2011 IEEE International Parallel & Distributed Processing Symposium

# Parallelizing BLAST and SOM algorithms with MapReduce-MPI library

Seung-Jin Sul, Andrey Tovchigrechko
J. Craig Venter Institute
Rockville, MD
atovtchi@jcvi.org

***Abstract*—Most bioinformatics algorithms are developed in a serial form due to a fast pace of changes in the subject domain and the fact that many bioinformatics tasks can be parallelized as collections of serial jobs communicating at the file system level (High-Throughput Computing, HTC). Recently, a MapReduce-MPI library was made available by Sandia Lab to ease porting of a large class of serial applications to the High Performance Computing (HPC) architectures dominating large federated resources such as NSF TeraGrid. Using this library, we have created two open-source bioinformatics applications. The first one addresses a problem of adapting existing complex and highly optimized serial bioinformatics algorithm to HPC architecture in a minimally invasive way. We built a parallel BLAST implementation that calls the high-level methods of unmodified NCBI C++ Toolkit. We demonstrated scaling for up to 1000 cores on TACC Ranger cluster when processing the sufficiently large input datasets. Using unmodified NCBI Toolkit ensures that the results are compatible across the multitude of settings in the original serial algorithm, and that future versions of the upstream code can be easily integrated. The second application is a Self-Organizing Map (SOM) machine-learning algorithm, popular in bioinformatics applications such as metagenomic binning. The nature of the SOM requires a global synchronization step with a frequency that necessitates the use of an HPC environment. Our implementation of the "batch SOM" uses a mix of MapReduce-MPI and direct MPI calls and scales to 1000 cores as well. This allows easy processing of datasets with a size that is out of range of the serial SOM implementations. Both implementations are available in the open source at http://andreyto.github.com/mgtaxa/.**

***Keywords - bioinformatics; parallel algorithms; map-reduce; high-performance computing***

## I. INTRODUCTION

The drastic drop in the cost of genomic sequencing in recent years is changing the landscape of the field of bioinformatics. Before, only the big sequencing centers needed comparatively large compute farms to annotate and analyze the raw data. They could also easily afford to acquire the necessary hardware because the cost of computations was only a fraction of the cost of the sequencing itself. Now, a single NextGen sequencing machine, affordable to even a separate academic Lab, will produced a stream of data that can only be processed in parallel using hundreds of cores even on the latest generation of hardware. So far, the curve that measures the amount of genomic sequence that can be produced for a dollar is outpacing the Moore's law.

Most of the bioinformatics algorithms have been designed as sequential, or with a shared memory parallelism with a limited scaling ability. There are two main reasons behind this:

One is the nature of the bioinformatics processing. Bioinformatics can be loosely defined as the analysis of biological sequences. Typically, many sequences have to be processed independently, with some global but relatively inexpensive step at the end. Examples are database searches, where millions of query sequences, such as metagenomic reads or predicted on such reads protein fragments have to be searched against a database of characterized sequences that in turn contains millions of entries. Even when a single completely sequenced genome has to be annotated, the searches are done for the protein sequences, which again are in the thousands of entries.

Thus, many bioinformatics tasks are embarrassingly parallel by nature and easily lend themselves to the coarse grained task-based parallelism where data are typically exchanged through the files on a shared file system.

The second reason is the fast pace of development in the subject domain. Because the biological knowledge has been expanding so fast, many concepts were appearing and undergoing substantial changes. New bioinformatics algorithms or substantial modifications of the existing ones had to be constantly developed. Creating or modifying a distributed parallel algorithm carries a substantial effort overhead compared to a serial version. Thus, even in cases where a more tightly coupled parallel formulation could provide a significant speed up compared to the file-based work splitting, the extra effort was often not justified in the eyes of the developers.

Different sequential tools are often organized into complex workflows, which is a typical mode of operation for the High-Throughput Computing (HTC) clusters.

Although many organizations dealing with sequencing have built their own local HTC clusters, with the mentioned democratization of sequencing this approach is becoming less affordable for smaller groups entering the sequencing field. Using bigger federated resources would be a solution providing better turn-around time while ensuring good utilization of resources.

However, the public funding agencies have a very substantial investment into High-Performance Computing (HPC) systems, designed primarily for the needs of physics, earth science and molecular modeling communities. For example, NSF TeraGrid is dominated by such clusters.

This work was sponsored by NSF grant DBI-0850256

1530-2075/11 $26.00 © 2011 IEEE
DOI 10.1109/IPDPS.2011.180

Although having ability to run bioinformatics algorithms on HPC systems would be beneficial, their scheduling policies are geared towards allocating large parallel jobs using hundreds or thousands of cores.

There are several execution engines that through the concepts of so called "glide-ins" emulate a serial execution environment (examples are SWIFT [1] and GlideinWMS [2]). They work through a two-level scheduling: allocating relatively large MPI jobs at the local resource manager on the cluster, and then having each processor rank act as an execution daemon that starts sequential tasks farmed out from the scheduler in a load-balancing mode.

Such systems allow running existing sequential workflows with minimal modifications. The glide-in systems, however, need either direct connections from compute nodes to the scheduler (typically running at external user's host) or having an intermediate proxy running on the cluster gateway node. Both of these options might be disallowed by the cluster policies. They also need to perform the *fork()* call on the compute nodes, which is not always available on some HPC platforms.

In this work, we demonstrate an approach to porting bioinformatics algorithms to HPC architectures that aims to find a middle ground between a deep re-design of an existing sequential algorithm with MPI calls, and an attempt at farming out execution of unmodified sequential programs. This approach will work even without *fork()* availability when the original algorithm can be compiled from source within an MPI program. With *fork()*, external compiled binaries can also be wrapped.

Specifically, we use the MapReduce framework implemented as a regular MPI program (as available from Sandia Lab [3]) to parallelize two applications.

One is NCBI BLAST [4], which is a de-facto standard tool for the database sequence similarity search and is the program consuming 80-90% of compute cycles in bioinformatics overall, based on our experience at the J. Craig Venter Institute as well as informal communication with other large centers. This is an example of a highly optimized and very complex existing sequential implementation. By the nature of the search, it also demonstrates a very irregular and unpredictable execution time for a given query sequence.

Another one is the Self Organizing Map (SOM) dimensionality reduction and clustering algorithm. In the bioinformatics domain, SOM is a popular tool for unsupervised clustering and semi-supervised classification of metagenomic sequences in a multi-dimensional sequence composition space due to its robustness to noise that is inevitable in real biological data, as well as easy visual presentation of results. The algorithm itself is simple. In its so-called "batch" formulation it maps very well to the coarse-grained parallelism model of the MapReduce. We further optimize the implementation with a little direct MPI programming in critical spots.

## II. BACKGROUND AND SEQUENTIAL ALGORITHMS

### A. MapReduce-MPI framework

MapReduce is a parallel processing pattern that can fit well many bioinformatics computations. It has been popularized by Google in the implementation that uses a distributed file system to exchange data [5]. Since then, other implementations have appeared that target different architectures and communication channels, such as multi-core systems with shared memory or distributed systems with only in-core communications.

In this work, we are using MPI implementation called MapReduce-MPI [3]. As the name suggests, it is implemented with standard MPI calls. Because this is a C++ library that is compiled into a regular MPI program, it does not require any special administrative support on the HPC cluster system. Thus, the application behaves as a normal MPI process.

Additionally, the programmer can always take advantage of using MPI calls directly. This is important for situations where fairly restrictive MapReduce programming paradigm creates friction with the required communication patterns, such as in all-against-all computations. The price for this extra flexibility and portability is a lack of fault-tolerance inherent in the underlying MPI execution model.

### B. BLAST

Because the task of searching for similarity in sequence databases has such a central role in the bioinformatics, BLAST algorithm has undergone multiple rounds of improvements and deep optimizations. The algorithm is quite complex, both in its statistical foundations, as well as in the implementation. It has been described in a series of publications over many years, where the latest one is probably a good starting point [6].

In a very short narrative, the algorithm is optimized to find statistically significant pairwise similarities between any number of query sequences and potentially very large collections of database sequences. It does it by breaking the search into three stages, with each stage discarding progressively larger numbers of candidate matches by applying progressively more expensive filters.

The first stage scans for matches between fixed size words; the second stage extends each matching word as an ungapped alignment on the condition that there is another word match nearby, and the third stage performs gapped alignment for those matches that passed the second stage.

At each stage, the remaining candidates have to pass the test for statistical significance, typically controlled by the user through the *E-value* cutoff parameter, which limits the absolute number of alignments that can be found for a given query sequence in the current database strictly by chance.

The implementation iteratively loads the next concatenated subset of query sequences, builds a word lookup table out of them, and streams the database past this lookup table, storing the positions of matches. After selecting the candidate set of matches, it loads the corresponding database sequences again to perform the final alignment.

Where supported by the platform, the database access is implemented by caching memory-mapped regions of the DB while trying to keep the total resident memory under an optimal limit.

The current de-facto standard implementation is the one maintained by NCBI. Recently, it has undergone a deep refactoring as part of moving to the NCBI C++ Toolkit. Now, the core is implemented in C, with the higher-level interfaces and the default sequence database module implemented in C++. The legacy NCBI C Toolkit is now in the maintenance mode, but it uses this new BLAST C core. The new C++ implementation called BLAST+ [6] has important improvements as well as new options to control its behavior.

The NCBI implementation is optimized for shared memory machines with limited thread parallelism. Node-level parallelism is achieved by splitting the query set and partitioning the database, and running the resulting matrix of serial jobs across a cluster of hosts, with one or more combiner jobs at the end merging results for each query across different database partitions.

### C. *Existing parallel or hardware accelerated implementations of BLAST*

We found two available MPI-parallelized ports of NCBI BLAST: mpiBLAST [7] and ScalaBLAST [8]. Initially, we intended to use nucleotide mpiBLAST in the context of our metagenomic taxonomic classification study. It has been already installed on TACC Ranger system where we had a TeraGrid allocation. Despite going through a matrix of (compiler / MPI library / mpiBLAST source version / runtime options), we were not able to make it run reliably on Ranger. We were getting interminent crashes or empty output files in some but not all runs with identical or nearly identical inputs. Following the advice obtained from the developers still did not help.

The mpiBLAST is a very efficient and optimized implementation that has demonstrated excellent scaling in a number of publications by its authors. We think that this degree of optimization is also a source of our problems in using it. In order to integrate the existing complex native BLAST codes, mpiBLAST goes on patching the upstream BLAST code and even the standard OS library. For example, it replaces the CLIB *open()* function call, in order to introduce virtual file access to DB partitions distributed through MPI calls in a way that is transparent to the NCBI BLAST DB module. It also separates the scanning stage code from the gapped extension stage code in order to achieve superlinear speed-ups compared to the matrix-split task-level parallelization.

We suspect that these efforts make mpiBLAST sensitive to site-specific changes such as either OS or networking stack updates.

We concluded that there is an inherent contradiction in an approach that tries a deep optimization on top of software that is already thoroughly optimized for a very different kind of execution environment.

Additionally, it makes it difficult to keep up with changes in an upstream NCBI code. mpiBLAST still integrates and patches the legacy NCBI C BLAST source instead of the current BLAST+ that is now used by NCBI itself.

Instead of trying our luck with similarly structured ScalaBLAST and learning the peculiarities of the latter, we decided to use MapReduce-MPI library to wrap pristine BLAST+ through its high-level NCBI C++ Toolkit API library calls. The advantages of this approach are: 1) it is trivial to keep up with upstream code updates 2) it is easy to support any of the multitudes of options implemented by the upstream algorithm 3) the implementation is simple and serves as an example of wrapping any other serial bioinformatics algorithm that is amenable to a matrix-split parallelization.

Aside from the MPI versions, we also had access to a TimeLogic DeCypherBLAST [9] that is a commercial (closed-source) ground-up re-implementation of BLAST algorithm. It accelerates the scan stage on FPGA card. However, we found out that its acceleration strategy targets mainly protein BLAST, and even that with a setting to search for only exact seed matches during the scan stage ("threshold" parameter is off by default).

### D. *SOM*

The SOM is a neural network of $K$ neurons that are organized into a 2-D grid. Each neuron is defined by its $X,Y$ position in the map and by an $n$-dimensional vector assigned to it ("weight vector" or "code-vector"). The matrix of all $K$ weight-vectors forms the complete description of the SOM called the codebook. To train the SOM in the original "online" SOM formulation, an input pattern vector $x(t)$ is presented to the network at time $t$ and the weight vectors $W_k(t)$ are updated as a result. Initially all weight vectors are either assigned random values or linearly generated from the first two PCA eigen-vectors. First, the distances between the presented input vector and all weight vectors are computed using the following Euclidean distance metric equation:

$$d_k(t) = \|x(t) - W_k(t)\|^2 \quad (1)$$

Then one neuron called the Best Matching Unit (BMU) is selected according to the criterion below:

$$d_c(t) = \min_k d_k(t) \quad (2)$$

Ties are broken with a random selection. Next the weight vector of the BMU and its neighbors are adjusted toward the input pattern according to the following equation:

$$W_k(t+1) = W_k(t) + \alpha(t) h_{ck}(t)[\mathrm{x(t)} - W_k(t)], \quad (3)$$

where $0 < \alpha(t) < 1$ is the learning-rate factor that decreases monotonically with time. The neighborhood function $h_{ck}(t)$ is a kernel function that rapidly decreases for neurons far away from the BMU in grid coordinates. Often the Gaussian is used for defining the neighborhood function:

$$h_{ck}(t) = e^{\frac{-\|r_k - r_c\|^2}{\delta(t)^2}}, \quad (4)$$

where $r_k$ and $r_c$ stand for the coordinates of the nodes, $k$ and $c$ in the SOM map. The width $\delta(t)$ of the neighborhood function monotonically decreases as iteration goes from a value no less than half of the largest diagonal of the map to a value equal to the width of a single cell.

In an alternative "batch" SOM training, the weight vectors are updated all at once at the end of a learning period after seeing a set of training vectors. The new weights are calculated using:

$$W_k(t_f) = \frac{\sum_{t'=t_0}^{t'=t_f} h_{ck}(t')x(t')}{\sum_{t'=t_0}^{t'=t_f} h_{ck}(t')}, \quad (5)$$

where $t_0$ and $t_f$ represent the beginning and the end of the current epoch, respectively. The BMU is selected based on $W_k(t_0)$.

Thus, unlike the online version, the batch algorithm is not influenced by the order in which the input vectors are presented.

### E. Existing parallel implementations of SOM

Parallelizing SOM in a batch formulation is fairly straightforward. The right side in Eq. 5 can be computed in parallel by either splitting the work along the input patterns, or the weight vectors, or both. Several publications exist that look at various aspects of doing it, such as [10], [11]. However, we could not find an actual parallel implementation that would have been made available for public use.

The time needed to train an SOM grows linearly with the dataset size. It also grows linearly with the number of neurons in the SOM. Although various spatial indexing data structures can significantly speed up the search for BMU in low-dimensional cases, no efficient solution exists for high-dimensional inputs such as our bioinformatics application of polynucleotide frequency vectors. PCA transformation of input vectors have been reported to help in stopping the distance comparisons earlier for each pair of vectors, but PCA is itself expensive for large datasets.

Thus, the sequential algorithm quickly becomes prohibitively slow for large input datasets that often arise in biological applications. This is especially true when large SOMs (more that 50x50 code vectors) are trained, which has been shown to be important to observe SOMs with the "emergent" properties [12].

## III. Parallel implementations with MapReduce-MPI

### A. BLAST

Within a fairly flexible MapReduce-MPI library, there are multiple ways for a programmer to structure the computation. In the very general terms, the user has to first define *map()* function as well as the work units that will be passed to *map()*, one unit per each invocation. The MPI process ranks make a collective call to a method of the MapReduce global object, that takes care of splitting all work units among ranks, delivering the work units to each rank and locally calling *map()* on them.

Each *map()* call emits key-value pairs, which are grouped into key-multivalue pairs with unique keys, and redistributed in a balanced way between ranks by a collective *collate()* call implemented by the library.

After that, a user-defined *reduce()* function can be called locally on each rank once for each key-multivalue pair, and can again emit key-value or key-multivalue pairs. Multiple iterations of MapReduce can be executed with the same or different mappers and reducers.

In our implementation of BLAST, we define a work item as a tuple that combines several query sequences ("query blocks") with one database partition (Fig. 1). The database partitions are created by running the standard NCBI BLAST tool formatdb on the entire database in FASTA format. Formatdb creates the DB partitions in a two-bit encoded format that is optimized for scanning with NCBI BLAST.

The query blocks are created before executing our MPI process by splitting the entire query set into multiple FASTA files of a specified target size each. The work items then become pairs *<query file name, DB partition name>*.

We use a run-time option of MapReduce-MPI that instructs it to use the process with rank 0 as a master that distributes work units to the remaining ranks ("workers") in a load-balanced way, such that each worker is kept occupied as long as there are remaining work units. This is especially important for an algorithm like BLAST which is characterized by a highly non-uniform and unpredictable execution time depending on each query.

The *map()* function uses high-level NCBI C++ Toolkit API calls to initialize both the query input and the DB input objects and to execute BLAST search for sequences from a given query block against a given DB partition. The DB object is cached between *map()* invocations on a given rank, and only re-initialized if the different DB partition is required.

As it is usually done in the DB-split BLAST computations, the DB length is overridden in the BLAST call to be the entire length of the DB instead of the length of the current partition.

The map() calls emit key-value pairs where keys are the query IDs, and values are High-Scoring Pairs (HSPs, or "hits") found for the query in the given DB partition.

The *collate()* call results in hits from all DB partitions grouped together for each key (query ID). The library uses a hash value computed on each key to assign keys to the process ranks, and to group the values by keys. This takes advantage of the large communication bandwidth available on HPC clusters.

The local *reduce()* call sorts each query hits by the E-value, selects the requested number of top hits if such cutoff is specified by the user and appends hits to the file that is owned by each rank.

Thus, the results of the computations are in a set of files, one per each MPI rank, with the hits for each query located in only one file, maintaining the original order of the queries and sorted by the E-value within each query.

In our experience, it is rarely needed for the practical downstream analysis of the large-scale BLAST searches to have the results merged into a single file.

In order to process arbitrarily large collections of the queries, we employ multiple iterations of the above MapReduce protocol within the same MPI process by looping over the consecutive subsets of the entire query set. This is done to control the size of the intermediate key-value dataset that has to be kept in the collective memory of the process ranks during each MapReduce cycle. Although the MapReduce-MPI library will transparently use file system paging when the working set size grows beyond a pre-defined limit ("out-of-core processing"), the performance will suffer, especially on typical cluster architecture that has no locally attached user scratch space on the compute nodes.

We used the release of the MapReduce-MPI library from August 26th 2010, and the release of NCBI C++ Toolkit from June 15th 2010 (both packages did not employ the version numbering).

*Complexity analysis*. The behavior of the implementation is primarily defined by the underlying NCBI BLAST algorithm, which is $O(N \times M)$ where $N$ is the DB size and $M$ is the combined query size. In the case when the limit $K$ on the number of output hits per query is requested by the user, our matrix-split parallelization has to perform extra work at the alignment extension stages compared to a sequential version, because we need to pass $K$ hits from each DB partition, and then discard all but top $K$ from a combined set after *collate()*. The mpiBLAST essentially does *collate()* of the candidate hits after the seed scan stage, and only does the extension for top N hits overall. However, in many practical applications of BLAST, a user is interested in all hits that satisfy a given E-value cutoff. The maximum number of hits is then set to some large value to limit the output from a few exceptional cases when the query matches some highly redundant DB sequence fragments. Additionally, the low-complexity filtering is usually requested. Thus, in most cases the extra work built into our implementation has to be done only for a very small subset of query sequences and does not present an overall performance issue.

### B. SOM

The control flow diagram of our parallel SOM implementation is presented on Fig. 2. Here the work unit for the *map()* call is a block of input vectors. We are again using the master-worker execution mode, although in the case of SOM this is not as critical as it is for BLAST. The copy of the codebook is distributed with *MPI_Broadcast()* from the master to all worker nodes at the start of each epoch.

Additionally, each worker has its own copy of a new codebook, initialized to zero at the start of an epoch, plus a matrix of floating point scalars with the same shape as the codebook.

Each *map()* call uses these two arrays to accumulate contributions from the input vectors to both numerator and denominator of Eq. 5.

At the end of the epoch, a collective MPI_Reduce() call is used to sum all newly computed numerators and denominators, and the new codebook is computed as per Eq. 5, after which the new epoch begins. No *reduce()* stage is used in this program.

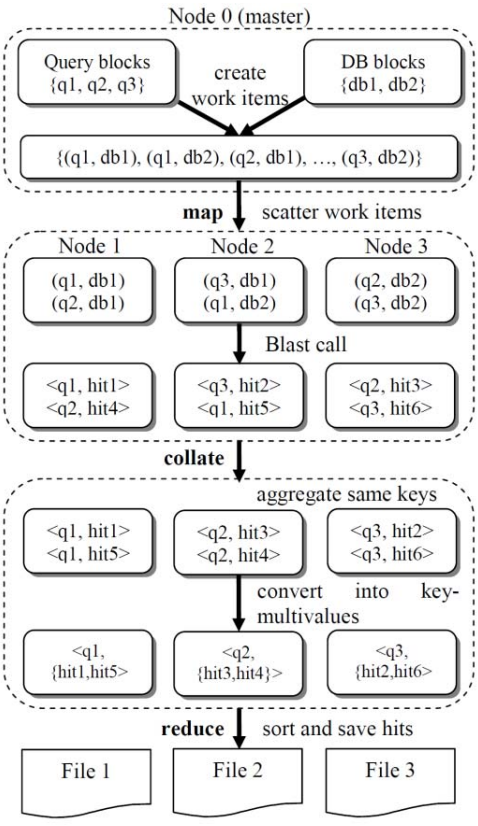


Figure 1. Control flow of the MR-MPI BLAST

The program takes the input vectors as a dense matrix saved on disk in the platform floating point representation, and uses memory mapped files to access them on the worker nodes, under an assumption that there is a shared file system mounted on the workers. Each work unit is thus described by

a pair of offsets in that memory mapped file. This allows processing input datasets larger than the available RAM size.

*Complexity analysis*. The complexity is O($N \times K \times M \times L$) where $N$ is the number of the input vectors, $K$ is the dimensionality of the vectors, $M$ is the number of cells (codebook vectors) in the SOM and $L$ is the number of training iterations (epochs).

## IV. Performance Analysis

We benchmarked both algorithms on TACC Ranger system. Each node has 16 AMD cores and 32 GB of RAM. The shared file system is Lustre, and no locally attached storage is available to the user programs. We used OpenMPI installed on the site with the default Infiniband networking transport and GNU C++ compiler.

Because the cluster always allocates entire nodes to the MPI job, our total core counts were always multiples of 16.

### A. BLAST

We benchmarked our parallel BLAST implementation in the nucleotide searches. Other types of searches such as BLASTP should not exhibit a fundamentally different behavior.

We downloaded the following nucleotide BLAST DBs from the NCBI FTP site in Jul 2010: RefSeq, NT, WGS and HTGS. The combined formatted DB had 109 partitions with on-disk size of 1GB each, and contained a total of 364 Gbp (Giga base-pairs) in 62 M sequences.

We have built the query dataset from those RefSeq sequences that belonged to bacteria, archaea, viruses, and unicellular eukaryotes, and shredded them into 400 bp fragments overlapping by 200 bp. This procedure simulated sequencing reads per our primary BLAST use case of the metagenomic taxonomic classification.

From 42M generated queries, we randomly selected 12K, 40K and 80K subsets for the performance benchmarking runs, the results of which are presented on Fig. 3.

The program was modified to exclude the hits of the RefSeq fragments against themselves from the output.

In the log-log scale used on the chart, the ideal scaling would be represented by a straight line. The results primarily show that the actual scaling behavior is sensitive to the total number and the size of the work units.

Specifically, the large core counts are only efficient for large input datasets, as it would be expected.

On Fig. 4 we present an alternative view of the scaling behavior for the two runs that used the largest dataset of 80,000 query sequences, but split into blocks of different size to form the work units.

For smaller core counts, the larger work units are more efficient. This is because the DB partitions have to be reloaded less often per single query sequence when larger chunks are used.

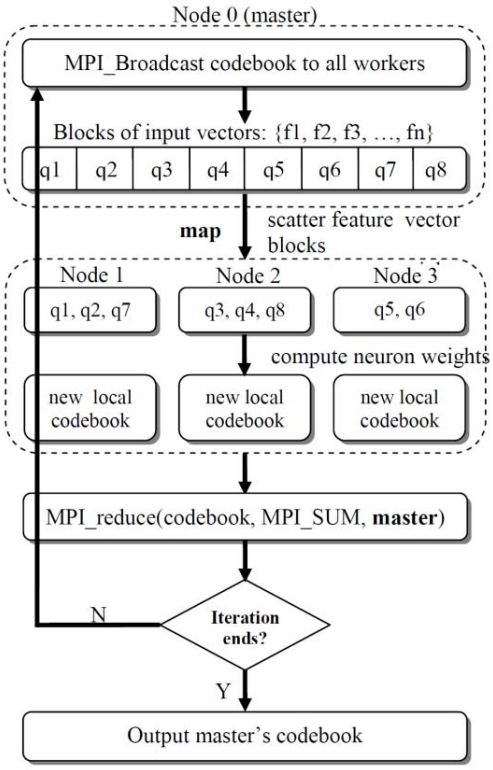


Figure 2. Control flow of MR-MPI Batch SOM

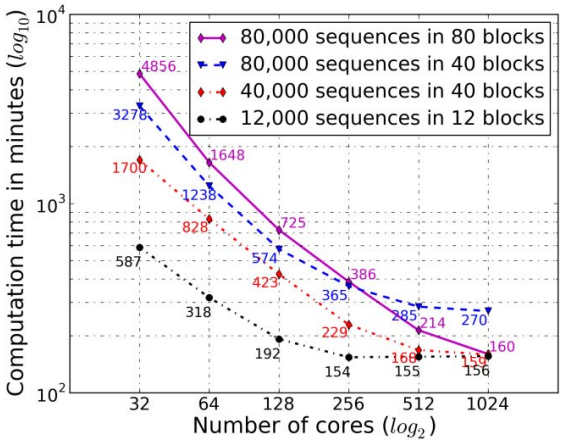


Figure 3. Scaling chart for MR-MPI BLAST showing process wall clock time at different total core counts in MPI job. Each data series corresponds to an indicated total number of query sequences split into blocks of 1000 sequences each, except for the series marked with blue rectangles that has 2000 sequences in each block. Each block, when combined with one DB partition, forms a sequential work unit for the MapReduce algorithm. The data point labels represent time in minutes.

For larger core counts, smaller query blocks lead to better performance because they result in more work units which is essential for better load balancing. This is because in that case we have less relative idling of cores at the end of each

processing stage when number of remaining work units becomes less than the number of cores.

Although for 1024 cores the efficiency is 95% of the efficiency observed for 32 cores, the efficiency is 167% for 128 cores (for 80,000 sequences in 80 blocks). This "superlinear" speed up at the medium core counts probably appears because all 109 1GB DB partitions begin to fit entirely into the combined RAM of the MPI process ranks (32 cores only have 64 GB of combined RAM). Thus, the memory mapped DB partitions stay cached in RAM after being loaded upon the first read access. The DB partitions still have to be reloaded occasionally to maintain the load balancing among ranks even at the larger core counts.

As the core counts grow larger, this speed-up due to RAM caching is eventually overwhelmed at a fixed query dataset size due to the idling of the cores. Specifically, for the series with 80 query blocks, the total number of work items is 80 query blocks × 109 DB partitions = 8720, which is 8.5 times the number of cores when 1024 core are used. If every work item was taking exactly the same amount of time to process, we could lose 15% of efficiency due to core idling at the end.

However, the BLAST search time can vary widely for specific query and DB sequences. Although mpiBLAST, for instance, uses a modified *formatdb* program to randomize the distribution of original DB sequences between partitions, we did not use this approach because we were interested in exploring the limits of the load balancing execution.

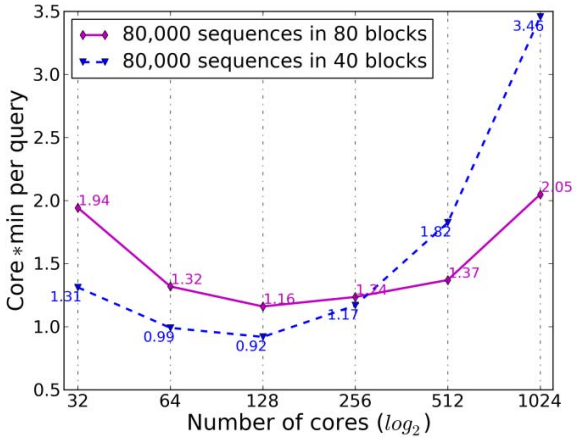


Figure 4. Scaling chart for MR-MPI BLAST showing the average number of wall clock core minutes spent per a single query sequence at different total core counts in MPI job

Thus, some combinations of the query blocks and DB partitions take much longer than others when all other work units have already been distributed in the *map()* stage. The entire MPI program then has to wait for that longest unit of work to finish, resulting in a delay above the above estimated 15%. This conclusion is supported by the fact the slowdown is more pronounced in the 40-blocks series, despite the fact that both series generate the same amount of key-value pairs, which then have to be exchanged in *collate()* and processed in *reduce()*.

We also benchmarked our application on a protein BLAST search. The query was represented by a subset of NCBI non-redundant environmental sequences (env_nr) containing 139846 proteins. The database was Uniref100 set from the UniProt collection, formatted into 58 partitions of 200,000 sequences each. The search was executed with the *E-value* cutoff of 10e-4.

As expected, the protein search demonstrated a very good scaling due to the considerably more CPU-bound nature of the protein search compared to the nucleotide search. This is because BLAST is able to detect the more remote homologies in protein space, and thus has to examine many more candidate matches per a given database set compared to the nucleotide space search. For example, the 1024 core run used only 6% more core*min per query compared to the 512 core run (294 min absolute wall clock time using 1024 cores).

On Fig.5, we show the average "useful" CPU utilization per core during the course of a protein BLAST run using 1024 cores. We define as the useful CPU utilization the ratio of user CPU time (as obtained with OS getrusage() call) to the wall clock time, both spend within each call to the NCBI BLAST search procedure. These values are summed over all calls taking place at any given moment and divided by the total core count allocated to the MPI program. The resulting value reflects the amount of time spent doing BLAST computations as opposed to either waiting for IO within the BLAST call or doing MapReduce book-keeping outside of that call. Ideal value of 1.0 would reflect running of the BLAST algorithm without the overheads of both BLAST IO and our parallel framework.

As we can see from Fig.5, protein BLAST scales very well. The tapering off at the end of the computation is due to cores idling without more workloads available to them. This is always compounded by the fact that BLAST search time for a given query sequence is highly variable and unpredictable.

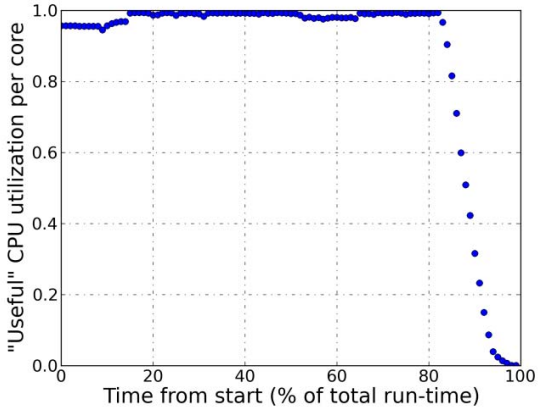


Figure 5. "Useful" CPU utilization per core during the course of the computation for the protein BLAST MPI run with 1024 cores. CPU user time used at any given moment within a BLAST call was divided by the

corresponding wall clock time, summed over all concurrent calls, and divided by a total number of cores allocated to the MPI program.

We executed the same protein BLAST search on a HTC cluster at the Venter Institute (JCVI). The search was controlled by a VICS workflow execution engine (unpublished internal software) that executed a matrix-split computation as a collection of 960 serial BLAST jobs followed by a few merge-sort and formatting jobs. The data files and intermediate results were stored on a shared Isilon storage cluster system.

Although a direct comparison of the absolute run-time is difficult, as the hardware at the JCVI is about two years younger than the hardware on Ranger, the user CPU utilization was similar to what we saw on Ranger with our own implementation. The longest VICS job took about the same wall clock time as our run at 1024 cores.

### B. SOM

For benchmarking the performance of our parallel SOM implementation, we generated 81,920 random vectors (the multiple of our core counts) of 256 dimensions each. Then, we trained a 50×50 SOM with different core counts in each MPI job (Fig. 6).

The implementation exhibited excellent linear scaling across all core counts with 96% efficiency at 1024 cores relative to the 32 core run.

Fig. 7 demonstrates that the implementation correctly clusters random RGB vectors, a visual test often used in evaluating the correctness of the SOM algorithm.

Fig. 8 as well demonstrates a well-defined U-matrix obtained by clustering 500-D random vectors.

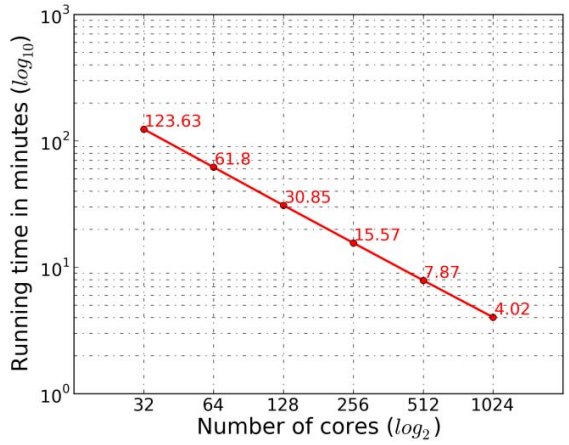


Figure 6. Scaling chart of MR-MPI Batch SOM algorithm with the input dataset of 81,920 random vectors of 256 dimensions. The work units for the MapReduce algorithm were blocks of 40 vectors. Work units of 80 vectors each produced the identical timings. The data point labels represent time in minutes.

## I. Conclusions and Discussion

We demonstrated that the MapReduce-MPI library can be successfully used to quickly parallelize a wide range of sequential algorithms in a way that is portable across typical HPC clusters. This ease of use is especially important in the domain of the bioinformatics applications.

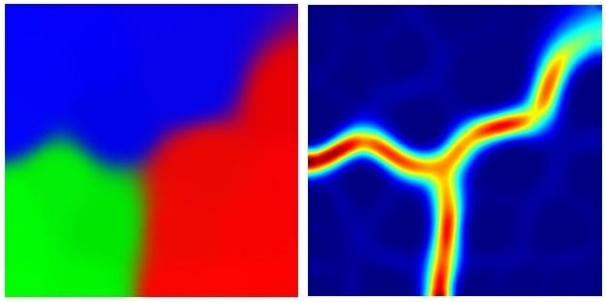

Figure 7. Clustering of input vectors viewed as RGB colors and U-Matrix of 50x50 SOM trained with 100 RGB feature vectors.

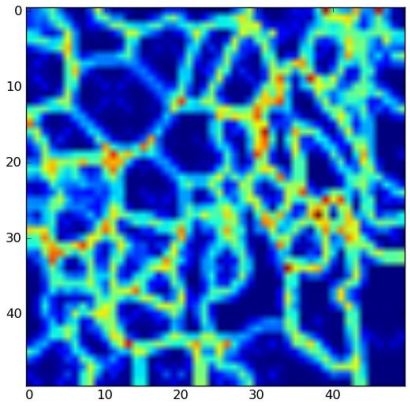


Figure 8. U-Matrix of 50x50 SOM trained by 10,000 random feature vectors with 500 dimensions.

Because NCBI BLAST is a highly complex application with a diverse range of use modes, fairly demanding I/O and non-uniform execution patterns, the efficiency of our parallelized implementation on high core counts requires sufficiently large input datasets as well as some tuning of the work distribution parameters.

The MapReduce framework takes care of a substantial part of the necessary book-keeping and data handling tasks. This leads to a very short and easily maintainable code that we have to implement.

We are currently working on improving the load-balancing properties of the implementation.

First, we are improving the location-aware work unit scheduler in order to distribute the work unit tuples to those ranks that have already been processing the same DB partitions in as many cases as possible. Improving the DB locality will in turn allow us to improve the load balancing by using smaller query blocks.

Second, we are eliminating the need to pre-partition the query dataset by building an index of sequence offsets in the

input FASTA file. This will allow selecting the size of the query blocks dynamically after the start of the program based on a small timing iteration at the beginning, thus eliminating the need for tuning by the user. This can be also used to make progressively smaller query chunks toward the end of each iteration and have a more uniform filling of the cores.

The really ground breaking parallel implementation of BLAST would be based on a global distributed index of the DB seeds, thus improving upon the linear complexity of the current implementations relative to the DB size. So far, this has been very challenging for the protein BLAST due to a large number of seeds that have to be checked in the index when a substitution matrix similarity threshold is employed. Recent efforts related to the use of the reduced protein alphabet in sequential BLAST searches [1] promise to overcome this problem. This would require a fairly deep re-implementation of the algorithm and break the compatibility with NCBI BLAST.

The SOM algorithm, on the other hand, is simple and maps very well to the coarse-grained parallelization model of the MapReduce when the direct MPI calls are used to distribute and combine the codebook updates. We intend to use our SOM implementation to visually explore the relationship between the metagenomic sequences and the universe of taxonomically characterized database sequences in the tetranucleotide composition space with the dataset sizes that so far have been out of reach.

## REFERENCES


[1] "SWIFT." [Online]. Available: http://omics.informatics.indiana.edu/SWIFT/. [Accessed: 24-Dec-2010].

[2] I. Sfiligoi, D. C. Bradley, B. Holzman, P. Mhashilkar, S. Padhi, and F. Würthwein, "The Pilot Way to Grid Resources Using glideinWMS," in *Computer Science and Information Engineering, World Congress on*, vol. 2, pp. 428-432, 2009.

[3] S. Plimpton and K. Devine, "MapReduce-MPI Library." [Online]. Available: http://www.sandia.gov/~sjplimp/mapreduce.html. [Accessed: 29-Nov-2010].

[4] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403-410, Oct. 1990.

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

[6] C. Camacho et al., "BLAST+: architecture and applications," *BMC Bioinformatics*, vol. 10, no. 1, p. 421, 2009.

[7] O. Thorsen et al., "Parallel genomic sequence-search on a massively parallel system," in *Proceedings of the 4th international conference on Computing frontiers*, pp. 59-68, 2007.

[8] C. Oehmen and J. Nieplocha, "ScalaBLAST: A Scalable Implementation of BLAST for High-Performance Data-Intensive Bioinformatics Analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 8, pp. 740-749, 2006.

[9] "DeCypherBLAST Solution." [Online]. Available: http://www.timelogic.com/decypher_blast.html. [Accessed: 24-Dec-2010].

[10] B. Silva and N. Marques, "A Hybrid Parallel SOM Algorithm for Large Maps in Data-Mining."

[11] I. Valova, D. Beaton, D. MacLean, and J. Hammond, "NIPSOM: Parallel Architecture and Implementation of a Growing SOM," *The Computer Journal*, vol. 53, no. 6, pp. 753 -771, Jul. 2010.

[12] A. Ultsch, "Emergence in Self Organizing Feature Maps," University Library of Bielefeld, 2007.