2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops

HiPGA: A High Performance Genome Assembler for Short Read Sequence Data

Xiaohui Duan, Kun Zhao, Weiguo Liu*

School of Computer Science and Technology
Engineering Research Center of Digital Media Technology, Ministry of Education
Shandong University, Jinan City, China 250101
Email: {sunrise.duan, kun.zhao}@mail.sdu.edu.cn, weiguo.liu@sdu.edu.cn

***Abstract*—Emerging next-generation sequencing technologies have opened up exciting new opportunities for genome sequencing by generating read data with a massive throughput. However, the generated reads are significantly shorter compared to the traditional Sanger shotgun sequencing method. This poses challenges for de novo assembly algorithms in terms of both accuracy and efficiency. And due to the continuing explosive growth of short read databases, there is a high demand to accelerate the often repeated long-runtime assembly task. In this paper, we present a scalable parallel algorithm – HiPGA to accelerate the de Bruijn graph-based genome assembly for high-throughput short read data. In order to make full use of the compute power of both shared-memory multi-core CPUs and distributed-memory systems, we have used a parallelized file I/O scheme as well as a hybrid parallelism for the whole assembly pipeline. Evaluations using three real paired-end datasets and the Yoruba individual dataset show that compared to two other well parallelized assemblers: ABySS and PASHA, HiPGA achieves speedups up to 7 while delivering comparable accuracy on 64 CPU cores of a compute cluster.**

***Keywords*-Genome Assembly; de Bruijn Graph; Short Read Data; MPI; Multi-threading.**

I. Introduction

In the past few years, a number of new-generation DNA sequencing technologies have been introduced. Compared to the traditional Sanger shotgun technique, these new technologies are able to generate a huge amount of read data at lower cost [14], [16]. Examples of already available such technologies are sequencers from 454 Life Sciences/Roche, Solexa/Illumina, and Applied Biosiciences/SOLiD. However, the length of generated reads is significantly shorter compared to the Sanger method. For example, the Illumina Genome Analyzer can generate up to 200 million (unpaired or paired) reads in a single run with a read length of 50–150.

There have been many well optimized methods and tools to do assembly for Sanger shotgun sequencing (i.e., read lengths of around 500bps and 6-to-10-fold coverage). However these methods generally do not scale well for high-coverage short read data. Thus, there is a high demand for scalable assembly tools that can deal with high throughput short read data. Consequently, several such tools have been introduced recently. SSAKE [19], SHARCGS [6], VCAKE [9], PE-Assembler [1], PASQUAL [12] and Taipan [17] are based on the k-mer extension approach. However, this approach is inaccurate for assembling repeat regions. In order to resolve repeats, the de Bruijn graph-based approach to assembly has been introduced in [15], which was then implemented in the Euler assembly package. Euler-SR [4], [3] is a further extension of Euler to assemble short read data. Other published de Bruijn graph-based short read *de novo* assemblers include ALLPATHS [2], Velvet [20], ABySS [18], SOAPdenovo [11], YAGA [8], and PASHA [13].

In this paper, we present HiPGA – a high performance de Bruijn graph-based genome assembler for high-throughput short read data. In order to overcome the large-scale file I/O bottleneck and make full use of the compute power of both shared-memory multi-core CPUs and distributed-memory systems, we have used a parallelized file I/O scheme as well as a hybrid parallelism for all stages of the assembly pipeline. Experiments show that our implementation achieves much better performance compared to two other well parallelized assemblers: ABySS and PASHA. And HiPGA also shows better scalability as the number of CPU cores increases.

The rest of this paper is organized as follows. In Section II, we introduce the de Bruijn graph-based genome assembly method and give a brief summary of the previous work on parallelization of short read assembly on different architectures. Section III presents our method to accelerate the whole assembly pipeline. Performance is evaluated in Section IV. Finally, Section V concludes the paper.

II. Related Work

In this section, we first briefly describe the de Bruijn graph-based genome assembly method and the four stages involved in its pipeline. Then, we introduce the previous work on accelerating short read assembly.

* Weiguo Liu is the corresponding author.

978-1-4799-4116-2/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPSW.2014.68

Table I
A SEQUENTIAL EXECUTION PROFILING (IN SECONDS) OF THE FOUR STAGES OF PASHA USING DIFFERENT SHORT READ DATASETS ON AN INTEL XEON E5650 2.67GHZ (BASED ON THE CODE FROM [13]).

Genome Datasets	K-mer Generation	Graph Construction	Contig Generation	Scaffolding	Overall
Bacillus	109.27 (19.85%)	138.53 (25.16%)	136.94 (24.88%)	165.75 (30.11%)	550.49
Bordetella	83.71 (19.12%)	112.62 (25.73%)	111.24 (25.41%)	130.14 (29.73%)	437.71
E.coli	155.59 (21.41%)	181.24 (24.94%)	178.78 (24.60%)	211.20 (29.06%)	726.81

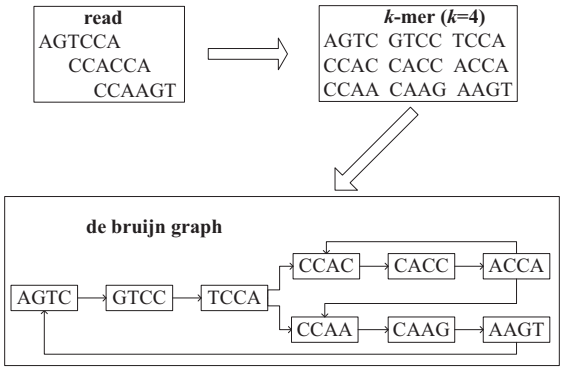

Figure 1. Illustrations of constructing the preliminary de Bruijn graph from k-mer set.

A. Genome Assembly using de Bruijn Graphs

The de Bruijn graph-based assembly method, introduced in [15], is very suitable for representing the short read overlap relationship. It uses k-mer as vertex, and read path along the k-mers as edges on the graph. The graph size is thus determined by the genome size and repeat content of the sequenced sample rather than the high redundancy of deep read coverage [11]. Hence, this method greatly reduces the memory consumption and increases the assembly efficiency in practice.

Once the k-mer set is ready, the de Bruijn graph-based assembler will start constructing a preliminary de Bruijn graph. That is, it will create an edge between two vertices (k-mers) if and only if they have a suffix-prefix overlap of $k-1$ bases. Fig. 1 gives an illustration of how the preliminary de Bruijn graph is built from a set of k-mers. Then, based on this preliminary de Bruijn graph, a set of operations will be done to generate the final scaffolds. Generally, the pipeline of the de Bruijn graph-based assembly method consists of four stages which are shown in Fig. 2. We briefly describe each stage in the following. More details can be found in [5] and [11].

Stage 1: This stage generates k-mers from short reads. Each k-mer is defined as a contiguous sequence of k bases.

Stage 2: Stage 2 first constructs the preliminary de Bruijn graph. Then a set of operations will be done to simplify the generated graph. These operations include: (a) clipping the short *tips*, (b) removing *low-coverage paths*, and (c) merging *bubbles*. At the end of Stage 2, a set of linear chains are generated.

Stage 3: Based on the previously generated linear chains, Stage 3 outputs the unambiguous sequence fragments as contigs.

Stage 4: Stage 4 realigns reads onto contigs and uses the paired-end information to merge contigs into scaffolds.

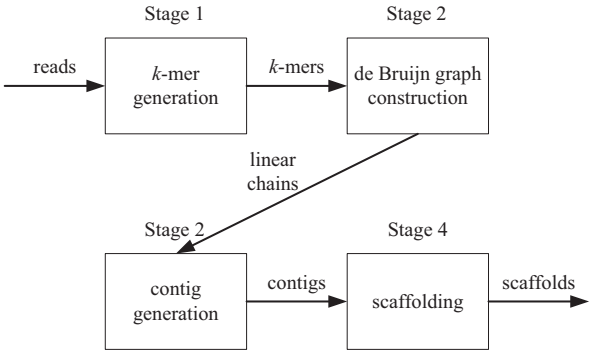

Figure 2. The typical assembly pipeline.

PASHA is one of the well parallelized de Bruijn graph-based short read assembler. It has been reported that compared with other assemblers such as Velvet and SOAPdenovo, PASHA can deliver better accuracy while achieve the fastest execution speed [13]. PASHA also follows the 4-stage assembly pipeline shown in Fig. 2. A sequential execution profiling of PASHA for different short read datasets (These datasets include *Bacillus*, *Bordetella*, and *E.coli* with accession numbers DRR000002, ERR007648, and SRR001665 in the NCBI Sequence Read Archive (SRA).) shows the breakdown of runtime in Table I. From Table I we can see that in order to achieve good speedups, all four stages need to be efficiently parallelized. We have also profiled the file I/O and compute time of PASHA for assembly these datasets (see Table II). From Table II we can see that the file I/O dominates the runtime of each stage. So in order to achieve better assembly efficiency, we should try to reduce the file I/O time.

Table II
A SEQUENTIAL EXECUTION PROFILING (IN SECONDS) OF THE FILE I/O AND COMPUTE TIME OF PASHA USING DIFFERENT SHORT READ DATASETS ON AN INTEL XEON E5650 2.67GHZ (BASED ON THE CODE FROM [13]).

Genome Datasets		K-mer Generation	Graph Construction	Contig Generation	Scaffolding	Overall
Bacillus	I/O	108.96 (99.72%)	103.70 (74.86%)	88.87 (64.90%)	89.51 (54.00%)	391.04 (71.03%)
	Computation	54.70 (50.06%)	79.02 (57.04%)	48.07 (35.10%)	76.24 (46.00%)	258.03 (46.87%)
	Overall	109.27 (19.85%)	138.53 (25.16%)	136.94 (24.88%)	165.75 (30.08%)	550.49
Bordetella	I/O	83.38 (99.61%)	81.28 (72.17%)	69.59 (62.56%)	71.12 (54.65%)	305.37 (69.77%)
	Computation	37.23 (44.48%)	74.16 (65.85%)	41.65 (37.44%)	59.02 (45.35%)	212.06 (48.45%)
	Overall	83.71 (19.12%)	112.62 (25.73%)	111.24 (25.41%)	130.14 (29.73%)	437.71
E.coli	I/O	155.28 (99.80%)	148.67 (82.03%)	125.24 (70.05%)	128.47 (60.83%)	557.66 (76.73%)
	Computation	77.21 (49.62%)	86.90 (47.95%)	53.54 (29.95%)	82.73 (39.17%)	300.38 (41.33%)
	Overall	155.59 (21.41%)	181.24 (24.94%)	178.78 (24.60%)	211.20 (29.06%)	726.81

B. Previous Work on Accelerating Short Read Assembly

Because of the prohibitive memory consumption and long execution time for assembling large genomes, there have been a lot of approaches to parallelize the de Bruijn graph-based assembler on parallel architectures. These approaches can be classified into two categories: 1) multi-threading, and 2) MPI. Multi-threaded approaches run on multi-core CPUs such as standard dual/quad-core CPUs. SOAPdenovo [11] is a multi-threaded implementation. It parallelizes the compute intensive portions of the assembly pipeline on shared-memory architectures. MPI implementations are designed for distributed memory systems which include PC clusters and supercomputers such as Blue Gene. Examples include ABySS [18], PASHA [13] and YAGA [8]. And we have also presented our preliminary work on parallelizing the de Bruijn graph-based assembly algorithm using multi-threading and MPI [21]. However our presented method at that time could only do assembly for small genomes and the supported read length is very limited (less than 40).

ABySS [18] and PASHA [13] are close to the approach presented in this paper since they also use both the small-scale shared-memory multi-threading parallelism and the large-scale distributed-memory parallelism to improve the assembly efficiency. They parallelize Stages 1 and 2 of the assembly pipeline using both multi-threading and MPI. However, each MPI process in them needs to repeatedly read the short read file sequentially, which makes the large-scale short read file I/O a bottleneck. Moreover, because the compute-intensive Stages 3 and 4 in PASHA and Stage 4 in ABySS are parallelized only using the multi-threaded parallelism, they can not make full use of the compute power of the distributed-memory architectures. Our solution overcomes these bottlenecks by using a parallelized file I/O scheme as well as a hybrid parallelism for the whole assembly pipeline. Experiments show that the de Bruijn graph-based assembly can be efficiently parallelized on both shared-memory and distributed-memory architectures using our approach.

III. DESIGN AND IMPLEMENTATION

Based on the characteristics of the assembly pipeline, we have designed parallel algorithms for all stages using a hybrid parallel scheme – the small-scale shared-memory multi-threading scheme and the large-scale distributed-memory scheme, to overcome the file I/O bottleneck and gain high assembly efficiency. And this hybrid parallel scheme makes our program suitable for both multi-core CPUs and distributed-memory systems. Fig. 3 shows the framework of our implementation – HiPGA. In HiPGA, the short read file is first divided evenly into a set of smaller sized subfiles. All distributed processes then handle Stages 1 to 4 in a multi-threaded fashion. In Stages 1 and 2, a parallelized file I/O scheme is used and all processes will communicate with each other to exchange k-mers and other linkage information. In Stages 3 and 4, all processes will calculate and send local linkage-related information to the master process in a multi-threaded parallel way. In HiPGA, we have packed the large number of small sized messages into message vector so that to decrease the message passing overhead in distributed-memory systems. We have used parts of the source code from Velvet and PASHA with some algorithmic and data structure changes for implementing HiPGA. The use of these open-source code greatly reduces the development time of our algorithm. HiPGA supports the standard FASTA or FASTQ dataset format for both the single-end and paired-end short reads.

A. Parallelized File I/O Scheme for k-mer Generation and de Bruijn Graph Construction

In order to achieve a high file I/O efficiency, we have used a parallelized file I/O scheme in HiPGA for the k-mer generation and de Bruijn graph construction stages (see Fig. 4 and Fig. 5). Experiments show that our parallelized file I/O scheme can greatly improve the file I/O efficiency in practice.

In the k-mer generation stage, all subfiles are first distributed evenly to all processes. Then each process launches m threads to extract k-mers from local subfiles in parallel.

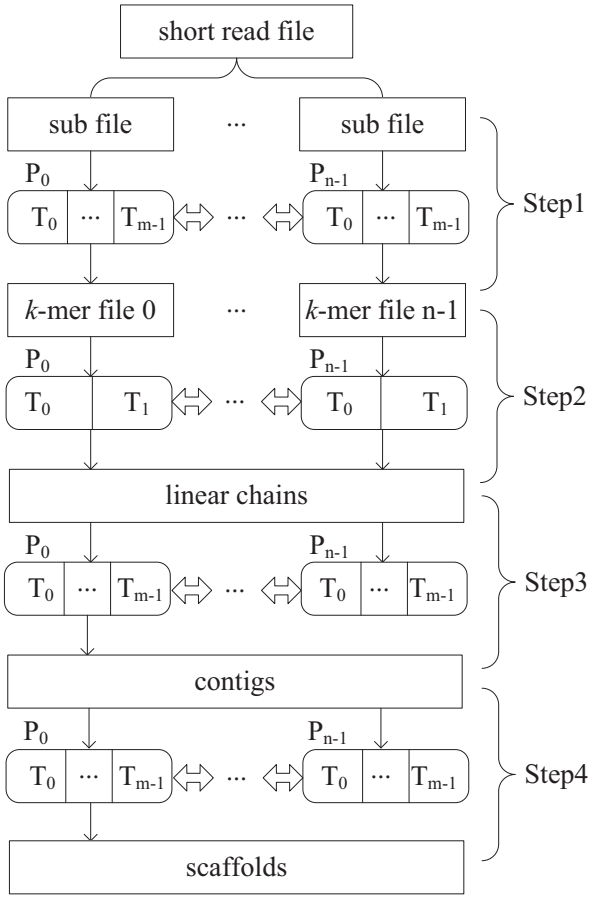

Figure 3. The HiPGA algorithm framework. In Stages 1 and 2, all processes will communicate with each other to exchange k-mers and other linkage information. In Stages 3 and 4, all processes will send local linkage-related information to the master process.

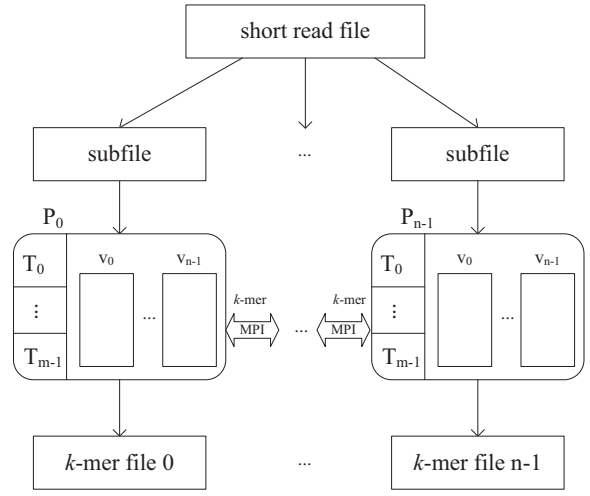

Figure 4. Illustration of our parallelized k-mer generation and distribution.

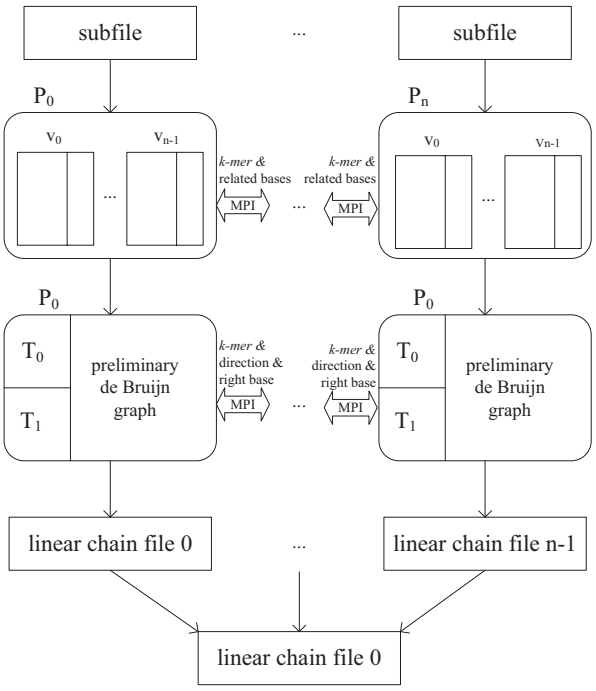

Figure 5. Illustration of our parallelized de Bruijn graph construction and simplification.

Each k-mer will be stored in one of the n local vectors v_i, where n is the total number of processes. The index i of the vector that owns a specific k-mer is computed as $i = h_k\%n$, where h_k is a hash value calculated using a linear congruential hash function. In our implementation, we have defined a threshold for the maximal number of k-mers in each vector. Once a vector reaches this threshold, a communication procedure will be done among all processes. In this procedure, process i will gather all k-mers in v_i from other processes. Algorithm 1 shows the pseudo code for this procedure.

And to gain memory efficiency, the collected k-mers will be locally distinguished and stored in the Google Sparse Hash library (http://code.google.com/p/google-sparsehash). After the communication procedure, all local vectors are cleaned up and each process continues extracting k-mers from local subfiles. At last, each process stores all the identified k-mers in the sparse hash set into a local k-mer file. In HiPGA, the total size of n vectors on each process depends on the RAM available. So, our scheme works well even only small sized RAM is available on each process.

In the de Bruijn graph construction stage, all processes first load the previously generated local k-mer file into a k-mer vector table. Then each process extracts k-mers and

Algorithm 1

```
1: procedure EXCHANGEKMERS
2:     for all processes do
3:         MPI_Gather the number of k-mers belong to it
4:         Malloc memory for local k-mer vectors
5:         Initialize the offset and receiveCount arrays
6:         MPI_Gather k-mers from other processes
7:         Insert k-mers into the Google hash heap
8:     end for
9: end procedure
```

corresponding left and right bases from local subfiles in parallel. Similar to the k-mer generation stage, we also use n local vectors to temporarily store these k-mers and related bases. And if a local vector is full, a communication procedure will be done among all processes to exchange information stored in these vectors. Once all k-mers and related bases are ready, each process will use them to search the k-mer vector table so that to get the multiplicity and linkage information for each k-mer. k-mers, together with the multiplicity and linkage information, are used by each process to generate the local preliminary de Bruijn graph. Since there are still spurious linkages, such as *tips*, *low-coverage paths*, and *bubbles*, in the generated preliminary de Bruijn graph, we need to further simplify them. In our method, each process launches two threads T_0 and T_1 to do the graph simplification. T_0 is used to do the spurious linkage identification and removal work. T_1 is used to exchange information such as k-mers, linkage directions, and right bases of k-mers among processes. After the graph simplification step, the linear chains are produced.

B. Distributed Contig Generation and Scaffolding

In order to make full use of the compute power of distributed-memory systems, we have used both the shared-memory multi-threading and distributed-memory parallelism in HiPGA to implement the contig generation and scaffolding stages (see Fig. 6 and Fig. 7).

In the contig generation stage, each process first produces a k-mer & chain location graph using all k-mers in the linear chains. Then each process aligns short reads in local subfiles to the k-mer & chain location graph in a multi-threaded parallel way. During this procedure, if two adjacent k-mers in a read belong to two different linear chains, the IDs of these two linear chains will be recorded in a local ID pair array. After all local k-mers are aligned, each process will send the local ID pair array to the master process. The master process will use the information in these received ID pair arrays to connect related linear chains into contigs.

Finally in the scaffolding stage, each process first produces a k-mer & contig location graph using all k-mers in previously generated contigs. Then each process aligns two pair-end short reads in local subfiles to the k-mer & contig location graph in a multi-threaded parallel way. Similar to the contig generation stage, if adjacent k-mers in a pair reads belong to two different contigs, the IDs of these two contigs will be recorded in a local ID pair array. And the mapping information of these two pair-end short reads, such

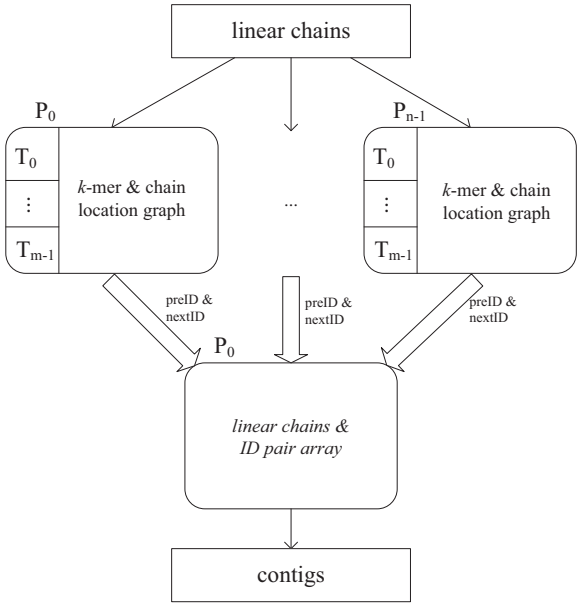

Figure 6. Illustration of our distributed contig generation method.

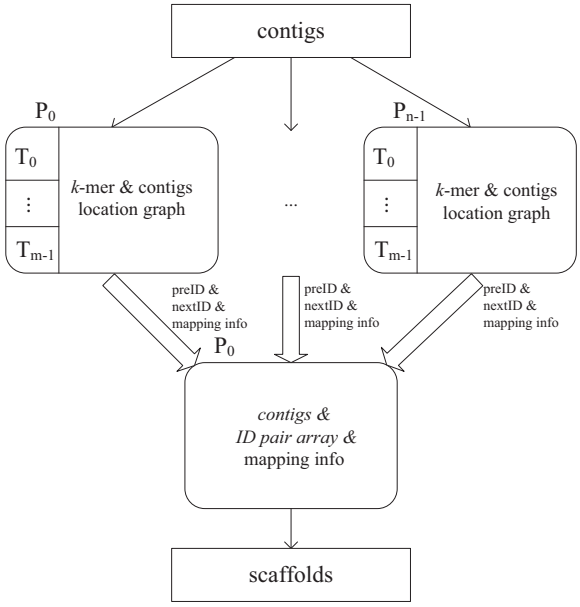

Figure 7. Illustration of our distributed scaffolding method.

as mapping locations and node identifiers, is stored in a local file. After all local pair-end short reads are aligned, each process will send the local ID pair array and the mapping information file to the master process. The master process will further connect related contigs into scaffolds.

IV. Performance Evaluation

HiPGA is currently implemented using standard C++ and the MPI library provided by OpenMPI 1.4.2 [7]. We have executed the program on a compute cluster. The cluster comprises eight nodes and each node contains two Intel Xeon E5650 six-core 2.67GHz CPU with 24GB shared RAM, running the system of Red Hat Enterprise Linux Server release 5.4. All the nodes are connected with each other by a high-speed Infiniband switch.

Table III
Short read datasets used for our experiments.

	Bacillus	**Bordetella**	**E.coli**	**Yoruba**
Accession no.	SRR1014757	SRR942682	SRR1002800	ERX004001
Read length	100	101	101	76
No. of reads	79,998,388	24,746,998	30,608,372	234,562,514
No. of k-mers*	234,122,021	100,377,811	106,878,246	4,362,205,386
Genome size	5,411,809	3,693,053	5,540,893	3,234,834,689

* k=29

We have assembled three paired-end short read datasets from three different genomes and the Yoruba individual dataset. The first three paired-end read datasets: $Bacillus$, $Bordetella$ and $E.coli$ have accession numbers SRR1014757, SRR942682 and SRR1002800 in the NCBI Sequence Read Archive (SRA). The reference genome of the *Bacillus* dataset is *Bacillus cereus* with the accession number NC_004722 in GenBank; the reference genome of the *Bordetella* dataset is *Bordetella holmesii* with the accession number NZ_AOEW00000000; and the reference genome of the *E.coli* dataset is *Escherichia coli J96* with the accession number NZ_ALIN02000000. The Yoruba individual dataset contains seven sub-datasets with accession numbers ERR010996, ERR010997, ERR010998, ERR010999, ERR011000, ERR011001 and ERR011002, all come from the library MMS6. We have compared the performance of HiPGA to two parallelized assemblers: PASHA (version 1.0.10) and ABySS (version 1.3.7). Because Velvet requires a large amount of memory to store the read mapping locations and paired-end information along with the graph, we can not execute it to assemble the above mentioned datasets with the hardware used in our tests.

The assembly quality of HiPGA, ABySS and PASHA is compared in terms of NG50, NG80, and the maximum & mean scaffold sizes. To compute the NG50 and NG80, first we need to order all scaffolds by their lengths. And then we add the lengths from the largest to the smallest until the sum equals or exceeds 50% and 80% of the reference genome size respectively. In this paper, we use the same reference genome size to calculate the NG50 & NG80 scaffold size for all assemblers. For the first three paired-end short read datasets, we only consider scaffolds with the length $\geq$ 150 bases and for the Yoruba individual dataset, we only consider scaffolds with the length $\geq$ 100. The genome coverage is computed from the results obtained from aligning scaffolds to their reference genomes using BLAT version 35 [10]. And in our experiments, if not specified, the parameter *'k=29'* (that is the k-mer size is 29) will be used for all of these three assemblers.

Table IV
Assembly results for Bacillus on 8 CPU cores.

	HiPGA	**PASHA**	**ABySS**
No. of scaffolds	232	236	46,632
Sum (bps)	5,269,941	5,297,869	5,419,381
Max	204,503	206,734	126,582
Mean	22,715	22,448	20,341
NG50	41,644	41,623	32,835
NG80	20,467	23,971	15,575
Genome coverage	97.38%	97.89%	99.02%
Time (in minutes)	72	170	220

Table V
Assembly results for Bordetella on 8 CPU cores.

	HiPGA	**PASHA**	**ABySS**
No. of scaffolds	92	86	184
Sum (bps)	3,452,634	3,452,948	3,462,289
Max	164,187	163,775	123,943
Mean	33,849	40,150	23,749
NG50	77,064	82,835	29,330
NG80	37,675	32,938	16,304
Genome coverage	93.49%	93.50%	93.75%
Time (in minutes)	30	61	70

Table VI
Assembly results for E.coli on 8 CPU cores.

	HiPGA	**PASHA**	**ABySS**
No. of scaffolds	65	66	133
Sum (bps)	5,143,160	5,143,080	5,133,924
Max	1,235,423	739,214	405,566
Mean	79,120	77,925	83,259
NG50	326,170	325,964	145,158
NG80	107,334	118,569	75,373
Genome coverage	92.82%	92.82%	92.65%
Time (in minutes)	38	74	83

First, we use $Bacillus$, $Bordetella$ and $E.coli$ to evaluate different assemblers in terms of assembly quality and execution speed. Because the last two stages of PASHA and

the last stage of ABySS can only run on a single node, we first evaluate these three assemblers on 8 CPU cores of a cluster node. The parameters of all assemblers have been carefully tuned to gain the highest quality of each dataset. In our test, HiPGA uses four processes (each one has two threads) for all stages; PASHA uses four processes (each one has two threads) for the first two stages and uses 8 threads for the last two stages; ABySS uses four processes (each one has two threads) for the first three stages and uses 8 threads for the scaffolding stage. Table IV, V, VI report the assembly quality and performance comparison of HiPGA, ABySS, and PASHA. From them we can see that in our tests, HiPGA achieves speedups up to 3.1 while producing comparable accuracy compared to ABySS and PASHA.

In order to compare the scalability of HiPGA, ABySS, and PASHA on the cluster, we have run them using different number of CPU cores. Fig. 8 shows the execution time of them using up to 64 CPU cores on the cluster. From it we can see HiPGA has a much better scalability when the number of CPU cores increases. And because ABySS and PASHA parallelize partial stages only using the multi-threaded parallelism, their scalability on the cluster is very limited. Table VII, VIII, IX show the assembly quality and performance comparison of HiPGA, ABySS, and PASHA using 64 CPU cores. From them we can see that HiPGA achieves much better performance (with speedups up to 7) while producing better or similar accuracy compared to ABySS and PASHA.

Table VII
ASSEMBLY RESULTS FOR BACILLUS ON 64 CPU CORES.

	HiPGA	**PASHA**	**ABySS**
No. of scaffolds	233	236	392
Sum (bps)	5,260,094	5,297,239	5,388,145
Max	204,504	206,739	125,521
Mean	22,814	22,637	22,834
NG50	41,623	41,644	32,712
NG80	23,727	20,467	15,659
Genome coverage	97.19%	97.88%	99.56%
Time (in minutes)	27	151	189

To compare the performance of HiPGA, ABySS, And PASHA for processing large genomes, we have conducted experiments using the Yoruba individual dataset. For the whole genome of Yoruba individual dataset, the number of k-mers (k=29) is 4,362,205,386 and the size of the simplified de Bruijn graph is about 5GB. Because the last two stages of PASHA and the scaffolding stage of ABySS can only execute on a single node which has only 24GB RAM, PASHA and ABySS are unable to complete the assembly procedure for the whole genome of Yoruba individual. In our experiments, we only use the first three sub-datasets of the Yoruba individual dataset. The number of reads in the first three sub-datasets is 95,420,416 and the corresponding number of k-mers (k=29) in the preliminary de Bruijn graph is 2,482,802,325. We have used 64 CPU cores in our tests. Table X shows the assembly results. From it we can see that HiPGA also achieves the fastest execution speed while delivering comparable accuracy compared to ABySS and PASHA.

Table VIII
ASSEMBLY RESULTS FOR BORDETELLA ON 64 CPU CORES.

	HiPGA	**PASHA**	**ABySS**
No. of scaffolds	103	93	184
Sum (bps)	3,451,818	3,452,818	3,453,186
Max	163,782	163,676	121,482
Mean	37,127	37,127	28,415
NG50	77,624	87,624	29,124
NG80	44,746	42,796	17,092
Genome coverage	93.49%	93.49%	93.50%
Time (in minutes)	8	42	51

Table IX
ASSEMBLY RESULTS FOR E.COLI ON 64 CPU CORES.

	HiPGA	**PASHA**	**ABySS**
No. of scaffolds	66	67	135
Sum (bps)	5,142,768	5,142,067	5,124,237
Max	1,230,241	739,214	404,243
Mean	85,964	80,335	84,153
NG50	325,964	325,823	144,131
NG80	108,569	118,568	78,343
Genome coverage	92.81%	92.80%	92.48%
Time (in minutes)	10	60	63

Table X
ASSEMBLY RESULTS FOR THE YORUBA INDIVIDUAL DATASET ON 64 CPU CORES.

	HiPGA	**PASHA**	**ABySS**
No. of scaffolds	1,546,244	1,557,368	2,021,375
Sum (bps)	571,606,978	575,784,992	583,259,736
Max	69,592	69,589	59,261
Mean	376	376	329
NG50	424	420	403
NG80	247	248	235
Time (in minutes)	96	234	391

Fig. 9 shows the runtime of each stage of HiPGA and PASHA for processing the Yoruba individual dataset. From it we can see that each stage of HiPGA achieves much better performance.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented HiPGA – a parallelized de Bruijn graph-based genome assembler for high-

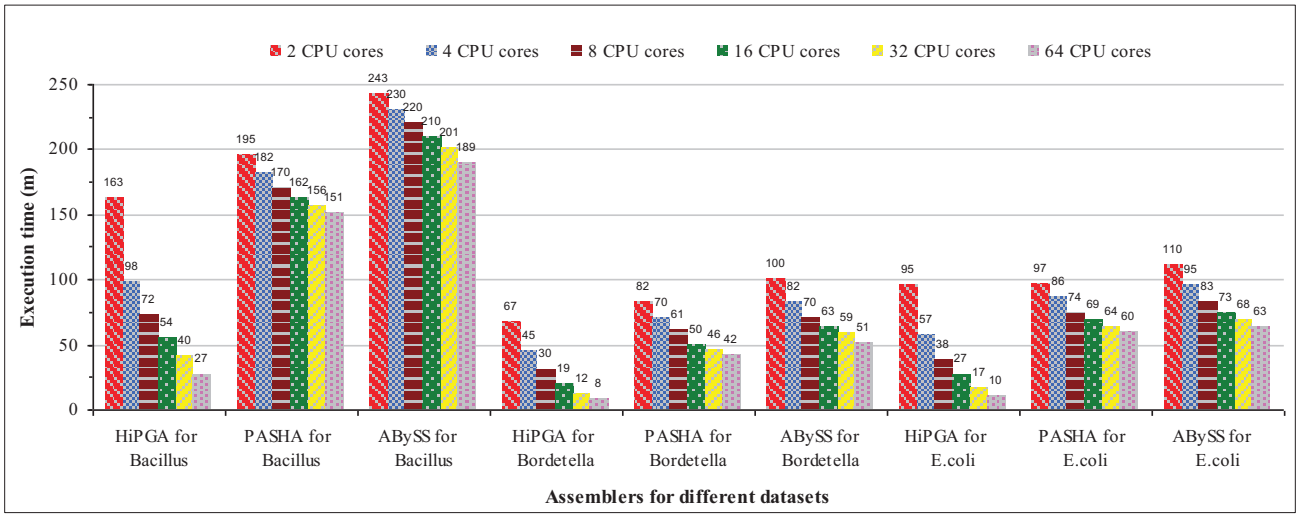

Figure 8. Runtime comparison using up to 64 CPU cores for assembling the Bacillus, Bordetella, and E.coli datasets.

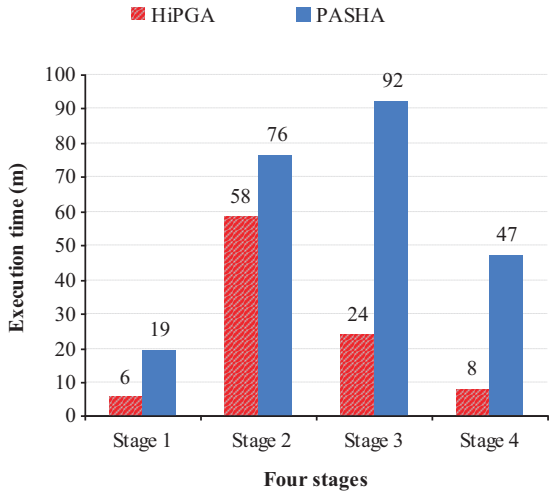

Figure 9. Stage runtime comparison of HiPGA and PASHA on 64 CPU cores for assembling the Yoruba individual dataset.

throughput short read data. In order to make full use of the compute power of both shared-memory multi-core CPUs and distributed-memory systems, we have used a parallelized file I/O scheme as well as a hybrid parallelism for the four stages of the assembly pipeline. Evaluations using three real paired-end small genome datasets and the Yoruba individual large genome dataset show that compared to two other well parallelized assemblers: ABySS and PASHA, HiPGA achieves speedups up to 7 while delivering comparable accuracy on 64 CPU cores of a compute cluster.

The exponential growth of short read datasets demands even more parallel and distributed assembly solutions in the future. Because the performance of many-core architectures grows much faster than the performance of standard multi-core CPUs, our future work includes designing efficient parallel assembly algorithms on many-core architectures such as GPUs and MIC.

REFERENCES

[1] P. Ariyaratne and W. Sung. PE-Assembler: de novo assembler using short paired-end reads. *Bioinformatics*, 27(2), 2011.

[2] J. Butler, I. MacCallum, M. Kleber, I. Shlyakhter, M. Belmonte, E. Lander, C. Nusbaum, and D. Jaffe. Allpaths: De novo assembly of whole-genome shotgun microreads. *Genome Res*, 18(5):810–820, 2008.

[3] M. Chaisson, D. Brinza, and P. Pevzner. De novo fragment assembly with short mate-paired reads: Does the read length matter? *Genome Res*, 19(2):336–346, 2009.

[4] M. Chaisson and P. Pevzner. Short read fragment assembly of bacterial genomes. *Genome Res*, 18(2):324–330, 2008.

[5] P. Compeau, P. Pevzner, and G. Tesler. How to apply de Bruijn graphs to genome assembly. *Nature Biotechnology*, 29(11):987–991, 2011.

[6] J. Dohm, C. Lottaz, T. Borodina, and H. Himmelbauer. SHARCGS, a fast and highly accurate short-read assembly algorithm for de novo genomic sequencing. *Genome Res*, 17(11):1697–1706, 2007.

[7] G. Edgar, E. Graham, B. George, A. Thara, J. Jack, M. Jeffrey, S. Vishal, K. Prabhanjan, B. Brian, L. Andrew, H. Ralph, J. David, L. Richard, and S. Timothy. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, 2004.

[8] B. Jackson, M. Regennitter, X. Yang, P. Schnable, and S. Aluru. Parallel de novo assembly of large genomes from high-throughput short reads. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, 2010.

[9] W. Jeck, J. Reinhardt, D. Baltrus, M. Hickenbotham, V. Magrini, E. Mardis, J. Dangl, and C. Jones. Extending assembly of short dna sequences to handle error. *Bioinformatics*, 23(21):2942–2944, 2007.

[10] W. Kent. BLAT–The BLAST-Like Alignment Tool. *Genome Research*, 12(4):656–664, 2002.

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

[12] X. Liu, P. Pande, H. Meyerhenke, and D. Bader. PASQUAL: Parallel techniques for next generation genome sequence assembly. *IEEE Trans. Parallel Distrib. Syst.*, 24(5):977–986, 2013.

[13] Y. Liu, B. Schmidt, and D. Maskell. Parallelized short read assembly of large genomes using de bruijn graphs. *BMC Bioinformatics*, 12:354, 2011.

[14] E. Mardis. The impact of next generation sequencing on genetics. *Trends Genet*, 3(24):133–141, 2008.

[15] P. Pevzner, H. Tang, and M. Waterman. An eulerian path approach to dna fragment assembly. *Proc Natl Acad Sci USA*, 98(17):9748–9753, 2001.

[16] M. Pop and S. Salzberg. Bioinformatics Challenges of New Sequencing Technology. *Trends Genet*, 3(24):142–149, 2008.

[17] B. Schmidt, R. Sinha, B. Beresford-Smith, and S. Puglisi. A fast hybrid short read fragment assembly algorithm. *Bioinformatics*, 25(17):2279–2280, 2009.

[18] J. Simpson, K. Wong, S. Jackman, J. Schein, S. Jones, and I. Birol. Abyss: a parallel assembler for short read sequence data. *Genome Res*, 19(6):1117–1123, 2009.

[19] R. Warren, G. Sutton, S. Jones, and R. Holt. Assembling millions of short dna sequences using ssake. *Bioinformatics*, 23(4):500–501, 2007.

[20] D. Zerbino and E. Birney. Velvet: Algorithms for de novo short read assembly using de bruijn graphs. *Genome Res*, 18(5):821–829, 2008.

[21] K. Zhao, W. Liu, G. Voss, and W. Muller-Wittig. Accelerating de bruijn graph-based genome assembly for high-throughput short read data. In *The 19th IEEE International Conference on Parallel and Distributed Systems*, 2013.