Scattered” MPI algorithm shown in figure 6.



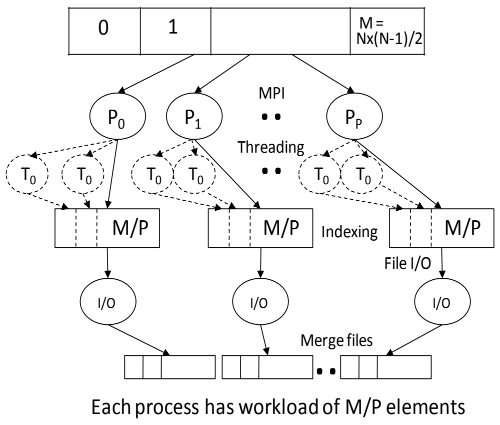
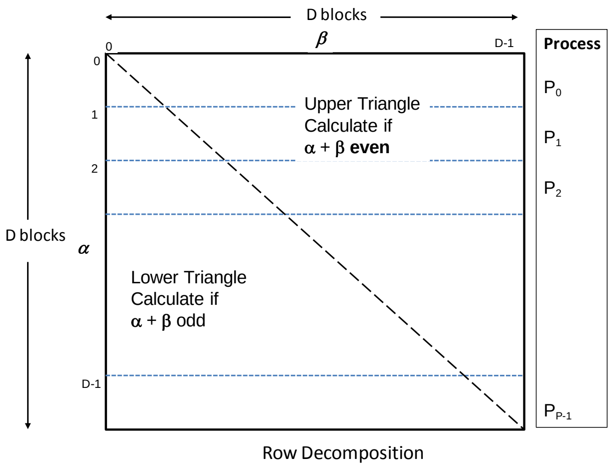


Fig. 5. Space Filling MPI Algorithm: Task decomposition (left) and SW-G implementation calculation (right).



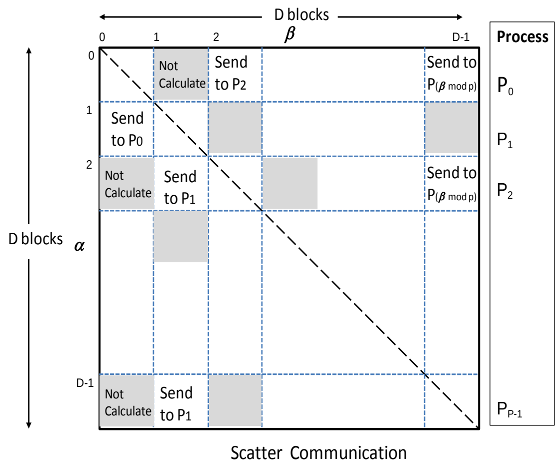


Fig. 6. Blocked MPI Algorithm: Row decomposition (left) and Scattered communication (right) to construct full matrix.

Points are divided into blocks such that each processor is responsible for all blocks in a simple decomposition illustrated in the figure 6 (left). This also illustrates the initial computation, where to respect symmetry, we calculate half the D(α,β) using the same criterion used in DryadLINQ implementation:

If β >= α, we only calculate D(α,β) if α+β is even while in the lower triangle, β < α, we only calculate D(α,β) if α+β is odd.

This approach can be applied to points or blocks. In our implementation, we applied it to blocks of points -- of size (N/P)x(N/P) where we use P MPI processes. Note we get better load balancing than the “Space Filling” algorithm as each processor samples all values of *β*. This computation step must be followed by a communication step illustrated in Figure 6 (Right) which gives full strips in each process. The latter can be straightforwardly written out as properly ordered file(s).

3.2 CAP3 Application EST and Its Software CAP3

3.2.1 EST and Its Software CAP3

An EST (Expressed Sequence Tag) corresponds to messenger RNAs (mRNAs) transcribed from the genes residing on chromosomes. Each individual EST sequence represents a fragment of mRNA, and the EST assembly aims to re-construct full-length mRNA sequences for each expressed gene. Because ESTs correspond to the gene regions of a genome, EST sequencing has become a standard practice for gene discovery, especially for the genomes of many organisms that may be too complex for whole-genome sequencing. EST is addressed by the software CAP3 which is a DNA sequence assembly program developed by Huang and Madan [17]. CAP3 performs several major assembly steps including computation of overlaps, construction of contigs, construction of multiple sequence alignments, and generation of consensus sequences to a given set of gene sequences. The program reads a collection of gene sequences from an input file (FASTA file format) and writes its output to several output files, as well as the standard output.

CAP3 is often required to process large numbers of FASTA formatted input files, which can be processed independently, making it an embarrassingly parallel application requiring no inter-process communications. We have implemented a parallel version of CAP3 using Hadoop and DryadLINQ. This application resembles a common parallelization requirement, where an executable, a script, or a function in a special framework such as

Matlab or R, needs to be applied on a collection of data items. We can develop DryadLINQ & Hadoop applications similar to the CAP3 implementations for all these use cases.

3.2.2 DryadLINQ Implementation

As discussed in section 3.2.1 CAP3 is a standalone executable that processes a single file containing DNA sequences. To implement a parallel application for CAP3 using DryadLINQ we adopted the following approach: (i) the input files are partitioned among the nodes of the cluster so that each node of the cluster stores roughly the same number of input files; (ii) a "data-partition" (A text file for this application) is created in each node containing the names of the input files available in that node; (iii) a DryadLINQ "partitioned-file" (a meta-data file understood by DryadLINQ) is created to point to the individual data-partitions located in the nodes of the cluster.

Then we used the "Select" operation available in DryadLINQ to perform a function (developed by us) on each of these input sequence files. The function calls the CAP3 executable passing the input file name and other necessary program parameters as command line arguments. The function also captures the standard output of the CAP3 program and saves it to a file. Then it moves all the output files generated by CAP3 to a predefined location.

3.2.3 Hadoop Implementation

Parallel CAP3 seqeunce assembly fits as a "map only" application for the MapReduce model. The Hadoop application is implemented by writing map tasks which execute the CAP3 program as a separate process on a given input FASTA file. Since the CAP3 application is implemented in C, we do not have the luxury of using the Hadoop file system (HDFS) directly. Hence the data needs to be stored in a shared file system across the nodes. However we are actively investigating the possibility of using Hadoop streaming and mountable HDFS for this purpose.

4 PERFORMANCE ANALYSIS

In this section we study the performance of SW-G and CAP3 applications under increasing homogeneous workloads, inhomogeneous workloads with different standard deviations and the performance in cloud like virtual environments. A 32 nodes IBM iDataPlex cluster, with each node having 2 quad core Intel Xeon processors (total 8 cores per node) and 32 GB of memory per node was used for the performance analysis under the following operating conditions, (i) Microsoft Window HPC Server 2008, service Pack 1 - 64 bit (ii) Red Hat Enterprise Linux Server release 5.3 -64 bit on bare metal (iii) Red Hat Enterprise Linux Server release 5.3 -64 bit on Xen hypervisor (version 3.0.3).

4.1 Scalability of different implementations

4.1.1 SW-G

In order to compare the scalability of Dryad, Hadoop and MPI implementations of ALU SW-G distance calculations with the increase of the data size using data sets of 10000 to 40000 seqeunces. These data sets correspond to 100 million to 1.6 billion total sequence distances. The actual number distance calculations performed by the applications are about half the above numbers due to optimisations mentioned in the implementation section. Data sets were generated by taking a 10000 sequence random sample from a real data set and replicating it 2 to 4 times. The Dryad & MPI results were adjusted to counter the performance difference of the kernel programs for fair comparison with the Hadoop implementation. NAligner on windows performs on average ~.78 times slower than Jaligner on Linux in the hardware we used for the performance analysis.

The results for this experiment are given in the figure 7. The time per actual calculation is computed by dividing the total time to calculate pairwise distances for a set of sequences by the actual number of comparisons performed by the application. According to figure 7, all three implementations perform and scale satisfactorily for this application with Hadoop implementation showing the best scaling. As expected, the total running times scaled proportionally to the square of the number of sequences. The Hadoop & Dryad applications perform and scale competitively with the MPI application.

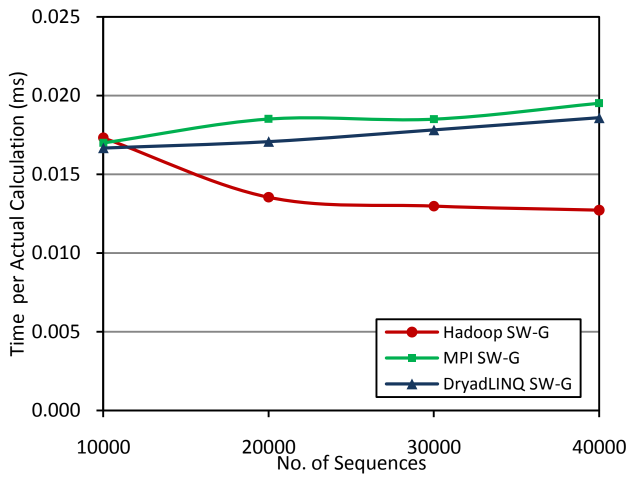


Fig. 7. Scalability of Smith Waterman pairwise distance calculation applications.

We can notice that the performance of the Hadoop implementation improving with the increase of the data set size, while Dryad performance degrades a bit. Hadoop improvements can be attributed to the diminishing of the framework overheads, while the Dryad degradation can be attributed to the memory management issues in the Windows and Dryad environment.

4.1.2 CAP3

We analysed the scalability of DryadLINQ & Hadoop implementations of the CAP3 application with the increase of the data set using homogeneous data sets. We prepare the data sets by replicating a single fasta file to represent a uniform workload across the application. The selected fasta sequence file contained 458 seqeunces.

The results are shown in the figure 8. The primary vertical axis (left) shows the total time vs the number of files. Secondary axis shows the time taken per file (total time / number of files) against the number of files. Bot the DryadLinq and Hadoop implementations show good scaling for the CAP3 application, although Dryad scaling is not as smooth as Hadoop scaling curve. Standalone CAP3 application used as the kernel for these applications performs better in the windows environment than in the Linux environment, which must be contributing to the reason for Hadoop being slower than Dryad.

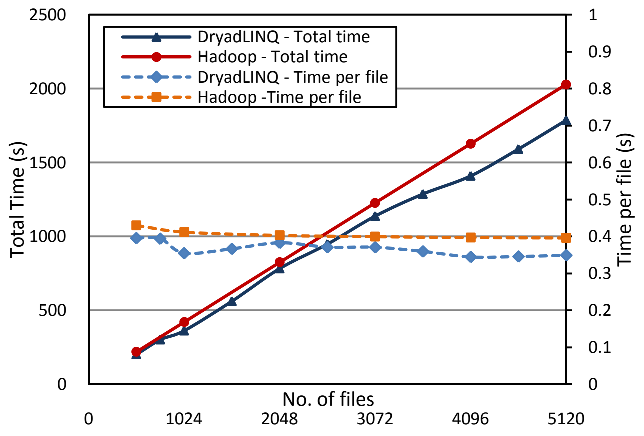


Fig. 8. Scalability of Cap3 applications.

4.2 Inhomogeneous Data Analysis

New generation parallel data processing frameworks such as Hadoop and DryadLINQ are designed to perform optimally when a given job can be divided in to a set of equally time consuming sub tasks. On the other hand most of the data sets we encounter in the real world are inhomogeneous in nature, making it hard for the data analyzing programs to efficiently break down the problems in to equal sub tasks.This motivated us to study the effects of inhomogeneity in the applications implemented using these frameworks.

4.2.1 SW-G Pairwise Distance Calculation

The inhomogeneoity of data applies for the gene sequence sets too, where individual sequence lengths and the contents vary among each other. In this section we study the effect of inhomogeneous gene sequence lengths for the performance of our pairwise distance calculation applications.

$$SWG(A,B) = O(mn)$$

The time complexity to align and obtain distances for two genome sequences A, B with lengths m and n respectively using Smith-Waterman-Gotoh algorithm is approximately proportional to the product of the lengths of two sequences (O(mn)). All the above described distributed implementations of Smith-Waterman similarity calculation mechanisms rely on block decomposition to break down the larger problem space in to sub-problems that can be solved using the distributed components. Each block is assigned two sub-sets of sequences, where Smith-Waterman pairwise distance similarity calculation needs to be performed for all the possible sequence pairs among the two sub sets. According to the above mentioned time complexity of the Smith-Waterman kernel used by these distributed components, the execution time for a particular execution block depends on the lengths of the sequences assigned to the particular block.

Parallel execution frameworks like Dryad and Hadoop work optimally when the work is equally partitioned among the tasks. Depending on the scheduling strategy of the framework, blocks with different execution times can have an adverse effect on the performance of the applications, unless proper load balancing measures have been taken in the task partitioning steps. For an example, in Dryad vertices are scheduled at the node level, making it possible for a node to have blocks with varying execution times. In this case if a single block inside a vertex takes a larger amount of time than other blocks to execute, then the whole node have to wait till the large task completes, which utilizes only a fraction of the node resources.

Since the time taken for the Smith-Waterman pairwise distance calculation depends mainly on the lengths of the sequences and not on the actual contents of the sequences, we decided to use randomly generated gene sequence sets for this experiment. The gene sequence sets were randomly generated for a given mean sequence length (400) with varying standard deviations following a normal distribution of the sequence lengths. Each sequence set contained 10000 sequences leading to 100 million pairwise distance calculations to perform. We performed two studies using such inhomogeneous data sets. In the first study the sequences with varying lengths were randomly distributed in the data sets. In the second study the sequences with varying lengths were distributed using a skewed distribution, where the sequences in a set were arranged in the ascending order of sequence length.

Figure 9 presents the execution time taken for the randomly distributed inhomogeneous data sets with the same mean length, by the two different implementations, while figure 10 presents the executing time taken for the skewed distributed inhomogeneous data sets. The Dryad results depict the Dryad performance adjusted for the performance difference of the NAligner and JAligner kernel programs. As we notice from the figure 9, both implementations perform satisfactorily for the randomly distributed inhomogeneous data, without showing significant performance degradations with the increase of the standard deviation. This behavior can be attributed to the fact that the sequences with varying lengths are randomly distributed across a data set, effectively providing a natural load balancing to the execution times of the sequence blocks.

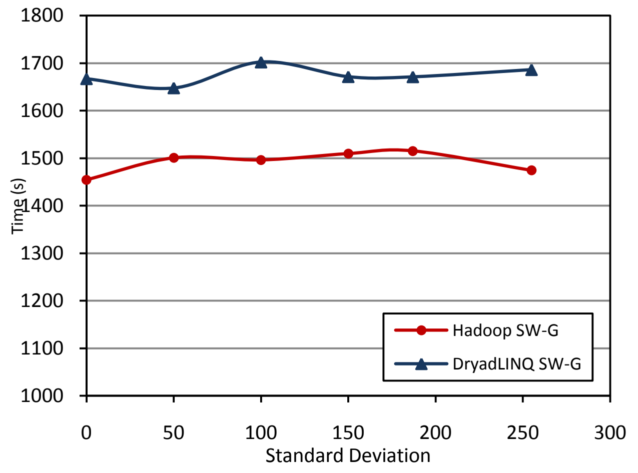


Fig. 9. Performance of SW-G pairwise distance calculation application for randomly distibuted inhomogeneous data with '400' mean sequence length.

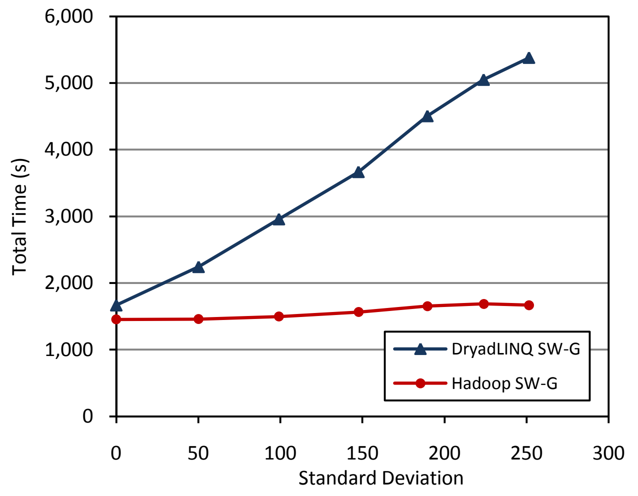


Fig. 10. Performance of SW-G pairwise distance calculation application for skewed distibuted inhomogeneous data with '400' mean sequence length.

For the skewed distributed inhomogeneous data, we notice clear performance degradation in the Dryad implementation. Once again the Hadoop implementation performs consistently without showing significant performance degradation, even though it does not perform as well as its randomly distributed counterpart. The Hadoop implementations' consistent performance can be attributed to the global pipeline scheduling of the map tasks. In the Hadoop Smith-Waterman implementation, each block decomposition gets assigned to a single map task. Hadoop framework allows the administrator to specify the number of map tasks that can be run on a particular compute node. The Hadoop global scheduler schedules the map tasks directly on to those placeholders in a much finer granularity than in Dryad, as and when the individual map tasks finish. This allows the Hadoop implementation to perform natural global level load balancing. In this case it might even be advantageous to have varying task execution times to iron out the effect of any trailing map tasks towards the end of the computation. Dryad implementation pre allocates all the tasks to the compute nodes and does not perform any dynamic scheduling across the nodes. This makes a node which gets a larger work chunk to take considerable longer time than a node which gets a smaller work chunk, making the node with a smaller work chuck to idle while the other nodes finish.

4.2.2 CAP3

Unlike in Smith-Waterman Gotoh (SW-G) implementations, CAP3 program execution time does not directly depend on the file size or the size of the sequences, as it depend mainly on the content of the sequences. This made is hard for us to artificially generate inhomogeneous data sets for the CAP3 program, forcing us to use real data. When generating the data sets, first we calculated the standalone CAP3 execution time for each of the files in our data set. Then based on those timings, we created data sets that have approximately similar mean times while the standard deviation of the standalone running times is different in each data set. We performed the performance testing for randomly distributed as well as skewed distributed (sorted according to individual file running time) data sets similar to the SWG inhomogeneous study. The speedup is taken by dividing the sum of sequential running times of the files in the data set by the parallel implementation running time.

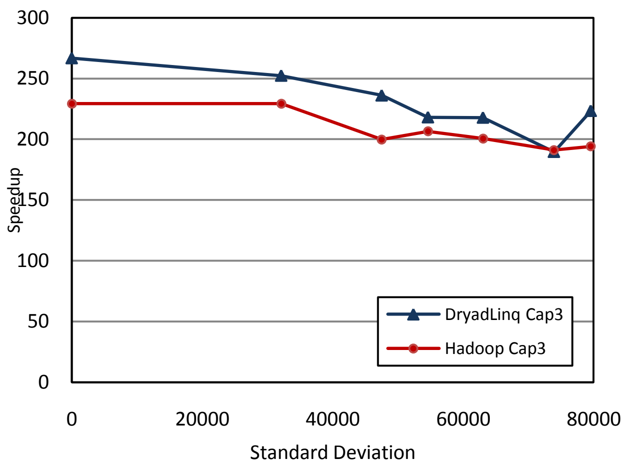


Fig. 11. Performance of Cap3 application for random distributed inhomogeneous data.

Figures 11 & 12 depict the CAP3 inhomogeneous performance results for Hadoop & Dryad implementations. Hadoop implementation shows satisfactory scaling for both randomly distributed as well as skewed distributed data sets, while the Dryad implementation shows satisfactory scaling in the randomly distributed data set. Once again we notice that the Dryad implementation does not perform well for the skewed distributed inhomogeneous data due to its' static non-global scheduling.

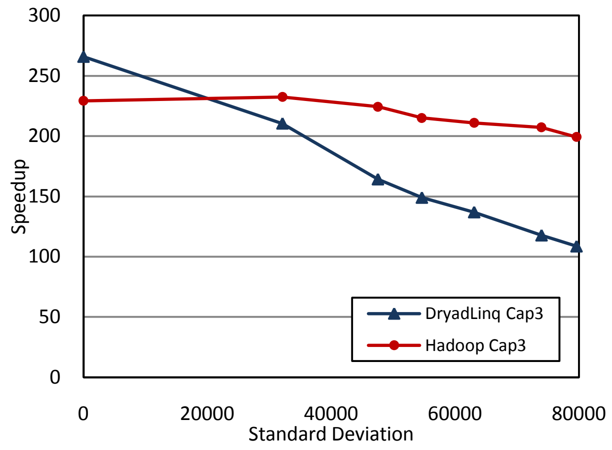


Fig. 12. Performance of Cap3 Applications for skewed distributed inhomogeneous data

4.3 Performance in the Cloud

With the popularity of the computing clouds, we can notice the data processing frameworks like Hadoop Map Reduce and DryadLINQ are becoming popular as cloudy parallel frameworks. We measured the performance and virtualization overhead of several MPI applications on the virtual environments in an earlier study [24]. Here we present extended performance results of using Apache Hadoop implementation of SW-G and Cap3 in a cloud environment by comparing Hadoop on Linux with Hadoop on Linux on Xen [26] para-virtualised environment.

While the Youseff, Wolski, et al. [27] suggests that the VM's impose very little overheads on MPI application, our previous study indicated that the VM overheads depend mainly on the communications patterns of the applications. Specifically the set of applications that is sensitive to latencies (lower communication to computation ration, large number of smaller messages) experienced higher overheads in virtual environments. Walker [28] presents benchmark results of the HPC application performance on Amazon EC2, compared with a similar bare metal local cluster, where he noticed 40% to 1000% performance degradations on EC2. But since one can not have complete control and knowledge over EC2 infrastructure, there exists too many unknowns to directly compare these results with the above mentioned results.

Above studies further motivated us to study the VM overhead for applications written using frameworks like Hadoop & Dryad. We set up a dynamic cluster to automatically switch the operating environment between hadoop on Linux, Hadoop on Linux on Xen and Dyrad on Windows HPCS. We used IBM's xCAT to enable switching controlled by messages managed by a publish-subscribe infrastructure. As described in [29], switching environments cost about 5 minutes — a modest cost to pay for reliable performance measurements on identical hardware. In our study we used the same cluster as bare metal and the virtual environment, so that we have full control over the infrastructure for a better comparison. All the tests used one VM per node to best match the bare metal environment. In all the nodes we setup the HDFS file system in a direct local disk partition. We used performance degradation computed using the $((T_{vm} - T_{bm}) / T_{bm})$ equation as our measure of VM overhead where T_{vm} stands for the running time in the virtual environment and T_{bm} stands for the running time in the bare metal environment.

4.3.1 SW-G Pairwise Distance Calculation

Figure 13 presents the virtualization overhead of the Hadoop SW-G application comparing the performance of the appication on linux on bare metal and on linux on xen virtual machines. The data sets used is the same 10000 real sequence replicated data set used for the scalability study in the section 4.1.1. The number of blocks is kept contant across the test, resulting in larger blocks for larger data sets. According to the results, the performance degradation for the Hadoop SWG application on virtual environment ranges from 25% to 15%. We can notice the performanace degradation gets reduced with the increase of the problem size.

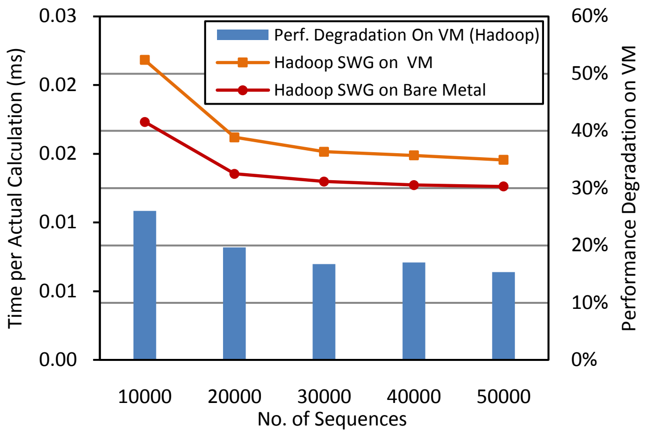


Fig. 13. Virtualization overhead of Hadoop SW-G on Xen virtual machines.

In the xen para-virtualization architecture, each guest OS (running in domU) perform their I/O transfers through Xen (dom0). This process adds startup costs to I/O as it involves startup overheads such as communication with dom0 and scheduling of I/O operations in dom0. Xen architecture uses shared memory buffers to transfer data between domU's and dom0, thus reducing the operational overheads when performing the actual I/O. We can notice the same behavior in the Xen memory management, where page table operations needs to go through Xen, while simple memory accesses can be performed by the guest Oss without Xen involvement. Accoring to the above points, we can notice that doing few coarser grained I/O and memory operations would incur relatively low overheads than doing the same work using many finer finer grained operations. We can conclude this as the possible reason behind the decrease of performance degradation with the increase of data size, as large data sizes increase the granularity of the computational blocks.

4.3.2 CAP3

Figure 14 presents the virtualization overhead of the Hadoop CAP3 application. We used the same scalability data set we used in section 4.1.2 for this analisys too. The performance degradation in this application remains constant near 20% for all the data sets. CAP3 application does not show the decrease of VM overhead with the increase of problem size as we noticed in the SWG application. Unlike in SWG, the I/O and memory behavior of the CAP3 program does not change based on the data set size, as irrespective of the data set size the granularity of the processing (single file) remains same. Hance the VM overheads does not get changed even with the increase of workload.

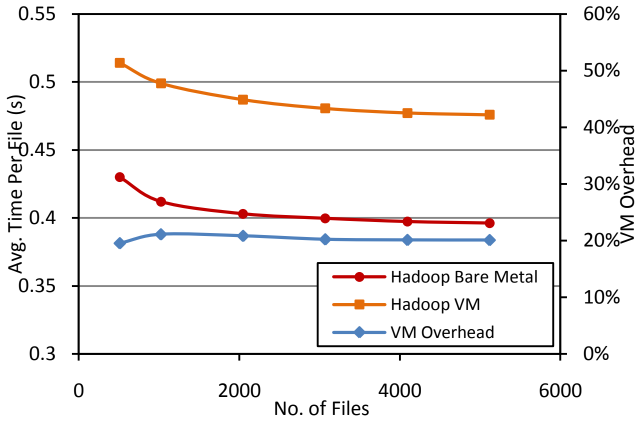


Fig. 14. Virtualization overhead of Hadoop Cap3 on Xen virtual machines

5. Comparison of Programming Models

The category of "many-task computing" also belongs to the pleasingly parallel applications in which the parallel tasks in a computation perform minimum inter-task communications. From our perspective independent jobs or jobs with collection of almost independent tasks represents the "many-tasks" domain. Apart from the EST and pairwise distance computations we have described, applications such as parametric sweeps, converting documents to different formats and brute force searches in cryptography are all examples in this category of applications.

One can adopt a wide range of parallelization techniques to perform most of the many-task computations in parallel. Thread libraries in multi-core CPUs, MPI, classic job schedulers in cloud/cluster/Grid systems, independent "maps" in MapReduce, and independent "vertices" in DryadLINQ are all such techniques. However, factors such as the granularity of the independent tasks, the amount of data/compute intensiveness of tasks determine the applicability of these technologies to the problem at hand. For example, in CAP3 each task performs highly compute intensive operation on a single input file and typically the input files are quite small in size (compared to highly data intensive applications), which makes all the aforementioned techniques applicable to CAP3. On the other hand the pairwise distance calculation application that requires a reduction (or combine) operation can easily be implemented using MapReduce programming model.

To clarify the benefits of the cloud technologies let's consider the following pseudo code segments representing Hadoop, DryadLINQ and MPI implementations of the pairwise distance (Smith Waterman dissimilarities) calculation application (discussed in section 3.1.2).

```
main{
    blockFiles[]=PartitionDataToRowBlocks(seqFile);
    paths[] = UploadToHDFS(blockFiles)
    addToDistributedCache (paths);
    //write a metadata file (row index & column
   // index) in HDFS for each block to compute
    IdentifyBlocksToCompute&WriteMetadata();
}

//value is a string containing row block index &
//column block index (contents of the metadata file)
map (key, value){
    rowSequences[] = Parse(row-block-file);
    columnSeqeunces[] = Parse(column-block-file);
    distances[][] = calculateDistances (
                rowSequences, columnSeqeunces);
    context.write(row-block-index , distances);
    context.write(column-block-index,
                        inverse(distances);
}

// key is a row-block-index, values are all the blocks
//belonging to a row block
reduce (key, blocks){
    Foreach(block in blocks){
        RowBlock +=block;
    }
}
```

Fig. 15. Code segment showing the MapReduce implementation of pairwise distance calculation using Hadoop.

In Hadoop implementation the *map* task calculates the Smith Waterman distances for a given block of sequences while the reduce task combine these blocks to produce row blocks of the final matrix. The MapReduce programming model executes map and reduce tasks in paral lel (in two separate stages) and handles the intermediate data transfers between them without any user intervention. The input data, intermediate data, and output data are all stored and accessed from the Hadoop's distributed file system (HDFS) making the entire application highly robust.

```
Main(){
//Calculate the block assignments
 assignBlocksToTasks();

//Perform allignment calculation for each block
//Group them according to their row numbers
//Combine rows to form row blocks
 outputInfo  =  assignedBlocks
        .SelectMany(block =>
                PerformAlignments(block))
        .GroupBy(x => x.RowIndex);
        .Apply(group=>PerformMerge(group));

    //Write all related meta data about row blocks
and
    // the corresponding output files.
    writeMetaData();
}

//Homomorphic  property  informs  the  compiler
//that each block can be processed separately.
[Homomorphic]
PerformAlignments(block){
     distances[]=calculateDistances(block);
     writeDistancesToFiles();
 }

[Homomorphic]
PerformMerge(group){
   mergeBlocksInTheGroup(group);
   writeOutputFile();
}
```

Fig. 16. Code segment showing the DryadLINQ implementation of pairwise distance calculation.

The DryadLINQ implementation resembles a more query style implementation in which the parallelism is completely abstracted by the LINQ operations which are performed in parallel by the underlying Dryad runtime. The "PerformAlignments" function has a similar capability to a map task and the "PerformMerge" function has a similar capability to the reduce task in MapReduce. DryadLINQ produces a directed acyclic graph (DAG) for this program in which both "SelectMany" and the "Apply" operations are performed as a collection of parallel vertices (tasks). The DAG is then executed by the underlying Dryad runtime. DryadLINQ implementation uses Windows shared directories to read input data, store and transfer intermediate results, and store the final outputs. With replicated data partitions, DryadLINQ can also support fault tolerance for the computation.

Input parameters of the MPI application include a FASTA file, thread, process per node, and node count. In the data decomposition phase, subsets of sequences are identified as described in section 3.1.5. Corresponding to MPI runtime architecture in Figure 6(right), we use MPI.net [18] API to assign parallel tasks to processes (de

```
using (MPI.Environment env = new
MPI.Environment(ref args)){
    sequences = parser.Parse(fastaFile);
    size = MPI.Intracommunicator.world.Size;
     rank = MPI.Intracommunicator.world.Rank;
    partialDistanceMatrix[][][] = new short
                        [size][blockHeight][blockWidth];

    //Compute the distances for the row block
    for (i =0; i< noOfMPIProcesses; i++){
        if (isComputeBlock(rank,i)){
            partialDistanceMatrix[i] =
                        computeBlock(sequences, i,rank);
         }
    }
    MPI_Barrier();

    // prepare the distance blocks to send
    toSend[] =transposeBlocks(partialDistanceMatrix);

    //use scatter to send  & receive distance blocks
    //that are computed & not-computed respectively
    for (receivedRank = 0; receivedRank < size;
                                    receivedRank++){
        receivedBlock[][] =
              MPI_Scatter<T[][]>(toSend, receivedRank);
        if (isMissingBlock(rank,receivedRank)){
           partialMatrix[receivedRank] = receivedBlock;
        }
    }
    MPI_Barrier();

    //Collect all distances to root process.
    if (rank == MPI_root){
        partialMatrix.copyTo(fullMatrix);
        for(i=0;i<size;i++){
           if (rank != MPI_root){
              receivedPartMat=MPI_Receive<T[][]>(i, 1);
              receivedPartMat.copyTo(fullMatrix);
           }
        }
        fullMatrix.saveToFile();
    }else{
        MPI_Send<t[][]>(partialMatrix, MPI_root, 1);
    }
}
```

Fig. 17. Code segment showing the MPI implementation of pairwise distance calculation.

fault is one per core). One can use fewer processes per node and multi-threading in each MPI process (and this is most efficient way to implement parallelism for MDS and PWClustering) but we will not present these results here. Each compute node has a copy of the input file and out put results written directly to a local disk. In this MPI example, there is a Barrier call followed by the scatter communication at the end of computing each block. This is not the most efficient algorithm if the compute times per block are unequal but was adopted to synchronize the communication step.

The code segments show clealy how the higher level parallel runtimes such as Hadoop and DryadLINQ have abstracted the parallel programming aspects from the users. Although the MPI algorithm we used for SW-G computation only uses one MPI communication construct (MPI_Scatter), in typical MPI applications the programmer needs to explicitly use various communication constructs to build the MPI communication patterns. The low level communication contructs in MPI supports parallel algorithms with variety of communication topologies. However, developing these applications require great amount of programming skills as well.

On the other hand high level runtimes provide limited communication topologies such as map-only or map followed by reduce in MapReduce and DAG base execution flows in DryadLINQ making them easier to program. Added support for handling data and quality of services such as fault tolerance make them more favorable to develop parallel applications with simple communication topologies. Many-task computing is an ideal match for these parallel runtimes.

There are some important differences such as MPI being oriented towards memory to memory operations whereas Hadoop and DryadLINQ are file oriented. This difference makes these new technologies far more robust and flexible. On the other the file orientation implies that there is much greater overhead in the new technologies. This is a not a problem for initial stages of data analysis where file I/O is separated by a long processing phase. However as discussed in [12], this feature means that one cannot execute efficiently on MapReduce, traditional MPI programs that iteratively switch between "map" and "communication" activities. We have shown that an extended MapReduce programming model named i-MapReduce[19][7] can support both classic MPI and MapReduce operations. i-MapReduce has a larger overhead than good MPI implementations but this overhead does decrease to zero as one runs larger and larger problems.

6. Related Work

There have been several papers discussing data analysis using a variety of cloud and more traditional cluster/Grid technologies with the Chicago paper [20] influential in posing the broad importance of this type of problem. The Notre Dame all pairs system [21] clearly identified the "doubly data parallel" structure seen in all of our applications. We discuss in the Alu case the linking of an initial doubly data parallel to more traditional "singly data parallel" MPI applications. BLAST is a well known doubly data parallel problem and has been discussed in several papers [22][23]. The Swarm project [6] successfully uses traditional distributed clustering scheduling to address the EST and CAP3 problem. Note approaches like Condor have significant startup time dominating performance. For basic operations [24], we find Hadoop and Dryad get similar performance on bioinformatics, particle physics and the well known kernels. Wilde [25] has emphasized the value of scripting to control these (task parallel) problems and here DryadLINQ offers some capabilities that we exploited. We note that most previous work has used Linux based systems and technologies. Our work shows that Windows HPC server based systems can also be very effective.

7. Conclusions

We have studied two data analysis problems with three different technologies. They have been looked on machines with up to 768 cores with results presented here run on 256 core clusters. The applications each start with a "doubly data-parallel" (all pairs) phase that can be implemented in MapReduce, MPI or using cloud resources on demand. The flexibility of clouds and MapReduce suggest they will become the preferred approaches. We showed how one can support an application (Alu) requiring a detailed output structure to allow follow-on iterative MPI computations. The applications differed in the heterogeneity of the initial data sets but in each case good performance is observed with the new cloud MapReduce technologies competitive with MPI performance. The simple structure of the data/compute flow and the minimum inter-task communicational requirements of these "pleasingly parallel" applications enabled them to be implemented using a wide variety of technologies. The support for handling large data sets, the concept of moving computation to data, and the better quality of services provided by the cloud technologies, simplify the implementation of some problems over traditional systems. We find that different programming constructs available in MapReduce such as independent "maps" in MapReduce, and "homomorphic Apply" in DryadLINQ are suitable for implementing applications of the type we examine. In the Alu case, we show that DryadLINQ and Hadoop can be programmed to prepare data for use in later parallel MPI/threaded applications used for further analysis. We performed tests using identical hardware for Hadoop on Linux, Hadoop on Linux on Virtual Machines and DryadLINQ on HPCS on Windows. These show that DryadLINQ and Hadoop get similar performance and that virtual machines give overheads of around 20%. We also noted that support of inhomogeneous data is important and that Hadoop currently performs better than DryadLINQ unless one takes steps to load balance the data before the static scheduling used by DryadLINQ. We compare the ease of programming for MPI, DryadLINQ and Hadoop. The MapReduce cases offer higher level interface and the user needs less explicit control of the parallelism. The DryadLINQ framework offers significant support of database access but our examples do not exploit this.

ACKNOWLEDGMENT

The authors wish to thank our collaborators from Biology whose help was essential. In particular Alu work is with Haixu Tang and Mina Rho from Bioinformatics at Indiana University and the EST work is with Qunfeng Dong from Center for Genomics and Bioinformatics at Indiana University. We appreciate all SALSA group members, especially Dr. Geoffrey Fox, Scott Beason, and Stephen Tak-Lon Wu for their contributions. We would like to thank Microsoft for their collaboration and support. Tony Hey, Roger Barga, Dennis Gannon and Christophe Poulain played key roles in providing technical support.

REFERENCES

[1] J. Dean, and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM* vol. 51, no. 1, pp. 107-113, 2008.

[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," *European Conference on Computer Systems*, March 2007.

[3] Y.Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. Gunda, J. Currey, "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language," *Symposium on Operating System Design and Implementation (OSDI)*, 2008.

[4] Apache Hadoop, http://hadoop.apache.org/core/

[5] X. Huang, A. Madan, "CAP3: A DNA Sequence Assembly Program," *Genome Research*, vol. 9, no. 9, pp. 868-877, 1999.

[6] S.L. Pallickara, M. Pierce, Q. Dong, and C. Kong, "Enabling Large Scale Scientific Computations for Expressed Sequence Tag Sequencing over Grid and Cloud Computing Clusters," PPAM 2009 - Eighth International Conference on Parallel Processing and Applied Mathematics, 2009.

[7] G. Fox, S.H. Bae, J. Ekanayake, X. Qiu, H. Yuan. "Parallel Data Mining from Multicore to Cloudy Grids," *High Performance Computing and Grids workshop (HPC 2008,* Italy,2008, http://grids.ucs.indiana.edu/ptliupages/publications/CetraroWriteupJan09_v12.pdf

[8] M.A. Batzer, P.L. Deininger, "Alu Repeats And Human Genomic Diversity," *Nature Reviews Genetics* vol. 3, no. 5, pp. 370-379. 2002.

[9] C. Moretti, H. Bui, K. Hollingsworth, B. Rich, P. Flynn, D. Thain, "All-Pairs: An Abstraction for Data Intensive Computing on Campus Grids," *IEEE Transactions on Parallel and Distributed Systems*, 2009, DOI 10.1109/TPDS.2009.49

[10] J. Ekanayake, A.S. Balkir, T. Gunarathne, G. Fox, C. Poulain, N. Araujo, R. Barga. "DryadLINQ for Scientific Analyses," *5th IEEE International Conference on e-Science*, 2009.

[11] Source Code. Smith Waterman Software. http://jaligner.sourceforge.net/

[12] G. Fox, X. Qiu, S. Beason, J.Y. Choi, M. Rho, H. Tang, N. Devadasan, G. Liu, "Biomedical Case Studies in Data Intensive Computing," *Keynote talk at The 1st International Conference on Cloud Computing (CloudCom 2009)* at Beijing Jiaotong University, China, 2009.

[13] A.F.A. Smit, R. Hubley, P. Green, 2004. Repeatmasker. http://www.repeatmasker.org

[14] J. Jurka, "Repbase Update: a database and an electronic journal of repetitive elements," Trends Genet. 9, pp.418-420, 2000.

[15] O. Gotoh, "An improved algorithm for matching biological sequences," Journal of Molecular Biology, 162, pp.705-708, 1982.

[16] T.F. Smith, M.S. Waterman, "Identification of common molecular subsequences," Journal of Molecular Biology, 147, pp.195-197, 1981.

[17] X. Huang, A. Madan, "CAP3: A DNA Sequence Assembly Program," Genome Research, vol. 9, no. 9, pp. 868-877, 1999.

[18] MPI.Net: High-Performance C# Library for Message Passing http://www.osl.iu.edu/research/mpi.net/

[19] J. Ekanayake, S. Pallickara, "MapReduce for Data Intensive Scientific Analysis," *Fourth IEEE International Conference on eScience*, pp.277-284, 2008.

[20] I. Raicu, I.T. Foster, Y. Zhao, "Many-Task Computing for Grids and Supercomputers," *Workshop on Many-Task Computing on Grids and Supercomputers MTAGS*, IEEE pages 1-11, 2008.

[21] C. Moretti, H. Bui, K. Hollingsworth, B. Rich, P. Flynn, D. Thain, "All-Pairs: An Abstraction for Data Intensive Computing on Campus Grids," *IEEE Transactions on Parallel and Distributed Systems*, DOI 10.1109/TPDS.2009.49, 2009.

[22] M.C. Schatz, "CloudBurst: highly sensitive read mapping with MapReduce," *Bioinformatics 25(11),* pp. 1363-1369, 2009, doi:10.1093/bioinformatics/btp236

[23] K. Keahey, R. Figueiredo, J. Fortes, T. Freeman, M. Tsugawa, "Science clouds: Early experiences in Cloud computing for scientific applications," *In Cloud Computing and Applications, (CCA08)*, 2008.

[24] Jaliya Ekanayake, Xiaohong Qiu, Thilina Gunarathne, Scott Beason, Geoffrey Fox, "High Performance Parallel Computing with Clouds and Cloud Technologies", to appear as a book chapter of Cloud Computing and Software Services: Theory and Techniques, CRC Press (Taylor and Francis), ISBN-10: 1439803153. http://grids.ucs.indiana.edu/ptliupages/publications/cloud_handbook_final-with-diagrams.pdf

[25] M. Wilde, I. Raicu, A. Espinosa, Z. Zhang1, B. Clifford, M. Hategan, S. Kenny, K. Iskra, P. Beckman, I. Foster, "Extreme-scale scripting: Opportunities for large task parallel applications on petascale computers," *SCIDAC, Journal of Physics: Conference Series* 180. DOI: 10.1088/1742-6596/180/1/012046, 2009.

[26] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," *ACM Symposium on Operating System Principles*, 2003.

[27] Youseff, L., R. Wolski, et al. 2006. Evaluating the Performance Impact of Xen on MPI and Process Execution For HPC Systems. Proc of First International Workshop on Virtualization Technology in Distributed Computing, 2006.

[28] Walker, E. 2008. Benchmarking Amazon EC2 for high-performance scientific computing. http://www.usenix.org/publications/login/2008-10/openpdfs/walker.pdf.

[29] Judy Qiu, Jaliya Ekanayake, Thilina Gunarathne, Jong Youl Choi, Seung-Hee Bae, Yang Ruan, Saliya Ekanayake, Stephen Wu, Scott Beason, Geoffrey Fox, Mina Rho, Haixu Tang, "Data Intensive Computing for Bioinformatics", *to appear as a book chapter of Data Intensive Distributed Computing, IGI Publishers*,

http://grids.ucs.indiana.edu/ptliupages/publications/DataIntensiveComputing_BookChapter.pdf.,2010.

Jaliya Ekanayake is a Ph.D. candidate at the School of Informatics and Computing of Indiana University, Bloomington. His research advisor is Prof. Geoffrey Fox. Jaliya works in the Community Grids Lab as a research assistant and his Ph.D. research focuses on architecture and performance of runtime environments for data intensive scalable computing.

Thilina Gunarathne is a Ph.D. candidate at the School of Informatics and Computing of Indiana University, Bloomington advised by Prof. Geoffrey Fox. Thilina works in the Community Grids Lab as a research assistant. His research interests include parallel programming architectures and their composition.

Judy Qiu obtained her PhD from Syracuse University and is currently an assistant director in the Pervasive Technology Institute at Indiana University. Here she leads the SALSA group in data-intensive and multicore systems.