GPU-Accelerated Bidirected De Bruijn Graph Construction for Genome Assembly

Mian Lu[1], Qiong Luo[2], Bingqiang Wang[3], Junkai Wu[2], and Jiuxin Zhao[2]

[1] A*STAR Institute of High Performance Computing, Singapore
lum@ihpc.a-star.edu.sg
[2] Hong Kong University of Science and Technology
{luo,jwuac,zhaojx}@cse.ust.hk
[3] BGI-Shenzhen, China
wangbingqiang@genomics.org.cn

Abstract. De Bruijn graph construction is a basic component in de novo genome assembly for short reads generated from the second-generation sequencing machines. As this component processes a large amount of data and performs intensive computation, we propose to use the GPU (Graphics Processing Unit) for acceleration. Specifically, we propose a staged algorithm to utilize the GPU for computation over large data sets that do not fit into the GPU memory. We also pipeline the I/O, GPU, and CPU processing to further improve the overall performance. Our preliminary results show that our GPU-accelerated graph construction on an NVIDIA S1070 server achieves a speedup of around two times over previous performance results on a 1024-node IBM Blue Gene/L.

1 Introduction

Genome assembly refers to the process of reconstructing a genome sequence from a large number of sequence fragments (known as *reads*). These reads are generated by sequencing machines through randomly sampling the original sequence. The De Bruijn graph based genome assembly algorithms have been shown effective for assembling a large number of short reads and have been adopted in state-of-the-art assemblers [1–4]. In this paper, we focus on the bidirected De Bruijn graph construction, which is the first as well as one of the most expensive steps in genome assembly. We propose to utilize the GPU (Graphics Processing Units) to accelerate this process, in particular, the construction of a bidirected De Bruijn graph from a large set of short reads.

The bidirected De Bruijn graph construction is expensive in both memory consumption and running time. A previous study [2] has shown that the graph construction for human genome took around 8 hours on a 16-core machine with 2.3GHz AMD quadcore CPUs and consumed 140GB main memory. With limited GPU memory (up to 6 GB per GPU in the market), the first challenge in GPU-based De Bruijin graph construction is to develop algorithms that can handle data larger than the GPU memory. Second, given the superb computation power of the GPU, the overall performance is likely to be dominated by disk

Y. Ishikawa et al. (Eds.): APWeb 2013, LNCS 7808, pp. 51–62, 2013.

I/O and CPU processing. Therefore, we study how to utilize the hierarchy of disk, main memory, and GPU memory to pipeline processing and to involve the CPU for co-processing. These issues are essential for the feasibility and overall performance on practical applications; unfortunately, they are seldom studied in current GPGPU research.

Specifically, we address the GPU memory limit by developing a staged algorithm for GPU-based graph construction. We divide the reads into chunks and load the data chunk by chunk from disk through the main memory to the GPU memory. To estimate the chunk size that can fit into the GPU memory, we develop a memory cost model for each processing step. We further utilize the CPU and main memory to perform result merging each time the GPU finishes processing a chunk. Finally, we pipeline the disk I/O, GPU processing, and CPU merging (*DGC*) to improve the overall performance. We expect this *pipelined-DGC* processing model to be useful for a wide range of GPGPU applications that handle large data.

We have implemented the GPU-based bidirected De Bruijn graph construction and evaluated it on an NVIDIA Tesla S1070 GPU device with 4 GB memory. Our initial results show that this implementation doubles the performance reported on a 1024-node IBM Blue Gene/L and is orders of magnitude faster than state-of-the-art CPU-based sequential implementations.

The remainder of this paper is organized as follows. In Section 2, we briefly introduce the graph construction algorithm and related work. We present our design of the staged algorithm in Section 3. We describe the details of the pipelined-DGC model in Section 4. The experimental results are reported in Section 5. We conclude in Section 6.

2 Preliminary

2.1 Genome Assembly

The second generation of sequencing produces very short reads at a high throughput. Popular algorithms to reconstruct the original sequence for such a large number of short reads are based on the *De Bruijn [5]* or *bidirected De Bruijn* graph [6]. The graph is constructed through generating *k-mers* from reads as graph nodes. For example, suppose a short read is ACCTGC and $k = 4$, then this read can generate three 4-mers, which are ACCT, CCTG, and CTGC. The major difference between the two graph models is that for each *k-mer*, its reverse complement is represented by a separate node (De Bruijn) or the same node (bidirected De Bruijn).

At the beginning of assembly, each l-length read generates $(l - k + 1)$ *k-mers*. Then the De Bruijn graph is constructed using information about overlap between *k-mers*. Next, the graph is simplified and corrected by some heuristic algorithms. After the simplification and correction, several long *contigs* are generated. Finally, if reads are generated through paired-end sequencing, these *contigs* are joined to produce *scaffolds* using relevant information. Another alternative of joining contigs is through Eulerian paths. The detailed algorithms

of these steps can be found in previous work [2, 4]. In this work, we focus on the De Bruijn graph construction.

2.2 Bidirected De Bruijn Graph

The double-stranded structure of DNA sequences can be naturally mapped to the bidirected De Bruijn graph [6]. For a birdirected graph, each edge has independent directions at two ends. A valid path from node v_1 to v_k is represented as a sequence $v_1e_1v_2e_2...e_{k-1}v_k$, where e_i is the edge connecting two node v_i and v_{i+1}, and e_{i-1} and e_i have different directions at the node v_i for $1 < i < k$. Figure 1 shows an example of bidirected graph. A De Bruijn graph is a directed graph, where each node represents an ordered sequence, and all sequences have the same length. An edge from node A representing $a_1a_2...a_n$ to node B representing $b_1b_2...b_n$ exists when $a_2a_3...a_n$ is identical to $b_1b_2...b_{n-1}$.

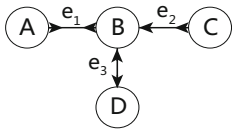


Fig. 1. An example of bidirected graph

For genome assembly, each node in a bidirected De Bruijn graph represents a *k-mer* containing k bases. Given the double-stranded structure of DNA sequences, each node also implicitly represents its reverse complementary *k-mer*, i.e., the base on each position in one sequence is the complement (A to T, C to G) of another sequence in the reverse order. For a given node v representing two reverse complementary *k-mers*, we denote the lexicographically greater one as v^+ to be the canonical form of these two *k-mers* as well as the node representative, and v^- the reverse complement of v^+. Note that, when k is an odd number, v^+ and v^- will never be identical.

Suppose an edge exists between nodes A and B. There are four cases when A overlaps B.

- Case 1. A^+ overlaps $B^+ \Rightarrow$ A $\triangleright\!-\!\triangleright$B
- Case 2. A^- overlaps $B^- \Rightarrow$ A $\triangleleft\!-\!\triangleleft$B
- Case 3. A^- overlaps $B^+ \Rightarrow$ A $\triangleleft\!-\!\triangleright$B
- Case 4. A^+ overlaps $B^- \Rightarrow$ A $\triangleright\!-\!\triangleleft$B

Note that duplicate edges are expected due to the high coverage of input reads. However, only distinct edges and their frequency counts (known as *multiplicities*) need to be maintained. The multiplicity is useful when removing errors from graphs. After constructing all distinct edges and recording their multiplicities, we generate an ordered pair $<c_X, c_Y>$ corresponding to an edge representing X overlaps Y. c_X and c_Y are the label to output when walking from node X to Y and from node Y to X, respectively. The label is generated as follows corresponding to the four cases of overlap. $X^+[1]$ and $X^-[1]$ denote the first character of the canonical form and its reverse complement of node X, respectively.

- Case 1. A^+ overlaps $B^+ \Rightarrow <A^+[1], B^-[1]>$
- Case 2. A^- overlaps $B^- \Rightarrow <A^-[1], B^+[1]>$
- Case 3. A^- overlaps $B^+ \Rightarrow <A^-[1], B^-[1]>$
- Case 4. A^+ overlaps $B^- \Rightarrow <A^+[1], B^+[1]>$

Distinct canonical *k-mers* are collected as representatives for graph nodes, and adjacency information is built for the graph representation for further manipulation.

2.3 Related Work

There are several genome assembly tools for short reads on the CPU. Velvet [4] is a pioneer software for short reads assembly, which is still one of the most popular short reads assemblers today. SOAPdenovo [2] has been used successfully for human genome projects. There are other similar assemblers, such as Euler-SR [7], Shorty [8], Edena [9], ALLPATHS [10], SSAKE [11], SHARCGS [12], ABySS [3], and YAGA [1]. These assemblers are different in implementation details, but are all based on the (bidirected) De Bruijn graph model.

Among parallel short reads assemblers, ABySS [3] was implemented using MPI. SOAPdenovo [2] has parallelized several key time-consuming steps. Jackson et al. [13] have reported their bidirected De Bruijn graph construction on an IBM Blue Gene/L. Later, the authors implemented the complete genome assembly procedure on the Blue Gene/L machine and reported the performance scalability [1]. Kundeti et al. [14] have further improved the parallel graph construction algorithm presented by Jackson et al. [13] to avoid a large amount of message passing. However, to the best of our knowledge, the only work on GPU-accelerated genome assembly is GPU-Euler [15], which implements a similar construction algorithm. The major issue of their work is that it cannot handle large genome data, e.g., human genome, which cannot fit into the GPU memory.

3 Design and Implementation

In this section, we first present the in-memory implementation of GPU-based bidirected De Bruijn graph construction assuming sufficient memory. Then, we propose a staged algorithm for out-of-core processing.

3.1 In-Memory Implementation

We consider the graph construction as a component in de novo assembly, where the program input is a plain text file storing short reads and the output is the adjacency list representation of the bidirected De Bruijn graph stored in the main memory. Overall, there are three steps for the graph construction.

Step 1. Encoding. This step loads short reads from the disk to the main memory and uses two bits per character to represent the reads.

Step 2. Canonical Edge Generation. We adopt an edge-oriented building method to avoid validating edges [1]. For an l-length short read, we generate *(k+1)-mers*, and the number of *(k+1)-mers* is $(l - k)$. Each *(k+1)-mer* corresponds to an edge connecting two overlapping *k-mers*. The two *k-mers* represent two nodes connected by the edge. We do not extract the two *k-mers* in this step since there are many redundant edges and nodes at this time. A 64-bit data type, such as *long*, is sufficient to hold a *(k+1)-mer* since $(k+1)$ is usually set to less than 32 in practice. Similarly, we do not extract edge directions and output labels in this step due to the redundancy.

In the GPU-based implementation, generating edges is done through a map primitive efficiently. Each thread takes charge of one encoded read and generates corresponding $(l - k)$ edges. In our implementation, the encoding and edge generation are done in one kernel program. We use the fast *shared memory* on the GPU to hold encoded reads, and bitwise operations to generate all *(k+1)-mers*. Duplicate elimination can be implemented through either sorting, or hashing. In our evaluation, the two methods have a similar performance. Considering the merging step in out-of-core processing, we have adopted the sorting-based approach to perform the duplicate elimination.

Step 3. Adjacency List Representation After edges are generated, we can extract two corresponding *k-mers*, edge directions, and output labels from each *(k+1)-mer*. This step can be implemented as a map on GPUs. To keep the adjacency information, we use a pair (*k-mer*, *edge_id*) to represent a node. An additional duplicate elimination step on the field *k-mer* is performed for all pairs to obtain distinct nodes as well as associated edges.

3.2 The Staged Graph Construction Algorithm

The in-memory implementation assumes unlimited GPU memory for the graph construction. In practice, a staged graph construction algorithm is necessary for the GPU-based implementation to handle data that cannot fit into the GPU memory. In common cases, the most memory consumption occurs in Step 2, when edges are generated from encoded reads before the duplicates are eliminated. Suppose there are n short reads of length l each, for a user-defined k, the total size of *(k+1)-mers* is $8 \times n \times (l - k)$ bytes, e.g., several hundreds of gigabytes for the human genome. Step 3 is executed using a similar staged algorithm when all distinct edges are produced. Therefore, we focus on introducing the staged algorithm for Step 1 and 2.

The basic idea in our staged algorithm is to load and process the input reads through multiple passes. In each pass, only a subset of input reads, denoted as a *chunk* of reads, is loaded from the disk to the memory for processing. Figure 2 shows the workflow of chunk-based processing. The chunk size is set so

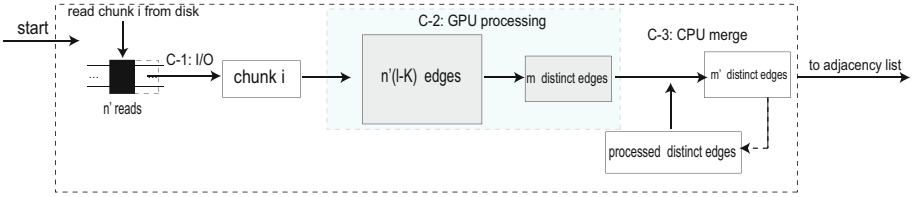


Fig. 2. The workflow of one chunk based processing of GPU-accelerated bidirected De Bruijn graph construction. The color coding of black, grey, and white indicates the data is stored in the disk, GPU memory, and CPU main memory, respectively

that the memory required for the chunk-based processing does not exceed the GPU memory size. Specifically, there are three processing steps for one chunk:

1. **C-1: I/O.** n' ($n' \leq n$) short reads are loaded from the disk to the main memory as a chunk of data.
2. **C-2: GPU processing.** We transfer the data chunk to the GPU memory. The GPU performs the encoding and generates $n' \times (l-k)$ edges. Duplicates are eliminated through a sorting-based method for these $n' \times (l-k)$ edges. After the duplicate elimination, there are m distinct edges for this chunk.
3. **C-3: CPU merge.** These m distinct edges are copied from the GPU memory to the main memory, and merged with the distinct edges generated from previously processed data chunks. Since both the newly generated edges and the existing edges are ordered, the merge step is efficient. The m' distinct edges after merging will be used for the next chunk. The multiplicity information are also updated in this step. Due to the high coverage of input reads, the number of distinct edges is around tens of times smaller than that of all generated edges. Therefore we assume that the main memory is sufficient to hold these distinct edges.

3.3 The Memory Cost Model

Given the GPU memory size M_g we estimate the memory consumption for each step to decide the suitable chunk size. Suppose each chunk contains n' reads, and each read is l-length. Then the memory size of input reads is $n' \times l$ bytes (one byte per character). After encoding, the memory size for encoded reads is $\frac{n' \times l}{4}$ bytes. Thus the total GPU memory consumption for this encoding step is: $(n' \times l + \frac{n' \times l}{4})$ bytes. Next we use encoded reads in the memory for generating edges. The number of edges generated for one chunk is $n' \times (l-k)$, and each edge is represented using a 64-bit word type. Thus the memory size required for all generated *(k+1)-mers* in one chunk is $8 \times n' \times (l-k)$ bytes. Therefore the total GPU memory consumption of this edge generation step for one chunk is: $\left(\frac{n' \times l}{4} + 8 \times n' \times (l-k)\right)$ bytes. We perform the sorting-based duplicate elimination algorithm on $n' \times (l-k)$ edges. The memory consumption of the GPU-based radix sort is an output buffer with the same size of the input array. Thus the total memory consumption for the radix sort can be estimated as $(2 \times 8 \times n' \times (l-k))$

bytes. Finally, we remove duplicates from these sorted edges. The memory consumption for this operation is at most $(2 \times 8 \times n' \times (l-k))$ bytes. Therefore, the maximum value of n' can be estimated as follows:

$$min\left(\frac{4M_g}{5l}, \frac{M_g}{8.25l-8k}, \frac{M_g}{16\times(l-k)}\right)$$

4 The Pipelined Processing Model

4.1 The DGC Model

Our staged algorithm can be generalized as a processing model, which can be adopted in many GPGPU applications to handle the data set that cannot entirely fit into the GPU memory or of which some steps are not suitable to be processed on the GPU. We call it the DGC (Disk-GPU-CPU) model.

The DGC model is defined as follows. There are three components: I/O, the computation on GPUs, and the processing on CPUs. Suppose the total size of input data is D, the chunk size d, there are $\lceil \frac{D}{d} \rceil$ passes. For each pass: (1) A chunk of data is loaded from the disk to the main memory. (2) The in-memory chunk is transferred from the main memory to the GPU memory, and processed by the GPU. (3) The result is transferred back to the main memory, and a merge (or other post-processing) step is performed on the CPU.

In the DGC model, at least three memory buffers are required: b_i is the CPU memory buffer to hold the data from the disk, b_g is the GPU memory buffer used to store the input data, and b_c is the CPU memory buffer used to hold the computation result transferred from the GPU memory. To simplify the presentation, we assume the computation result on the GPU is also stored in the input buffer, that is b_g.

4.2 The Pipelined-DGC Model

The DGC model can be improved using pipelining. Since b_g is the GPU memory buffer independently accessed from b_i and b_c, the DGC model can be pipelined without additional memory allocated for data exchange. Without loss of generality, we assume that during processing, b_g and b_c will not be released by the GPU and CPU programs, respectively.

We maintain three threads to independently take charge of the I/O, GPU, and CPU processing. The I/O thread will be blocked when b_i is full and also when the memory copy to b_g is ongoing. The GPU thread may be blocked in two cases. First, the data is not ready, i.e., b_g is not full. Second, the processing on the GPU has been done, but the data is being copied to b_c. The CPU thread may be blocked only when the data is not ready in b_c.

At the beginning, we load the first data chunk to b_i. When b_i is full, a memory copy from b_i to b_g is executed. The I/O is blocked when performing the memory copy. As long as the data has been uploaded to b_g, b_i becomes available to load the next chunk of data until it is full again. However, the second memory

copy from b_i to b_g (and all following memory copies) should wait until the GPU processing for the previous chunk is done to release the use of b_g. Similarly, we pipeline the GPU and CPU processing. A memory copy from the GPU to CPU is performed when the computation on the GPU is finished. The GPU processing is blocked when coping the data to b_c. After the memory copy, the CPU can perform the merge step for the data in b_c, and the the next chunk of data can be copied from b_i to b_g if the data is ready in b_i, otherwise the GPU thread is blocked to wait for the data in b_i becoming ready. We can see that with pipelining, the I/O, GPU processing, and CPU merge can overlap in time.

5 Experiments

5.1 Experimental Setup

Hardware Configuration. We conduct our experiments on a server machine with two Intel Xeon E5520 CPUs (16 threads in total) and an NVIDIA Tesla S1070 GPU. The NVIDIA Tesla S1070 has four GPU devices. In our current implementation, we only utilize one GPU device, and the CPU merge step also only uses one core on the CPU. This computation capability is sufficient for the overall performance as the I/O is the bottleneck. The main memory size of the server is 16 GB.

Data Sets. We use a small and a medium sized data set for the evaluation. As the previous research [13], we randomly sample known chromosomes to simulate short reads. Two data sets are sampled from Arabidopsis chromosome 1 (denoted as *Arab.*) and human chromosome 1 (denoted as *Human*), which can be accessed from the NCBI genome databases [16]. The details of the two data sets are shown in Table 1. Additionally, we fix the *k-mer* length k to 21, which is a common value for most genome assemblers.

Table 1. Data sets

	Arabidopsis chromosome 1	Human chromosome 1
#nucleotide	30 million	247 million
Read length	36	36
Coverage	17	36
#read	12 million	230 million
File size (FASTA)	800 MB	13 GB

5.2 Performance Results

Time Breakdown without Pipelining. We first study the performance bottleneck of our GPU-based graph construction without the pipeline optimization. Figure 3 shows the time breakdown of our GPU-accelerated bidirected De Bruijn graph construction. In this figure, the I/O, the encoding and edge generation on the GPU, the duplicate elimination for each chunk on the GPU, and the merge

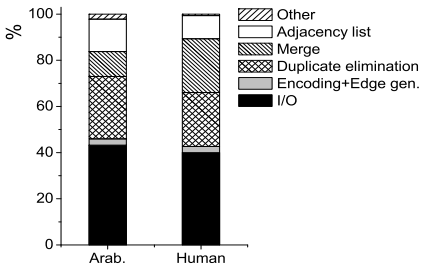
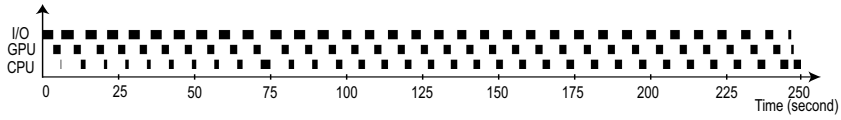
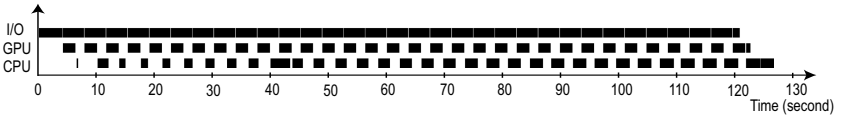


Fig. 3. The time breakdown of GPU-based bidirected De Bruijn graph construction without pipelining

step on the CPU together take around 90% of the execution time. The DGC model is applied to these four components. Particularly, among these four components, I/O takes around 50%, and both the GPU processing (including encoding and generating edges) and CPU processing take around 25% of the elapsed time. Since the efficiency of the pipeline is limited by the most time-consuming component, we expect that the overall performance of these four components can be improved by around two times through pipelining.



(a) Without pipelining



(b) With pipelining

Fig. 4. Executions of the staged algorithm without and with pipelining for GPU-based bidirected De Bruijn graph construction. The solid black rectangles represent the busy time of a component.

The Pipelining Optimization. We focus on the four steps (I/O, Encoding+Edge gen., Duplicate elimination, Merge) which as shown in Figure 3 that consume around 90% of the overall elapsed time. We take the larger data set human chromosome 1 to demonstrate the result. There are 33 passes in the staged algorithm for this medium size data set. Figure 4(a) shows the execution of the staged algorithm without pipelining. The I/O, GPU and CPU processing are done sequentially in each pass. As a result, the disk is idle around half of the overall time, and the utilization of the GPU and CPU is only around 25%. Figure 4(b) shows the execution status of the staged algorithm when pipelining optimization is adopted. With pipelining, I/O is nearly non-blocking.

Even though the GPU and CPU processing cannot completely overlap, their utilization is improved considerably. As a result, the overall execution time for these four steps are reduced from around 250 seconds to 128 seconds.

Comparison to Existing Implementations. There are four CPU-based implementations for comparison: Velvet [4], SOAPdenovo [2], ParBidirected [14], and Jackson's implementation [13]. We only compare the performance of graph construction. We perform the evaluation on the same machine for all these software except the Jackson's implementation. Note that SOAPdenovo, ParBidirected, and Jackson's implementation are parallel implementations.

Table 2. Comparison of the elapsed time in seconds

	Arab.	**Human 1**
GPU-accelerated	18	177
Velvet [4]	86	-
SOAPdenovo [2]	78	1,245
ParBidirected [14]	1,740	32,400
Jackson2008 [13]	15	327

Table 2 shows the comparison result for the running time of four different implementations. The memory required by preprocessing in Velvet for the Human data set exceeds our main memory limit and takes excessively long time, thus we do not report the performance number for the Human data set in the table. SOAPdenovo has parallelized the hash table building. In the evaluation, it consumes a similar main memory size (around 8 GB for the Human data set) to our implementation. However, the GPU-accelerated graph construction is around 4-7x faster than SOAPdenovo for the similar functionality. Since ParBidirected adopts a more conservative method to handle the out-of-core processing, intermediate results need to be written into the disk in most steps, which results in a slow execution time, but the memory consumption is stable and very low. In our experiments, we have already modified the default buffer size used in the external sorting to improve the performance. The published performance results from Jackson's implementation [13] is based on a 1024-node IBM Blue Gene/L. The input and output are the same as our program. However, they have adopted a node-oriented graph building approach, and the message passing is very expensive. In summary, compared with existing implementations, our GPU-based graph construction is significantly faster. Specifically, compared with the massively parallel implementation on the IBM Blue Gene/L, our implementation is still around two times faster.

6 Conclusion

In this paper, we have presented the design and implementation of our GPU-accelerated bidirected De Bruijn graph construction, which is a first step to build

a complete GPU-accelerated genome assembler. We have addressed the GPU memory limit issue through a staged algorithm, and the overall performance can be further improved by around two times through pipelining I/O, GPU and CPU processing. Furthermore, such optimized processing flow can be generalized as a pipelined-DGC model, which can be applied to other GPGPU applications to handle large data sets. Compared with existing implementations on CPUs, the performance of our implementation is up to two orders of magnitude faster. In particular, our GPU-accelerated graph construction is around 2X faster than the published performance numbers for a parallel implementation on a 1024-node IBM Blue Gene/L.

Acknowledgement. This work was supported by grants 617509 and 616012 from the Research Grants Council of Hong Kong, and grant MRA11EG01 from Microsoft China.

References

1. Jackson, B., Regennitter, M., Yang, X., Schnable, P., Aluru, S.: Parallel de novo assembly of large genomes from high-throughput short reads. In: IPDPS 2010: Proceedings of the 2010 IEEE International Symposium on Parallel&Distributed Processing, pp. 1–10 (April 2010)
2. Li, R., Zhu, H., Ruan, J., Qian, W., Fang, X., Shi, Z., Li, Y., Li, S., Shan, G., Kristiansen, K., Li, S., Yang, H., Wang, J., Wang, J.: De novo assembly of human genomes with massively parallel short read sequencing. Genome Research 20(2), 265–272 (2010)
3. Simpson, J.T., Wong, K., Jackman, S.D., Schein, J.E., Jones, S.J., Birol, I.: Abyss: a parallel assembler for short read sequence data. Genome Research 19(6), 1117–1123 (2009)
4. Zerbino, D.R., Birney, E.: Velvet: algorithms for de novo short read assembly using de bruijn graphs. Genome Research 18(5), 821–829 (2008)
5. Pevzner, P.A., Tang, H.: Fragment assembly with double-barreled data. Bioinformatics 17(suppl. 1), S225–S233 (2001)
6. Medvedev, P., Georgiou, K., Myers, G., Brudno, M.: Computability of models for sequence assembly. In: Giancarlo, R., Hannenhalli, S. (eds.) WABI 2007. LNCS (LNBI), vol. 4645, pp. 289–301. Springer, Heidelberg (2007)
7. Chaisson, M.J., Pevzner, P.A.: Short read fragment assembly of bacterial genomes. Genome Research 18(2), 324–330 (2008)
8. Hossain, M.S.S., Azimi, N., Skiena, S.: Crystallizing short-read assemblies around seeds. BMC Bioinformatics 10(suppl. 1) (2009)
9. Hernandez, D., François, P., Farinelli, L., Østerås, M., Schrenzel, J.: De novo bacterial genome sequencing: Millions of very short reads assembled on a desktop computer. Genome Research 18(5), 802–809 (2008)
10. Butler, J., MacCallum, I., Kleber, M., Shlyakhter, I.A., Belmonte, M.K., Lander, E.S., Nusbaum, C., Jaffe, D.B.: Allpaths: De novo assembly of whole-genome shotgun microreads. Genome Research 18(5), 810–820 (2008)
11. Warren, R.L., Sutton, G.G., Jones, S.J., Holt, R.A.: Assembling millions of short dna sequences using ssake. Bioinformatics 23(4), 500–501 (2007)

12. Dohm, J.C., Lottaz, C., Borodina, T., Himmelbauer, H.: Sharcgs, a fast and highly accurate short-read assembly algorithm for de novo genomic sequencing. Genome Research 17(11), 1697–1706 (2007)
13. Jackson, B.G., Aluru, S.: Parallel construction of bidirected string graphs for genome assembly. In: International Conference on Parallel Processing, pp. 346–353 (2008)
14. Kundeti, V., Rajasekaran, S., Dinh, H.: Efficient parallel and out of core algorithms for constructing large bi-directed de bruijn graphs. CoRR abs/1003.1940 (2010)
15. Mahmood, S.F., Rangwala, H.: Gpu-euler: Sequence assembly using gpgpu. In: Proceedings of the 2011 IEEE International Conference on High Performance Computing and Communications, HPCC 2011, pp. 153–160. IEEE Computer Society (2011)
16. National Center for Biotechnology Information, `http://www.ncbi.nlm.nih.gov/`