2016 IEEE International Parallel and Distributed Processing Symposium Workshops

A memory and time scalable parallelization of the Reptile error-correction code

Vipin Sachdeva
Computational Science Center
IBM T. J. Watson Research
Cambridge, MA
Email: vsachde@us.ibm.com

Srinivas Aluru
Georgia Institute of Technology
Atlanta, GA
Email: aluru@cc.gatech.edu

David A. Bader
Georgia Institute of Technology
Atlanta, GA
Email: bader@cc.gatech.edu

Abstract— **This paper details a distributed memory implementation of Reptile, a scalable and accurate spectrum based error-correction method. Reptile uses both k-mer and adjoining k-mers (called tiles) information along with the quality scores of bases to correct substitution-based errors from next generation sequencing machines. Previous approaches to parallelize Preptile have replicated the spectrums on each node which can be prohibitive in terms of memory needed for huge datasets. Our approach distributes both the k-mer and the tile spectrum amongst the processing ranks, relying on message passing for error correction. This allows hardware with any memory size per node to be employed for error-correction using Reptile's algorithm, irrespective of the size of the dataset. As part of our implementation, we have also implemented several heuristics which can be used to run the algorithm optimally based on the advantages of the hardware used. We present our results on IBM's BlueGene/Q architecture for the E.Coli, Drosophila and the human datasets showing excellent scalability with increasing number of nodes. Using 256 nodes of BlueGene/Q, we are able to error correct E.Coli and Drosphila datasets in less than 200 seconds and 600 seconds respectively. The human dataset consisting of 1.55 billion reads is corrected in a little more than two hours using 1024 nodes of BlueGene/Q. All three datasets are corrected with Reptile's memory intensive algorithm with less than 512 MB per process.**

I. Introduction

Next-generation sequencing (NGS) technologies have steadily replaced Sanger sequencing as the preferred method of genome sequencing [1]. They provide far more information than the Sanger method, at a much lower cost per DNA base. However, NGS methods suffer from two critical disadvantages: they produce significantly shorter pieces of genome called reads that have to be assembled together, and are more error-prone than the Sanger method.

Error correction for datasets from NGS technologies is one of the most important steps for correct assembly results. The errors associated with NGS technologies can be classified into substitution errors which happen when a single base is altered to a different base, and insertion-deletion errors in which the entire region of a genome consisting of several characters has been changed. Correction of these errors greatly enhances the performance of many subsequent steps using this data such as de novo genome and RNA sequencing, re-sequencing and metagenomics besides others [2]. Error correction methods are broadly distinguished into k-spectrum, suffix tree based and multiple sequence alignment methods [3]. K-spectrum methods decompose reads into the set of all k-mers. In this approach, erroneous k-mers are converted to the consensus k-mer (or the highest frequency k-mer). Suffix tree based methods generalize the k-mer based approach and multiple sequence alignment methods identify reads co located on the unknown reference genome by using k-mers as seeds. Please refer to [3] for a comprehensive evaluation of error correction of reads.

Reptile [4] is a scalable substitution error correction method that has been shown to outperform several other error correction methods [3]. Reptile is broadly based on the k-mer spectrum approach, but also uses adjoining k-mer information (called tiles) to reliably suggest error corrections. Since Reptile relies on both k-mer and tile spectrum for more accurate error correction, there are memory limits on the size of datasets it is able to error-correct.

Previous approaches to parallelize Reptile [5] [6] have relied on replication of k-mers and tile spectrum on every process or node. For large datasets, the replication of spectrum on every node leads to very high memory requirements per node. Furthermore, with the exponential increase in the size of the datasets from NGS technologies, replication of spectrums on every node or process might not be feasible. Error correction of datasets from RNA sequencing, population genetics and metagenomics can lead to datasets in hundreds of gigabytes, often leading to k-mer spectrum sizes of over a terabyte [7]. In such cases, replication of the k-mer and tile spectrum can be prohibitive in terms of memory needed per node.

In this work, we detail a distributed memory approach to parallelization of Reptile which distributes both the k-mer and tile spectrum amongst the processing ranks. Our approach allows hardware (with any memory size per node) to be employed for error correction of any dataset using Reptile's algorithm. The only requirement is that a minimum number of nodes is needed such that the combined memory of all the nodes exceeds the storage of the entire k-mer and tile spectrum.

During error correction of a read, it is expected that a rank will need the k-mers and tiles it does not store in its own local memory. We rely on message passing for such k-mers and tiles. The goal of our work is to enable Reptile to run on

/16 $31.00 © 2016 IEEE
DOI 10.1109/IPDPSW.2016.59

IEEE computer society

any hardware irrespective of the memory per node or the size of the dataset. In addition, our approach also shows excellent scalability as the number of nodes are increased (both in terms of time and memory). Besides a distributed k-mer and tile spectrum, our implementation also has support for numerous heuristics that allow parallel Reptile to use the features of the underlying hardware efficiently.

We present the results of our approach on IBM's BlueGene/Q architecture [8], where each node has 16 GB of memory; using 32 ranks per node, we demonstrate that we can complete error correction of E.Coli, Drosophila and human datasets with less than 512 MB per process. Our results show excellent scalability with increasing number of nodes for all three datasets. Using 256 nodes of BlueGene/Q, we are able to error correct E.Coli and Drosphila datasets with read coverages of 96X and 75X in approximately 200 and 600 seconds respectively. The human dataset consisting of 1.55 billion reads with a read coverage of 47X is corrected in a little more than two hours using 1024 nodes of BlueGene/Q.

In Section II, we summarize other error-correction applications and detail the algorithm behind Reptile, along with a summary of the previous parallelization approaches of Reptile. We also contrast our approach to the previous work in this section. Next, in Section III we give a detailed description of our parallelization scheme and its implementation. Results are given in Section IV, and we conclude in Section V with an outlook on future improvements.

II. Related Work

There are several error-correction algorithms and implementations; among k-mer spectrum based methods, besides Reptile, Quake [9] uses a maximum likelihood approach incorporating quality values. For suffix-tree approaches, SHREC [10] was the first implementation to use a generalized suffix trie as the data structure employed for error correction. Among multiple sequence alignment methods, ECHO [11] performs error correction by finding overlaps between the reads. Several authors have published on parallelization of error-correction approaches; many of the approaches implement shared-memory parallelization only [12] [13] [14]. Only a few implementations exist for distributed memory architectures. This includes Reptile parallelization which we discuss in the next subsection. DecGPU [15] uses both GPUs and distributed memory CPUs to perform error correction.

There has been prior work to use distributed hash tables with messaging to tackle memory-intensive tasks in computational biology such as assembly [16] [17]. A viable alternative to message passing in assembly algorithms is to use a global address space using Unified Parallel C [18].

A. Reptile algorithm description

Reptile is a spectrum based substitution error-correction method; instead of only relying on the k-mer spectrum (consisting of k-mers), Reptile also constructs and subsequently uses another spectrum consisting of tiles. Tiles can be defined as a sequence of two or more k-mers with a fixed overlap length between the k-mers. During error correction, both the k-mer and tile spectrum are used for error correction of a read. Spectrum-based methods often correct k-mers in a read with their Hamming distance neighbors; a Hamming distance neighbor of a k-mer is defined as the number of positions the two k-mers differ. However, this reduces exactness when an erroneous k-mer has to be corrected since there are multiple candidates for the k-mer. To avoid this scenario, Reptile corrects tiles instead of k-mers. Since a tile has almost twice the character count as the k-mer, error correction at the tile level has far fewer candidates than at the k-mer level. Using the tiles leads to more accuracy in error-correction [3]. The drawback of using both the k-mer and the tile spectrum is that Reptile's algorithm is memory-intensive as it keeps two spectrums in its memory. Please refer to [4] for further details.

B. Previous Reptile Parallelization Approaches

Previous approaches to parallelize Reptile have either replicated k-mer and tile spectrum on each process or on each node. Both approaches limit the hardware on which parallel Reptile can be run; for example in the work by [5], the spectrums were replicated per process. The approach by Jammula et al. [6] improved upon the original existing parallel Reptile implementation by replicating the spectrum on every node. Since this approach is an improvement over the work by Shah et al. [5], we contrast our approach with this work.

- The k-mer and the tile spectrum were replicated per node compared to the previous approach of replication per process. Multiple threads of each node share the k-mer and tile spectrums during the error correction phase.
- K-mer and tile spectrums are stored as sorted lists with look-up operations involving repeated binary searches over the spectrum. A cache-aware layout of k-mer spectrum was presented which lowered the search time from the original $O(log_2 N)$ to $O(log_{(B+1)} N)$ where B represents the number of elements that can fit into a cache line.
- A dynamic work allocation scheme that depends upon a global master which coordinates the entire work allocation mechanism and a local master that is responsible for getting work from the global master. The actual error correction is performed by worker threads running on the node who fetch chunks of sequences from the work-queue.

Our approach differs from the the previous work in the following three respects:

- Instead of replicating the entire dataset on a node or a rank, we distribute the k-mer and tile spectrum amongst the processing ranks. This lowers the memory footprint significantly and we have shown that we can perform error correction with a memory footprint of less than 512 MB per process even for the human dataset comprising of over 1.55 billion reads. This implementation also provides highly scalable error-correction times. Thus, our approach provides a memory and time scalable implementation to spectrum based error correction. Besides distributing k-mers and tiles amongst the processing ranks, our implementation also has the ability to replicate the k-mer and tile spectrum on every node. This mode does not require any communication between the processes during error-correction and is designed to be run on machines where the entire spectrum can be replicated on every node. We have also implemented several heuristics which can be employed based on the traits of the datasets and the hardware.

- We store the k-mer and tile spectrum in hash tables instead of arrays; this prevents any need for sorting the arrays or for repeated binary searches. We use separate hash tables for the k-mer and the tile spectrum. The hash tables are created in parallel during the k-mer construction phase.
- We rely on static work allocation for load-balancing. Our approach does not rely on a master-slave policy, instead it redistributes sequences to the processing ranks. We present our results with and without this load balancing policy in Section IV, and show that such a static allocation policy balances the load very effectively.

III. Parallelization of Reptile

Related work shows that parallel implementations of Reptile so far have only been run in modes requiring replication of k-mer and tile spectrum. Our approach which differs markedly is detailed below; please note that while most of the changes are detailed with respect to the k-mer spectrum, the same changes also apply to the tile spectrum.

I As a first step, the file consisting of short reads is read in parallel by each rank. The input to parallel Reptile consists of a configuration file, which specifies the fasta file and the quality file to be used for the error correction. At this point, Reptile is not capable of reading the fastq format. The fasta file consists of the sequences along with the sequence names; the names have been pre-processed to be sequence numbers (in ascending order beginning with number 1). The second file to be read is the quality score files, which has information on the quality score associated with every base of the sequence and the sequence number as well. Both files are read in parallel; each rank computes its subset of the reads whose size is simply the file size divided by the number of ranks. The subset of reads are processed beginning with an offset from the start of the file. The offset is based on the rank. Each rank starts reading the fasta file from this offset and records the starting sequence number. It then looks up the same sequence number in the quality score file as well to ensure that the quality scores corresponding to the same set of reads as the fasta file is processed. Similarly, the end sequence number is also computed. Each rank is responsible for the set of reads corresponding to the starting sequence number up to the ending sequence number. This subset of reads is read in chunks by each rank; the chunk size is also defined in the configuration file.

II In this step, each rank builds a k-mer and a tile spectrum from its set of reads. The k-mer spectrum is represented by key-value pairs with k-mer ID as the key and the count of the k-mer as the value. The k-mer ID is a number constructed from the characters of the sequence. The tile spectrum is similarly represented except the tile ID is a long integer as the number of characters of the tile can be up to $2k$ where k represents the number of characters of the k-mer. The k-mer and tile spectrum are stored in separate hash tables on each rank. With each read, the k-mers and tiles corresponding to the reads are processed, and added to the k-mer and tile hash tables respectively. During the current step, the k-mer and tile spectrum are separated into the *hashKmer* and *readsKmer* hash tables, and the *hashTile* and the *readsTile* hash tables respectively. Each k-mer (and tile) are defined to have an owning rank; the owning rank in our implementation is defined as the rank p (out of the number of ranks np) for which $hashFunction(kmer)\%np == p$ (and similarly for the tile). The rank adds the k-mer it has processed to the *hashKmer* if it is the owning rank, else the k-mer is inserted into the *readsKmer* hash table. The process continues until the entire allocated subset of reads are processed by the rank. Once this phase is over, the rank stores the k-mers extracted from its reads in either a hash table consisting of the k-mers it owns (*hashKmer*) or a hash table consisting of the k-mers (*readsKmer*) it does not own (and similarly for the tiles). As can be observed, this is an embarrassingly parallel phase requiring no communications amongst the ranks.

III After the previous phase, each rank has two hash tables, each for the k-mer and tile spectrum. However, it can be observed that no rank has the true global counts for the k-mers and the tiles in their hash tables. This is because each k-mer might exist on multiple ranks (as part of their *readsKmer* hash table), besides the owning rank. The counts of the k-mer on the owning rank thus need to be added to the counts of the same k-mer that exist on every other rank to get the true global count of the k-mer. This next phase thus requires communication such that all k-mers and tiles (along with their true global counts) exist on only the owning rank. For this step, each rank processes each k-mer in its *readsKmer* hash table; for each k-mer, the owning rank is computed ($hashFunction(kmer)\%np$) and the k-mer and its local count is placed into a vector for the owning rank. This is then followed by an *MPI_alltoallv* communication that sends a vector of k-mers and their counts to their owning ranks. Each rank then processes the k-mers it has received from the other ranks. This step involves adding to the count of the k-mer if the k-mer exists in the hash table, or adding the k-mer (along with its count) if it does not exist. Following this, each rank now has a hash table of k-mers it owns, with the true global counts (or frequencies) of these k-mers.

Finally, based on the threshold set in the configuration file, k-mers and tiles below a threshold are subsequently removed from their hash tables by the ranks. A memory-efficient alternative to this step is usage of a Bloom filter [18]. Note that the with these steps, each rank only retains now a subset of the k-mer and tile spectrum with their true global counts; the storage requirements of each rank for the k-mer (and tile spectrum) now depend on the hashing function. With the inbuilt hashing function of the C++ standard templates library, we have found the number of k-mers and tiles to be remarkably consistent across the total number of ranks. The total time taken up by the the steps I-III is printed as the k-mer construction time in our execution; besides the error correction times, we also show the k-mer construction times in Section IV. Figure 1 shows the steps of the k-mer construction including parallel reading, construction of the hash and the reads k-mer tables followed by the collective communications for a hypothetical execution of 128 ranks and k-mer size of 3. These steps are similarly executed for tiles as well.

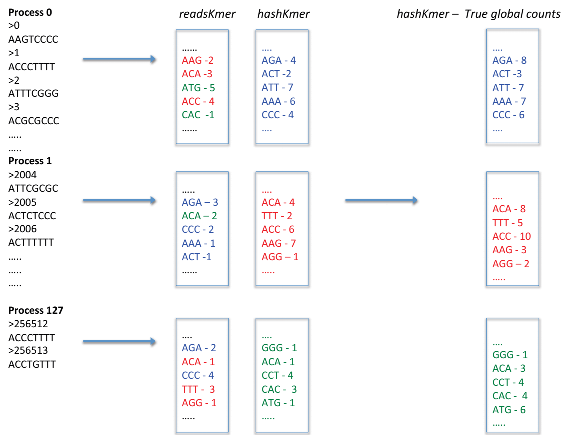

Fig. 1. A diagram representing the steps of the k-mer construction for a hypothetical execution of 128 ranks and a k-mer size of 3

IV After the k-mer construction steps, each rank now has a hash table of the k-mers and tiles it owns (and an optional hash table of k-mers and tiles that it has processed from its reads dataset). Once all the ranks have finished the two steps, the error correction step can now begin. For our experiments, the short reads are again processed from the file. This is because the total memory consumed by storing the reads will increase the memory footprint significantly. Most of our experiments are run with only 512 MB per rank, thus storing the reads is not a feasible option for us. The error correction of each read requires a set of k-mers and tiles (and their Hamming distance neighbors). Each rank at the beginning of this step forks two separate threads - one thread is responsible for the error correction of the reads in its part of the file, while the other thread acts as a communication thread. If a rank during error correction does not have a k-mer (or tile), it first finds out if it is the owning rank. In case the processing rank p is the owning rank, this implies that the k-mer or tile does not exist; in case the processing rank is not the owning rank, it looks up its *readsKmer* hash table (in case of the corresponding mode of execution). If the k-mer is not found, it sends a message to the owning rank, requesting the count of the k-mer or tile. The communication thread of each rank probes any incoming messages – based on the probe, it first finds out the nature of the request (if it a k-mer or a tile lookup). The thread then looks up the corresponding hash table (k-mer or tile) based on the request) and sends the appropriate response. The response is either the count of the k-mer or tile or a response like (-1) implying that the k-mer or tile does not exist. If a k-mer or tile does not exist at its owning rank, it can be inferred that the k-mer or tile does not exist at all in the entire k-mer spectrum.

Each rank can continue with the error correction of its reads subset using the strategy above; the communication thread of the rank responds to incoming requests, while the non-communication thread continues with the error correction of the rank's subset of reads using Reptile's algorithm. Once all the ranks have finished their error correction step, each rank shuts down its communication threads and outputs the reads it has corrected.

A. Load Balance Through Randomization

One issue we faced with the strategy above is load imbalance which Jammula et al. [6] have also explained in their work. This issue is because in many cases, the errors appear localized in several parts of the file. Since the reads in the file are divided up into chunks amongst the ranks, this leads to certain ranks having considerably more erroneous sequences compared to the other ranks. Since the work done with erroneous sequences is much higher than the other ranks, this leads to load imbalance between the ranks. In Section IV we show the variation in times between the slowest and the fastest ranks, along with a breakdown of their times for 128 ranks processing the E.Coli dataset. While Jammula et al. [6] have relied on a dynamic load balance approach based on a global master, a local master and worker threads, our approach of load balance is a static scheme. As we noticed that most of the load imbalance is caused due to errors being localized in parts of the file, a "randomization" of the entire file might remedy the problem.

Our strategy is for a sequence to be processed by a rank only if it is the owning rank; a sequence is designated to be owned by a rank p if $hashFunction(seq)\%np == p$ (similar to our definition for k-mers and tiles). Therefore, in addition to the Step I above, we also perform additional steps for load-balancing: after each rank has read their batch of short reads (or sequences), they find out the owning rank for each sequence in their batch of reads. The sequences are then placed in separate buckets corresponding to the owning ranks. Subsequently, a collective communication *MPI_Alltoallv* is performed; each rank then processes the sequences for which they are the owning rank. This hashing of sequences has the same effect as the "randomization" of the file might have.

B. Heuristics for Efficient Parallelization

Besides the core steps above, we have also implemented heuristics to be employed for efficient execution based on the dataset and the architecture. The primary purpose of these heuristics is to lower the runtime or memory footprint based on the hardware being tested and the requirements of the dataset. We show the results for all the heuristics in Section IV for the E.Coli dataset for 32 nodes only. Since we had limited access to higher number of nodes, we are unable to present the results of the heuristics for higher node counts and other datasets. However, the results for E.Coli give us an understanding of the effectiveness of these heuristics. We give a brief description of each the heuristics below.

- **Universal**: We rely on message passing between the rank's communication threads to get the counts of k-mers and tiles. Based on the request (k-mer or tile), the sending rank puts different tags on the messages. The receiving rank probes any incoming messages (with any sending rank or any tag) and subsequently based on the tag, looks up the hash table corresponding to either the k-mers or the tiles. In *universal* mode of execution, the message is itself a structure with the tag included as part of the message (and the k-mer ID and the tile ID). The receiving rank now does not have to probe the message for the tag, but accepts any message; once the message is received, it looks up the tag (which is part of the message received) to find the nature of the request and subsequently does a lookup of its hash tables. This increases the size of the message but makes the call to MPI_Probe unwarranted.
- **Read K-mers/Tiles**: Each rank retains the k-mers it owns. Besides the owned k-mers, it can also have k-mers and tiles from its own set of reads. To implement this heuristic, an additional collective communication step is needed where each rank sends the k-mers it does not own to the owning rank, requesting the global count for the k-mer. This is again implemented as an *MPI_alltoallv*, where each rank creates a vector of k-mers to be sent to the owning ranks. Thus, in this mode, each rank has two hash tables each for k-mer and tiles (or four in total). If a k-mer is not found in the *hashKmer*, it is looked up in the *readsKmer* and then a message is sent to the owning rank. This increases the local lookup time, but can potentially decrease the time spent in communication.
- **Allgather k-mers/tiles/both**: This heuristic replicates the entire k-mer spectrum and/or tile spectrum on every node. This mode is designed to be used on machines with enough memory to keep either or both the spectrums. This mode does not employ any message passing between the ranks during the error-correction step. The only communication in the entire execution is the collective communication calls to exchange the owned k-mers and tiles amongst the ranks. As expected, this mode decreases the runtime significantly, while increasing the memory footprint.
- **Add remote k-mer/tile lookups**: This mode adds any k-mer or tile lookups from the remote ranks; once a k-mer or tile count is received from the owning rank, it adds those k-mers to the local *readsKmer* hash table. This mode can only be run with the *read kmers* mode as the remote k-mers and tiles are added to the *readsKmer* and the *readsTile* hash table. The lookup strategy follows the *read kmers* heuristic; a k-mer is first looked up in the owned k-mers hash table *hashKmer*, followed by the reads hash table *readsKmer* and finally requested from the owning rank. This mode will be useful if the k-mers or tiles needed from remote ranks, will be needed in the future.
- **Batch Reads Table**: In this mode, the reads table which is maintained during the k-mer construction phase (and optionally during the error correction phase), is kept to a minimum size with an increased communication overhead. Each rank reads a chunk of their subset of reads (specified by the chunk size in the configuration file). This mode performs Step III of the parallel algorithm after each batch of reads instead of performing it in the end once all the ranks have processed their set of reads. In this mode, after all the processes have read their batches, the processes synchronize and complete a *MPI_alltoallv* operation to assign the k-mers and tiles to their owning ranks from the batch of reads just processed. Each rank subsequently processes the set of k-mers and tiles it has received and adds them to their hash tables for the k-mers and the tiles. Following this, the reads hash table is emptied out before the next batch is read. Thus, the size of the reads hash table can be kept to a minimum by varying the chunk size as the reads hash tables only contain k-mers and tiles from a single chunk than from all the chunks. One point of observation is that different processes might have been allocated slightly different number of batches; thus, before this step, a *MPI_Reduce* is carried out to find the maximum number of batches amongst all the processes. Each process thus continues this process for the maximum number of batches even though it might have exhausted its set of reads. This is because other ranks might still be continuing to process their set of reads and require every rank participation in the *MPI_alltoallv* operation. This mode lowers the memory consumed with an increase in the communication overhead. However, since the k-mer and tile construction time is a negligible percentage of the total time for error correction, the overhead is not substantial.

Genome	Number of reads (millions)	Length (chars)	Genome Size	Read Coverage
E.Coli	8874761	102	$4.6 * 10^6$	96X
Drosophila	95674872	96	$1.22 * 10^8$	75X
Human	1549111800	102	$3.3 * 10^9$	47X

TABLE I
E.COLI, DROSOPHILA AND HUMAN DATASETS USED FOR EXPERIMENTATION

IV. RESULTS

In this Section, we discuss the results of our implementation for 3 datasets - E.Coli, Drosophila and human dataset. We have followed exactly the same methodology of preparing the datasets as [6]; our datasets are very similar to Jammula et al. with minor differences being introduced in the conversion of the downloaded fastq file format to separate fasta and quality score files which are needed by Reptile. Table I shows the details for the datasets including the number of the reads and the length of the reads. The smallest dataset is the E.Coli dataset with less than 9 million reads, with the human dataset roughly 1500 times the size of the E.Coli dataset. The read coverages for all the genomes have been calculated as (Length*Number of Reads)/(Genome Size)

We have tested our implementation on IBM's BlueGene/Q architecture. Blue Gene/Q (BG/Q) is the third generation of highly scalable, power efficient supercomputers in the IBM BlueGene line, following Blue Gene/L and Blue Gene/P. The BG/Q SoC has 16 cores for user code, and a 17th core is reserved for use by the system software. Each core has four hardware threads. The 4 threads are simultaneously multi-threaded (SMT) threads. Each node has a wake-up unit that allows SMT threads to "sleep" waiting for an event; this allows faster OpenMP work handoff and lowers messaging latency. The 64-bit, in-order, PowerPC cores run at 1.6 GHz. 32 compute nodes are electrically interconnected to form a 2x2x2x2x2 grid on a node card. 16 node cards comprise a 512-node midplane and two midplanes stack vertically to form a 1024-node rack, with electrical links within midplanes and optical links between midplanes.

Our BlueGene/Q hardware has 16 GB of memory per node. For most of our experiments, we have decided to run 32 ranks per node, with each rank running 2 threads (communication and error correction) during the error correction phase. During error correction for most of our experiments, we run 64 threads per node which is the maximum possible number of threads on the BG/Q node. Using 32 ranks only allows each rank to have 512 MB per process; this includes memory for both the messaging buffers and the application's data structures. Using multiple ranks per node also gives us a benefit: it allows any communication between the ranks on the same node to use the shared memory on the node (and not use the messaging interface). Considering the nature of the problem, most of our effort has been to lower memory footprint per process. We have not made an extra effort to optimize the communication between the ranks; this has been done purposefully to observe if we can get good results without any tuning that may be specific to the datasets.

To find the effect of running multiple ranks per node on runtime, we varied the number of ranks per node from 8 to 32. Figure 2 shows the results for the E.coli dataset using 128 ranks for this run; the number of ranks per node are varied from 8 to 32. Thus, the number of nodes are varied from 16 to 4 for this experiment. As can be seen from Figure 2, the time taken using 32 processes per node is slower than using 8 or 16 processes per node by almost 30%. Most of the increase comes from slowdown in communication. This slowdown is expected as each node of BlueGene/Q only has 16 physical cores per node; with 8 processes per rank and 2 threads per rank, the cores are fully occupied. Increasing the number of threads beyond 16 per node leads to usage of the hardware threads. The setting for least run time is 8 ranks per node; however our primary focus is to minimize the number of nodes (and not necessarily the runtime) for execution, and all our experiments for the E.Coli, Drosophilla and the human datasets are run with 32 ranks per node. The runtimes will further reduce if the number of processes are reduced to 8 or 16 per node for our experiments.

Figure 2 also provided some observations about the application overall; looking at the times in k-mer construction and error correction, it can be seen that the k-mer construction time is a negligible percentage of the error correction time. Most of the error-correction time is spent in communication as expected; it can also be observed that the majority of the communication time is spent in communication of tiles especially tiles which are not part of the tile spectrum (non-existent on any rank).

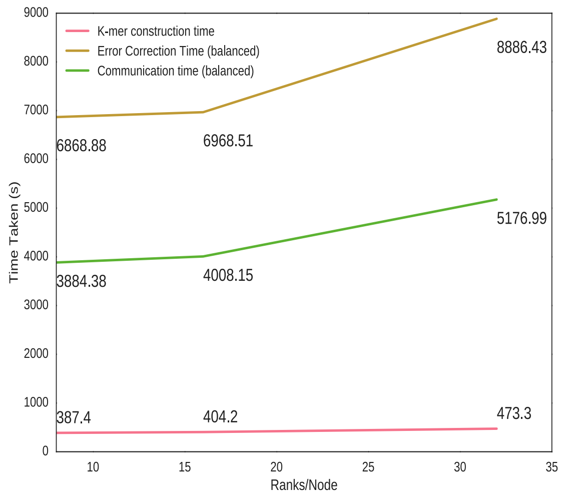

Fig. 2. Execution time of 128 ranks for the E.Coli dataset varying the number of nodes from 4 to 16 nodes.

For our implementation, it is key to keep the memory footprint of the processes uniform. This implies no particular rank has a substantially higher count of k-mers and/or tiles that could become a bottleneck; if a rank has a much higher count of k-mers or tiles, that could lead to more messaging overhead for that rank during error correction. Figure 3 shows the total number of the k-mer and tiles for all 128 ranks of the E.Coli dataset. For this run, the variation between the ranks having the highest and the lowest number of k-mers is less than 1%, with the variation in the number of tiles slightly less than 2%. This shows that the distribution of the k-mers and tiles is uniform across all the ranks, making the memory footprint and the messaging overhead of each rank consistent.

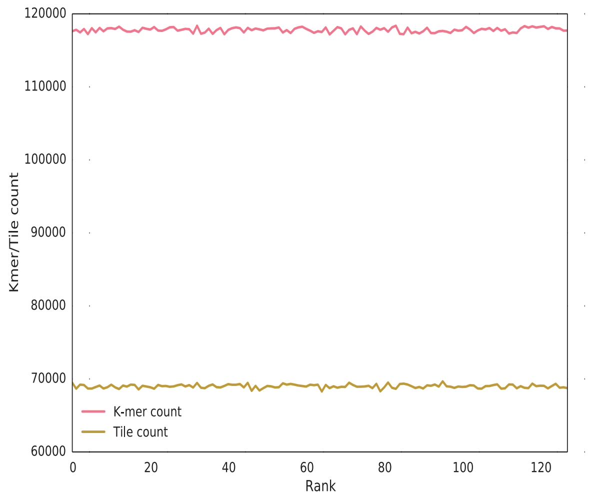

Fig. 3. K-mer and tile count of each rank for 128 processes

Figure 4 shows the effect of load balance on our results for the E.Coli dataset for 128 ranks on 4 nodes; our static load balancing algorithm reduces the total runtime almost by a factor of 2. Without load balance, there is a huge variation in the number of errors corrected per rank. The lowest number of errors corrected amongst all the ranks is 33886, while the highest number of errors corrected are almost 50% higher to 47927. This leads to the load imbalance; the fastest rank in this case takes 4948 seconds, while the slowest rank takes more than triple that time to more than 16000 seconds. The majority of the time is taken up in the communication time varying from 2891 seconds for the fastest rank to more than 10800 seconds for the slowest rank. Most of the communication time is taken up in the communication for tiles. The breakdown of the communication time shows that while the fastest rank needs almost 31 million remote tile lookups, the slowest rank needs more than 118 million tile lookups (not shown in Figure 4).

The load balancing strategy makes a major difference in the results; almost all the ranks uniformly take 8886 seconds. The number of errors corrected per rank only vary from 39127 to 39997 (only 2%), with the range of the communication time from 5073 seconds to 5268 seconds (less than 4%). The remote tile lookups needed per rank stays remarkably consistent with 64 million lookups per rank. Since there is a considerable improvement with load balancing, all of our future experiments are also completed with static load balancing.

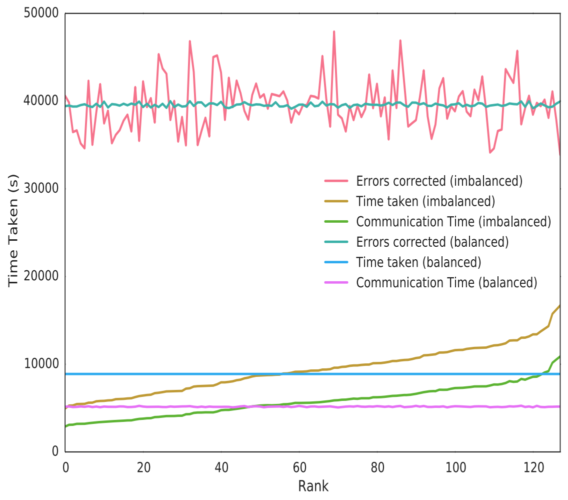

Fig. 4. Time taken (total and communication) and errors corrected for 128 processes

As detailed in Section III, our implementation also has support for various heuristics for optimal execution. For all the heuristics, we only show results for 1024 ranks running on 32 nodes for the E.Coli dataset. It is possible that other datasets might show different results for the heuristics, but since we had limited access to the BlueGene/Q for node counts higher than 32, we only experimented with these heuristics for the E.Coli dataset.

Figure 5 shows the results for all the heuristics in terms of time taken and the highest memory footprint rank after the k-mer construction and the error correction steps. We can make several observations from Figure 5 about the effect of the heuristics on the runtime and the memory footprint. We detail the observations of each heuristic below:

- Universal mode is faster than non-universal mode by 8.8%. The increase in performance doesn't consume any extra memory, and thus this mode is advantageous to the non-universal mode.
- Replicating the k-mer spectrum on every process leads to a deterioration in performance; these runs were completed with 8 ranks per node (or 256 total ranks) as the memory footprint was noticeably higher. Due to the lower number of ranks, the improvement by replicating the k-mer spectrum on every rank is offset by the increased workload of the ranks. The memory footprint increases to 928 MB per rank as well.
- Replicating the tile spectrum on every process reduces the runtime of the error-correction step to 975 seconds (from 1178 seconds of the base mode). With the replication of tile spectrum, no communication is needed for the

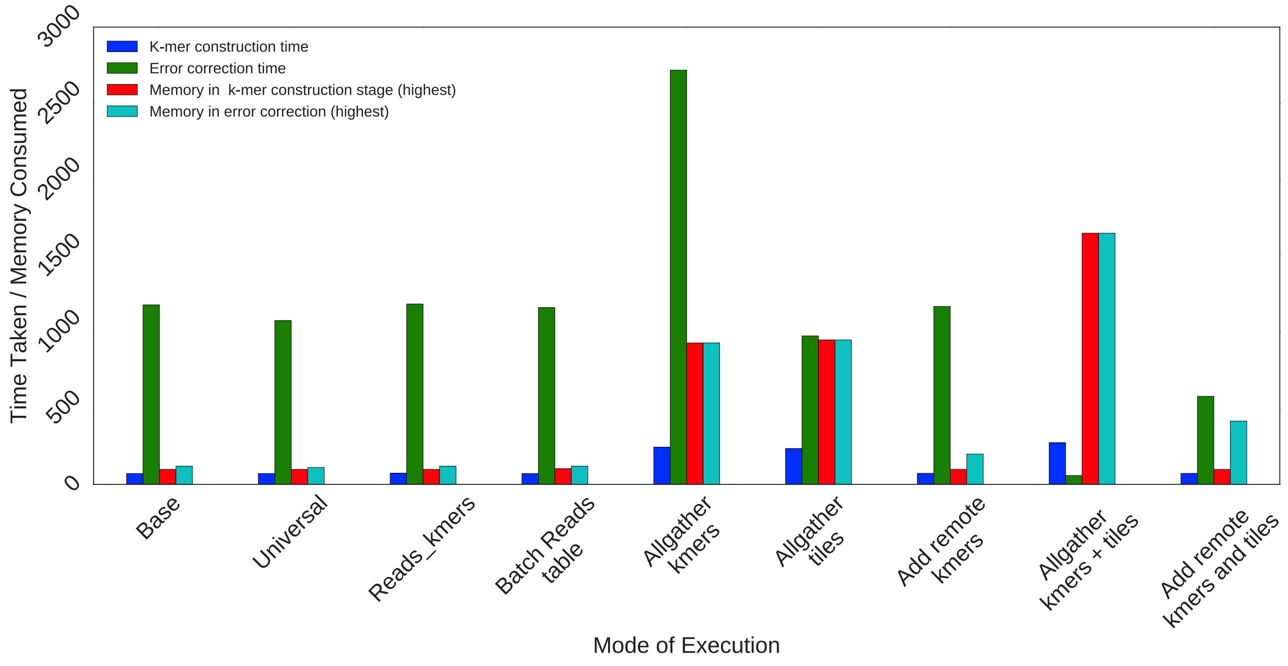

Fig. 5. Time of execution and memory footprint with different heuristics

tiles; since the runtime is dominated by the communication time of tiles, the runtime decreases even with the lower number of ranks. The replication also increases the memory footprint to 948 MB per rank. Thus, instead of replicating both the k-mer and the tile spectrum on every rank, it is highly advantageous to replicate only the tile spectrum, relying on communication for the k-mer spectrum. This run was also completed with 256 ranks only.

- The effect of adding remote k-mers (both the ones which are on other nodes and the ones which are non-existent) does not improve the runtime of the error-correction step. The memory footprint increases to 199 MB from 119 MB for this heuristic.
- Batch reads table is useful in lowering the memory footprint further by keeping the size of the reads hash table to a minimum. This run was completed with a chunk size of 2000 reads; the reads table is processed and cleared after each rank has finished their chunk of reads. Since the number of sequences processed by each rank is 8657, each rank does this step 5 times. This heuristic is highly advantageous as it was used for the runs for the human dataset.
- Adding the k-mers and tiles belonging to the reads of the rank does not improve the performance of the error correction step. This is because most of the communication time is spent in remote tile lookups.
- Finally, with the k-mers and tiles replicated on every node, the error-correction time is only 58 seconds. The memory footprint of this mode is almost 1648 MB per rank. This run was completed with only 1 rank per node and 64 threads per rank.

For our purposes, the advantageous heuristics are *universal* which reduces the runtime, and *batch reads table* which reduces the memory footprint of collective communications. We did not use any of the replication settings in our experiments.

Figure 6 shows the results for the E.Coli dataset as the number of ranks are increased from 1024 to 8192; since each node is running 32 ranks, this translates into an increase in the number of nodes from 32 to 256. No heuristics were employed in this scalability graph. Figure 6 shows that our implementation is scalable both in k-mer construction and error correction times. The parallel efficiency for E.Coli dataset at 8192 ranks when the error-correction time is approximately 180 seconds is 0.81. Figure 6 also shows the improvement over imbalanced run; there is a marked improvement in runtimes especially at lower node counts. For example for 32 nodes, the runtime more than halves due to our strategy of redistribution of sequences for load balance. At 256 nodes, the total time taken to correct the dataset is less than 200 seconds using the load-balancing approach.

Figure 7 shows similar scalability results for the Drosophila dataset as the number of ranks are increased from 128 to 512 BlueGene/Q nodes. This figure also shows excellent scalability from 1024 ranks to 8192 ranks. In this graph as well, each node is running 32 ranks per node. The load balancing improves the performance significantly; the runtime improves by more than a factor of seven at 8192 ranks (or 256 nodes). The runs using the imbalanced approach for node counts 1024 and 2048 did not finish in a reasonable time. Also, it can be

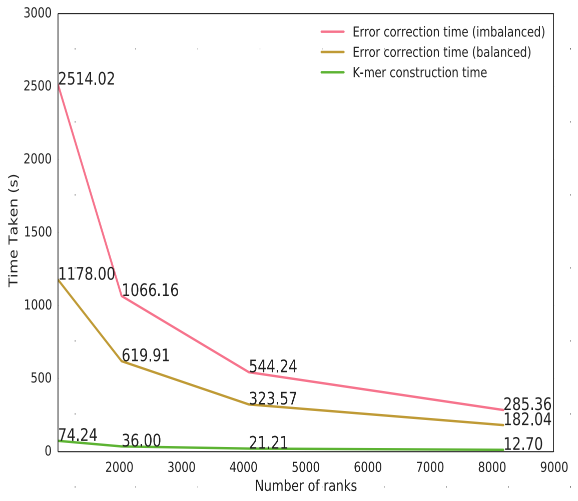

Fig. 6. Scaling graphs for E.Coli dataset varying the number of nodes from 32 to 256

seen that for 1024 ranks, the K-mer construction time takes 981 seconds. This run was completed with the heuristic *batch reads table*, which reduces the memory footprint of the k-mer construction stage, but leads to an increase in runtime. The parallel efficiency at 8192 ranks for Drosophilla dataset is 0.64.

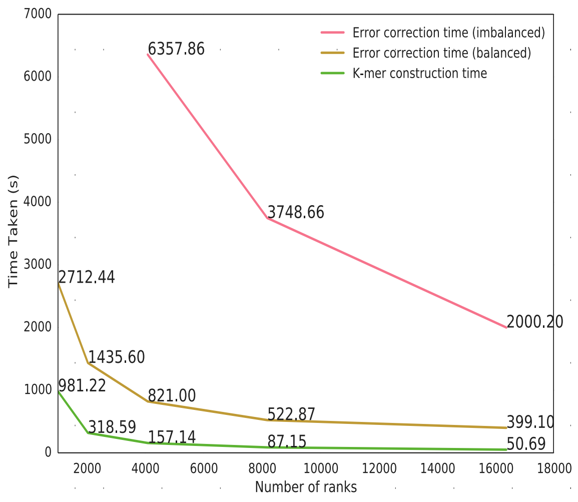

Fig. 7. Scaling graphs for Drosophila dataset varying the number of nodes from 128 to 512

Finally, Figure 8 shows the runtimes for the human dataset consisting of over 1.55 billion reads varying the number of nodes from 128 to 1024. The runs were completed with 32 ranks per node, so the total number of ranks are varied from 4096 to 32768. All the runs were completed with the heuristic *batch_reads* and the load balancing strategy enabled; the reason for the *batch_reads* heuristic being employed is the size of communication buffers for the collective communication in Step II of the k-mer construction time exceeds the memory available on each process. As explained before, this heuristic leads to multiple collective communication calls in the k-mer construction; each call is executed after all the processes have processed a batch of reads. For the 128 and the 256 nodes run, the batch size was only set to 5000 reads, while for the 512 and 1024 node runs, the batch size was set to 10000 reads. A BlueGene/Q specific environmental flag to lower the memory requirements for collective communication by implementing the collective calls as multiple point to point communication was also used. **This result shows we can complete error correction of the human dataset in less than 2.5 hours using a single rack of BlueGene/Q.**

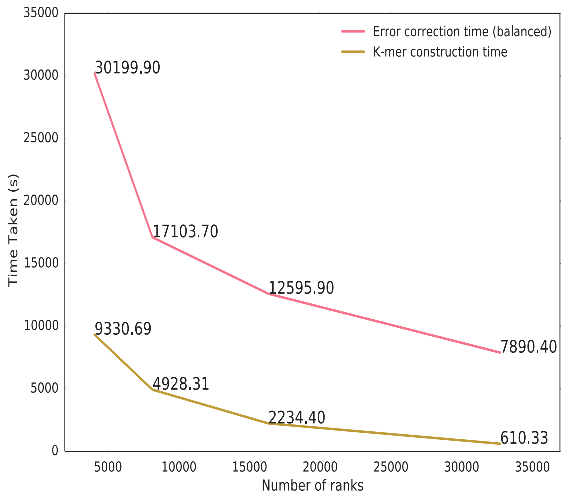

Fig. 8. Scaling graphs for human dataset varying the number of nodes from 256 to 1024

V. Conclusions and Future Work

This paper detailed a time and memory scalable parallelization of Preptile, a popular and accurate code used in error-correction of short reads from next-generation sequencing machines. Our approach allowed error-correction to be completed on IBM's BlueGene/Q with only 512 MB per process for all datasets including the human dataset with over 1.55 billion reads. Using 256 nodes of BlueGene/Q, we are able to error correct E.Coli and Drosphila datasets in approximately 200 and 600 seconds respectively. We achieve a parallel efficiency of 0.81 and 0.64 for the two experiments at 8192 ranks. The human dataset consisting of 1.55 billion reads is corrected in a little more than two hours using 1024 nodes of BlueGene/Q. We have also experimented with many heuristics that can be employed based on the advantages of the underlying hardware. For load-balancing, we relied on a static strategy that considerably improved the load balance between the ranks. This strategy does not depend on a master-slave;

instead it redistributes the reads amongst the processing ranks. Our approach can be employed on hardware regardless of the memory size per node and the dataset being error-corrected. Besides the low memory footprint, we also achieved excellent scalability both in terms of memory and time with increasing number of nodes.

An area of further improvement is experimentation with partial replication: as the number of nodes is increased, the number of k-mers and tiles per rank also decreases. This leads to very low memory footprints at the highest node counts for all three datasets, for example, the footprint of E.Coli dataset at 256 nodes is less than 50 MB per rank and the footprint of the Drosophila is close to 80 MB at 512 nodes. The footprint of the billion plus reads human dataset at 1024 nodes is about 120 MB per process (with the *batch reads dataset*). However, one of the approaches could be to only lower the memory footprint as much as needed. One potential strategy is for each rank to store the k-mers and tiles of a subset of other ranks, besides the k-mers and the tiles the rank owns. This would allow the memory footprint to be low enough for a complete execution and reduce the communication overhead, which could enable a faster runtime. Another idea we plan to further explore is to keep the k-mers and tiles belonging to the reads of the rank; while it was not advantageous for Reptile's performance, we believe other algorithms might benefit from this approach.

Acknowledgment

We would like to thank Sriram Chockalingam for providing us the original parallel Reptile implementation and the instructions for the datasets used in this paper. We also wish to thank IBM T. J. Watson Research Center facilities especially Karen Bard for providing us access to BlueGene/Q.

References

[1] S. C. Schuster, "Next-generation sequencing transforms today's biology," *Nature*, vol. 5, pp. 16–18, 2008.

[2] M. D. Macmanes and M. B. Eisen, "Improving transcriptome assembly through error correction of high-throughput sequence reads," *PeerJ*, p. e113, 2013, https://peerj.com/articles/113/.

[3] X. Yang, S. P. Chockalingam, and S. Aluru, "A survey of error-correction methods for next-generation sequencing," *Briefings in Bioinformatics*, vol. 14, no. 1, pp. 56–66, 2013.

[4] X. Yang, K. S. Dorman, and S. Aluru, "Reptile: representative tiling for short read error correction," *Bioinformatics*, vol. 26, no. 20, pp. 2526–2533, 2010.

[5] A. R. Shah, S. Chockalingam, and S. Aluru, "A parallel algorithm for spectrum-based short read error correction," in *Proc. 16th Int'l Parallel and Distributed Processing Symp. (IPDPS)*, Shanghai, CN, May 2012, pp. 60–70.

[6] N. Jammula, S. Chockalingam, and S. Aluru, "Parallel Read Error Correction for Big Genomic Datasets," in *Proc. Int'l Conf. High-Performance Computing (HiPC)*. Bengaluru, India: IEEE Press, Dec. 2015.

[7] C. S. Kim, V. Sachdeva, M. Winn, K. Jordan, and K. Hassani-Pak, "de novo transcriptome assembly using Trinity for large RNA-Seq datasets," in *Proc. High Throughput Sequencing Algorithms and Applications (HitSeq 2014)*, Boston, MA, 2014, poster session.

[8] R. A. Haring, M. A. Ohmacht, T. W. Fox, M. K. Gschwind, D. L. Satterfield, K. Sugavanam, P. W. Coteus, P. Heidelberger, M. A. Blumrich, R. W. Wisniewski, A. Gara, G. L.-T. Chiu, P. A. Boyle, N. H. Chist, and C. Kim, "The IBM Blue Gene/Q Compute Chip," *IEEE Micro*, vol. 32, no. 2, pp. 48–60, 2012.

[9] D. R. Kelley, M. C. Schatz, and S. L. Salzberg, "Quake: quality-aware detection and correction of sequencing errors," *Genome Biology*, vol. 11, no. 11, 2010, http://www.genomebiology.com/2010/11/11/R116.

[10] J. Schrder, H. Schrder, S. J. Puglisi, R. Sinha, and B. Schmidt, "SHREC: a short-read error correction method," *Bioinformatics*, vol. 25, pp. 2157–2163, 2009.

[11] W.-C. Kao, A. H. Chan, and Y. S. Song, "ECHO: A reference-free short-read error correction algorithm," *Genome Research*, vol. 21, pp. 1181–1192, 2011.

[12] M. H. Schulz, D. Weese, M. Holtgrewe, V. Dimitrova, S. Niu, K. Reinert, and H. Richard, "Fiona: a parallel and automatic strategy for read error correction," *Bioinformatics*, vol. 30, pp. 356–363, 2014.

[13] A. Wirawan, R. S. Harris, Y. Liu, B. Schmidt, and J. Schrder, "HECTOR: a parallel multistage homopolymer spectrum based error corrector for 454 sequencing data," *BMC Bioinformatics*, vol. 15, 2014.

[14] L. Illie and M. Molnar, "RACER: Rapid and accurate correction of errors in reads," *Bioinformatics*, vol. 29, no. 19, pp. 2490–2493, 2013.

[15] Y. Liu, B. Schmidt, and D. L. Maskell, "DecGPU: distributed error correction on massively parallel graphics processing units using CUDA and MPI," *BMC Bioinformatics*, vol. 12, 2011.

[16] S. Boisvert, F. Raymond, E. Godzaridis, F. Laviolette, and J. Corbeil, "Ray Meta: scalable de novo metagenome assembly and profiling," *Genome Biology*, vol. 13, no. 12, 2012, http://genomebiology.biomedcentral.com/articles/10.1186/gb-2012-13-12-r122.

[17] P. Carrier, B. Long, R. Walsh, J. Dawson, B. Haas, T. Trickle, C. P. Sosa, and T. William, "The impact of high-performance computing best practice applied to next-generation sequencing workflows," http://biorxiv.org/content/biorxiv/early/2015/04/07/017665.full.pdf.

[18] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick, "Parallel de bruijn graph construction and traversal for de novo genome assembly," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2014)*, New Orleans, LA, Nov. 2014.