# Accelerating *De Novo* Assembler WTDBG2 on Commodity Servers

Ming Dun[1], Yunchun Li[1,2], Xin You[2], Qingxiao Sun[2], Zerong Luan[4], and Hailong Yang[2,3]

School of Cyber Science and Technology[1]
School of Computer Science and Engineering[2]
Beihang University[1,2], Beijing, China, 100191
State Key Laboratory of Mathematical Engineering and Advanced Computing[3]
College of Life Sciences and Bioengineering[4]
Beijing University of Technology[4], Beijing, China, 100083

**Abstract.** *De novo* genome assembly reconstructs the chromosomes from massive relatively short fragmented reads and serves as fundamental for studying new species where there is no reference genome. Wtdbg2 is a *de novo* assembler for long reads that is up to hundreds of kilobases. It is based on fuzzy-Bruijn graph (FBG) and is ten times faster than the cutting-edge assemblers such as Canu. However, the performance of wtdbg2 still requires further improvement: *1)* it requires up to terabytes of memory to compute the assembly, which is infeasible to run on commodity server; *2)* it requires tens of hours for assembling on large datasets such as genomes of *homo sapiens*. To address the above drawbacks, we propose several optimization techniques for accelerating wtdbg2 on commodity server, including a memory auto-tuning scheme, sequence alignment optimization and intermediate result elimination in the output procedure. We compare the optimized wtdbg2 with the original implementation and two cutting-edge assemblers on real-world datasets. The experiment results demonstrate that optimized wtdbg2 achieves maximum and average speedup of 2.31× and 1.54× respectively. In addition, our proposed optimization reduces the memory usage of wtdbg2 by 39.5% without affecting the correctness.

**Keywords:** genome assembly · wtdbg2 · performance optimization · computational biology · auto-tuning · load balance

## 1 Introduction

*De Novo* genome assembly aims to generate a new genome from DNA fragments (named as reads) without the reference genome. It is of great significance in bioinformatics for identifying previously uncharacterized genomes [23] and analyzing the structural genomic changes [22]. Moreover, with the prosperities and advances of DNA sequencing technologies from Oxford Nanopore Technologies (ONT) and Pacific Bioscience (PacBio), the length of reads has been increased up to several hundreds of base pairs (bps). The most popular assembly methods such as *de Bruijn* Graph are developed for short read assembly in second-generation DNA sequencing. However, recent research works have taken efforts to modify the *de Bruijn* Graph for long and error-prone reads in next-generation sequencing such as A-Bruijn graph in Flye [13] and fuzzy-Bruijn graph (FBG) in wtdbg2 [22].

Wtdbg2 [22] is one of the fastest status-quo assemblers for long noisy reads and has been adopted in Novel Sequence Insertion (NSI) tools such as rCANID [12] in addition to analysis for genome datasets [24]. Wtdbg2 is based on FBG that wraps each 256bp of reads into a unit named as bin and utilizes hash lists to encode the reads. In wtdbg2, a k-bin of FBG contains K consecutive bins. To improve the error toleration of FBG, a vertex can represent various k-bins if aligned together in sequence alignment routine. For the algorithm details of FBG, the readers can refer to [22].

However, to further improve the performance of wtdbg2, there are two major challenges to be addressed. The first challenge is the prohibitive memory consumption generated during the execution of wtdbg2. For instance, the wtdbg2 takes up to 1,788GB memory on *Axolotl* [5], thus it is infeasible to assemble large datasets on commodity servers that usually contains less than 512GB memory. The second challenge is the tremendous computation power required by wtdbg2. For instance, the wtdbg2 takes up to tens of hours for mammalian genome assembly on datasets such as *homo sapiens*.

Therefore, to alleviate the memory consumption and to further enhance the performance of wtdbg2, we propose a memory auto-tuning scheme based on regression model to reduce memory usage. In addition, we improve the parallelization of sequence alignment routine, which is one of the major bottlenecks, by enhancing the thread efficiency and load balance in wtdbg2. Moreover, we optimize the output procedure by eliminating the redundant intermediate results. We compare our optimization strategies with the original wtdbg2 and other two status-quo assemblers Canu and Flye under five real-world genome datasets and the results show that the maximum acceleration rate is of 2.31×, with the average acceleration rate is of 1.54× and the memory cost can be reduced up to 39.5%, without affecting the correctness. With our proposed optimizations, massive parallel genome assembling that is previously infeasible with wtdbg2 on commodity server now can be run successfully.

– We perform comprehensive analysis of the performance bottlenecks in wtdbg2 and identify that the all-vs-all sequence alignment and output procedure are the major hotspots in large genome assembly.
– We propose a memory auto-tuning scheme based on regression model to alleviate the memory usage of wtdbg2. In addition, we accelerate the sequence alignment and optimize the output procedure to further improve the performance of wtdbg2.
– We evaluate our optimized wtdbg2 with five real-world genome datasets and compare to the original wtdbg2 as well as two cutting-edge genome assemblers. The experiment results demonstrate our optimization strategies are

effective to reduce the memory usage in addition to improve the performance of wtdbg2 with correctness validation.

This paper is organized as follows. In Section 2, we present the background of genome assembly and wtdbg2. We present the bottleneck analysis and optimization strategies of wtdbg2 in Section 3. The detailed implementations are described in Section 4. Section 5 presents the evaluation results of the optimized wtdbg2. Section 6 describes the related work. Section 7 concludes this paper.

## 2 Background

### 2.1 Genome Assembly

Genome assembly is a computational representation of a genome sequence. Since we are not able to sequence along the whole length of DNA, genome assembly provides a computational method to reconstruct a DNA sequence from a large set of short reads of sequenced DNA fragments, which may be overlapped by each other. Moreover, the *de novo* genome assembler is a type of assembler aiming at assembling short reads to construct full-length sequences of DNA without any reference template. Among the methodologies for implementing a *de novo* genome assembler, *De Bruijn* graph is one of the most popular method, which aligns *k-mers* based on $k-1$ sequence conservation to create contigs. The short k-mers allow fast hashing to decrease the computationally intensity and enhance the overall performance.

Nowadays, a number of software such as Canu [14] and MECAT [25] is developed for *de novo* assembly. Most of them reported that *sequence alignment* process, which includes *k-mer* counting and contig generation, is the major bottleneck of these existing assemblers [10] and consumes large memory footprint when parallelized on shared-memory system.

### 2.2 Wtdbg2

Wtdbg2 [22] is an efficient long-read genome assembler. It implements fuzzy-Bruijn graph (FBG) which extends the basic ideas of De Bruijn graph (DBG) to support long noisy reads on shared memory system with *pthread* parallelization. Similar to DBG, a ”base” in FBG is a 256bp bin and a k-mer in FBG consists of k consecutive bins on reads. FBG is made up of vertices of k-bins and edges between two vertices which indicates their adjacency on a read. The difference is that a single vertex may represent different k-bins if they are aligned together based on all-vs-all read alignment, which tolerates errors in noisy long reads.

As shown in Figure 1, the workflow to obtain the constructed contigs of input reads contains four steps: (a) binning and pairwise alignment, (b) graph construction, (c) graph clearing and (d) consensus. In the first stage, input reads are all loaded from files to memory and each base is encoded with 2 bits. For all-vs-all read alignment, a Smith-Waterman-like dynamic programming is applied. After alignment, FBG is then constructed by adding vertices according to the obtained all-vs-all alignments and edges of two vertices if they are in the same read. In the phase of graph clearing, wtdbg2 retains only one edge if there are multiple edges between two k-bins. Besides, edges covered by less than 3 reads are omitted. The resulting FBG is then consensused to obtain the final long contigs.

In assembling, the execution of wtdbg2 is split into two phases. The first phase is binning and pairwise alignment, whose results are written to disk as intermediate results. The second phase includes graph constructing, graph cleaning and consensus, which consumes large memory footprint.

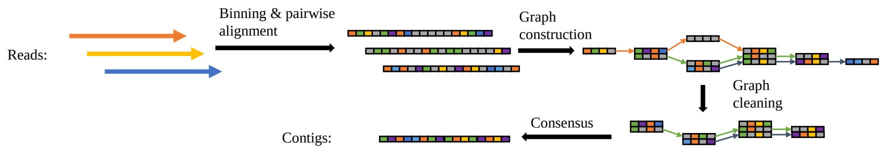


Fig. 1: The overview of the long-read assembling process in wtdbg2.

## 3 Methodology

In this section, we first provide the analysis for bottleneck as well as for the memory consumption of wtdbg2 to guide our optimization. Then we present the strategies for reducing the memory usage and optimizing execution hotspots, including the sequence alignment and output procedure.

### 3.1 Bottleneck Analysis

The reported [5] memory consumption of wtdbg2 under different datasets are shown in Table 1. Since most of the commodity servers contain less than 256GB memory, the large mammalian datasets especially the genome of *homo sapiens* cannot be directly assembled by wtdbg2 on the commodity servers. Though the original wtdbg2 provides a variable *kbmparts* to control the memory usage, it still requires manual tuning, and thus facing the risk of assembly failure due to memory overflow.

We profile the execution of wtdbg2 using HPCToolkit [6]. The result of bottleneck analysis is shown in Figure 2, where the execution time for output procedure is excluded. The description of the datasets is shown in Table 2. It is clear that the KBM indexing is the major bottleneck of wtdbg2 under small datasets such as *E.coli* and *C.elegans*. However, when the dataset's genome size increases, especially in the genomes of *homo sapiens*, the sequence alignment process dominates the execution time and becomes the bottleneck of wtdbg2. Since the large datasets usually take hours to be processed and thus worth the efforts for acceleration, we focus on optimizing the sequence alignment routine to improve the performance of wtdbg2 on large datasets.

In addition, the size of output contigs of wtdbg2 can take up to 30GB under dataset *Human HG00733*. On the other hand, it can take more than 41.3% of entire execution time to write the contigs under dataset *E.coli* (e.g., 2.03 seconds of the overall 4.91 seconds). What's more, the performance of the output procedure also needs to be improved to further accelerate wtdbg2. Moreover, the wtdbg2 also writes the intermediate alignment results to the hard disks, whose size is at the same order of magnitude as the contigs. These intermediate alignment results are redundant during the execution. Therefore, we propose to eliminate the redundant intermediate results in order to accelerate the output procedure.

Table 1: The memory cost of wtdbg2 under different datasets.

| | *C.elegans* | *D.melanogaster A4* |
|---|---|---|
| **Memory Cost(GB)** | 1.0 | 19.4 |
| | *Human NA24385* | *Human HG00733* |
| **Memory Cost(GB)** | 112.9 | 338.1 |

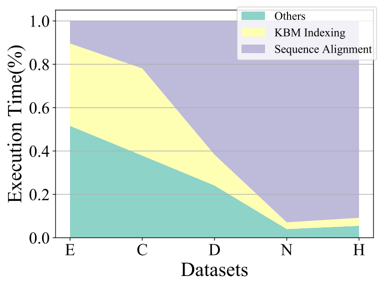


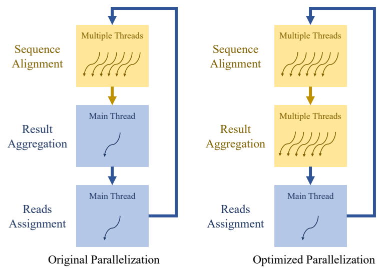
Fig. 2: The result of bottleneck analysis of wtdbg2. E, C, D, N and H represents the datasets: *E.coli*, *C.elegans*, *D.melanogaster A4*, *Human NA24385* and *Human HG00733*. The time for output is excluded.

### 3.2 Memory Auto-tuning

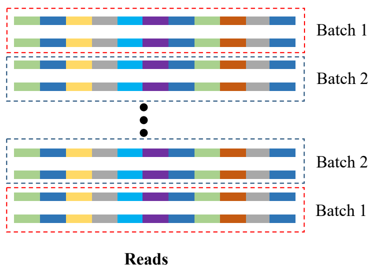
We develop an auto-tuning scheme based on linear regression model to satisfy the memory demand of the large genome datasets. We identify that the highest memory usage is at the phase of sequence alignment where wtdbg2 easily crashes due to memory shortage at large datasets. Another execution phase that is unlikely to crash due to memory shortage is the reads collecting phase whose memory usage approximately equals to the size of the input dataset. We also notice that there is a parameter *kbm_parts* in wtdbg2 to control the number of iterations of alignment, which exhibits a negative linear relationship with memory usage in sequence alignment routine. Based on the above observations, we develop a regression model to estimate the memory usage of sequence alignment based on the memory measurement of reads collecting. At the reads collecting phase, the model estimates how much sequence alignment would exceed the system memory capacity and then adjusts the *kbm_parts* automatically to satisfy the memory demand. The regression model is built on the statistics collected in previous execution of wtdbg2. Since running the regression model only requires monitoring the system memory usage and executing a simple linear function, the overhead of memory auto-tuning is negligible.

## 3.3 Sequence Alignment Optimization

**Thread Efficiency** We improve the parallelization of sequence alignment routine in wtdbg2 to accelerate the major bottleneck under large datasets. In the original wtdbg2, as shown in Figure 3a, after processing the alignment with threads at the granularity of a single read, the alignment results of each read are merged into the graph in the master thread. In addition, the read assignment among threads is also performed in the master thread. After analyzing the performance of sequence alignment, the result merging function *map2rdhit_graph* dominates the execution time of the master thread, which increases the time delay between two reads alignment for a thread and thus decreases the efficiency of multi-threading. Since wtdbg2 adopts the shared-memory parallelization using pthread, we distribute the work of result merging among threads by leveraging the mutual exclusive locks [16] as shown in Figure 3a, which improves the multi-threading efficiency of the sequence alignment routine.



(a) Original and optimized parallelization of sequence alignment



(b) Batched read assignment scheme in sequence alignment

Fig. 3: The illustration of optimization strategies in sequence alignment routine of wtdbg2.

**Load Balance** When the sequence alignment routine analyzes a specific read $R_i$ in wtdbg2, it only compares $R_i$ with reads that have larger identifiers. Hence, as the identifier of the $R_i$ increases, the number of reads that $R_i$ needs to be aligned with decreases. Since a thread only performs sequence alignment for reads in ascending order in wtdbg2, load imbalance becomes severe as the identifier of the read increases. Moreover, reducing the number of assignments can alleviate the load imbalance between master and slave threads. However, fewer assignments means more reads in each assignment, which in turn increases the load of slave threads.

To address load imbalance among threads and to mitigate the load in the master thread, we develop a batched read assignment scheme for the sequence alignment. The proposed allocation scheme is shown in Figure 3b, where *batch-size* is set to 4. We partition all the reads into $M$ *batches*, each with $N$ reads. For the $i$th *batch*, the index of its reads ranges from $Ni$ to $Ni + \frac{N}{2}$ in ascending or descending order. Thus, the number of reads each thread needs to compare is the same theoretically, which also decreases the number of read assignments in the master thread. Since the solution space of $M$ is huge and the best $M$ is different for various datasets, we develop a linear regression model to automatically search for the optimal setting of $M$.

## 4 Implementation

In this section, we provide the implementation details of the optimization methodolgies proposed in Section 3.

### 4.1 Memory Auto-tuning

The processing logic of memory auto-tuning is shown in Algorithm 1. After the reads collecting process finishes, our implementation uses function *check_meminfo* to obtain the total memory $M_T$ and available memory $M_A$ of the system from the file *meminfo*. Then the memory usage of reads $M_{READ}$ can be calculated by $M_T - M_A$ and the highest memory usage $M_{MAX}$ can be estimated by the regression model. At the end of the auto-tuning process, the parameter *kbm_parts* is determined using $kbm\_parts = \frac{M_{MAX}}{M_T}$. The value of *kbm_parts* is later used in sequence alignment to avoid exceeding the system memory constraint.

**Algorithm 1** The logic of memory auto-tuning

```
1: Input: regression model $f : M_{READ} \rightarrow M_{MAX}$
2: Output: $kbm\_parts$
3: $(M_T, M_A) \leftarrow$ check_meminfo()
4: $M_{READ} \leftarrow M_T - M_A$
5: $M_{MAX} \leftarrow f(M_{READ})$
6: $kbm\_parts \leftarrow \frac{M_{MAX}}{M_T}$
```

### 4.2 Sequence Alignment Optimization

**Thread Efficiency** As shown in Algorithm 2, for each read $R$ in the sets of read $\mathbf{R}$, the master thread finds an idle slave thread $k$ and then assigns the sequence to it. The thread $k$ first encodes its read $R$ with a specific hash list to get the binned sequence $BS_k$ and then performs the sequence alignment between $BS_k$ and other binned sequences whose indices are bigger than the index of $R$. After the sequence alignment, it generates the alignment results including the cigars $cigars_k$ and the overlapped bins with other sequences $hits_k$. With the results generated, the thread $k$ requests for the mutex lock and merges the local results to the FBG graph. The reason for applying mutex lock is to avoid writing conflict among multiple threads. Once the result merging finishes, the thread $k$ releases the lock and becomes an idle thread. Then the master thread continues to assign remaining reads to thread $k$ unless all reads have already been assigned. The parallel result merging method reduces the assignment delay for the slave threads as well as alleviating the load for the master thread.

**Algorithm 2** The logic of thread efficiency optimization

```
1: Input: Reads $\mathbf{R}$
2: Output: cigars $g\_cigars$, hits $g\_hits$
3: for each $R \in \mathbf{R}$ do
4:     $k$ = get-idle-thread-id()
5:     assign $R$ to thread $k$
6:     $BS_k \leftarrow$ query_index($R$, $k$)
7:     ($cigars_k$, $hits_k$) = kbm_alignment($BS_k$, $k$) /*the new intermediate result in thread $k$*/
8:     get-mutex_lock($k$)
9:     ($g\_cigars$, $g\_hits$) = result-gather($cigars$, $hits$, $cigars_k$, $hits_k$)
10:    free-mutex_lock($k$)
11:    turn-idle-thread($k$)
12: end for
```

**Load Balance** The implementation of batched read assignment is shown in Algorithm 3. The variable *batchtime* is calculated by the regression model based on the number of reads $\mathbf{R}.size$, which is then used to determine the optimal setting of *batchsize*. After the number of batched reads (*batchsize*) is determined, the master thread computes the indices of reads that belong to a specific *batch* $i$. Once the *batch* is assigned, thread $k$ will finish the sequence alignment process for the reads in batch $i$, generating the cigars as well as hits and merging them to the graph.

### 4.3 Output Optimization

We use a condition parameter *WITHALIGNMENT* to control the output of intermediate alignment results written to hard disks. Once the *WITHALIGNMENT* = *true*, the output of intermediate results is skipped. In addition, we use another condition parameter *PRINTGRAPH* to control the output of runtime information such as that for FBG graph.

**Algorithm 3** The logic of load balance optimization

```
1: Input: all the reads $\mathbf{R}$
2: Output: cigars $g\_cigars$, hits $g\_hits$
3: $batchtime \leftarrow f_1(\mathbf{R}.size)$
4: $batchsize \leftarrow \mathbf{R}.size/(batchtime \times ncpu - 1)$
5: for $i \leftarrow 0 \rightarrow batchtime \times ncpu$ do
6:     $\mathbf{batch}_i \leftarrow$ get_newbatch()
7:     $k$ = get-idle-thread-id()
8:     $BS_k \leftarrow$ query_index_batch($\mathbf{batch}_i$, $k$)
9:     ($cigars_k$, $hits_k$) = kbm_alignment_batch($BS_k$, $k$) /*the new intermediate result in thread $k$*/
10:    turn-idle-thread($k$)
11: end for
```

## 5 Evaluation

We implement the proposed optimizations for wtdbg2 and compare its performance with the original wtdbg2 and two state-of-art sequence assemblers under five real-world datasets.

### 5.1 Experiment Setup

**Assemblers For Comparison.** We choose two cutting-edge genome assemblers Canu [14] and Flye [13] for comparison. These two assemblers use both PacBio and Oxford Nanopore datasets as input. However, the assemble and consensus processes are integrated in Canu and Flye, whereas they are separate in wtdbg2. Canu is based on MinHash Alignment Process (MHAP) [8] and Celera Assembler [9], whereas Flye is based on A-Bruijn Graph.

**Genome Datasets.** We choose five datasets to compare the performance of different genome assemblers, including E.coli [5], C.elegans [1], D.melanogaster ISO1 [2], Human NA24385 [4] and Human HG00733 [3]. The details of the datasets are shown in Table 2. For convenience, we abbreviate the datasets to **E, C, DISO, NA** and **HG** respectively.

**Experiment Platform.** We conduct all the experiments on a commodity server that is memory constrained. The server has 2 Intel Xeon E5-2680v4 CPUs, each with 14 hyper-threaded cores. The memory capacity of the server is 256 GB, which is much smaller than the fat node with 2 TB memory used in the experiment of original wtdbg2 [22].

**Evaluation Criteria.** We first compare the memory usage of wtdbg2 using memory auto-tuning optimization with the original wtdbg2 to measure the effectiveness of the memory auto-tuning. To evaluate the performance improvement, we compare the optimized versions of wtdbg2 with Canu and Flye. We only measure the execution time of assemble process in Canu and Flye. We choose wtdbg2 applied with memory auto-tuning optimization as the baseline since the original wtdbg2 fails to run on large datasets such as *HG* due to out of memory error. For validation of correctness, we use QUAST [11] with assembly indicators including N50, NGA50, genome fraction and total genome length. All assembly indicators derived for both original and optimized wtdbg2 are based on pre-polished assembly results.

Table 2: The genome datasets.

| Dataset | Dataset Type | Coverage | genome size |
|---|---|---|---|
| *E.coli* | PacBio RSII | 20× | 4.6MB |
| *C.elegans* | PacBio RSII | 80× | 100MB |
| *D.melanogaster ISO1* | Oxford Nanopore | 32× | 144MB |
| *Human NA24385* | PacBio CCS | 28× | 3GB |
| *Human HG00733* | PacBio Sequel | 93× | 3GB |

### 5.2 Performance Analysis

**Memory Auto-tuning** The memory usage of the optimized wtdbg2 normalized to the original wtdbg2 is shown in Figure 4. We adopt the memory usage of the original wtdbg2 reported from [5], which indicates the dataset *HG00733* consumes 338.1GB memory that exceeds the memory capacity of our experiment server. From Figure 4, we can see that the regression model used in the optimized wtdbg2 can successfully predict the memory usage and adjust the parameter *kbm_parts* accordingly so that it satisfies the memory demand from large datasets. Specifically, for dataset *HG00733*, the memory auto-tuning scheme reduces the memory demand by 39.5% compared to the original wtdbg2, and thus enables a successful execution on this dataset.

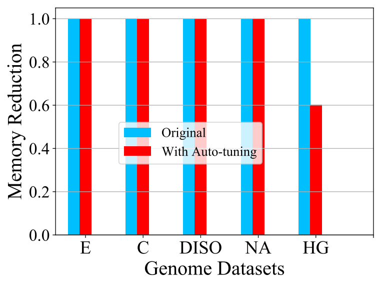


Fig. 4: The memory usage after applying the memory-auto tuning optimization on wtdbg2.

**Performance Improvement** The performance comparison results between the optimized wtdbg2 and two state-of-art assemblers Canu and Flye are shown in Figure 5. The metrics for assembly quality including N50, NGA50, genome fraction and total genome length compared to the reference genome are shown in Table 3, Table 4, Table 5 and Table 6, respectively. The missing bar in Figure 5 and the '-' symbol in Table 3 to 6 indicate the execution failure due to out of memory error or extreme long execution time (e.g., more than 7 days).

From Figure 5, we can see that the optimized wtdbg2 with both sequence alignment and output optimizations achieves the best performance under all datasets. Based on the results from Table 3 to Table 6, we can see that the optimized wtdbg2 achieves comparable or even better assembly quality when compared to the original wtdbg2. The highest speedup achieved by A-wtdbg2 is 2.31× under dataset **NA** and the average speedup of A-wtdbg2 is 1.54× compared to the baseline. The reason for A-wtdbg2 to achieve the lower speedup with datasets **E** and **C** is that the sequence alignment routine takes up a small portion of the entire execution time on these datasets, which is also described in the bottleneck analysis in Sec 3.1.

Table 3: The N50 (million base pairs (Mbps)) of the assembly results from different assemblers.

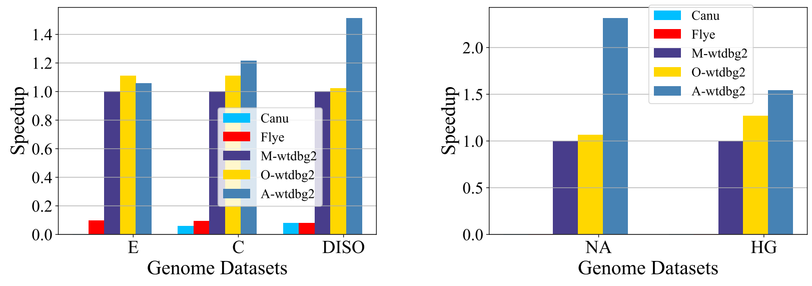
| Datasets | Canu | Flye | wtdbg2 | M-wtdbg2 | O-wtdbg2 | A-wtdbg2 |
|---|---|---|---|---|---|---|
| **E** | 4.6680 | 4.6371 | 4.6719 | 4.6358 | 4.6358 | 4.6341 |
| **C** | 2.0681 | 2.8587 | 1.9726 | 1.9726 | 1.9726 | 1.9718 |
| **DISO** | 4.2986 | 16.431 | 6.2255 | 6.2258 | 6.2259 | 6.2268 |
| **NA** | - | - | 15.512 | 15.512 | 15.512 | 15.511 |
| **HG** | 34.637 | - | - | 21.293 | 26.354 | 24.699 |

Table 4: The NGA50 (million base pairs (Mbps)) of the assembly results compared to reference genome from different assemblers.

| Datasets | Canu | Flye | wtdbg2 | M-wtdbg2 | O-wtdbg2 | A-wtdbg2 |
|---|---|---|---|---|---|---|
| **E** | 0.033895 | 0.033539 | 0.033539 | 0.033540 | 0.033539 | 0.033935 |
| **C** | 0.56166 | 0.56513 | 0.558,99 | 0.55899 | 0.55899 | 0.55752 |
| **DISO** | 1.0913 | 1.6581 | 1.2923 | 1.2923 | 1.2038 | 1.1852 |
| **NA** | - | - | 2.0664 | 2.0531 | 2.0531 | 2.0714 |
| **HG** | 2.4485 | - | - | 1.6464 | 1.9423 | 1.9534 |

Table 5: The genome fraction(%) of the assembly results compared to reference genome from different assemblers.

| Datasets | Canu | Flye | wtdbg2 | M-wtdbg2 | O-wtdbg2 | A-wtdbg2 |
|---|---|---|---|---|---|---|
| **E** | 73.451 | 73.478 | 72.97 | 72.992 | 73.056 | 72.994 |
| **C** | 99.685 | 99.647 | 98.662 | 98.66 | 98.662 | 98.686 |
| **DISO** | 93.516 | 93.273 | 88.361 | 88.35 | 88.322 | 88.411 |
| **NA** | - | - | 87.607 | 87.705 | 87.705 | 88.417 |
| **HG** | 91.726 | - | - | 87.284 | 88.39 | 88.39 |



(a) Small Non-mammalian Datasets (b) Large Mammalian Datasets

Fig. 5: Performance comparison among optimized wtdbg2, Canu and Flye, where M-wtdbg2, O-wtdbg2 and A-wtdbg2 is the original wtdbg2 applied with memory auto-tuning optimization (baseline), both memory and output optimizations and all proposed optimizations, respectively.

Table 6: The total genome length (million base pairs(Mbps)) of the assembly results from different assemblers.

| Datasets | Canu | Flye | wtdbg2 | M-wtdbg2 | O-wtdbg2 | A-wtdbg2 |
|---|---|---|---|---|---|---|
| **E** | 4.6476 | 4.6371 | 4.6358 | 4.6719 | 4.6719 | 4.6702 |
| **C** | 108.19 | 102.46 | 106.10 | 106.09 | 106.09 | 106.31 |
| **DISO** | 140.57 | 144.43 | 136.90 | 136.86 | 136.78 | 136.82 |
| **NA** | - | - | 2,749.3 | 2,752.6 | 2,752.4 | 2,774.5 |
| **HG** | 3,038.5 | - | - | 2,775.5 | 2,804.1 | 2,802.3 |

### 5.3 Parameter Sensitivity Analysis

To better understand the impact of parameter *batchtime* described in Section 4.2 on the performance of wtdbg2, we adjust the setting of *batchtime* within the range of A±16 under four different datasets, where A is the setting after applying memory auto-tuning optimization described in Section 4.2. From Figure 6 we can see that when the *batchtime* increases, which in turn decreases the *batchsize*, the performance of wtdbg2 usually gets improved. The reason for that is a finer-grained partitioning improves the load balance, while at the same time hardly increases the load for threads in a single execution of alignment. Moreover, we notice that the performance impact of *batchtime* varies across different datasets and thus an auto-tuning scheme is required to search for the optimal *batchtime*.

## 6 Related Work

As genome assembly is widely used to obtain genome information, various genome assemblers are developed to construct assembly graphs from a large set of reads,

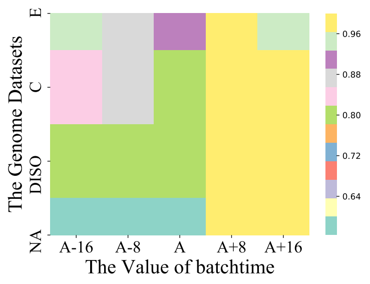


Fig. 6: The performance impact of parameter *batchtime* under different datasets, where A equals the performance when applying memory auto-tuning optimization. The results under each dataset have been normalized by the best one.

including A-Bruijn assembly graph [13] and de Bruijn graph. Among them, de Bruijn graph (DBG) is the most commonly used method for genome assembling. Among recent implementations on a single machine, IDBA-UD [19] algorithm can reconstruct long contigs with higher accuracy through multi-thread parallelism, which applies progressive relative depth, local assembly and error correction. In addition, Canu [14] is developed for scalable and accurate long-read assembly. It implements adaptive k-mer weighting and repetitive separation methods, and parallelizes the overlap computation into multiple jobs and merges these results with parallel bucket sort algorithm. On the other hand, an open-source *de novo* genome assembly, MECAT [25], combines fast mapping, error correction and de novo assembly. MECAT indices reads with hash tables and accelerates the computation by sampling with sliding window and thus reduces the number of searched k-mers from the degree of sampling number.

Recently, research works are proposed to scale genome assembly to multiple nodes. Pan *et al.* [17] develop distributed memory parallel hash tables for DNA k-mer counting and evaluate their methods on 4,096 cores of the NERSC Cori supercomputer. In addition, Pakman [10] is one of the most recent parallel implementations, which enables large-scale genome assembly with distributed memory parallelism up to 8K cores. In addition to parallel I/O and load-balanced counting of k-mers, Pakman proposed a new type of graph named PakGraph for better parallelism. Other optimizations such as dynamic memory allocation and runtime power control [20, 21] can also be applied to further accelerate genome assembly.

Moreover, there are research works focusing on accelerating the genome sequencing or assembly on modern GPU. Nvidia provided its own CUDA library NVBIO [18] for sequence analysis with high throughput. To outperform NVBIO, Ahmed *et al.* proposed specially designed APIs (GASAL [7]) to provide GPU accelerated kernels for local, global and semi-global alignment routines, which achieved notable speedup. In addition, CUDASW++ 3.0 [15] accelerates Smith-Waterman algorithm by the use of CPU and GPU SIMD operations as well as the collaborated processing on CPU and GPU. However, the above GPU acceleration methods are not applicable for wtdbg2 due to its implementation with hash list and varying read length.

## 7 Conclusion

In this paper, we analyze the hotspots of wtdbg2, and identify that sequence alignment and output procedure are the two major performance bottlenecks. In addition, we also reduce the prohibitive memory usage of the original wtdbg2. Specifically, we propose a memory auto-tuning scheme to satisfy the memory demand when running wtdbg2 with large datasets on commodity servers. We also propose sequence alignment optimization to improve the multi-threading efficiency and load balance. Moreover, we apply output optimization to eliminate the redundant intermediate alignment results to further improve the performance of wtdbg2. The experiment results demonstrate that the optimized wtdbg2 achieves a maximum speedup of 2.31× and an average speedup of 1.54×. In addition, our optimization reduces the memory usage by 39.5% without affecting the correctness.

## Acknowledgment

This work is supported by National Key Research and Development Program of China (Grant No.2016YFB1000304), National Natural Science Foundation of China (Grant No. 61502019), and the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing (Grant No. 2019A12). Hailong Yang is the corresponding author.

## References

1. C.elegans genome dataset (2019), `http://datasets.pacb.com.s3.amazonaws.com/2014/c_elegans`
2. D.melanogaster iso1 genome dataset (2019), `https://www.ebi.ac.uk/ena/data/view/SRR6702603`
3. Human hg00733 genome dataset (2019), `https://www.ebi.ac.uk/ena/data/view/SRR7615963`
4. Human na24385 genome dataset (2019), `https://ftp-trace.ncbi.nlm.nih.gov/giab/ftp/data/AshkenazimTrio/HG002_NA24385_son/PacBio_CCS_15kb`
5. wtdbg2 (2019), `https://github.com/ruanjue/wtdbg2`
6. Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: Hpctoolkit: Tools for performance analysis of optimized parallel programs. Concurrency and Computation: Practice and Experience **22**(6), 685–701 (2010)
7. Ahmed, N., Mushtaq, H., Bertels, K., Al-Ars, Z.: Gpu accelerated api for alignment of genomics sequencing data. In: 2017 IEEE International Conference on Bioinformatics and Biomedicine (BIBM). pp. 510–515. IEEE (2017)

8. Berlin, K., Koren, S., Chin, C.S., Drake, J.P., Landolin, J.M., Phillippy, A.M.: Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. Nature biotechnology **33**(6), 623 (2015)
9. Denisov, G., Walenz, B., Halpern, A.L., Miller, J., Axelrod, N., Levy, S., Sutton, G.: Consensus generation and variant detection by celera assembler. Bioinformatics **24**(8), 1035–1040 (2008)
10. Ghosh, P., Krishnamoorthy, S.: Pakman: Scalable assembly of large genomes on distributed memory machines. In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS) (2019)
11. Gurevich, A., Saveliev, V., Vyahhi, N., Tesler, G.: Quast: quality assessment tool for genome assemblies. Bioinformatics **29**(8), 1072–1075 (2013)
12. Jiang, T., Fu, Y., Liu, B., Wang, Y.: Long-read based novel sequence insertion detection with rcanid. IEEE Transactions on NanoBioscience (2019)
13. Kolmogorov, M., Yuan, J., Lin, Y., Pevzner, P.A.: Assembly of long, error-prone reads using repeat graphs. Nature biotechnology **37**(5), 540 (2019)
14. Koren, S., Walenz, B.P., Berlin, K., Miller, J.R., Bergman, N.H., Phillippy, A.M.: Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. Genome research **27**(5), 722–736 (2017)
15. Liu, Y., Wirawan, A., Schmidt, B.: Cudasw++ 3.0: accelerating smith-waterman protein database search by coupling cpu and gpu simd instructions. BMC bioinformatics **14**(1), 117 (2013)
16. Nichols, B., Buttlar, D., Farrell, J., Farrell, J.: Pthreads programming: A POSIX standard for better multiprocessing. " O'Reilly Media, Inc." (1996)
17. Pan, T.C., Misra, S., Aluru, S.: Optimizing high performance distributed memory parallel hash tables for dna k-mer counting. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 135–147. IEEE (2018)
18. Pantaleoni, J., Subtil, N.: Nvbio: A library of reusable components designed by nvidia corporation to accelerate bioinformatics applications using cuda. URL http://nvlabs. github. io/nvbio (2014)
19. Peng, Y., Leung, H.C., Yiu, S.M., Chin, F.Y.: Idba-ud: a de novo assembler for single-cell and metagenomic sequencing data with highly uneven depth. Bioinformatics **28**(11), 1420–1428 (2012)
20. Qiu, M., Chen, Z., Niu, J., Zong, Z., Quan, G., Qin, X., Yang, L.T.: Data allocation for hybrid memory with genetic algorithm. IEEE Transactions on Emerging Topics in Computing **3**(4), 544–555 (2015)
21. Qiu, M., Ming, Z., Li, J., Liu, S., Wang, B., Lu, Z.: Three-phase time-aware energy minimization with dvfs and unrolling for chip multiprocessors. Journal of Systems Architecture **58**(10), 439–445 (2012)
22. Ruan, J., Li, H.: Fast and accurate long-read assembly with wtdbg2. Nature Methods **17**(2), 155–158 (2020)
23. Simpson, J.T., Durbin, R.: Efficient de novo assembly of large genomes using compressed data structures. Genome research **22**(3), 549–556 (2012)
24. Wenger, A.M., Peluso, P., Rowell, W.J., Chang, P.C., Hall, R.J., Concepcion, G.T., Ebler, J., Fungtammasan, A., Kolesnikov, A., Olson, N.D., et al.: Highly-accurate long-read sequencing improves variant detection and assembly of a human genome. bioRxiv p. 519025 (2019)
25. Xiao, C.L., Chen, Y., Xie, S.Q., Chen, K.N., Wang, Y., Han, Y., Luo, F., Xie, Z.: Mecat: fast mapping, error correction, and de novo assembly for single-molecule sequencing reads. nature methods **14**(11), 1072 (2017)