Stony Brook University

Development and Application of an Integrated Parallel Platform on Short–read Sequences Assembly

A Dissertation Presented

by

Fei He

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Applied Mathematics and Statistics

Stony Brook University

May 2016

Stony Brook University

The Graduate School

Fei He

We, the dissertation committee for the above candidate for the

Doctor of Philosophy degree, hereby recommend

acceptance of this dissertation.

Song Wu – Dissertation Advisor
Assistant Professor, Department of Applied Mathematics and Statistics

Wei Zhu - Chairperson of Defense
Professor, Department of Applied Mathematics and Statistics

Xuefeng Wang - Committee Member
Assistant Professor, Department of Applied Mathematics and Statistic

Jiangyong Jia - Outside Member
Associate Professor, Department of Chemistry

This dissertation is accepted by the Graduate School

Charles Taber
Dean of the Graduate School

Abstract of the Dissertation

Development and Application of an Integrated Parallel Platform on Short–read Sequences Assembly

by

Fei He

Doctor of Philosophy

in

Department of Applied Mathematics and Statistics

Stony Brook University

2016

Rapid and automated next generation sequencing (NGS) methods have emerged recently and significantly accelerated the research in biological and medical fields. The high-throughput NGS usually generates billions of shorter reads, which poses great bioinformatics challenges on extracting meaningful information from these massive data, one of which is *de novo* assembly. At the same time, the fast development of massive parallel processing (MPP) systems presents a substantial opportunity for processing larger datasets. Therefore, using supercomputer innovations on NGS research promises a good strategy; however, this application is not straightforward and requires new algorithms and parallel design for efficient implementations.

In this thesis, we develop and present PPLAT, an integrated hierarchical multitasking parallel platform framework, and PPASSEM, a novel genome assembler built on PPLAT. PPLAT is designed for distributed storage and distributed processing of big data by enabling asynchronous computing and message passing, and provides a hybrid of multithreading- and

MPI-based solution for MPP systems with simple APIs and great flexibility. We demonstrate the power of PPLAT to significantly reduce the coding and debugging complexity as well as facilitate high performance of derived parallel programs. PPASSEM is a novel application built on PPLAT, which employs the small-scale shared-memory multithreading and the large-scale distributed-memory parallelism using de Bruijn graph data structure for short–read sequences data.

Our parallel platform has been tested on commodity computer clusters, based on both simulated and real data. Our results show that PPLAT can effectively handle billions of short reads (~500GB), and PPASSEM can generate accurate assembly constructs with much less time, compared with other well-known benchmark assembler like ABySS and PASHA. As new additions to the existing NGS toolbox, we expected that PPLAT and PPASSEM will greatly facilitate the future NGS-based research.

Table of Contents

List of Tables

List of Figures

Acknowledgments

First I would like to express my sincere gratitude and kind regards to my committee member and my family for their guidance, support and help to me for during my PhD study.

I would like to express my special thanks to my advisor, Professor Song Wu. Professor Wu took me to his group when I was completely dazed about my future and instructed me to statistical research and parallel programming, which bright up my sky and ignite my passion for research. Not only did Professor Wu patiently provide instruction and guidance to my work, he taught me about responsibility, critical thinking, problem solving and many more that I would eternally benefit from.

I would like to thank to my committee member Professor Zhu, Professor Wang and Professor Jia, for kindly helping me walk through my dissertation and providing valuable insights and improvements suggestions.

I would like to thank to all my friends. It is my luck to have them around. They bring me help and happiness, hanging out with them has always been my fondest memory.

To me the most important, I would like to thank my parents and my wife for their unconditional love and support for so many years. Together with my family, I have never been more blessed and brave.

Chapter 1

Introduction to Genome Sequencing and Genome Assembly

1.1 Introduction

Through examine the complete DNA sequences of an organism, genome sequencing proves to be an important tool towards understanding the mystery of life. Ever since the first genome was published (Sanger et al., 1977), more rapid, automated sequencing methods have emerged and significantly accelerated the research in biological and medical fields. These methods are particularly useful in determining the genomic and functional structures of new species (Cho and Blaser, 2012), detecting protein to DNA binding pattern (Wu et al., 2010), guiding therapeutic intervention (Mooney, 2014), identifying differentially expressed genes (Velculescu et al., 1995) and understanding the transcriptional landscape (Carninci et al., 1995).

Technically, it is rather difficult for genome sequencers to sequence the whole genome as one piece (usually at a length of around 1 billion bases). Instead, many short stretches of DNA are generated at a time. Either “clone-by-clone” approach or “whole-genome shotgun” approach (Weber and Myers, 1997) involves extracting DNAs, breaking them into random small pieces and then sequencing the pieces. A natural problem followed by sequencing is assembly of these short reads, i.e., put them back in order to form sets of continuous sequences (or called contigs) in the original genome. Assembly programs for Sanger sequences, such as, Arachne (Batzoglou

et al., 2002), Celera (Myers et al., 2000), PCAP (Huang et al., 2003) and Phusion (Mullikin and Ning, 2003) were developed using an overlap–layout–consensus (OLC) approach (Li et al., 2011). In general, OLC methods first search overlapping among all reads, then lay them out on a graph, which gives inference about consensus sequences.

Early Sanger sequencing technologies were assiduous, slow and costly. With the advent of high throughput technologies, sequencers were able to automatically process millions of reads in parallel rather than 96 at one time (Mardis, 2008). These next-generation sequencing technologies notably reduced the sequencing time and cost, while producing shorter reads lengths (35-250 bp, varied by platform) than traditional Sanger reads (650-800 bp) and achieving greater coverage depths. When handling the large quantities of short reads from NGS (usually at the order of 10^9 reads), these early successful programs mentioned above became extremely time-consuming and memory-intensive, or even failed to run because they need pairwise comparison among all the reads (Zerbino, 2009). Since assembly algorithms optimized for long reads are fundamentally different from approaches targeted for short reads, novel designs and assembly algorithms have been developed for these next-generation sequencing data.

In the following, we will describe some of the most popular next-generation sequencing technologies and review most recent approaches in several *de novo* assemblers.

1.2 Next-generation sequencing (NGS)

Nowadays, complex genomic research questions demand information that is beyond the capacity of traditional DNA sequencing technologies. Recent short read, massively parallel sequencing technologies were fundamentally different with Sanger-based sequencing technologies, and revolutionized the sequencing capabilities to a “next-generation” era. Although

reads produced by next-generation sequencing is somewhat shorter in length and have relatively higher error rates, the next-generation sequencing is far faster and cheaper than Sanger sequencing. Marvelously noticed, the sequencing of first human genome, came out in 2001(Venter et al., 2001; Lander et al., 2001), took 15 years at the cost of nearly 3 billion dollars; however, 13 years later, the HiSeq X Ten, has the ability to sequence over 45 human genomes in a single day and make $1000 per genome a reality.

Broadly speaking, the sequencing principle used in NGS is similar to Sanger sequencing – DNA polymerase catalytically incorporates fluorescently labeled deoxy ribonucleotide triphosphates (dNTPs) into a DNA template strand during sequential cycles of DNA syntheses. During each cycle, the incorporated nucleotides are identified by fluorophore excitation. Major difference lays in other than sequence a single DNA fragment, NGS parallel this process across millions of fragments.

Commercialized next-generation sequencers include Roche (454) Sequencing Systems, Illumina Sequencing Systems and Applied Biosystems SOLiD Sequencer (Mardis, 2008). By far, the Illumina sequencing technology is the most popular NGS platform and has been widely used in research communities. Below are brief reviews of the Illumina sequencing and its applications.

1.2.1 Illumina Sequencing

Illumina sequencing originated from Solexa sequencing, which launched first NGS sequencer in 2006 and then were acquired by Illumina in 2007. Due to its different amplification and sequencing techniques compared to other sequencers, Illumina sequencing by synthesis (SBS) chemistry can generate reads with higher throughput, significant lower cost and higher

accuracy. Therefore, it is becoming the most widely applied chemistry in industry. Nowadays, Illumina occupies more than 75% of sequencing market share.

Illumina sequencing works in 4 basic steps, as shown in Figure 1:

1) Library Preparation – The sequencing library is prepared by random fragmentation of the DNA or cDNA sample, followed by 5' and 3' adapter ligation. Or, fragmentation and ligation reaction combined into single step to increase the library preparation efficiency. Adapter-ligated fragments are then PCR amplified and gel purified.
2) Cluster Generation – In cluster generation, the library of DNA fragments is loaded into a flow cell where they are captured on a lawn of surface mounted with oligoes complementary to the library adapters. Each fragment is then bridge amplified into distinct, clonal clusters through several cycles of PCRs.
3) Sequencing – Illumina SBS method used a proprietary reversible terminator-based approach that ensures only one single base to be incorporated into a DNA template strand each cycle. Since all 4 reversible terminator-bound dNTPs are present during each sequencing cycle, natural competition minimizes incorporation bias and greatly reduces raw error rates compared to other technologies.
4) Data Analysis – In data analysis, Illumina provides a DNA-to-Data solution, the newly sequenced reads are aligned to a reference genome. After alignment, many variations in genome can be identified, such as single nucleotide polymorphism (SNP) or insertion-deletion (INDEL), read counting for RNA methods, phylogenetic or metagenomics analysis, and more.

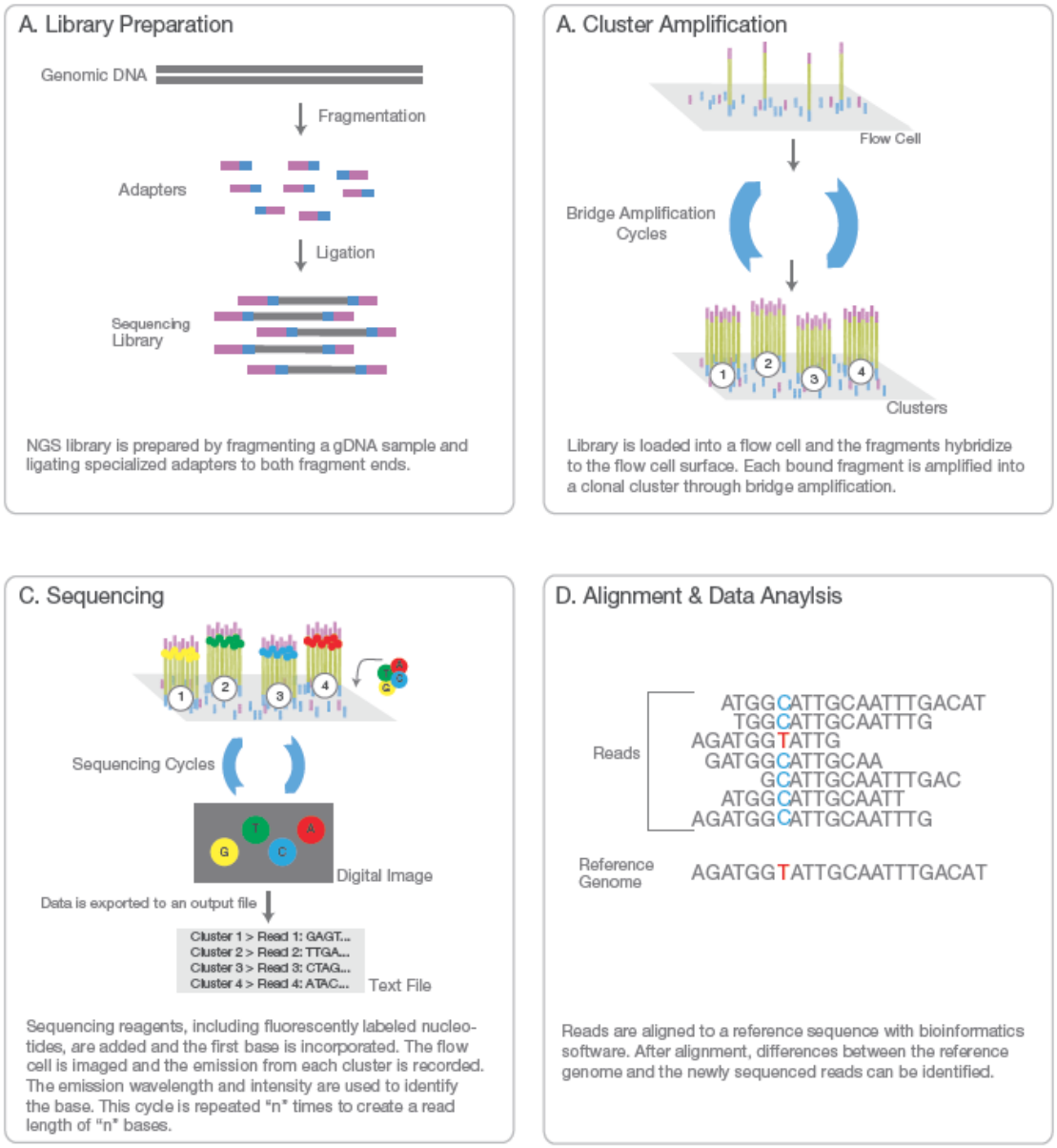

Figure 1: Illumina sequencing overview (http://www.illumina.com/content/dam/illumina-marketing/documents/products/illumina_sequencing_introduction.pdf)

Pair-end sequencing is a major advance in NGS, involves sequencing both end of a single DNA fragment form the sequencing library. In addition to producing twice the number of reads for the same time and effort in library preparation, sequences alignment with as paired reads

enables more accurate read mapping and the ability to detect important genomic structures such as indels and translocations. Analysis of differential read-pair spacing also allows removal of PCR duplicates, a common artifact resulting from PCR amplifications. Also, since the distance between each paired read is known, alignment algorithms can use this information to map the reads over repetitive regions more precisely. What's more, pair-end sequencing produces a higher number of SNV calls following read-pair alignment. Most researchers currently use pair-end reads data, particularly in the field of read assembly.

1.2.2 Application of next-generation sequencing

The emergence of next generation sequencing facilitates a broad area of researches:

- Genome sequencing: targeted resequencing, mutation detection …
- Transcript expression profiling: RNA-seq, polyadenylation site …
- Transcription factor binding: ChIP-seq …
- Structural variation: tandem duplication, translocation …
- Metagenomics: study of genomes recovered from environmental samples
- Epigenomic variation
- And more.

1.3 *De novo* assembly of next-generation sequencing data

De novo assembly is a method of build long sequences from short reads without reference sequences, compared to comparative assembly, which assembles reads against existing closed related organism as reference sequences. In the view of computer science, *de novo* genome

assembly is a NP-hard problem that no efficient solution is known (Myers, 1995). Below are reviews of several most recent assembly algorithms for NGS data.

1.3.1 Assembly

Assembly relies on the assumption that reads share common substrings, which is made possible by the over-sampled genome at sequencing. Through these overlapping substrings, assembly reconstructs the reads data to a putative genome. It groups reads into contigs and then to scaffolds (Miller et al., 2010). The scaffolds characterize the order and orientation of the contigs, as well as the gap information between contigs.

Popular input file formats for assemblers are in FASTA or FASTQ. For example, the FASTQ format is a text-based format storing both nucleotide sequences and corresponding quality score (Cock et al., 2010). A FASTQ file usually uses four lines per sequence:

- Line 1 starts with a '@' character and is followed by a sequence identifier and an optional description
- Line 2 reflects the raw sequence, a string of A, C, G, T and possibly N
- Line 3 begins with a '+' character and is optionally followed by the same sequence identifier/ description again
- Line 4 is the quality values for the sequence in Line 2 and has the same length.

For the assembly outputs, size and accuracy of the contigs and scaffolds are most commonly adopted for performance evaluation. Maximum length, total length, number of contigs/scaffolds and N50 are generally used for size measurement. Given a set of sequences of varying lengths, the N50 length is defined as the length N for which 50% of all bases in the

sequences are in a sequence of length L < N. For accuracy, align output to reference genome for coverage information is good, provided reference genome is trustworthy.

Graph algorithms are widely used for NGS read assembly. For instance, the overlap-layout-consensus methods depend on overlap graph, while de Bruijn graph methods employ k-mer graph. Briefly, a graph is an abstract data structure in computer science. A graph data structure consists of a set of vertices or nodes, together with a set of unordered pairs of these vertices for an undirected graph or a set of ordered pairs for a directed graph. These pairs are known as edges or arcs for a graph.

An overlap graph contains sequencing reads and overlaps between them (Myers, 1995). In this graph, reads are the nodes and overlaps are the edges. The overlaps were computed by a series of exhaustive pair-wise sequence alignments. Potential contigs are derived from the paths in the graph.

De Bruijn graph is a directed graph that represents overlaps between sequences and was found of enormous potential in assembly (Pevzner et al., 2001). In this approach, reads are first decomposed into several k-mers by a moving window of fixed length k, in another word, k-mers is a continuous substring of length k. Nodes in the de Bruijn graph are k-mers and a directed edge was created when an overlap of k-1 bases was found between two k-mers. Reads with high similarity should share k-mers in their overlapping regions, and finding shared k-mers are computationally easier than the all-against-all pairwise alignments in overlap graph.

1.3.2 Challenge in assembly

Genome assembly algorithms and implementations are generally complex: high throughput and computation intensity require high-performance computing platform; random

and systematic errors in sequencing data need robust error correction algorithms; non-uniform sequencing coverage calls for gap closure. Aside from the above, genome itself also brings complications: repetitive sequences in the genome can be indistinguishable especially when they are longer than the reads (Kececioglu and Ju, 2001); single nucleotide polymorphisms (SNPs) and structural variations in the genome are more difficult to address in the presence of sequencing errors; DNA is not always extracted from a haploid genome, but in many cases from heterozygous diploid genomes.

Computationally, assembly problem is NP-hard with no known efficient solution (Nagarajan and Pop, 2009), so it may need polynomial time to solve. In overlap graph, the number of nodes equals the reads number that increases linearly with sequencing depth, and the number of edges increase at the logarithmic scale. In de Bruijn graph, the number of nodes equals the genome size and hence is the number of edges, which is irrelevant to sequencing depth. But in practice, nodes in de Bruijn graph are much higher because of sequencing errors. In the current high throughput era, assembly of large genome is still computationally prohibitive for most assemblers on accessible computing resources. Parallel computing shows prominent potential in addressing the assembly problem.

Assembly is also confounded by several read-world genomic structures, such as double-strandedness of DNA, palindromes, sequencing errors and genomic repeats. These structures usually complicate assembly graphs quite significantly.

- Double-strandedness of DNA: DNA molecules consist of two biopolymer strands coiled around each other to form a double helix. Because of the way sequencing is done, the forward sequence of any given read may overlap the forward or reverse complement

sequence of other reads. Several methods are developed to take care of this property: one works by storing forward and reverse complement sequence together to a block (Zerbino and Birney, 2008); another works by storing forward and reverse complement sequence but avoiding results output twice (Idury and Waterman, 1995).

- Palindromes: palindrome is a DNA sequence whose reverse complement is itself. In assembly, palindromes induces paths folds back on themselves. Usually force k-mer length to be odd can prevent this issue.

Sequencing errors typically induce three types of topological structures in the graph, as presented below.

- Tips: short, dead-end divergences from the main path. They are caused mostly by sequencing error toward one end of a read. In rare cases, caused by zero sequencing coverage at certain region.
- Bubbles: paths that diverge then converge. Bubbles can happen due to several reasons: sequencing error in the middle of a read; polymorphism of bases; insertion or deletion of bases (INDELs); random overlaps of two nearby tips.
- Chimeric connections: sequences that connect true contigs in artificial ways, branching out otherwise continuous contigs. They are caused by connecting a read incorrectly to another part of the genome or an actual chimeric read in which two different parts of the genome are physically ligated.

Sequencing errors can be reduced pre-assembly by filtering out low quality raw reads and/or post-assembly by graph reduction that is based on reads multiplicity information. Usually de Bruijn graph method benefits more from pre-assembly error correction because false k-mers

will consume more memory and create meaningless branch paths (Batzoglou et al., 2002). There are generally two ways for pre-assembly error corrections, one based on reads alignment, the other based on k-mer frequency spectrum. The first one is CPU-intensive, and works by first performing multiple alignments then detecting sequencing errors through a probability model, as adopted in assemblers including Allpath-LG (Gnerre et al., 2011). The second one is less time consuming, and works by first counting k-mer frequencies then correcting low frequency k-mers, adopted in assemblers including Euler (Pevzner et al., 2001) and SOAPdenovo (Li et al., 2010).

Repetitive sequences are patterns of nucleic acids that have multiple copies throughout the genome. Repeats complicate the graph, and is another big challenge for assembly since interspersed repeat DNA is found in all eukaryotic genomes. DNA regions that share identical repeats are very hard to differentiate, especially when repeats are longer than read length. Fortunately, current sequencing technologies provide pair-end sequencing which gives further long-range linkage information and is helpful to cross repeats, but the analysis is complicated and at the risk of introducing false positive joins.

There are certain repeat structures usually be resolved together with error corrections:

- Frayed rope pattern: paths converge and then diverge.
- Loops: paths converge on themselves. For example, short tandem repeats (a pattern of two or more nucleotides is repeated and the repetitions are adjacent to each other) induce small loops.

By removing the above error structures and simple repeats removed, assemblers can avoid contigs being broken up by small isolated differences, can usually detect much longer homologous regions.

1.3.3 Overlap-layout-consensus (OLC) assembly

OLC approach uses an overlap graph and is suited for longer reads. As showed in its name, it usually operates in three stages: overlap - build overlap graph; layout - bundle stretches of the overlap graph into contigs; consensus - pick the most likely nucleotide sequence for each contig and correct read errors. Figure 2 shows an example of OLC graph.

- Overlap: find best match between suffix of one read and prefix of another. Usually a seed-based strategy is adopted and K-mers are used as indices. Only reads that share a seed are compared for alignments (Schatz et al., 2010). Overlap discovery heavily depends on the choice of parameters: k-mer size, overlap length and percent of identity. It is also the most time consuming stage.
- Layout: create local multiple alignments from the overlapping reads. Overlap graph is typically big and messy, and layout stage can remove transitively-inferable edges and output contigs that belongs to non-branching stretches. By nature, this stage is a Hamiltonian path problem.
- Consensus: derive the DNA sequence implied by reads arrangement along the edge through the graph. Each consensus base is identified by weight voting from multiple sequence alignment (MSA). Since no known optimal MSA is available (Wang and Jiang, 1994), progressive pairwise alignments are adopted here.

Figure 2: Example of OLC graph.

The first line is a genome segment, R1 to R6 are reads mapped to this region, and the bottom connected graph is overlap graph built from these reads.

This OLC approach has some disadvantages, especially in the era of next-generation sequencing when billions of reads are considered. Although, the computation of pairwise overlaps in OLC approach can be improved by heuristic methods (Pearson and Lipman, 1988) or filter methods (Rasmussen et al., 2005), its inherent quadratic complexity hinders its usage.

1.3.4 De Bruijn graph assembly

De Bruijn graph method is not as intuitive as OLC method. It works by first decomposing NGS reads into k-mers, then building de Bruijn graph over these k-mers and finally deriving DNA sequences from the graph. This method was first brought up by Idury and Waterman (Idury and Waterman, 1995) but did not catch up much attention by the research community until Pevzner et al. (2001) extended this idea, which was further refined by Chaisson and Pevzner (2008), and Zerbino and Birney (2008). Based on this method, several assemblers have been

developed a, including Velvet (Zerbino and Birney, 2008), ALLPATHS (Butler et al., 2008), ABySS (Simpson et al., 2009), SOAPdenovo (Li et al., 2010) and PASHA (Liu et al., 2011).

In the de Bruijn graph, nodes represent all fixed-length subsequences (called k-mers) retrieved from the reads. A directed edge indicates two nodes occur consecutively in one or more reads. The problem is to build the connected graph from the reads, clean the graph and find a path that traverses every edge exactly once. De Bruijn graph method is widely applied on Illunima and SOLiD data, which generate shorter reads and higher sequencing depth. Compared to OLC method, de Bruijn graph method is much less CPU-intensive reads alignment and achieves better CPU efficiency. Also, instead of storing all reads and their overlaps, here k-mers are stored only once no matter how many times they show up in the reads, thus alleviating the pressure on physical memory. Figure 3 shows an example of de Bruijn graph.

It usually operates in following stages:

- K-mer generation: generate k-mers from reads file and load them inot memory as nodes
- Graph construction: construct edges between nodes
- Error correction and contig generation: identify errors by their structure in the graph, remove the error nodes and generate unambiguous stretches of sequence as contigs
- Scaffolding: realign reads onto contigs and employ pair-end information to merge contigs into scaffolds

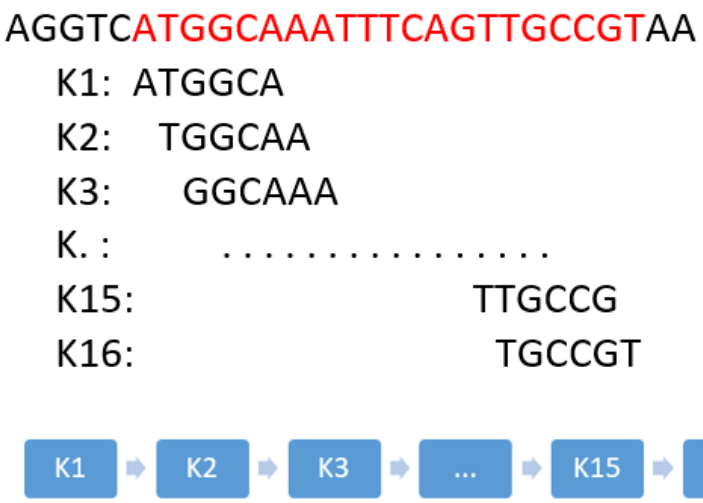

Figure 3: Example of de Bruijn graph.

The first line is a genome segment, K1 to K16 are k-mers mapped to this region, and the bottom connected graph is de Bruijn graph built from these k-mers.

The choice of k-mers length *k* is subtle. A proper *k* should be large enough that false overlaps are included as few as possible, while at the same time small enough that most true overlaps are captured. Say in extreme cases, a *k* equals 2 gives completely no useful information; a *k* equals read length missed most true overlaps because the shotgun sequencing cannot guarantee every region of read length in the genome is covered by multiple reads. The choice of *k* should be carefully tuned depending on the data (sequencing depth, error rate, etc.).

Compared to OLC assemblers, de Bruijn graph assemblers initially create multiple nodes for each reads. One should notice that as more reads are added in those nodes, a linear path may not be formed. Also, de Bruijn graph is not read coherent (Myers et al., 2005), that is, there may exist paths that doesn't supported by underlying reads. All these considerations should be carefully taken to output meaningful results.

Velvet

Velvet is a sound and popular de Bruijn graph based assembler, which implemented a full pipeline of assembly (Zerbino and Birney, 2008; Zerbino et al., 2009). It provides a collection of graph simplification methods to deal with genomic variant structures, like duplications, inversions or transpositions and sequencing errors.

In Velvet, each node represents a series of overlapping k-mers and is attached to its reverse complement k-mers, which takes care of the double-strand property, to form a "block". Nodes are connected by directed "arcs", because of the symmetry of the blocks. If an arc connects node A to node B, there would be a symmetric arc connects reverse complement of node B to reverse complement of node A. The graph in Velvet is indeed an implicit bi-graph (or bi-directed graph) (Medvedev et al., 2007).

Tips are iteratively removed from the graph based on two criteria - length and minority count - to avoid chopping out actual sequences that are discontinued by coverage gaps. Tips usually occur when a sequencing error occurs within *k* bp from either start or end of a read. If we allow two consecutive errors, which sum up to *2k* bp, a tip should have a length of less than *2k* bp and be removed. A standout sequence longer than *2k* has a much better chance to be an actual sequence than an accumulation of two errors. A tip is expected to have minority count because other more common paths should be superior to the one walking through the tip.

Velvet further addresses the issue of bubbles by an algorithm called Tour Bus, which is based on Dijkstra-like breadth-first search method. The Tour Bus starts at nodes with multiple out-going arcs and visits nodes by the increasing distance to the outset. If a node has been previously visited, trace back to find closest common ancestor and record the two paths. If the two paths are similar enough (identity threshold can be specified), the path with lower

multiplicity would be removed in a majority vote manner. This process is done iteratively and essentially analogous to bubble detection and bubble smoothing in OLC assemblers (Fasulo et al., 2002). Velvet further reduces graph by removing paths that represented fewer reads than a threshold. Although this process operates at the risk of eliminating true low-coverage sequences, empirically it does work well on removing sequencing errors induced spurious linkages.

After the above processes, a set of contigs can be generated by breaking paths at branching points. Then the long reads / paired-end reads information for repeats can be exploited. Original version of Velvet employs a Breadcrumb algorithm to correctly extend and connect contigs through repeated regions by using long contigs to anchor groups of mate-pairs. Breadcrumb could resolve simpler repeats but was limited in handling longer ones in large genomes. This algorithm was then replaced by a Pebble algorithm in later versions of Velvet, which use insert lengths to resolve more complex cases.

Velvet is extremely successful for small-size genomes, such as bacteria genomes. However during the assembly, Velvet records the read locations and paired-end information in the graph, memory issues preclude Velvet from assembling larger mammalian-sized genomes.

SOAPdenovo

SOAPdenovo is a novel short read assembler specially designed for Illumina GA reads that can build a de novo draft assembly for the human-sized genomes (Li et al, 2010). It uses threaded parallelization on supercomputer with multi-cores and large shared memory to resolve memory issues and computational complexities.

SOAPdenovo starts with preassembly sequencing error correction using k-mer frequency information to save memory. Error correction is the most time consuming part and works as

follows: at each inferred erroneous site, the impact of replacing the current nucleotide with the other three allele types can be tested and the nucleotide will be revised to the one with the highest frequency. After de Bruijn graph construction, erroneous connections on the graph are also handled, including clipping tips that are shorter than *2k* and have lower frequency than other alternative paths connected to a common node, removing low-coverage links that connected by only one or few nodes, resolving tiny repeats that display a frayed-rope pattern, and merging bubbles that have parallel paths very similar to each other.

SOAPdenovo breaks the simplified de Bruijn graph at repeat boundaries into contigs, then discards the de Bruijn graph and builds a contig linkage graph by transferring paired-end relationship to linkage information between contigs. Two steps are used to simplify the contig linkage graph and to extract unambiguously linear paths for scaffold construction: subgraph linearization, which removes compatible transitive lineages among a group of contigs and merges contigs into one node; and repeat masking, which isolates contigs traversed by multiple, incompatible paths.

SOAPdenovo provides a full pipeline for assembly and is able to assemble large plant and animal genomes. When working on large genomes, supercomputer with huge amount of memory is required (~500GB), which may not be accessible in many research facilities.

ABySS

ABySS is the first paralleled assembler that is based on distributed memory and is targeted to address memory issues and computational complexities for large genomes, by using a distributed de Bruijn graph method (Simpson et al., 2009). The distributed representation of de Bruijn graph allows parallel computation of assembly across a network of computing nodes, each

associate with independent memory that can be scaled up to deal with genomes of theoretically any size. ABySS shows its ability to work on a Yoruba individual genome that has billions of reads.

ABySS first converts k-mer and its reverse complement to numeric values by assigning {0, 1, 2, 3} to bases {A, C, G, T}, then on which hash values is computed. Two values are combined by XOR operation on their bit representation. a single k-mer is a vertex in the graph and its location is determined by its hash value moduling the number of computing nodes. Adjacency information between k-mers are stored in 8 bits per k-mer, representing existence or non-existence of four possible edges in either direction. After data are loaded, de Bruijn graph is built on exhaustive search on all eight neighbors.

ABySS uses similar approaches for graph simplification as in Euler (Pevzner et al., 2001) and Velvet, but is implemented in a parallel way: iteratively removes tips that are shorter than certain threshold; identifies bubbles and omits the paths supported by less reads. Initial contig are generated by merging unambiguously stretched paths. Pair-end information is not used at the stage of distributed memory parallelism, but is used to resolve ambiguities and merge contigs to final assembly. The job of working with contigs does not need the distributed memory architectures, and can usally work efficiently on a single computing node with the multi-threading parallelism,

ABySS is implemented in C++ and uses the MPI (Message Passing Interface) protocol for communication between nodes. One should note that since data are physically distributed over a cluster of computing nodes, when a path traverses a vertex located on a different computing nodes, a request for information has to be made through message passing. With the

slow inter-processor communication, ABySS typically consumes more time to assemble human genome than SOAPdenovo (87 hours vs 40 hours).

PASHA

PASHA is another distributed memory paralleled assembler (Liu et al., 2011). Noticing that the most time consuming part in assembly is generating and distributing *k*-mers, and constructing and simplifying the distributed de Bruijn graph. PASHA concentrates its effort on parallelizing these two stages to improve its efficiency.

K-mer representation in PASHA is similar to ABySS by assigning numeric value to bases {A, C, G, T}. Load balancing can hardly be achieved because using hash value to module the number of processes significantly relies on how hash function is implemented. PASHA uses a sorted vector data structure to store the k-mers and their graph-related information. Unlike ABySS to check the existence of all possible neighbors of a k-mer, PASHA builds linkage directly from the adjacency information of k-mers in the input reads. This method wipes out the possibility of introducing spurious edges at the expense of lots of file I/O between disk and memory. Graph simplification and scaffolding in PASHA are inherited from Velvet, but uses multithreading to speedup.

PASHA only allows a single process for tasks such as bubble merging, contig generation and scaffolding, which limits its degree of parallelism.

Chapter 2

PPLAT – An integrated distributed-memory parallel platform

2.1 Introduction

Parallel computing is a type of computation in which a vast amount of calculations are carried out simultaneously, under the assumption that a large problem can be divided into many smaller ones, which are then solved at the same time (Almasi and Gottlieb, 1988).

Traditional computer software has been written for serial computation. That is, to solve a problem, an algorithm is built and implemented as a discrete series of instructions, which are then executed on a central processing unit (CPU) in a sequential manner, i.e. only one instruction is executed at a time - after that instruction is finished, the next one is proceeded (Barney, 2010). Nowadays, for large computation tasks, parallel computing that simultaneously uses multiple computing resources has been developed. This is accomplished by breaking the computing task into independent subtasks that can be solved concurrently. The computing resources are typically a single computer with multiple processors, or several networked computers.

In modeling, simulating and understanding complex, real world phenomena, parallel computing has several advantages over serial computing.

- Efficiency: Parallel computing can greatly speed up task execution. Parallel clusters can also be built from commodity hardware, which costs much less than supercomputers.
- Capability: Parallel computing is able to solve complex problems, which are difficult or even impossible for a single computer with the limitation of physical memory and frequency scaling.
- Concurrency: parallel computing resources can work on many things simultaneously while a single computer can only work one thing at a time.
- Flexibility: Parallel computing can use not only local computing resources, but also those over a network.

Over the past few decades, parallel computing has been employed in many areas of science and engineering. Also, recent industrial and commercial success by introducing parallel computing is persuasive. The ever faster networks, distributed systems, and multiprocessor computer architectures demonstrate that parallelism is the future of computing.

Parallelism is an abstract idea of using more than one flow of instructions to complete a computation task. The critical aspect of all parallel techniques is how to efficiently and effectively communicate between flows. In the following, we will describe more details about general parallel computing and our designed distributed-memory parallel software platform - PPLAT.

2.2 Parallel Computing

2.2.1 Terminologies

Hereafter, the following terminologies will be used:

Node: refers to a standalone "computer", usually comprised of multiple CPUs/processors/cores, memory, network interfaces, etc. Many nodes networked together to form a cluster.

Task: typically refers to a program or program-like set of instructions to be executed by a processor. A parallel program consists of multiple tasks running on multiple processors.

Shared Memory: refers to a computer architecture where all processors have direct access to the same physical memory. In a programming sense, it describes a model where parallel tasks all have the same logical memory that can be directly addressed and accessed.

Symmetric Multi-Processor (SMP): refers to a shared memory hardware architecture where multiple processors share a single address space and have equal access to all resources.

Distributed Memory: refers to network based memory access for physical memories that are not the same. In a programming sense, tasks can only logically access local machine memory and must use communications to access memory on other machines where other tasks are executing.

Communication: refers to a way of accomplishing data exchange that is often needed in parallel tasks, such as through a shared memory bus or over a network. The actual event of data exchange is commonly referred to as communications regardless of the method employed.

Synchronization: refers to coordination of parallel tasks in real time, often associated with communications. Synchronization is usually implemented usually by establishing a synchronization point within an application where a task may not proceed further until another tasks reach the same or logically equivalent point. In general, synchronization involves waiting for at least one task, therefore can cause a parallel application's execution time to increase.

Parallel Overhead: refers to the amount of time required to coordinate parallel tasks, as opposed to "real" computation time. Parallel overhead includes factors such as task start-up and termination time, synchronization, communications, etc.

Scalability: refers to the ability of a parallel system to demonstrate a proportionate increase in speedup as more computing resources are included. Factors such as, memory-CPU or communication network bandwidths, application algorithms and parallel task coordination can affect the scalability.

Observed Speedup: observed speedup of a parallelized code is defined as:

$$Observed\ Speedup = \frac{wall - clock\ time\ of\ series\ execution}{wall - clock\ time\ of\ parallel\ execution}$$

It is one of the simplest and most widely used indicators for a parallel program's performance.

2.2.2 Parallel computer memory architectures and models

There are generally three types of memory architectures: shared memory, distributed memory and hybrid distributed-shared memory.

Shared memory

In shared memory architecture, all processors share same memory resources as global address space while operating independently. There are certain advantages to this architecture: programming is relatively easy because of global address space; data sharing between tasks is fast and uniform without redundant copies. However, its primary issue is lack of scalability in CPUs in terms of memory usage. As the number of CPUs increase, they can cram the traffic on the shared memory-CPU path geometrically; also, for cache coherent systems, more CPUs can geometrically increase traffic associated with cache/memory management. To guarantee accurate access of global memory, synchronization or memory locking are usually needed to prevent paging and swapping. Last but not the least, it becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors. Figure 4 shows an example of shared memory machine.

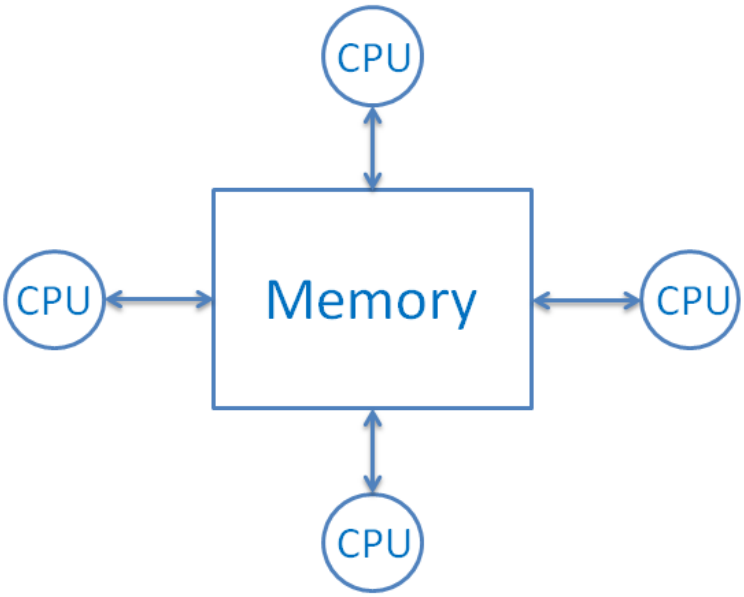

Figure 4: Example of shared memory architecture.

Shared memory architecture involves multiple CPUs having equal access to the same memory.

Multithreading is one type of parallel model widely used with the shared memory in multiprocessing systems. A thread is a flow of instructions with its own stack to keep a record of local variables and function calls, and communicates with the other flows implicitly through shared global memory. Different threads are executed at the same time, leading to efficient use of resources of the system. However, it should be noted that since threads share the memory space, synchronization may be required to ensure that no more than one thread is updating the same memory address at any time.

Distributed memory

For all processors in distributed memory architecture, they have their own local memory, so there is no concept of global address space. Memory address in one processor do not map to another, so a communication network is required to connect inter-processor memory. It is usually the programmer's responsibility to explicitly define how and when data is communicated when a processor needs access to data in another processor, and likewise, synchronization between tasks. Advantages of distributed memory architecture include: 1) when the number of processors increase, the size of memory can increase accordingly, 2) each processor can readily access its own memory without interference and without the overhead incurred with trying to maintain global cache coherency, and 3) it can be built with commodity hardware.

As of disadvantages, data communication between processors has to be carefully implemented. Data located on a different processor takes longer time to access than local data, as communication between processors are relatively slow. Also, new algorithms and designs may be required because those based on global memory sometime are hard to apply here. Figure 5 shows an example of shared memory machine.

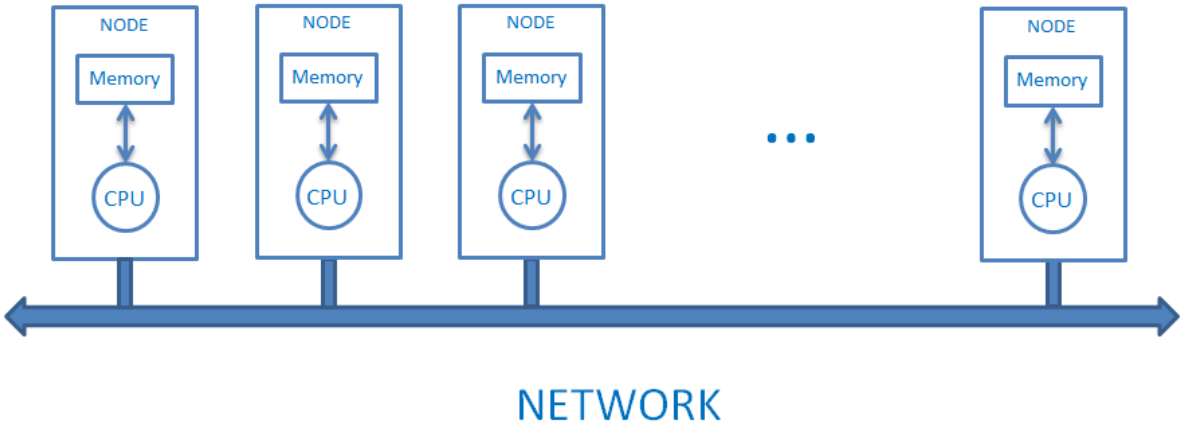

Figure 5: Example of distributed memory architecture.

Distributed memory architecture involves one CPU having its own memory to form a node, then nodes are connected by network to form a cluster. CPU can access other memories only by sending requests to other CPUs.

Message passing is one type of parallel model widely applied with distributed memory in computer clusters. This model has the following characteristics: a set of tasks resides on their own local memory during computation on the same physical machine and/or across a number of machines; tasks exchange data through communications by sending and receiving messages; cooperative operations needed to perform data transfer. Message Passing Interface (MPI) is an implementation of message passing model and has its own standardized protocol that is widely accepted.

Hybrid distributed-shared memory

A hybrid distributed-shared memory system consist of multiple independent SMP modules which are connected by a general interconnection network. The most advanced computers in the world today employ both shared and distributed memory architectures.

This architecture possesses the advantages of both shared memory architecture and distributed memory architecture, as well as their disadvantages. Increased scalability comes with complicated programming efforts. Figure 6 shows an example of distributed-shared memory machine.

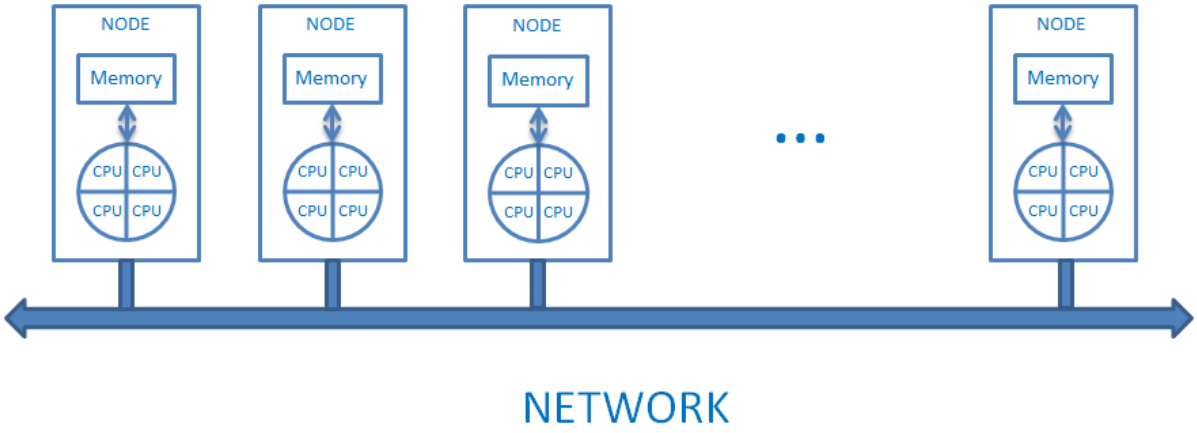

Figure 6: Example of hybrid distributed-shared memory architecture.

Hybrid distributed-shared memory architecture involves multiple CPUs having same access to same memory to form a node, then nodes are connected by network to a cluster. CPUs on the same node have access to its memory, access other memories only by sending requests to other CPUs.

A common hybrid model is the combination of the message passing model with the threads model: threads perform computationally intensive modules using local data, whereas communications between processes on different nodes for data transfer over the network using MPI. This hybrid model lends itself well to the most popular hardware environment of clustered multi-core machines.

2.2.3 Parallel program designs

The first step in developing parallel program is to understand the problem to be solved and determine whether the problem can actually be parallelized. For example, heavy data

dependency is an inhibitor to parallelism. The core and bottleneck in the program need to be identified. A careful consideration of these factors is key to a successful parallel program, which usually involves following steps: partitioning, communications, synchronization and load balancing.

Partitioning

Parallel processing requires breaking a problem to discrete flows of work that can be assigned to different tasks, known as partitioning. Two basic principles are usually employed to partition computational work among parallel tasks: domain decomposition and functional decomposition. In domain decomposition, the data associated with a problem is decomposed, and then each parallel task works on a portion of the data. In functional decomposition, the problem is decomposed according to the essential jobs, and then each task performs a portion of the overall jobs.

Communications

When designing inter-task communications, many important elements need to be considered:

- Cost of communication: is dependent on a variety of features including the programming model semantics, the network topology, data handling and routing, and associated software protocols. Message passing between nodes are much more time consuming so it may not be a good idea if there are too many communications involved in the design. Communications usually require some type of synchronization between tasks, which would put tasks on hold. Also, heavy message passage in the traffic may jam up the network, results in longer communication time.

- Latency and bandwidth: latency is the time it takes to send a minimal message from one point to another; bandwidth is the amount of data that can be communicated per unit of time. Sending many small messages can cause latency to dominate communication overheads. It is therefore more efficient to package small messages into a larger message, thus increasing the effective communication bandwidth.
- Synchronous vs Asynchronous communications: Synchronous communications require exchanging agreements between tasks that are sharing data, often referred to as blocking communication since other work must wait until the communications have completed. Asynchronous communications allow tasks to transfer data independently from one to another, often referred to as non-blocking communication since other work can be done while the communications are taking place. Using non-blocking communication allows computation and communication to overlap in a single process, leading to improved performance.
- Communication Scopes: It has to be known that when and how each task would communicate with others. The scope can either be point to point, between one and another; or collective, same operations between multiple tasks in a common group, including broadcast, scatter, gather and reduction.

Synchronization

Synchronization is a mechanism that multiple processes are to join up or handshake at a certain point, in order to reach an agreement or commit to a certain sequence of action. Generally two types of synchronization are widely used: barrier and lock.

- Barrier: Each task performs its own work until reaches the barrier, then stop and wait until last task reaches the barrier. When all tasks are synchronized, they are released to resume working following certain instructions.
- Lock: Lock is usually used to securitize the access to global data or a section of code, that is, at any time, only one task owns the lock. The first-coming task sets the lock so it can safely access the data or code, other tasks attempting to access have to wait until the one owns the lock releases it.

Load Balancing

Load balancing attempts to optimize resource use by distributing approximately equal amounts of work among tasks so that all tasks are kept busy all the time. Just like the capacity of a barrel with staves of unequal length is limited by the shortest stave, running time of a parallel program with barrier synchronization point is determined by the slowest task. Equally partition the work each task receives and/or use dynamic work assignment are helpful to achieve load balancing and increase overall computing efficiency.

2.3 PPLAT

Coding with multi-threading on shared-memory machines is relatively easy with the help of many integrated packages like OpenMP and POSIX Threads. However, writing parallel codes on distributed-memory clusters is difficult even with the help of MPI, and debugging can become quite challenging.

Hadoop is a large scale, open source software framework dedicated to scalable, distributed, data-intensive computing on clusters. The framework is to first partition large data

into smaller parallelizable chunks, and map each chunk to an intermediate value then reduce intermediate values to a solution. Hadoop adopts a master-slave paradigm, that is, a master node starts slave computations, and the slave computations return their results to the master. This loosely-coupled parallelism is suitable for problems with insignificant dependencies among the slave computations. Usually when point to point communication is necessary, master-slave paradigm cannot serve as an efficient solution.

Here we develop PPLAT, an integrated hierarchical multitasking parallel platform framework enabling easy asynchronous computing as well as both point to point and collective message passing. PPLAT provides a hybrid of multithreading-based and MPI-based solution especially for massive parallel processing (MPP) systems.

2.3.1 Design

PPLAT is built at the intention to reduce the coding complexity and facilitate high performance of parallel programs. The platform framework consists of three major components: a scheduler, multiple engines and an interconnection network. Figure 7 shows the design scheme of PPLAT, and detailed functionalities are stated below.

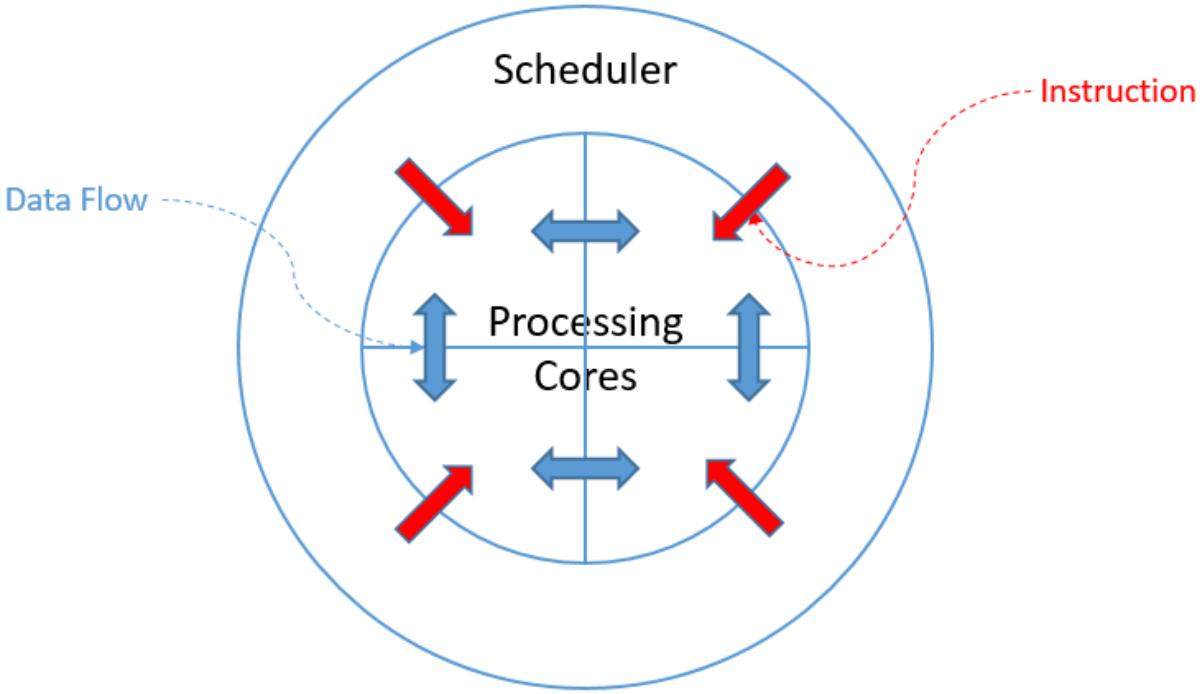

Figure 7: Design scheme of PPLAT

PPLAT has one scheduler and multiple processing cores, scheduler directs instruction/data to each processing core without doing actual computation. Processing cores do the actual computation and data storage and can have direct communication with other processing cores. Communication of instruction and data flow are achieved by intercommunication network.

Scheduler

Scheduler is the headquarter of the PPLAT, as it executes superiorly over the entire platform including network communication and processing cores' functionality. The scheduler has the following responsibilities:

- initiation of the whole project
- balanced distribution of jobs to processing cores by analyzing the input file sizes and characteristic
- supervision and synchronization of all processes to follow the same routine
- intermediate and post analysis of results fed back by the processing cores

- termination of the project

Control flows are distributed through interconnection network by wrapping instructions as coded messages. Detailed functionalities and/or actions to be executed are completely defined by users.

Processing Cores

Processing cores are the computing nodes that listen to the instructions from the scheduler, performing all the data storage and computing tasks. They can also exchange data freely with other processing cores with a built-in interconnection network. All the data flows are wrapped up as coded messages with different message types. Specific computations and communications are implemented by users.

Interconnection Network

Interconnection network here is an architecture-neutral and portable communicator protocol, mainly for packing and transferring data as well as predefined control flows. Our interconnection network package provides a library of data-oriented communication protocols and tools with flexible interfaces, including:

- task assignment interfaces that invoke and automatically supervise sequential routines on individual processing cores;
- data transfer interfaces that support collective and point-to-point communicators for importing and exporting data;
- control message interfaces that route control flows among scheduler and processing cores.

The synchronous and asynchronous messaging mechanisms are interwoven for interruptive control and high throughput data transfer.

2.3.2 Development

Our PPLAT is implemented in C++ for performance and portability. Interconnection network is leveraging on the standard MPI protocols and hidden in the scheduler and processing core class since standard MPI is too general and code redundant to handle complicated communication tasks, resulting in inefficient communication.

Multithreading is enabled to leverage the full computing power of the scheduler and processing cores. A base class of working thread is provided and defined as class Processor. Class Processor is for user to inherit and redefine the functionality in virtual function drive(). Both scheduler and processing cores are inherited instances from class Processor. Threading on scheduler and processing cores are also realized as inherited class of class Processor with detailed thread activities specified by users. Implementation details are discussed in next section.

In PPLAT, as soon as the project is invoked, scheduler and processing cores are automatically embedded with two communication channels: outgoing channel for sending out messages, and incoming channel for receiving messages. The channels are provided with simple application program interfaces (API) that are very easy to use: for P2P communication, after a message is encoded with specified destination and message type, the function addMessage() can move it to the outgoing channel for sending; the incoming channels listen to any incoming messages and store them when the queue is not empty. The function getMessage() is used to collect and decode the messages. In this way, users no longer need to write lengthy and tricky MPI functions, which saves substantial programming and debugging time.

These channels are indeed thread-safe lock-free queues that utilize the interconnection network to send/receive messages to their designated locations and from scheduler and other processing cores, respectively. PPLAT supports multithreading, which may bring the issue of multiple processes simultaneously accessing the same queue to push or pop messages. Lock-free-queue data structure do not rely on locks and mutexes to ensure thread-safety. Our program is set up so that each thread can always advance regardless of what the other is doing by the wait-free structure to avoid expensive locks and ensure data is perceived in the right order.

2.3.3 Implementation

To better underline the structure of PPLAT, below is a brief bottom-up description of the implementation of the framework. Although only non-blocking communication is described below, blocking communication is also supported in PPLAT. Figure 8 shows the Unified Modeling Language (UML) diagram of PPLAT.

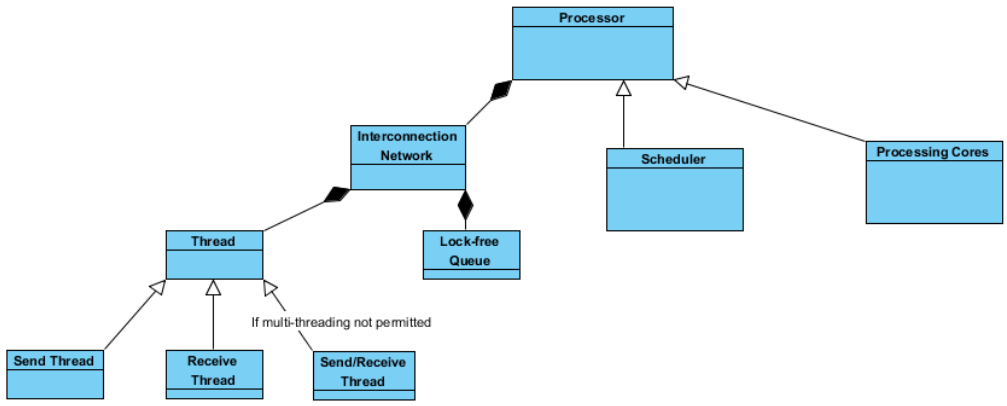

Figure 8: The UML diagram of PPLAT

mtpiThread is the fundamental threading class; message send and/or receive class inherit mtpiThread class and do actual message passing; messages are stored in lock free queue; the netComm class is the interconnection network, combines message passing and message storage; scheduler and processing core inherit from processor class, which does actual operations specified by users with incorporated interconnection network.

Fundamental classes

ThreadWrapper class is a wrapper of C++ standard library <*thread*> for easier use. It is the base class and could be subclassed for different tasks.

```
class ThreadWrapper
{
public:
   ThreadWrapper();
   template <typename T>
   ThreadWrapper(T&){}
   void start();
   void join();
   void finish();
   void yield(){std::this_thread::yield();}
   bool joinable(){return this->_joinable;}
   void enableJoin(){this->_joinable= true;}
   static void runThread(void *p);
   thread::id self();
   bool  isfinished() {return !_inRunning;}
   virtual~ThreadWrapper()=0;
private:
   virtual void run();
protected:
   bool            _inRunning;
   thread*         _pThread;
   bool            _joinable;
};
```

Class mtpiThread is inherited from the class ThreadWrapper, serving as a fundamental class for working threads at each node. Basically, this thread belongs to a MPI process and communication group.

```
class mtpiThread : public ThreadWrapper
{
public:
   mtpiThread(MPI_Comm group);
   int getRank()const{
      int me = 0;
      MPI_Comm_rank(_group,&me);
      return me;
```

```
    }
    int groupSize()const{
        int sz = 0;
        MPI_Comm_size(_group,&sz);
        return sz;
    }
    MPI_Comm group()const{return this->_group;}
    virtual ~mtpiThread(){}
private:
    MPI_Comm                _group;
};
```

Message send and/or receive class

Support for multithreading in MPI varies by the MPI implementation (MPICH, OPEN MPI, etc.) and its version user installed. When MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provide) is invoked, if provide < MPI_THREAD_MULTIPLE, multiple threads may call MPI at the same time without restrictions, which is not supported, and therefore we cannot send/receive messages; Otherwise, send and receive processes could work simultaneously, leading to dramatic improvement on the message passing efficiency. For example, MPI_THREAD_MULTIPLE is included if Open MPI is configured with the --enable-mpi-thread-multiple configure switch.

If MPI multithreading is supported, PPLAT operates message sending and receiving on two different threads, implemented in messageSender and messageReceiver class respectively. Both classes are inherited from the mtpiThread class. messageSender can grab a message from the shared queue, send it and release the corresponding memory.

```
class messageSender : public mtpiThread
{
public:
    messageSender(lockFreeQueue<rawMessage*>&  que,
                  lockFreeQueue<rawMessage*>&  inque,
                  bool& finished,int maxTry, MPI_Comm group,comMode m):
```

```
            mtpiThread(group),_que(que),_inque(inque),
            _finished(finished),_maxTry(maxTry),_mode(m){

  }
private:
  void run();

private:
  lockFreeQueue<rawMessage*>&          _que;
  lockFreeQueue<rawMessage*>&          _inque;
  bool&                                _finished;
  int                                  _maxTry;
  comMode                              _mode;
};
```

Pseudo code for messageSender run function:

```
WHILE process not finish OR outgoing queue not empty
        IF get one non-empty message THEN
                IF the message is to myself THEN
                        add to incoming queue
                ELSE
                        non-blocking MPI_P2P::ISend to destination
                ENDIF
        ELSE
                thread yield
        ENDIF
ENDWHILE
```

messageReceiver listens to incoming messages. It receives messages from the network, and add them to incoming queue.

```
class messageReceiver : public mtpiThread
{
public:
  messageReceiver(lockFreeQueue<rawMessage*>& queue,bool& finished,
                  int maxtry,MPI_Comm group,commode m):
        mtpiThread(group),_que(queue),_finished(finished), _maxTry(maxtry),_mode(m){
  }
private:
  void run();
private:
  lockFreeQueue<rawMessage*>&          _que;
```

```
    bool&                                   _finished;
    int                                     _maxTry;
    comMode                                 _mode;
};
```

Pseudo code for messageReceiver run function:

```
WHILE process not finish
        try receive incoming message from the network
        IF get no message THEN
                jump to while loop
        ENDIF
        add the message to incoming queue
ENDWHILE
```

If MPI multithreading is not supported, PPLAT operates message sending and receiving on the same single thread, implemented in messageSenderReceiver class, inherited from mtpiThread class.

```
class messageSenderReceiver:public mtpiThread
{
public:

messageSenderReceiver(lockFreeQueue<rawMessage*>&,lockFreeQueue<rawMessage*>&,bo
ol&,int,MPI_Comm,comMode);
private:
    void run();
private:
    lockFreeQueue<rawMessage*>&         _inque;
    lockFreeQueue<rawMessage*>&         _outque;
    std::vector<rawMessage*>            _waitMessages;
    std::vector<MPI_Request>            _bufferedReq;
    int                                 _buffered;
    int                                 _bufferCap;
    bool&                               _finished;
    int                                 _maxTry;
    comMode                             _sendMode;
};
```

Pseudo code for messageSenderReceiver run function:

```
WHILE process not finish OR outgoing queue not empty
        try receive incoming message from the network
        IF get a message THEN
                add the message to incoming queue
        ENDIF
        IF send buffer is full THEN
                clean the send buffer
        ELSE
                IF get one non-empty message THEN
                        IF the message is to myself THEN
                                add to incoming queue
                        ELSE
                                non-blocking MPI_P2P::ISend to destination
                        ENDIF
                ENDIF
        ENDIF
ENDWHILE
```

Message framework class

MPI predefines its primitive data types, such as MPI_CHAR, MPI_INT, etc., which corresponds to character, integer, etc. Since too many datatypes makes the program complicated to write and difficult to debug, one safe and robust choice is to encode all data to be sent into the string type, and then decode them back to their original form after receiving. This involves user defined methods for encoding and decoding. More recently, a more flexible way is using google protocol buffers, a language-neutral, platform-neutral, extensible mechanism for serializing structured data. More details about protocol buffers can be found in future work section.

In PPLAT, we support sending and receiving C++ string type by implementing a rawMessage class. The process is rather simple: construct a rawMessage instance, specify the source, destination, message itself as well as the message tag, and then add it to the outgoing queue. User can define their own tags in enumerator MTPI_Task_Tag located at mtpi_types.h.

```
class rawMessage
{
```

```
public:
   rawMessage(void):_src(-1),_dest(-1),_msg(""),_tag(UNKNOWN_TAG){}
   rawMessage(int src,int dest,const std::string& msg,MTPI_Task_Tag tag): _src(src),
_dest(dest), _msg(msg), _tag(tag) {}
   rawMessage(const rawMessage& cp){
      this->init(cp);
   }
   rawMessage& operator=(const rawMessage& cp){
      this->init(cp);
      return *this;
   }
   ~rawMessage(){
   }
public:
   int source()const{return this->_src;}
   int destination()const{return this->_dest;}
   const std::string& message()const{return this->_msg;}
   MTPI_Task_Tag tag()const{return this->_tag;}
   size_t       size()const{return this->_msg.size();}
private:
   void init(const rawMessage& cp){
      this->_src = cp.source();
      this->_dest = cp.destination();
      this->_msg.assign(cp.message());
      this->_tag = cp.tag();

   }
private:
   int                  _src;
   int                  _dest;
   std::string          _msg;
   MTPI_Task_Tag        _tag;
};
```

Interconnection network class

The interconnection network class combines the lock-free send/receive queue and message sender/receiver to take care of message passing, embedded in scheduler and processing cores, implemented as netComm class:

```
class netComm
{
```

```
public:
   netComm(MPI_Comm,int quesize,int maxTry,comMode m,int threadNum);

   // Send message to network
   bool addMessage(rawMessage* m);

   // Grab one message from the network
   // indicator shows whether this action is success or not
   rawMessage* getMessage(bool& succ);

   // pause the receive and send threads
   void stop();

   // restart the send and receive thread
   void resume();

   //check if the in que and out que is empty
   bool empty(){
      return this->_inque.empty() && this->_outque.empty();
   }
   ~netComm(){
   }
private:
   lockFreeQueue<rawMessage*>              _inque;    // message received from other processes
   lockFreeQueue<rawMessage*>              _outque;   // message ready for sent
   std::shared_ptr<messageSender>          _sender;
   std::shared_ptr<messageReceiver>        _receiver;
   std::shared_ptr<messageSenderReceiver>  _sendRecver;
   bool                                    _finished;
   MPI_Comm                                _group;
   int                                     _maxTry;
   bool                                    _running;
   comMode                                 _mode;
   int                                     _numThread;
};
```

Processor class

This is the prototype and building block for scheduler and processing cores, since they are by nature the same but with different responsibilities. Both scheduler and processing cores class inherit this class and detailed operations are specified by users. This class provides barrier operation, an interconnection network and other parameters. Notice that netComm instance

_network are set to be static since all use the same interconnection network. Implemented as

Processor class:

```
class Processor : public mtpiThread
{
public:
   typedef std::unordered_map<int,int> LookupTable;
public:
   Processor(std::shared_ptr<Parameter> params);
   Processor(Processor* me):Processor(me->params()){}
   virtual  ~Processor(){}
   bool              addMessage(rawMessage* m);
   rawMessage*    getMessage(bool& succ);
   rawMessage*    Barrrior(rawMessage* msg){return this->barriorImp(msg);}
   virtual bool      finished()const{throw 1;}
   int               getProcessID(int prefix){
      assert(Processor::_rankLookUpTable.count(prefix));
      return Processor::_rankLookUpTable.at(prefix);
   }
   std::shared_ptr<Parameter>     params(){return this->_params;}

   // initialize the network configuration and look up table
   static void initGlobal(std::shared_ptr<Parameter> params);
private:
   virtual void drive(){throw 1;}
   void run(){
      this->_network->resume();
      this->drive();
   }
private:

// barrier implementation
   rawMessage* barriorImp(rawMessage*);
   void init();

public:
   static std::shared_ptr<netComm>              _network;
   static LookupTable                           _rankLookUpTable;
private:
   std::shared_ptr<Parameter>                   _params;
};
```

As scheduler and processing cores classes are inherited from this Processor class, user only need to implement the virtual drive() function. For example, in the main function, user can use the following code:

```
if(rank == 0){
	th = new scheduler(param);
	th->start();
	th->join();
}
else{
	th = new processing_cores(param);
	th->start();
	th->join();
}
```

2.3.4 Applications of PPLAT

PPLAT is an integrated parallel framework that allows distributed data storage and processing in a clusters of computers with simple APIs and great flexibilities. Its scalability up to thousands of cores, each of which provides local storage and computation as well as enabled direct communications, offers solutions both to repetitive work and big data.

In a sense, MapReduce algorithm is a special case in PPLAT. Here we will show some applications of PPLAT. Tests are executed on a computing cluster with 8 nodes connected by InfiniBand switch. Each cluster node contains two twelve-core 2.6 GHz CPUs and a 128GB RAM.

Communication efficiency

We test PPLAT on sending different size of messages on different number of cores. Each core send *one* message to and receive *total core number-1* messages from other cores, then

barrier to wait for others to finish, repeat this process for multiple rounds. The time recorded in Table 1 is the average wall-clock time, from the start to the completion of this test.

Table 1: Communication efficiency of PPLAT

	Message Size 500KB	Message Size 5MB
24 Cores	18 seconds	24 seconds
48 Cores	42 seconds	44 seconds

One of the key observation from Table 1 is that the time needed to transfer a big chunk of data is relatively the same as to transfer a small chunk of data. Even though the data volume increases by 10 times, the time used just increases by 1.3 times for 24 cores and 1.05 times for 48 cores. This special characteristic motivates us to come up with better algorithm to make larger messages and it is one of the reasons we group the k-mers in the next chapter.

Testing and simulations using PPLAT

Simulation studies in statistics and finance are extremely common while time consuming, such as resampling (like, permutation test) and Monte Carlo simulation. These tasks often involve iterative evaluations. If each round would be complex and cost considerable amount of time, the whole processing time would sum up to a massive number. A natural solution is to use parallel clusters with each node simulating one case, and then results are pooled together to summarize a final solution. However, rewrite serialized code to parallel code is hard and sometime impossible for people with little parallel computing experience.

PPLAT offers a solution to this embarrassingly parallel situation. Given that there are numerous interface libraries to help call C++ scripts from other programming languages, such as Boost.Python for Python and Rcpp for R (Eddelbuettel et al., 2011), and the work on each processing core has no communication at all, the coding would be extremely easy.

For example, suppose we want to perform Monte Carlo simulation on calculating the price of complicated exotic derivatives. Suppose the function written in R is derivPrice(), which returns results of 10 simulated cases in a vector, and a total of 200 nodes are used to run simultaneously. Input are specified in params. The added C++ code in processing_cores would be:

```
#include <Rcpp.h>
Using namespace Rcpp;
NumericVector callFunction(NumericVector x, Function f){
        NumericVector res = f(x);
        Return res;
}
```

The drive() function is simply:

```
void engine_thread::drive(){
        std::vector<double> vec = Rcpp::as<std::vector <double> >(
        callFunction(params, derivPrice));

        // Gather to scheduler for final answer or just output
        …
}
```

Pool of tasks using PPLAT

In the case of two processes: master process and slave process. Master process holds pool of tasks for slave processes to do, sends slave a task when requested, and collects results from

slaves. Slave process gets task from master process, performs computation, and sends results back to master. Scheduler and processing cores in PPLAT also work in this situation.

Slave processes do not know before runtime which portion of work they will handle or how many tasks they will perform, dynamic work assignment helps achieve load balancing at run time: the faster response slave will get more work to do.

A Pseudo code for solution:

```
IF I am scheduler THEN
        send every processing core a task
        WHILE there are still more tasks OR not all collected
                try receive incoming message from the network
                IF can't get a message THEN
                        jump to while loop
                ENDIF
                collect results from message and send next job to that processing core
        ENDWHILE
        terminate
ELSE
        WHILE not terminated
                try receive task from the network
                IF can't get a message THEN
                        jump to while loop
                ENDIF
                process task and send results to scheduler
        ENDWHILE
ENDIF
```

Here we test on the following scenarios, scheduler holds 100 tasks for processing cores to do, while processing cores come at first-come-first-serve basis to get the tasks and process. In the first scenario, all 100 tasks require 10 seconds to finish; in the second scenario, the time to finish each task is random from 1 second to 100 seconds. We will show how PPLAT performs.

Table 2: Time needed to finish 100 equal time pool of tasks

	10 Cores	20 Cores	50 Cores
Execution Time	100 seconds	50 seconds	21 seconds

In Table 2, the serial processing of this 100 equal time pool of tasks requires 1000 seconds, PPLAT achieves almost linear speedup as the number of processing cores increase.

Table 3: Time needed to finish 100 varying time pool of tasks

	10 Cores	20 Cores	50 Cores
Theoretical Time	573 seconds	307 seconds	159 seconds
Execution Time	573 seconds	308 seconds	160 seconds

In Table 3, time to finish each task is sampled from 1 to 100 with replacement and the serial processing of this 100 varying time pool of tasks requires 5304 seconds, Theoretical time assumes no communication cost, execution time is the wall-clock time recorded. PPLAT shows nice scheduling and execution ability with little communication lost.

Numerical analysis problems using PPLAT

For big-data problems in numerical analyses , like matrix-matrix multiplication, numerical integration and partial differential equation, they are usually too big for a single computer. Parallel processing makes this calculation possible and communication among the

tasks is required. In this case, PPLAT can facilitate clean coding with simpler implementation and robust performance.

For example, the number of operation required to multiply an m*r matrix by an r*n matrix is m*n*(2r-1). For square matrices, it is of complexity $O(n^3)$. Partition the matrices into blocks, distribute blocks to processing cores, communicate with neighbors for data, compute locally and gather back to scheduler are all within the capability of PPLAT.

More complicated application

With the development of new technologies, genomic problems nowadays usually come with high throughput and complicated algorithms. In this thesis, we will show the PPLAT framework can be applied in the assembly problem, and demonstrate the performance of the PPASSEM, which is given in the next Chapter.

Chapter 3

PPASSEM– A paralleled assembly software for NGS data

3.1 Introduction

As shown in previous chapters, past few decades have seen fast development and tremendous growth in sequencing technologies, as well as massive parallel processing (MPP) systems. To address the computation time and memory constraints on *de novo* assembly of genomes with billions of base pairs, we intend to develop algorithms running on MPP systems that can fast and accurately assemble this vast amount of data.

On distributed parallel clusters, data are spread over a number of computer nodes. One critical aspect of these systems is that the time it takes to successfully exchange a big chunk of data is relatively the same as to exchange a small chunk a data, given the sufficient network bandwidth. Since communication in parallel assembly is inevitable and relatively time consuming, this special characteristic urges us to design new algorithms with reduced number of communications.

In this Chapter, we will present PPASSEM - an efficient genome assembler that employs both small-scale shared-memory multithreading and large-scale distributed-memory parallelism. PPASEM is mainly based on the de Bruijn graph data structure.

The general workflow in PPASSEM is given as follows: take input short read file(s), generate and distribute k-mers over processing cores, build the de Bruijn graph, iteratively condense the graph and correct the errors, and output the contigs. Figure 9 shows the pipeline of PPASSEM.

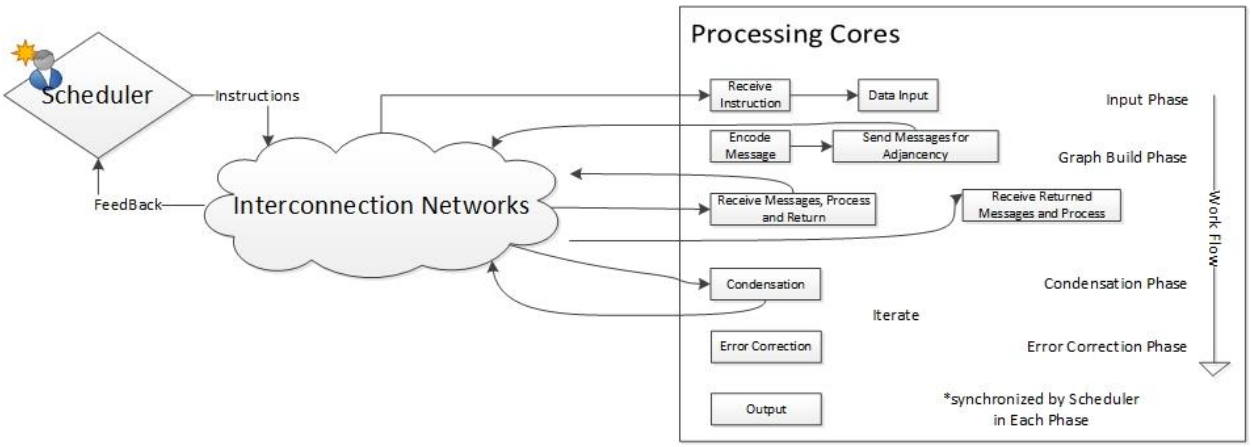

Figure 9: Pipeline of PPASSEM

PPASSEM generally has three stages: (i) k-mer representation and distribution; (ii) distributed de Bruijn graph building; and (iii) iterative graph condensation and error correction.

3.2 k-mer representation and distribution

The first phase in de Bruijn graph assembly is generating k-mers from reads file. PPASSEM contains a k-mer counting module that supports input files with the standard FASTA or FASTQ format. For small dataset, a concise python code is used for k-mer generation in a single computing node; however, for large dataset, single-node memory may not be big enough to hold all the k-mers, and a parallel module called PPKmer may be used to count the k-mers. PPASSEM can also take binary format k-mer files, like output from software JELLYFISH (Marçais and Kingsford, 2011).

When counting k-mers, both the occurrence and frequency of each k-mer are recorded. Frequency information are used at error correction phase and stored using one byte by trimming the counts of k-mers appearing more than 255 times to be 255 for memory purpose.

K-mers in PPASSEM are not stored as in its string form, since each char in C++ occupies 1 byte. Since there are {A, C, G, T} four bases in DNA and 4 equals 2^2, this echoes a natural translation between DNA bases and the binary representation. Therefore, the bases {A, C, G, T} are represented using 2 bits {00, 01, 10, 11}. If stored in string form, a k-mer of length 29 would take 29 bytes space to store; whereas using binary representation, the k-mer can be stored in an unsigned integer (uint_64), which in a 32-bits system only takes 8 bytes. This design saves the memory quite substantially, especially for large dataset with billions of k-mers. Additionally, the design also takes advantage of the bit-wise operation in C++, which is extremely fast.

Since each k-mer can have up to 8 neighbors, i.e., four possible one base extension of {A, T, G, C} in both directions, to store a k-mer's adjacency information efficiently, we use 1 bit (0/1) to represent the existence of each edge, which adds up to one byte per k-mer. At beginning, all adjacencies are set to be 0, not updated until the graph building phase. Figure 10 shows how adjacency information is stored.

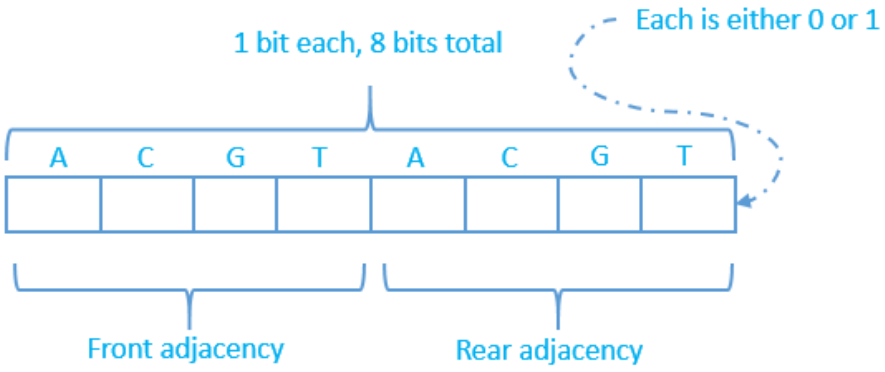

Figure 10: The adjacency information storage

A uint8_t is an integer type that can store 8 bits (one byte). First 4 bits are used to store front adjacency and latter 4 bits are used to store rear adjacency. If there is an adjacent k-mer it is a 1, otherwise, it is a 0.

The k-mer, its frequency and adjacency information are made to a struct type, stored together as key and value in hash table. Hash table has the feature of on average constant lookups in time. When searching for existence of a certain k-mer, O(1) complexity certainly outperforms search tree which complexity is O(logn). A good hash function and implementation algorithm are essential for good hash table performance, PPASSEM uses google sparse hash library.

A balanced load of k-mers among processing cores is of vital importance to the performance of distributed parallel assembly algorithm, in terms of both execution time and memory usage. In an unbalanced load, it is likely that some processes may consume more memory for k-mer storage or longer time for execution, thus resulting in a system failure due to memory leakage or program idle waiting for one or few nodes to finish. ABySS uses a naïve way of distribution, by module the hash value to the number of nodes to determine the index to assign the k-mer to one of the nodes, which may have serious unbalanced loading issue.

In PPASSEM, we notice the nature of adjacency in the graph is:

If two k-mers are adjacent, their number of 1s in binary representation differ by at most two.

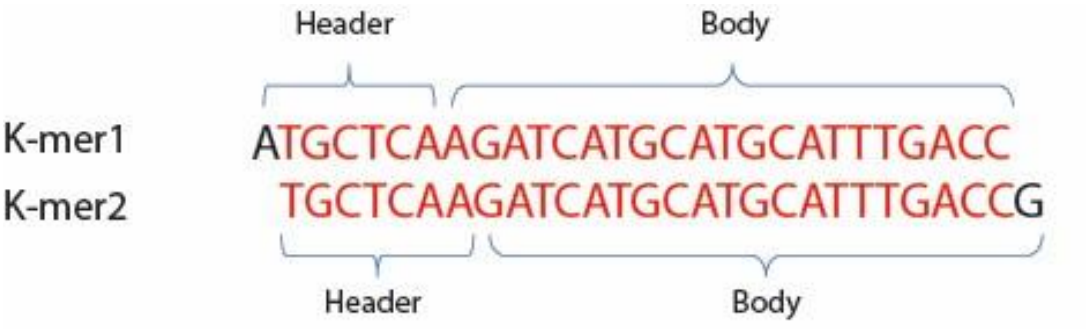

Figure 11: Two adjacent k-mers of length 28 with header length of 7

Two length 28 k-mers k-mer1 and k-mer2 are adjacent, all letters labelled in red are exactly the same, so it is the same for header and body.

As shown in Figure 11, since the two k-mers overlap by *(k-1)* bases, meaning that the bases in red have to be the same, in binary representation, difference in the number of 1s of two adjacent k-mers is the same as the difference in the number of 1s between the first nucleotide of k-mer1 and last nucleotide of k-mer2, which can only be one fo the 5 values: -2, -1, 0, 1, 2.

Hereby comes our unique grouping of k-mers: k-mers with same prefix (referred to as header with length of *l*) are placed together, which are what we called jobs; within each job, we group those k-mers with same number of 1s in the body part together to what we called bins. Notice that the difference in the number of 1s in the body part of two k-mers also differ at most by 2. Figure 12 shows the design of a job.

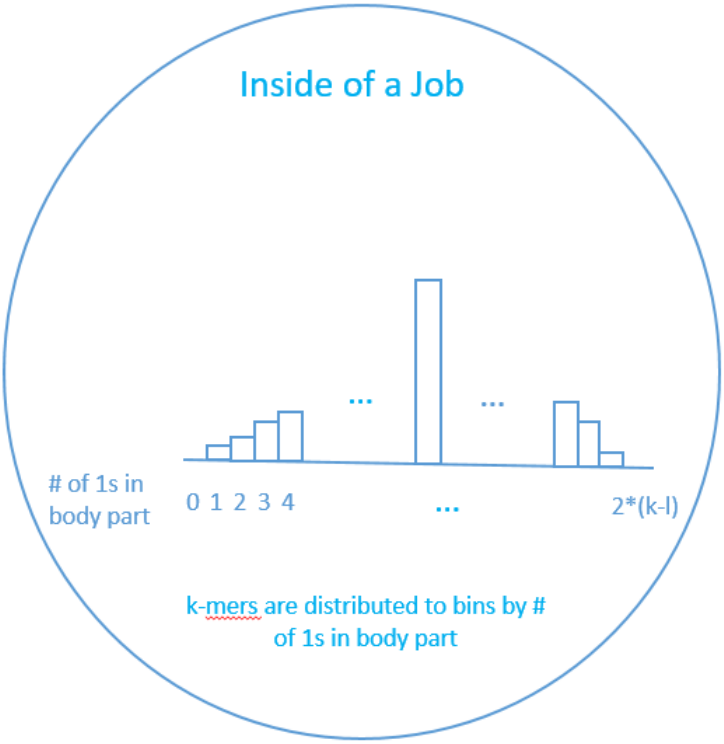

Figure 12: The design of a job

A job is comprised of 2*(k-l)+1 bins and each bin are loaded with k-mers based on number of 1s in k-mer's binary representation.

The advantages of this grouping and job-bin design are:

- For all the k-mers in the same job, their neighbors can only be found in at most 8 jobs; more interestingly, for k-mers in a particular bin, their neighbors c can only be found in at most 5 bins. Therefore, we significantly narrow the search space for restoring adjacency information. Actually in real-data applications, instead of searching in all 8 possible jobs, we only need to search 4 jobs for rear adjacencies for each job, because the front adjacencies of a job are updated when its front jobs are searching for rear adjacencies.
- Instead of passing single short messages between processing cores for each k-mer, in each bin we pack all these short messages into 4 long messages and deliver them to

the designated location (where destined job is stored at). This significantly reduces communications between processing cores.

Header length is specified as one of the input parameters, and the number of jobs depends on the choice of header length. For a given l, there are at most 4^l jobs. Since k-mer counting process enumerates the size of each job, jobs can then be distributed to processing cores in a dynamically balanced way based on their sizes. The program sorts the job size from large to small and each time the next job is given to the processing core that has the smallest total job size. The choice of l need to be cautious as it cannot be too small so that one job cannot be held by one single computing node. Although Korf's Complete Karmarkar-Karp algorithm (Korf, 1998) can give a theoretically better data partition it is quite complicated to implement. Our current dynamic balanced partition works fine even on human data. Figure 13 illustrate the data on processing cores.

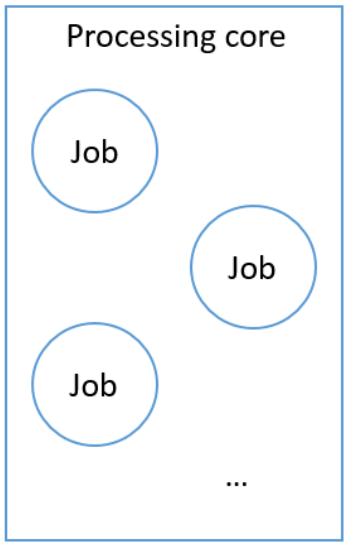

Figure 13: Data on processing cores

One or more jobs are distributed to each processing cores based on their size.

In actual implementation, each bin is a hash table and each job is a map of hash table. Where each job is located is in global information and shared to all processing cores by scheduler before assembly.

3.3 Distributed de Bruijn graph building

As discussed previously, k-mers are grouped and physically distributed at different computing nodes. The next phase is to restore the adjacency information of these k-mers.

Because one base extension in the rear direction of all k-mers in the same bin can only be found in 4 other jobs, we start by packing these k-mers in the same bin altogether into 4 discrete "request" messages and deliver the messages to the nodes that may contain their neighboring k-mers. For example, in job with header AAGT and bin 17, all k-mers' rear adjacencies can only be found in jobs with header AGTA, AGTC, AGTG and AGTT. All k-mers in bin 17 are packed into four strings/messages with each string goes to a specific job (k-mer starting with AAGTA goes to job AGTA, k-mer starting with AAGTC goes to job AGTC, etc.), then to fours rawMessage instances; finally, the addMessage() function is used to to put them into outgoing channel.

All processing cores work simultaneously, with one thread packing and deposing the messages to outgoing channel and the other thread retrieving and processing the "request" messages from incoming channel. Upon receipt, the "request" messages were then unpacked and each k-mer's adjacency information were searched within the possible bins. Notice that in the example above, within each of the neighboring jobs, rear adjacencies can only be found in bin 15, bin 16, bin 17, bin 18 and bin 19. If a rear adjacent k-mer is found, the front adjacency of this k-mer is updated by changing the corresponding flag from 0 to 1.

Since during the searching process, the existence of adjacencies is recorded for each k-mer, after the searching is completed, k-mers in the "request" messages, together with the existence or non-existence of adjacency information, are then packed into "response" messages and sent through the outgoing channel to where the "request" messages originate. Upon receiving the "response" messages, each k-mer retrieves all its one base rear extension and update the corresponding adjacency information. Figure 14 and Figure 15 show the design of request message and response message respectively.

The total messages exchanged in this phase is bounded by $2 * 4^l * (2(k-l)+1)*4$, where the first number 2 represents a send request always coupled with a receive request; the second term 4^l represents the total number of jobs; the third term $(2(k-l)+1)$ represents there are $(2(k-l)+1)$ bins in each job; the last number 4 represents 4 messages coming out for each bin. By this design, messages used to build the graph is drastically reduced, compared to other parallel assemblers, such as ABySS and PASHA, which try to pack more fragmented messages.

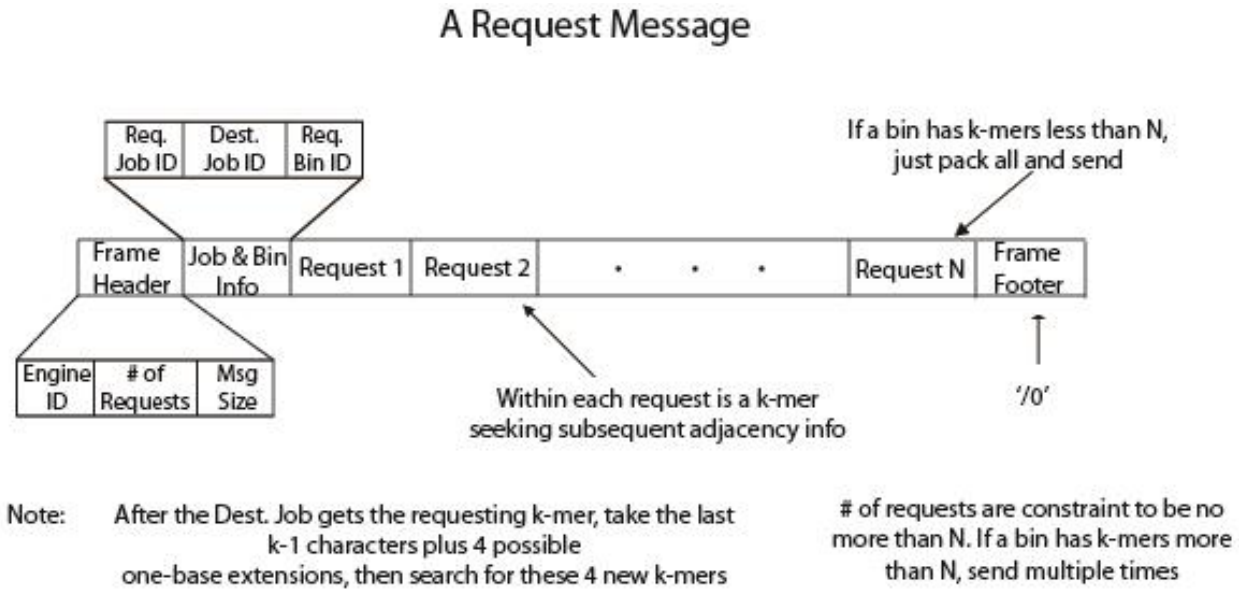

Figure 14: Design of request message

A request message is a string by its nature. Data in the message are separated by delimiter to avoid confusion. A user defined encode method is required to guarantee safe data transfer.

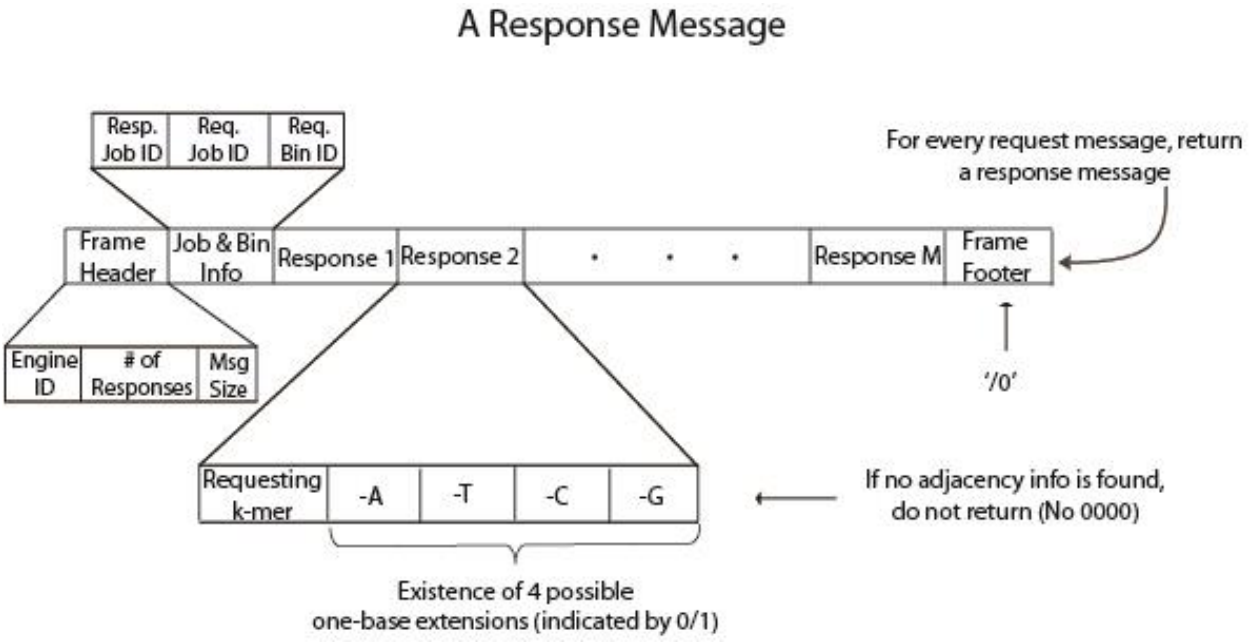

Figure 15: Design of response message

Like request message, a response message is also a string by its nature. Data in the message are separated by delimiter to avoid confusion. A user defined decode method is required to guarantee safe data transfer.

When every bin finishes these operations in each job, the adjacency information was restored and the distributed de Bruijn graph was established.

3.4 de Bruijn graph condensation

Right before the start of this condensation phase, nodes in the graph are k-mers with same length *k*. However, each k-mer only carries information about one useful base by the overlapping property of de Bruijn graph along an unambiguously extended path, resulting in high redundancy. In this phase, we merge those non-ambiguity linear chains of nodes into single node with longer length and more information. After "condensation" of the information, the nodes in the graph are contigs with varying lengths.

The process starts from the “dead-end” k-mers that have no front adjacency with the following steps:

- Merge linear chains of k-mers (one incoming edge and one outgoing edge) into contigs all the way until ambiguities occur (k-mers with multiple adjacencies in either direction) and store the contigs on the same computing cores along with the ambiguity nodes.
- Then from the ambiguity nodes, continue merging linear chains of k-mers until new ambiguities occur: the contigs were stored with the ambiguity nodes as usual, but also had the information of their prior ambiguities nodes; plus, inform the prior ambiguities nodes with current contigs information.
- If an ambiguity node is previously visited, STOP.
- Contigs that reaches “dead-end” k-mers with no rear adjacency were stored back to the ambiguities nodes where they originated.
- Propagate this process until all nodes in de Bruijn graph have been traversed, similar to a domino effect.

More points regarding the above process:

- Contig here is now new data structure, with ID to be the last k-mer it takes in. Ambiguity node is a special contig and keep the ID of k-mer. The quality of a contig is the average frequency of k-mers it condensed.
- The adjacency of two contigs is stored in each contig’s data structure, including the other contig’s location, ID, length and quality.
- Bins and jobs are no longer necessary, so old k-mer data can be safely removed.

- For self-ended-loop path, which starts and ends at the same nodes is possibly resulted by repetitive structure in the genome, we simply remove and keep it in the log for future analysis.

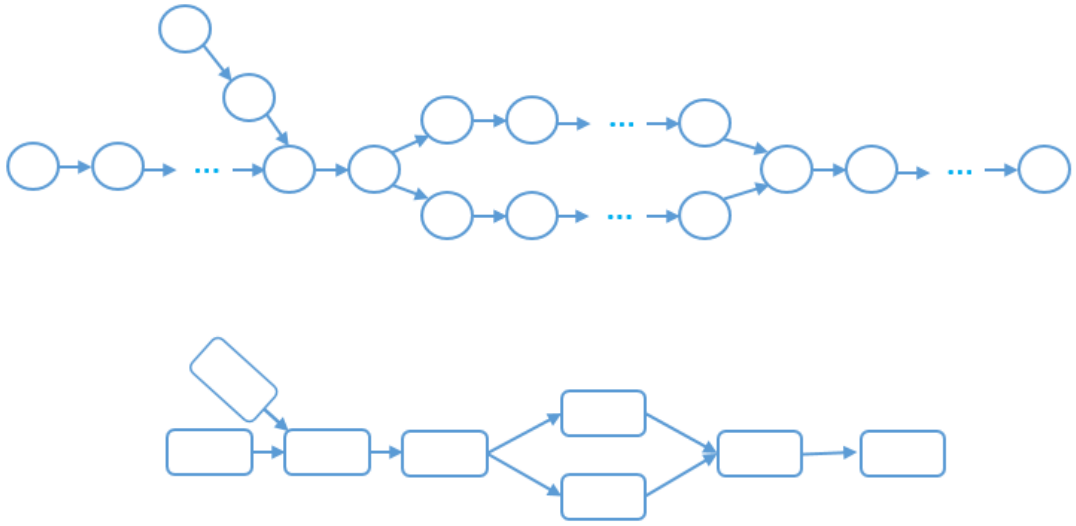

Figure 16: Example of Condensation

The upper one is the de Bruijn k-mer graph (each circle is a k-mer) while lower one is the of contig graph (each rounded rectangle is a contig) after condensation.

Figure 16 shows an example of the effect of condensation, upper one is the graph of k-mers (each circle is a k-mer) before condensation and lower one is the graph of contigs (each rounded rectangle is a contig) after condensation.

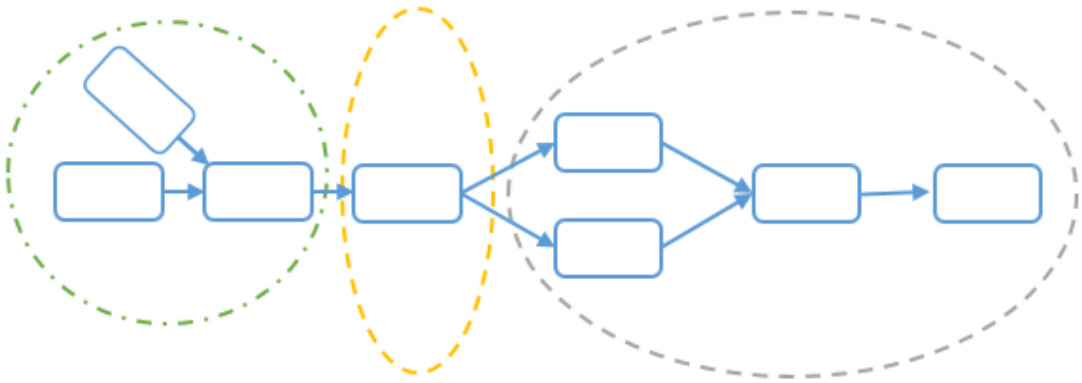

Figure 17: Location of contigs

Following the example of Figure 16, here we show the location of the contigs. This condensation design has significant impact for error correction later.

Figure 17 shows the location of contigs after condensation. This design of contigs storage is important for operations in error correction phase. Details are covered in next section.

This condensation is the core phase in PPASSEM and its implementation is pretty tricky to include all possible scenarios. Condensation has the subsequent advantages:

- significantly removes the redundancy thus leaves much less nodes in the graph
- makes the following error correction phase easier without having communication between computing nodes

3.5 Error correction of the graph

As seen in Chapter 1, assembly is confounded by several read-world factors. In the graph, some special genomic structures need to be handled carefully, including: 1) tips, short and low-

coverage dead-end path that possibly resulted by sequencing errors on either end of a read; 2) bubble, a path divergence at a location that converges later and possibly resulted by sequencing errors in the middle of a read or structural variation in genome (like, SNPs and INDELs); 3) scissor bolt (or frayed rope pattern), a vertex has multiple indegrees and outdegrees on both side that possibly resulted by small repeats.

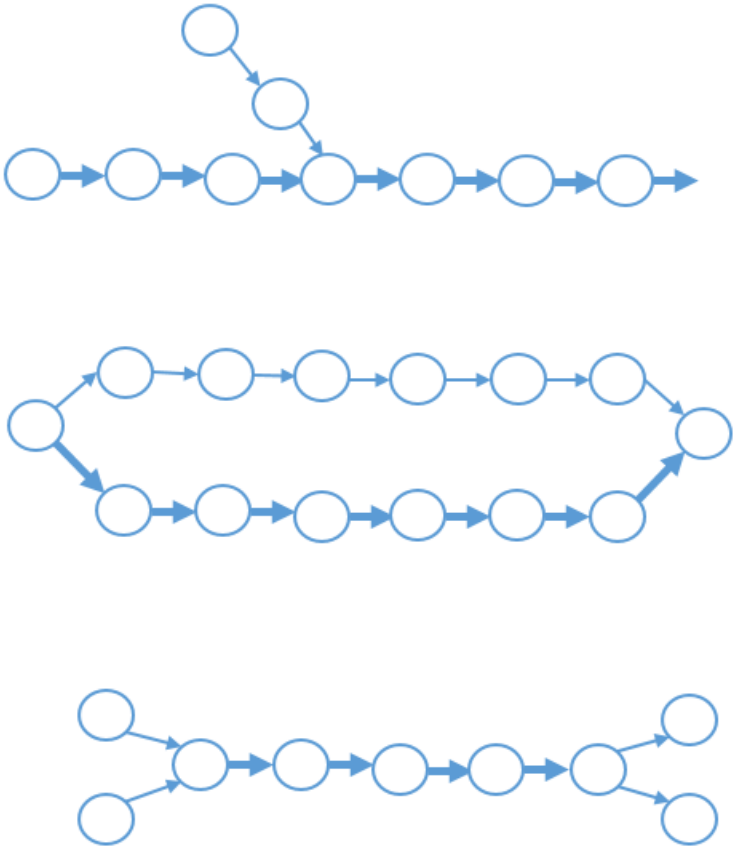

Figure 18: Structures need to be addressed.

First one is tip structure, second one is bubble structure and last one is scissor bolt structure. Thicker arrows in the graph means paths have high quality/coverage.

Figure 18 illustrates some of the structures in the graph that need to be addressed. The process can improve the assembly quality and remove erroneous structures introduced by sequencing errors.

Tip removal

During the process of condensation, all "dead-end" contigs are identified and marked, among which those shorter than *2k* bases pair (bp) and having low frequencies are removed. *2k* bp are set as threshold because an error happens at the two ends of a read causes a tip structure of length less than *k* (otherwise it is a bubble instead of a tip). We tolerate two accumulated errors thus the length of a tip should less than *2k* bp. A standout sequence longer than *2k* bp likely represents either a genuine sequence or an accumulation of errors that is hard to distinguish from novel sequence. Figure 19 shows the graph after tip removal.

Since tips are stored in the same location as ambiguity nodes and ambiguity nodes hold all the adjacency information, tip removal are performed locally and simultaneously across the processing cores.

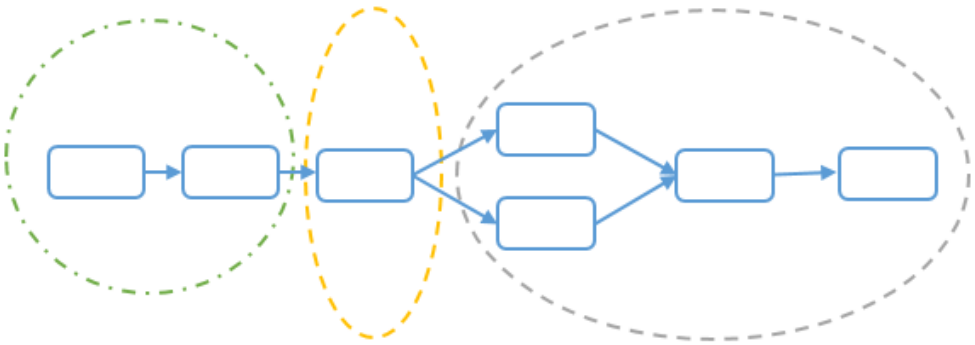

Figure 19: Example of tip removal

Following Figure 16, the tip path has shorter length and lower coverage so it is removed.

Tip removal in shared memory is easy since each process can access the global picture. Other parallel assemblers that involve communication in tip removeal thus make the coding difficult and take substantially more time.

Bubble removal

Ambiguity nodes that have more than two indegree may possibly be the convergent point of multiple divergent paths. As mentioned in condensation phase, the contigs linked to this ambiguity node are stored on the same processing cores, along with their prior ambiguities node's information.

If these two contigs connect exactly to the same two ambiguity nodes and if they have same length and differ at the first nucleotide, it's possibly a SNP or a sequencing error in the middle of a read; if they have different length and last few nucleotides are the same, it is possibly an INDEL. Either way, we merge the bubbles into single path by deleting lower frequency paths. This bubble removal also can be performed locally and simultaneously.

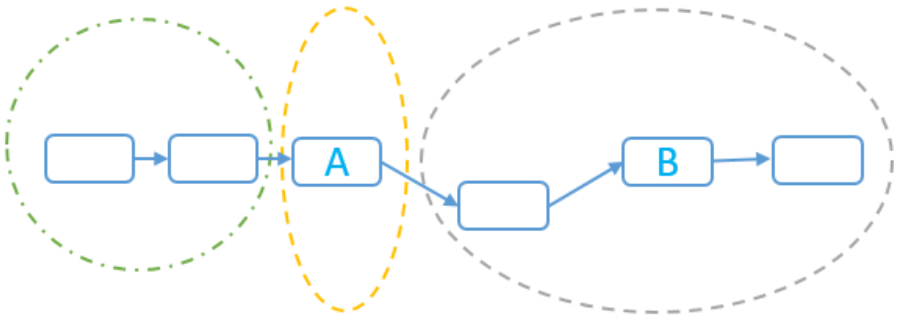

Figure 20: Example of bubble removal

Following Figure 16, the bubble path has lower coverage so it is removed.

Bubble removal in shared memory is as natural as tip removal because of the global access. Other parallel assemblers that involve communication to find and deal with the bubble thus make the coding unnecessarily difficult and the running time unnecessarily longer. After bubbles are removed, PPASSEM needs to notify the prior ambiguity node of the removal of path, like in Figure 20, node A may locate at a different processing core to node B, when upper path is removed it needs the update from node B.

Scissor bolt resolve

Scissor bolt is a vertex in the graph that makes a scissor-like structure, and two paths share the same segment of sequences would cause this structure. The scissor bolt is recognized when a vertex in a graph has multiple indegrees and multiple outdegrees. Out of strong cautions, we only address those having the same indegrees and outdegrees.

In PPASSEM, the vertex in the scissor bolt is removed by splitting the connections into parallel paths. By the time of next round of condensation, this structure will be merged into longer contigs.

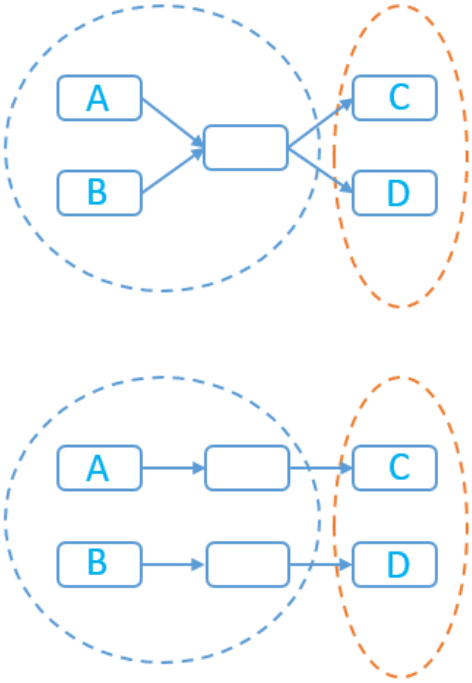

Figure 21: Scissor bolt resolve

The upper one is the structure in the graph, the lower one is after the scissor bolt resolution. Those contigs share similar coverage are split out in to linear paths and then further condensed.

In actual implementation, because of the way of how condensation is performed, the ambiguity node stores all adjacency information locally and even adjacent contigs may not locate at the same processing cores. For the example in Figure 21, there are two possible ways of splitting the connections into parallel paths: (AC, BD) and (AD, BC). The safest way is to map the reads back and resolve this structure, which will take enormous amount of time and memory. Instead, we propose to merge paths that have similar quality/coverage. Here we choose (AC, BD) over (AD, BC) since A, C have similar quality than A, D.

By removing these tips, bubbles and scissor bolts, contigs can be unambiguously extended further. Hence we need to condense the graph iteratively for a few cycles, because after

each new condensation, these special structures may show up again, like longer tips and bigger bubbles. Condensation and error correction will be conducted until no correction can be made.

3.6 Scaffolding discussion

All the above phases have not made use of pair-end information yet, which may be helpful in finding the correct ordering of the assembled contigs and joining contigs to longer sequences. This process needs to map the pair-end reads onto the contigs. Since the graph is significantly simplified and do not require much RAM for storage, it would not be a good idea to work this phase on parallel cluster. Other assemblers, like ABySS and PASHA also execute this phase on a single computing node. Here we will simply use a stand-alone scaffolding software like SSPACE (Boetzer et al., 2011).

After these steps, the contigs can then be outputted into FASTA files.

3.7 Results

Experiment data

To evaluate PPASSEM, we use three short-read datasets generated by Illunima sequencer. They represent three different genome sizes: a small *E.coli* dataset (accession number SRR001665 in the NCBI Sequence Read Archive (SRA)), a medium human chromosome 14 dataset (from GAGE project (Salzberg et al. 2012)) and a large Yoruban male dataset (accession number SRA000271 in NCBI SRA). Table 4 shows the summary of the experiment data.

Table 4: Experiment data summary for assembly

	E.coli	Human chromosome 14	Yoruban male
Genome size(Mb)	4.6	88.3	3101.8
# of reads(million)	21	62	3759
Read length(bp)	36	101	36~42
coverage	162×	69×	44×

Assembly quality assessment

For quality assessment, we compare PPASSEM with another two parallel assembler that are also based on de Bruijn graph: ABySS (version 1.5.2) and PASHA (version 1.0.10). The following criteria will be used:

- number of contigs: contigs generated by clipping at ambiguity point, we only consider contigs of length >100 bps
- N50 size: the contig length such that using equal or longer contigs produces half the bases of the genome, a widely used measure
- maximum contig length: longest contig size
- coverage: calculated from aligning contigs to their reference genomes using BLAT (Kent, 2002) Version 35
- speed: time used to finish the assembly

Tests are executed on a workstation with two six-core 2.3GHz CPUs and 64 Gb RAM (Bubble merging and contig generation stage of PASHA only runs on single node), and a computing cluster with 8 nodes connected by InfiniBand switch. Each cluster node contains two eight-core 2.6 GHz CPUs and 128GB RAM.

K-mer size *k* can neither be too small which can hardly find linear paths, nor too large which make the de Bruijn graph very fragmented. Chikhi and Medvedev (Chikhi and Medvedev, 2013) has a nice discussion on the k-mer size selection. Generally k-mer size is recommended to be an odd number between 23 and 31 for PPASSEM. K-mers in the bin increase as dataset gets larger, a smaller header length may cause a message be too big to send. It is recommended larger dataset comes with a larger header length.

We first use the small dataset *E.coli*, with k-mer length *k* set to be 27 for all three assembler and header length *l* to be 3 for PPASSEM. These programs were run on clusters using 8 cores. Results are shown in Table 5.

Table 5: Assembly results for *E.coli*

	PPASSEM	**ABySS**	**PASHA**
# of contigs	423	184	344
N50	25203	62449	32215
Max	132865	178752	184046
Genome coverage	94.60%	95.98%	96.88%
Time(minutes)	15	41	37

ABySS tends to give better N50 size of 62449 while as PPASSEM tends to generate larger number of contigs. Genome coverages are comparable for all three assemblers. However, PPASSEM shows significantly less execution time than both ABySS and PASHA.

We continue to use the medium size data human chromosome 14, with k-mer length *k* set to be 29 for all three assembler and header length *l* to be 4 for PPASSEM. These programs were run on clusters using 64 cores. Results are shown in Table 6.

Table 6: Assembly results for human chromosome 14

	PPASSEM	**ABySS**	**PASHA**
# of contigs	87426	106543	88793
N50	1925	1876	1648
Max	22759	25614	21235
Genome coverage	82.37%	84.23%	85.86%
Time(minutes)	38	96	105

Here, PPASSEM shows better N50 size of 1925, and ABySS generates more number of contigs. Genome coverages are again comparable for all three assemblers. PPASSEM once again shows significantly less execution time than both ABySS and PASHA.

Detailed runtime summary for human chromosome 14 data is shown in Figure 22. We see that condensation and error correction take majority of the execution time (61%), k-mer generation and distribution require least of the time (14%). Execution time of PPASSEM on different number of CPU cores is shown in Figure 23. Both the total and each stage execution time needed decrease as the number of cores increases.

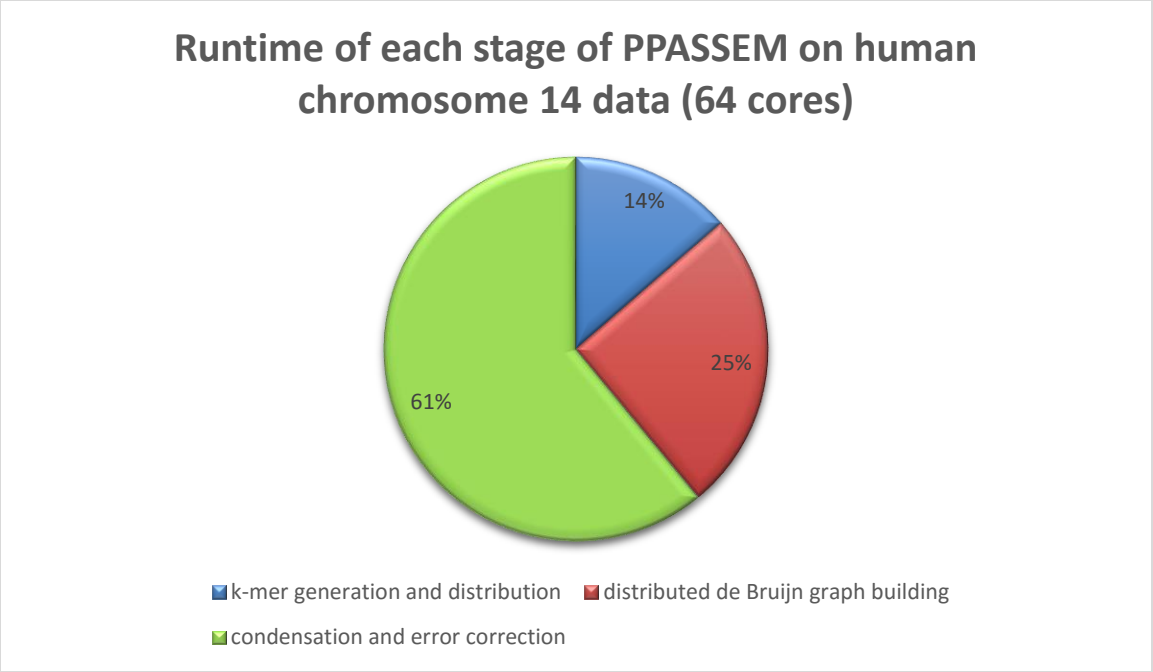

Figure 22: Runtime segmentation of each stage of PPASSEM on human chromosome 14 data using 64 cores

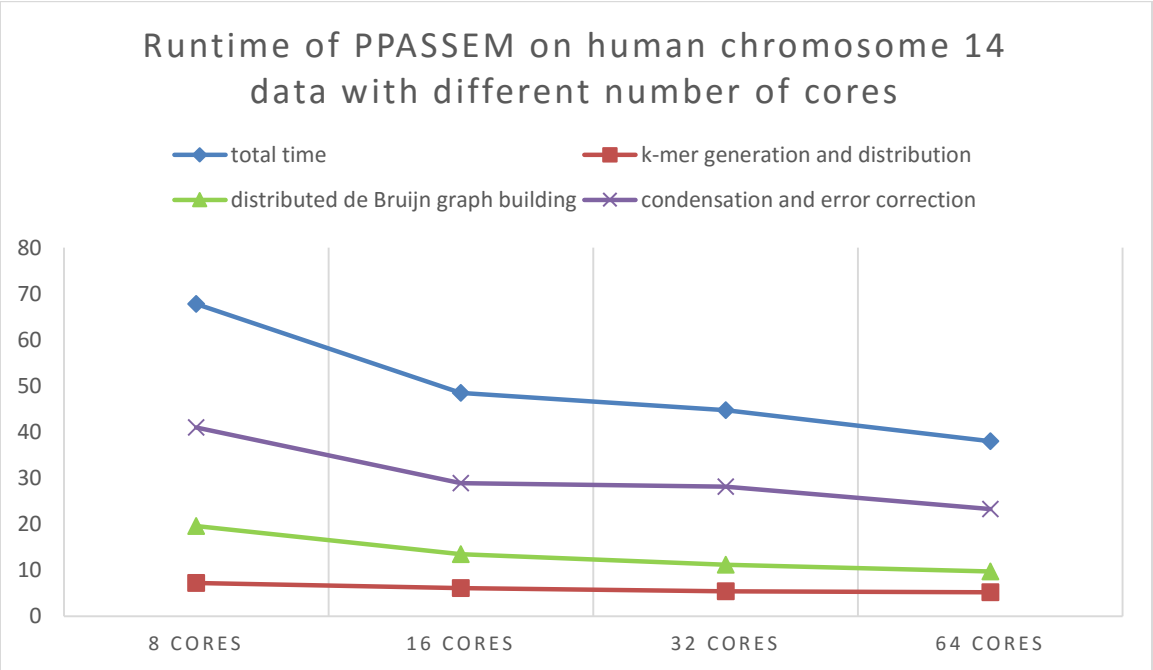

Figure 23: Runtime (in minute) of PPASSEM on human chromosome 14 data with different number of cores

Finally, to demonstrate the ability to handle large genomes, we test on the Yoruban male dataset. K-mer length *k* is set to be 31 for ABySS and PPASSEM and header length *l* to be 5 for PPASSEM. These programs were run on clusters using 128 cores. PASHA has a runtime error that could not generate any result so it is excluded from this comparison. Results on Yoruban male are shown in Table 7.

Table 7: Assembly results for Yoruban male

	PPASSEM	**ABySS**
# of contigs	37804682	34716926
N50	581	617
Max	17981	19074
Time(hours)	8	32

PPASSEM and ABySS shows comparable N50 size of 581 and 617, respectively. Number of contigs and largest contig size are also in line with each other. However, PPASSEM use only 8 hours to complete the assembly of a whole human genome while ABySS takes 32 hours.

In summary, PPASSEM is able to process billions of reads in parallel on commodity computer clusters, producing results comparable to well-known parallel assembler ABySS and PASHA but use significantly less time.

Chapter 4

Conclusion and Future Work

4.1 Conclusion

With the rapid developments in both sequencing technology and massive parallel processing, it is natural to combine them together as a novel tool for genomic researches. In this work, we first develop PPLAT - an integrated parallel platform framework for distributed storage and distributed processing of big data, allowing asynchronous computing and message passing. This platform is intended to reduce the coding and debugging complexity by providing flexible interfaces and simplified APIs. It will greatly facilitate development of new high-performance software by enabling shared-memory multithreading and distributed-memory message passing. Second, we apply PPLAT to massive short-read sequences to construct PPASSEM, a parallel assembler that is based on de Bruijn graphs. PPASSEM contributes several novel algorithms for the large genome assembly, like unique k-mers grouping to address communication issues and condensation to reduce redundancy and enable local error correction in the graph. Our real data examples show that PPASSEM is capable of processing billions of reads on commodity clusters. Tests on datasets from small to large sizes all show its ability to produce comparable results but use much less time to the well-known parallel assembler ABySS and PASHA.

4.2 Future work on parallel platform

There are several functionalities and designs that could be added to PPLAT, such as a robust file system or a smarter scheduler. Because of C++ comprehensive standard library and powerful compatibility with other library or software, data stored on processing cores can be in multiple format and structure. However, it is still up to users to specify and include the data structure which may be an issue for non-proficient C++ users when working on linear algebra problems. Future direction may include providing PPLAT supported data structure by wrapping some widely used template libraries. For example, vectors, matrix and numeric solvers are commonly used in numerical analysis. We may provide these data structures and operations as built-in types and functions, or include wrap template libraries like Eigen. Once this is done, PPLAT will become even more powerful and easy to use, resulting in minimal efforts from end users.

For now, PPLAT only supports C++ string type in the messages with encoding and decoding methods provided by users. This is a bottleneck for easy coding if multiple types of data needs to be exchanged. Ideal message packaging would only need to specify which data and where it should be sent, but leave the details of packing, delivering and unpacking the data to platform. Future direction includes building a language-neutral, extensible mechanism for data transfer. For example, you have a matrix M on process A and would like to send to process B. Instead of encoding it row by row to a string then decoding it at destination, simply tell the platform to do this just by a function sendMessage (M, A, B). Google protocol buffer can be a good candidate to be included in the future PPLAT.

4.3 Future work on assembly

Repeats resolution

Output of the assembler is usually not the final results of a research project. For example, extensive post-assembly analyses may be required to produce chromosome sequences and deliver variants analysis. Real genomes present complex repeat structures including tandem repeats, inverted repeats, imperfect repeats, and repeats inserted within repeats. The graph itself is insufficient to distinguish the repeats. Assemblers typically turn to the reads, and pair-end information, in order to resolve these regions.

Recently, sequencing technologies are able to produce reads with longer insert sizes, and using multiple short-read libraries with different insert sizes is more effective than a single insert size library for the generation of a *de novo* assembly (van Heesch et al., 2013).

Long-insert pair-end libraries are very useful at determining whether two contigs are linked while short-insert libraries can help identify the exact sequence between two contigs. Using multiple insert length libraries is capable of producing even longer scaffolds and better blueprint of genome structures. Future work includes developing a scaffolding module for PPASSEM, preferably in parallel for single insert library; and when multiple insert libraries is available, algorithms to use these information for better assembly results.

Third generation sequencing

More recently, the third generation sequencing technology has also been discussed. The third generation sequencing allows real-time sequencing of single DNA (or RNA) molecules, which aims to increase throughput even more and correspondingly decrease the sequencing time

and cost, through eliminating the need for excessive reagents and harnessing the processivity of DNA polymerase (Schadt et al., 2010). Although still in development, Single Molecule, Real-Time (SMRT) method by Pacific Biosciences and Nanopore method by Oxford Nanopore Technologies have been commercialized to produce much longer reads (3000 to 10000 bp) but with higher error rate.

These new features of third generation sequencing data calls for novel error correction methods, special mapping and assembly algorithm and comprehensive pipeline for analysis. Future work includes extending PPASSEM to adapt to the new sequencing platform.

Methylation data processing

DNA methylation is a process in which methyl groups are added to DNA. To determine the methylation state of each cytosine position in the read, mapping method is generally used for these Bisulfite sequencing data (Krueger and Andrews, 2011). However, mapping method are fundamentally flawed, involving complicated procedure and producing poor results. Possible improvement for methyl-seq data analysis would be to apply assembly first and then map longer contig in genome to improve the quality. Future work includes the application of PPASSEM to whole genome methyl-seq data.

Reference

Almasi, G.S. and Gottlieb, A., 1988. Highly parallel computing.

An introduction to Next-generation Sequencing Technology. http://www.illumina.com/technology/next-generation-sequencing.html

Apache Hadoop. http://hadoop.apache.org/

Barney, B., 2010. Introduction to parallel computing. *Lawrence Livermore National Laboratory*, *6*(13), p.10.

Batzoglou, S., Jaffe, D.B., Stanley, K., Butler, J., Gnerre, S., Mauceli, E., Berger, B., Mesirov, J.P. and Lander, E.S., 2002. ARACHNE: a whole-genome shotgun assembler. *Genome research*, *12*(1), pp.177-189.

Boetzer, M., Henkel, C.V., Jansen, H.J., Butler, D. and Pirovano, W., 2011. Scaffolding pre-assembled contigs using SSPACE. *Bioinformatics*, *27*(4), pp.578-579.

Boost.Python. http://www.boost.org/doc/libs/1_60_0/libs/python/doc/html/index.html

Butler, J., MacCallum, I., Kleber, M., Shlyakhter, I.A., Belmonte, M.K., Lander, E.S., Nusbaum, C. and Jaffe, D.B., 2008. ALLPATHS: de novo assembly of whole-genome shotgun microreads. *Genome research*, *18*(5), pp.810-820.

Carninci, P., Kasukawa, T., Katayama, S., Gough, J., Frith, M.C., Maeda, N., Oyama, R., Ravasi, T., Lenhard, B., Wells, C. and Kodzius, R., 2005. The transcriptional landscape of the mammalian genome. *Science*, *309*(5740), pp.1559-1563.

Chaisson, M.J. and Pevzner, P.A., 2008. Short read fragment assembly of bacterial genomes. *Genome research*, *18*(2), pp.324-330.

Chikhi, R. and Medvedev, P., 2013. Informed and automated k-mer size selection for genome assembly. *Bioinformatics*, p.btt310.

Cho, I. and Blaser, M.J., 2012. The human microbiome: at the interface of health and disease. *Nature Reviews Genetics*, *13*(4), pp.260-270.

Cock, P.J., Fields, C.J., Goto, N., Heuer, M.L. and Rice, P.M., 2010. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic acids research*, *38*(6), pp.1767-1771.

Dayarian, A., Michael, T.P. and Sengupta, A.M., 2010. SOPRA: Scaffolding algorithm for paired reads via statistical optimization. *BMC bioinformatics*,*11*(1), p.1.

Eddelbuettel, D., François, R., Allaire, J., Chambers, J., Bates, D. and Ushey, K., 2011. Rcpp: Seamless R and C++ integration. *Journal of Statistical Software*, *40*(8), pp.1-18.

Eigen. http://eigen.tuxfamily.org/

Fasulo, D., Halpern, A., Dew, I. and Mobarry, C., 2002. Efficiently detecting polymorphisms during the fragment assembly process. *Bioinformatics*, *18*(suppl 1), pp.S294-S302.

Gnerre, S., MacCallum, I., Przybylski, D., Ribeiro, F.J., Burton, J.N., Walker, B.J., Sharpe, T., Hall, G., Shea, T.P., Sykes, S. and Berlin, A.M., 2011. High-quality draft assemblies of mammalian genomes from massively parallel sequence data. *Proceedings of the National Academy of Sciences*, *108*(4), pp.1513-1518.

Google sparse hash. https://github.com/sparsehash/sparsehash

Huang, X., Wang, J., Aluru, S., Yang, S.P. and Hillier, L., 2003. PCAP: a whole-genome assembly program. *Genome research*, *13*(9), pp.2164-2170.

Idury, R.M. and Waterman, M.S., 1995. A new algorithm for DNA sequence assembly. *Journal of computational biology*, *2*(2), pp.291-306.

Kececioglu, J. and Ju, J., 2001, April. Separating repeats in DNA sequence assembly. In *Proceedings of the fifth annual international conference on Computational biology* (pp. 176-183). ACM.

Kent, W.J., 2002. BLAT—the BLAST-like alignment tool. *Genome research*, *12*(4), pp.656-664.

Korf, R.E., 1998. A complete anytime algorithm for number partitioning.*Artificial Intelligence*, *106*(2), pp.181-203.

Lander, E.S., Linton, L.M., Birren, B., Nusbaum, C., Zody, M.C., Baldwin, J., Devon, K., Dewar, K., Doyle, M., FitzHugh, W. and Funke, R., 2001. Initial sequencing and analysis of the human genome. *Nature*, *409*(6822), pp.860-921.

Li, R., Zhu, H., Ruan, J., Qian, W., Fang, X., Shi, Z., Li, Y., Li, S., Shan, G., Kristiansen, K. and Li, S., 2010. De novo assembly of human genomes with massively parallel short read sequencing. *Genome research*, *20*(2), pp.265-272.

Li, Z., Chen, Y., Mu, D., Yuan, J., Shi, Y., Zhang, H., Gan, J., Li, N., Hu, X., Liu, B. and Yang, B., 2011. Comparison of the two major classes of assembly algorithms: overlap–layout–consensus and de-bruijn-graph. *Briefings in functional genomics*, p.elr035.

Liu, Y., Schmidt, B. and Maskell, D.L., 2011. Parallelized short read assembly of large genomes using de Bruijn graphs. *BMC bioinformatics*, *12*(1), p.1.

Marçais, G. and Kingsford, C., 2011. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, *27*(6), pp.764-770.

Mardis, E.R., 2008. The impact of next-generation sequencing technology on genetics. *Trends in genetics*, *24*(3), pp.133-141.

Medvedev, P., Georgiou, K., Myers, G. and Brudno, M., 2007. Computability of models for sequence assembly. In *Algorithms in Bioinformatics* (pp. 289-301). Springer Berlin Heidelberg.

Message Passing Interface Forum. http://www.mpi-forum.org/

Miller, J.R., Koren, S. and Sutton, G., 2010. Assembly algorithms for next-generation sequencing data. *Genomics*, *95*(6), pp.315-327.

Mooney, S.D., 2015. Progress towards the integration of pharmacogenomics in practice. *Human genetics*, *134*(5), pp.459-465.

Mullikin, J.C. and Ning, Z., 2003. The phusion assembler. *Genome research*, *13*(1), pp.81-90.

Myers, E.W., 1995. Toward simplifying and accurately formulating fragment assembly. *Journal of Computational Biology*, *2*(2), pp.275-290.

Myers, E.W., Sutton, G.G., Delcher, A.L., Dew, I.M., Fasulo, D.P., Flanigan, M.J., Kravitz, S.A., Mobarry, C.M., Reinert, K.H., Remington, K.A. and Anson, E.L., 2000. A whole-genome assembly of Drosophila. *Science*, *287*(5461), pp.2196-2204.

Nagarajan, N. and Pop, M., 2009. Parametric complexity of sequence assembly: theory and applications to next generation sequencing. *Journal of computational biology*, *16*(7), pp.897-908.

Pearson, W.R. and Lipman, D.J., 1988. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, *85*(8), pp.2444-2448.

Pevzner, P.A., Tang, H. and Waterman, M.S., 2001. An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences*, *98*(17), pp.9748-9753.

Protocol Buffers. https://developers.google.com/protocol-buffers/

Rasmussen, K.R., Stoye, J. and Myers, E.W., 2005, May. Efficient q-gram filters for finding all ε-matches over a given length. In *Research in Computational Molecular Biology* (pp. 189-203). Springer Berlin Heidelberg.

Rcpp for seamless R and C++ Integration. http://www.rcpp.org/

Salzberg, S.L., Phillippy, A.M., Zimin, A., Puiu, D., Magoc, T., Koren, S., Treangen, T.J., Schatz, M.C., Delcher, A.L., Roberts, M. and Marçais, G., 2012. GAGE: A critical evaluation of genome assemblies and assembly algorithms. *Genome research*, *22*(3), pp.557-567.

Sanger, F., G. M. Air, B. G. Barrell, N. L. Brown, A. R. Coulson, J. C. Fiddes, C. A. Hutchison, P. M. Slocombe, and M. Smith. "Nucleotide sequence of bacteriophage phiX174 DNA." *Nature* 265 (1977): 687-695.

Schadt, E.E., Turner, S. and Kasarskis, A., 2010. A window into third-generation sequencing. *Human molecular genetics*, p.ddq416.

Schatz, M.C., Delcher, A.L. and Salzberg, S.L., 2010. Assembly of large genomes using second-generation sequencing. *Genome research*, *20*(9), pp.1165-1173.

Simpson, J.T., Wong, K., Jackman, S.D., Schein, J.E., Jones, S.J. and Birol, I., 2009. ABySS: a parallel assembler for short read sequence data.*Genome research*, *19*(6), pp.1117-1123.

van Heesch, S., Kloosterman, W.P., Lansu, N., Ruzius, F.P., Levandowsky, E., Lee, C.C., Zhou, S., Goldstein, S., Schwartz, D.C., Harkins, T.T. and Guryev, V., 2013. Improving mammalian genome scaffolding using large insert mate-pair next-generation sequencing. *BMC genomics*, *14*(1), p.1.

Velculescu, V.E., Zhang, L., Vogelstein, B. and Kinzler, K.W., 1995. Serial analysis of gene expression. *Science*, *270*(5235), pp.484-487.

Venter, J.C., Adams, M.D., Myers, E.W., Li, P.W., Mural, R.J., Sutton, G.G., Smith, H.O., Yandell, M., Evans, C.A., Holt, R.A. and Gocayne, J.D., 2001. The sequence of the human genome. *Science*, *291*(5507), pp.1304-1351.

Wang, L. and Jiang, T., 1994. On the complexity of multiple sequence alignment. *Journal of computational biology*, *1*(4), pp.337-348.

Weber, J.L. and Myers, E.W., 1997. Human whole-genome shotgun sequencing. *Genome Research*, *7*(5), pp.401-409.

Whiteford, N., Haslam, N., Weber, G., Prügel-Bennett, A., Essex, J.W., Roach, P.L., Bradley, M. and Neylon, C., 2005. An analysis of the feasibility of short read sequencing. *Nucleic acids research*, *33*(19), pp.e171-e171.

Wu, S., Wang, J., Zhao, W., Pounds, S. and Cheng, C., 2010. ChIP-PaM: a n algorithm to identify protein-DNA.

Zerbino, D.R., 2009. Genome assembly and comparison using de Bruijn graphs (Doctoral dissertation, University of Cambridge).

Zerbino, D.R. and Birney, E., 2008. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome research*, *18*(5), pp.821-829.

Zerbino, D.R., McEwen, G.K., Margulies, E.H. and Birney, E., 2009. Pebble and rock band: heuristic resolution of repeats and scaffolding in the velvet short-read de novo assembler. *PloS one*, *4*(12), p.e8407.