*Bioinformatics*, 2016, 0–0

doi: 10.1093/bioinformatics/xxxxx

Advance Access Publication Date: DD Month YYYY

Manuscript Category

|  |
| --- |
| Genome Analysis  Omega 3: First Parallel OLC Metagenome Assembler  Corresponding Author1,\*, Co-author2 and Co-Author2  1Department of XXXXXXX, Address XXXX etc., 2Department of XXXXXXX, Address XXXX etc.  \*To whom correspondence should be addressed.  Associate Editor: XXXXXXX  Received on XXXXX; revised on XXXXX; accepted on XXXXX  Abstract  **Motivation:** The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.  **Results:** The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.  **Availability:** The quick brown fox jumps over the lazy dog.  **Contact:** example@example.org  **Supplementary information:** Supplementary data are available at *Bioinformatics* online. |

# Introduction

The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog..

The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.

# Methods

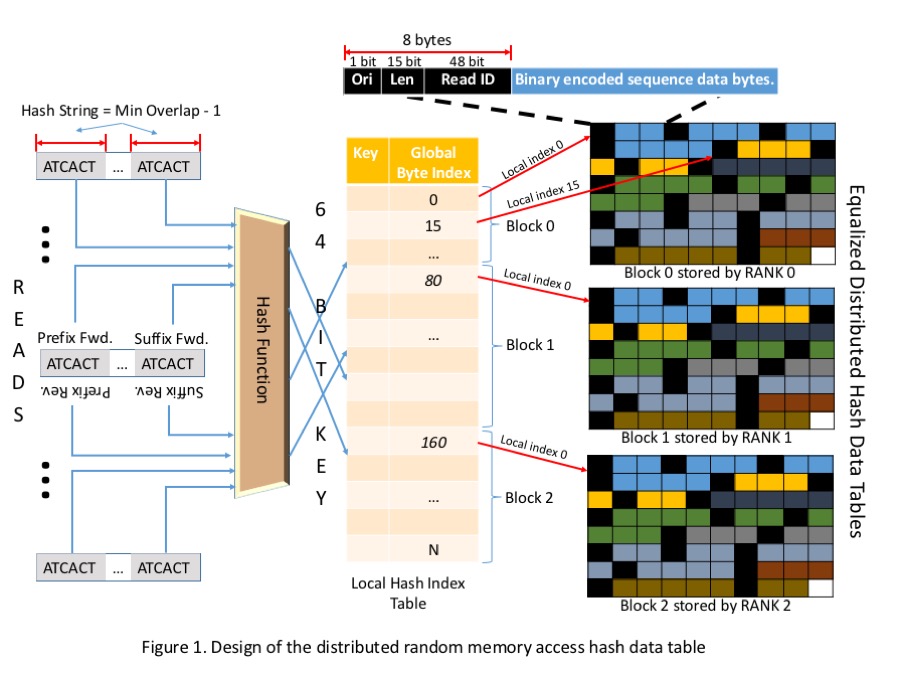
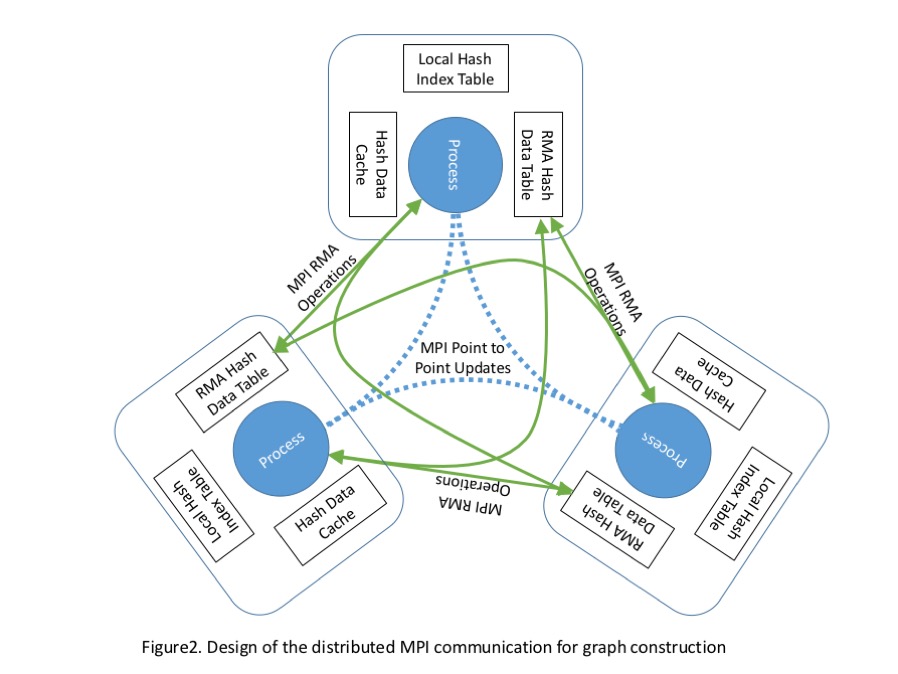
In this section we describe several novel optimizations developed to improve the computational performance and memory utilization of overlap graph construction and simplification algorithms. We have developed 3 version of Omega for three distinct usage patterns. Briefly they are:

1. OpenMP version: This version uses only openmp multithreading to speedup computation and runs on a single machine with multiple CPUs. The RAM usage of the version is approximately equal to the disk space size of the uncompressed reads. This version is best suited for small to medium size metagenomes as the memory requirement is the major limiting factor.
2. MPI HighMem version: This version uses multiprocess and multithreaded parallelism to improve execution time of the OpenMP version. Traditional point-to-point MPI communication is used to synchronize graph construction. The RAM usage of each MPI process is approximately equal to the disk space size of the uncompressed reads. This version is also best suited for very fast assembly of small to medium size metagenomes as the memory requirement is the major limiting factor.
3. MPI RMA version: This version uses multiprocess parallelism to reduce the memory usage of the individual processes. The read data storage is distributed equally across all the processes and random memory access feature of MPI-3 is used to for querying data from other processes. Traditional point-to-point MPI communication is still used to synchronize graph construction. The RAM usage of each MPI process can be significantly reduced by almost 70% provided enough compute nodes are available. This version is also best suited for very assembly of large metagenomes as the memory requirement is generally not a limiting factor.

We designed a distributed hash table structure to efficiently store and access data for overlap computation. We combined MPI remote memory operations and point-to-point communication for synchronizing graph construction and simplification. In the following section we will describe the design of out MPI RMA version and point out major differences from the other two versions as appropriate.

## Omega Algorithm

Omega can accept multiple input datasets with different insert sizes and variable read lengths in fasta or fastq format. The assembly and scaffolding are performed in eight steps :

1. **Hash table construction.** All unique reads are loaded to the memory and indexed in a hash table. Let *K* be the user-defined minimum overlap length. The keys of the hash table are DNA sequence substrings of length *K −*1. Each read is inserted to the hash table with four keys: prefix and suffix of length *K −*1 of both forward sequence and reverse complement sequence of the read. A value in the hash table is an array of pointers to the reads associated with the corresponding key. The hash table is initialized to be eight times of the total read number. Hash collision is resolved using linear probing. The hash table allows a nearly constant time search of all reads by their prefixes or suffixes. A read that is a substring of another read is called a *contained read*. To identify all contained reads of read *r*, every proper substring *s* of length *K* in read *r* is searched in the hash table. This produces a short list of reads that contains *s* as a prefix or suffix, which is then compared with read *r* to identify the contained reads of read *r*. The contained reads are used for coverage depth calculation and mate-pair linkage analysis below.
2. Contained Read Removal:
3. **Overlap graph construction.** Each read is represented by a vertex in a bi-directed overlap graph. An edge is inserted between two vertices if the two corresponding reads have an exact-match overlap of at least *K* bp. The bi-directed edges represent the four different orientations in which two reads can overlap: suffix with prefix (**•→−−→•**), suffix of the reverse complement with prefix (**•←−−→•**), suffix with prefix of the reverse complement (**•→−−←•**) and suffix of the reverse complement with prefix of the reverse complement (**•←−−←•**). To efficiently find all reads overlapping with a read *r*, every proper substring *s* of length *K −*1 in read *r* is searched in the hash table, and all retrieved reads are compared with the read *r*. If a read has the exact match with read *r* for their remaining overlap, an edge is inserted between the two reads’ corresponding vertices. After inserting all edges of read *r*, all transitive edges incident on read *r* are removed using a linear algorithm as described ([Haider, 2012](http://bioinformatics.oxfordjournals.org/content/30/19/2717.long#ref-3); [Myers, 2005](http://bioinformatics.oxfordjournals.org/content/30/19/2717.long#ref-9)). Briefly, suppose that *r* is connected with two other reads, *a* and *b*. If there is also an edge between *a* and *b* to form a triangle with *r* and the sequence represented by the edge (*r, b*) is the same as the sequence represented by the path through (*r, a*) and (*a, b*), then (*r, b*) is identified as a transitive edge and is deleted. Removing all transitive edges significantly simplifies the overlap graph without losing any information.
4. **Composite edge contraction.** While the bi-directed edges can be traversed in both directions, the vertices can be traversed only by entering a vertex in an in-arrow and exiting in an out-arrow (**−→•→−**) or by entering a vertex in an out-arrow and exiting in an in-arrow (**−←•←−**). A valid path in the overlap graph represents an assembled DNA sequence containing proper overlapping reads with appropriate orientation and sufficient overlap length. After removing transitive edges, simple vertices have exactly one in-arrow and one out-arrow, representing only one possible way to traverse such simple vertices. A read in a simple vertex uniquely overlaps with one other read in either direction. To simplify the overlap graph, a simple vertex, *r*, along with its in-arrow edge (*u, r*) and out-arrow edge (*r, w*), are replaced by a composite edge (*u, w*) in the overlap graph. The composite edge (*u, w*) contains the read *r*and all ordered reads in edge (*u, r*) and (*r, w*). The edge (*u, w*) has the same arrow types to *u* and *w* as the original edges, (*u, r*) and (*r, w*), respectively. Simple vertices are merged into composite edges iteratively, until there is no simple vertex remaining in the overlap graph.
5. **Sequence variation removal.** Sequence variations originate from uncorrected sequencing errors and natural sequence polymorphisms in microbial communities. Many reads with sequence variations do not overlap with any other reads and are represented as isolated vertices in the overlap graph. Reads with the same sequence variation may overlap with one another, which creates small branches and bubbles in the overlap graph. Small branches are short dead-end paths that contain <10 reads. Bubbles are two edges that connect the same two vertices with the same arrow types. The overlap graph is systematically traversed to trim off small branches and remove the edges containing less reads in bubbles. This may create new simple vertices that are then removed by repeating the composite edge contraction.
6. **Minimum cost flow analysis.** Each edge in the overlap graph is associated with a string copy number, representing how many times the edge’s sequence is present in the metagenome. String copy numbers of edges are estimated based on the topology of the overlap graph using minimum cost flow analysis as described ([Haider, 2012](http://bioinformatics.oxfordjournals.org/content/30/19/2717.long#ref-3); [Myers, 2005](http://bioinformatics.oxfordjournals.org/content/30/19/2717.long#ref-9)). Composite edges with sequences >1000 bp are set to have a minimum flow of 1, requiring such edges’ sequences to be present in the metagenome at least once. The minimum flow for short edges (<1000 bp) is set to 0. The CS2 algorithm ([Goldberg, 1997](http://bioinformatics.oxfordjournals.org/content/30/19/2717.long#ref-2)) is used to optimize the amount of flow passing through every edge such that the total cost of the flow network in the overlap graph is minimized. Edges with more than one unit of flow correspond to repeat regions shared among multiple genomes or multiple places in a single genome. Edges with zero flow represent short sequences that are not needed to connect long sequences together and are ignored. Tree structures in the overlap graph are simplified using the flows. A tree comprises two edges, (*p, t*) and (*q, t*), merging to a third edge (*t, r*), and the flow on (*t, r*) is equal to the total flow on (*p, t*) and (*q, t*). Such a tree is reduced to two new edges (*p, r*) and (*q, r*) that both contain the reads in vertex *t* and edge (*t, r*).
7. **Merging of adjacent edges with mate-pair support.** The insert size of each paired-end dataset is estimated separately to accommodate a mixture of datasets with different insert sizes. The overlap graph at this stage has long composite edges that contain both reads of many mate-pairs. The insert sizes of such pairs are determined from their relative locations on the long edges and are pooled to estimate the mean *μ* and SD *σ* of all mate-pairs’ insert sizes in each dataset. Mate-pairs that span multiple edges are used to merge adjacent edges in the overlap graph. For each of such mate-pairs, all possible paths of length within range (*μ* − 3*σ*, *μ* + 3*σ*) are enumerated. If all paths of a mate-pair travel through two adjacent edges, (*m, r*) and (*r, n*), the connection between these two edges is considered to be supported by this mate-pair. After processing all mate-pairs, if the connection between (*m, r*) and (*r, n*) is supported by more than three mate-pairs, these two edges are merged to one edge (*m, n*) containing a duplicated *r*.
8. **Scaffolding of long edges with mate-pair support.** Scaffolding uses mate-pairs that have no valid path between their paired reads in the overlap graph because of a gap in genome coverage. Scaffolding is attempted for every pair of non-adjacent edges >1000 bp. A mate-pair is considered to support the scaffolding of two edges if its two reads are uniquely mapped to the two edges at an appropriate distance apart. After processing all mate-pairs, the scaffolds of long edges with support of more than three mate-pairs are accepted.
9. **Resolving ambiguity by coverage depth.** Many unresolved vertices in the overlap graph have two incoming edges and two outgoing edges, which often originate from a short repeat region between two different genomes. The two genomes may have different coverage depths to separate their edges. The coverage depth is calculated for every position along an edge to estimate the mean *δ* and SD *θ* of coverage depth along the edge. Only unique reads in an edge are considered for coverage depth calculation. A pair of adjacent edges on an unresolved vertex are merged if | *δ*1 –*δ*2 | < *θ*1 + *θ*2.

## Hash Table and Distributed Read Data Storage

The hash table described in section 3.1.1 is the major read storage that can be distributed and stored across all nodes (Fig 1.). All reads are loaded to the memory and indexed in a partially distributed hash table for fast lookup. Let *K* be the user-defined minimum overlap length. The keys of the hash table are DNA sequence substrings of length *K –*1; 4 for each read corresponding to the prefix and suffix of length *K −*1 of both forward sequence and reverse complement sequence respectively. To reduce the number of times the read is stored in the hash table we apply a hash function that generates for same key for the prefix of the forward sequence and the suffix od the reverse complement sequence. Similarly, the key generated for the suffix of the forward sequence and prefix of the reverse complement sequence is the same. This allows us to store the read only twice in the hash table instead of all 4 times for all 4 substrings. We store a single orientation bit along with other read metadata such as read identifier and length for differentiating between the prefix or suffix. This bit is used to correctly compute the orientation of a hit when querying the hash table.

One major design consideration while distributing the hashed read across multiple processes is that sequences hashing to the same value must the stored together on the same node. Otherwise, each hash lookup will result is more than one costly remote memory access operation. A main consideration in our design of the distributed hash table is to ensure minimum number of remote access operation. Therefore, we introduced a local hash index table to store the global index of each hash block. This local hash index table initially counts the number of bytes required to store all the read entries for each hash key in a preprocessing step. This information allows to equally distribute the hash data across nodes and then a cumulative sum operation on the local hash index table gives us the global index for each hash key. This global index can be used to determine the rank of the MPI process storing the hash data and the local index of the hash data within that process’s remote memory access window. The length of the block of data to be fetched can be computed by looking and the global index of the next hash key and subtracting from it the queried global index. With these three bits of information i.e. rank, local index and length we can initiate a MPI\_Get operation to lookup remote memory and receive all the reads hashed to a particular key in one operation.

The same design of the hash table is used by the OpenMP and MPI HighMem versions with all the data blocks being stored on a single node and therefore requiring no remote memory operations.

## Distributed Data Transfer and Synchronization

The distributed hash data storage described in section 3.2 is highly efficient and performs no more than 1 remote memory operation for one hash lookup. Hash lookups are used for determining the potential set of overlapping reads for a given read. Computing all overlapping edges of a read requires N-K+1 hash lookups with each such operation requiring 1 costly remote access.

Consider the read alignment in the figure above. The large number hash lookups for the read R1 will also be performed for R2 and again repeated for R3. To avoid such repeated remote memory accesses, we use a hash data cache that stores the ash lookup results locally. We use least recently used cache algorithm to cache remote data and improve performance.

Our MPI remote memory access algorithm assumes that MPI\_THREAD\_SERIALIZED is available and implemented by the MPI library.

The graph construction process in each MPI process is further multithreaded. The main thread does not perform any computation but is only responsible for point to point MPI communication to reduce redundant neighborhood computation.

# Results & Implementation

We performed two major experiments to test the performance and accuracy of Omega3. We compare the accuracy of Omega3 against 6 other popular assemblers – SPADes (Bankevich *et al.*, 2012), ABySS, MegaHit (Li *et al.*, 2015), MetaVelvet , RAY and HipMer. The performance comparisons for the OpenMP version of Omega3 is shown against all the 6 above assembler. The MPI HignMem and RMA version of Omega3 are compared against RAY and HipMer which can scale to many nodes.

## Accuracy

We compare Omega3 accuracy with 5 other assemblers and report .

### 3.4.1 Evaluation Metrics

We use MetaQUAST tool to assess the quality of assembly (Gurevich *et al.*, 2013). We select 3 major assembly quality parameters to report from the various metrics that are reported by MetaQUAST. The quality parameters are:

1. # misassemblies: This is the measure of the the number of contigs that are incorrectly joined. This metric is the sum of all relocations, inversions and translocations errors. This metric is the best measure of correctness of an assembly. Local base calling errors are not included in this metric.
2. Median NGA50: NGA50 is calculated after splitting incorrect contigs and scaffolds to align then all to the reference genomes. The N50 score of these split contigs and scaffolds is NGA50. Since, the NGA50 is reported for each reference in the metagenome individually. We pick the median NGA50 of all the reference organisms that are at least covered 10 times by the read library. This metric is a good metric to measure contiguity of the assembly. Due to the fact that incorrect contigs and scaffolds are broken to align correctly with the reference this metric cannot influenced by incorrect long contigs.
3. Genome Fraction (%): Genome fraction is the ratio of bases in the reference that appear at least once in the assembled contigs. This metric is a good measure of completeness of the assembly. In cases of complex metagenome communities with large number of low coverage organisms this value is very useful to determine the capability of an assembler to assemble low coverage organisms.

Further details on QUAST assembly metric can be found in (Gurevich *et al.*, 2013).

### 3.4.2 Test Datasets & Experiment Setting

The comparison is conducted on 5 datasets of various sizes and community complexities. Table 1 shows the input parameters of the test datasets. The 64 genome dataset is a small low complexity dataset with 64 genomes covered almost uniformly. The MC04 and MC06 are mock datasets are simulated high complexity datasets with 868 microbes in the community. These two mock datasets are drawn from the same set of reference organisms however the abundance of the organisms are different. The mock 250 medium volume dataset is a real Illumina sequenced library from a low complexity mock community of only 26 reference organisms. The CAMI dataset is a medium complexity mock dataset with large sequence volume. These 4 datasets were trimmed and filtered using BBtools(cite?) and error corrected using the Tadpole(cite?).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Sl. No. | Name | Seq.  Param.(bp) | # of Reads  (Mbp) | # of  References | Read Disk Space(GB) |
| 1 | 64 Genome | 2 x 100 | 214 | 64 | 24 |
| 2 | MC04 | 2 x 150 | 128 | 144\*(868) | 42 |
| 3 | MC06 | 2 x 150 | 128 | 137\* (868) | 42 |
| 4 | MOCK 250 | 2 x 250 | 210 | 26 | 108 |
| 5 | CAMI | 2 x 150 | 725 | 304 | 241 |

The datasets were chosen to reflect both the sequencing volume of metagenomics reads and complexity of metagenomics communities.

The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.

* The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.
* The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.
* The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.

The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.

The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.

## Computational Performance

The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.

The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.

The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.

The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.

The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.

The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.

 (1)

The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.

**Fig. 1. Relation between τ and *t*.** This example has only two continuous Steppers, S1 and S2.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Table 1.**Benchmark results accuracy test | | | | | | | | | | | | | | | | | |
| Assembler | Datasets | | | | | | | | | | | | | | | | |
| 64 Genome | | | MC06 | | | MC04 | | | cami | | | | Mock250 | | | |
| Mis.a | NGA50b | GF(%) | Mis | NGA50 | GF(%) | Mis | NGA50 | GF(%) | Mis | NGA50 | GF(%) | Mis | | NGA50 | GF(%) |
| Omega3 |  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |
| SPADes |  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |
| ABySS |  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |
| MegaHit |  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |
| RAY |  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |
| HipMer |  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |
| This is table foot note sample text This is table foot note sample text This is table foot note sample text | | | | | | | | | | | | | | | | | |

The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.

The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.

The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick

brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.

**Table 1.**Benchmark results of the cascade oscillators model

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |S| | Predicted cost | Timing | Predicted speed | Speed |
| 1 | S219.20(100%) | 68m43s | 1.00 | 1.00 |
| 2 | 29.10+219.10(~50%) | 35m13s | 2.00 | 1.95 |
| 4 | 219.20(100%) | 68m43s | 1.00 | 1.00 |
| 10 | 29.10+219.10(~50%) | 35m13s | 2.00 | 1.95 |
| 20 | 219.20(100%) | 68m43s | 1.00 | 9.5 |

This is table foot note sample text This is table foot note sample text This is table foot note sample text

The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.The quick brown fox jumps over the lazy dog.

Acknowledgements

The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.

Funding

This work has been supported by the …..

*Conflict of Interest:* none declared.

References

Bankevich,A. *et al.* (2012) SPAdes: A New Genome Assembly Algorithm and Its Applications to Single-Cell Sequencing. *J. Comput. Biol.*, **19**, 455–477.

Gurevich,A. *et al.* (2013) QUAST: quality assessment tool for genome assemblies. *Bioinforma.* , **29**, 1072–1075.

Li,D. *et al.* (2015) MEGAHIT: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph. *Bioinforma.* , **31**, 1674–1676.