Parallelization of Velvet,"a *de novo* genome sequence assembler"

Nitin Joshi[§†*], Shashank Shekhar Srivastava[§†*],
M. Milner Kumar[†], Jojumon Kavalan[†], Shrirang K. Karandikar[†]and Arundhati Saraph[†]
[§]Department of Computer Science and Technology Goa University, Goa
[†]Computational Research Laboratories, Tata Sons Ltd. Pune

***Abstract*—Genome assembly is one of the most complex and computationally intensive tasks of genome sequence analysis. There are various parallel and serial tools available for genome assembly. Some parallel tools require as much as 2 days to assemble human genome. Serial tools are incapable of achieving this milestone even on a powerful machine with 256 GB of memory. Velvet is one such serial tool which requires 14 GB of memory even for a small bacteria genome, therefore memory is a major limitation of Velvet. In this paper we present our work on parallelization of VelvetH (the primer part of Velvet). As a demonstration of the capability of our parallel-VelvetH, we hashed the human genome in as low as 1 hour 30 minutes.**

I. Introduction

De novo assembly [1] is the process of reconstructing the genome of organisms not sequenced before or for which a reference comparative genome is unavailable. It is accomplished through the shotgun process where the genome of the organism is sheared into small fragments, each of which is sequenced separately and reconstructed using computational tools. This process is complex because genomes contain segments of identical sequences namely *repeats*. The length of the repeats varies very much and makes it impossible to recover the complete genome. Therefore, almost all *de novo* tools do not recover the complete genome. However, they report long segments of genome known as *contigs*. Furthermore, the complexity increases with the size of the genome.

There are primarily two categories in *de novo* genome assembling process namely *Overlap Layout and Consensus* (OLC) and *De Bruijn* graph based methods OLC based methods are computationally more expensive than *De Bruijn* graph based methods, whereas the latter are memory intensive. There are several tools available [2]–[4] based on *De Bruijn* graph. Velvet [5] is also based on *De Bruijn* graph approach, which is widely used. In this paper we present our work on parallelization of VelvetH, the module (which does the hashing of reads) of open source tool Velvet.

[*] Student Author

A. Motivation and problem definition

Even today, very few centers have the resources, in terms of both software and hardware, to assemble a genome. Most short read assemblers are single threaded applications, designed to run on a single processor. Velvet is one such widely used serial tool. However, the practical use of these assemblers are limited for large genomes due to computation time and memory constraints. One of the bottlenecks for practical assembly of short reads is the huge memory requirement in order to process repetitive fragments from large genomes. Another big challenge for the assembly of short reads is the intensive computational time requirement.

To decrease the time cost of the assembly procedure, threaded parallelization is implemented in a few assemblers. While this approach addresses the large run time required, the memory footprints is not resolved. In this paper we present our work on parallelizing VelvetH on a distributed memory architecture. This allows us to run VelvetH on a cluster of nodes in parallel. We obtain the benefit of run time speedup, and our approach ensures that the memory usages on an individual node is within reasonable limits.

B. Previous work on parallelization of sequence assembler tools

Several previous work has been done either to develop parallel sequence assembly tools or to parallelize existing serial sequence assemblers. One existing tool, SOAPdenovo [3] supports multithreaded parallel computing. The latest version of Velvet ($version1.1.03$) also supports multithreaded parallel computing. As mentioned before, these multithreaded tools address only the run time issue. Since the memory space is shared, their operation requires computers with large memory such as 256 GB RAM, thereby rendering their approach infeasible for large genomes. ABySS [4] is also a sequence assembler that is designed for short reads, whose parallel version is implemented using Message Passing Interface (MPI) and is capable of assembling larger genomes. However, genome assembly using ABySS on the human genome still required 15 hours to generate contigs [4].

II. Architecture of Velvet

Velvet has two parts, VelvetH for hashing and VelvetG for graph generation. VelvetH produces a hash-table of which two files namely *Sequences* and *Roadmaps* are the output and function as input to VelvetG. VelvetG creates a directed graph and generates the unique segments of chromosomes as output.

In this section, we describe the computational aspects of VelvetH, which is of interest from a parallelization perspective. The interested reader is referred to [5] for an exposition on the theory of genome assembly. We also highlight the dependencies between different computational steps, which makes parallelization difficult.

A. Working of VelvetH–serial

A *read* is a sequence of 4 letters, A, C, G, T which represent the four basic molecules (called *nucleotides or bases*) that composes the DNA. Every read can be of length between $25 - 100$ characters. All possible sub-strings of length k of this read, termed k-mers, are generated and stored in a hash-table.

An example of a read of length 6 and k-mers for k = 4 is shown in Figure 1. The value of k can be chosen by the user.

```
read  =  ATGCTT
kmers = {ATGC,
          TGCT,
           GCTT}
```

Fig. 1. *All possible k-mers for k = 4 from read of length 6.*

VelvetH starts by processing various input files, which have millions of reads. While processing input files, VelvetH scans these reads and converts them to an internal format. The converted reads are written to a file called *Sequences*. k-mers are generated from reads. VelvetH maintains a hash-table with n entries and checks for the presence of a k-mer in this hash-table. If present, it stores a reference of that entry in the *Roadmaps* file. If not, the k-mer is inserted into the hash-table. The hash-table and *Roadmaps* are used by VelvetG to assemble the complete genome. Figure 2 shows the insertion of k-mers into hash-table.

B. Challenges involved in parallelization of VelvetH

Most of the time (60%) and memory of VelvetH is spent during insertion of k-mers into the hash-table. Each k-mer is inserted sequentially into the hash-table. This process can be expedited by inserting k-mers simultaneously into the hash-table. However, this approach has two problems. First, if the same k-mer is encountered (as in k_2 of Read I and k_1 of Read II in the example in Figure 2), we have a race condition, which enforces sequentiality - the 1^{st} k-mer has to be inserted in the hash-table and the 2^{nd} in the *Roadmaps* file. Second, if two k-mers hash to the same entry, once again we have to add them sequentially, else the state of the data structure can get destroyed.

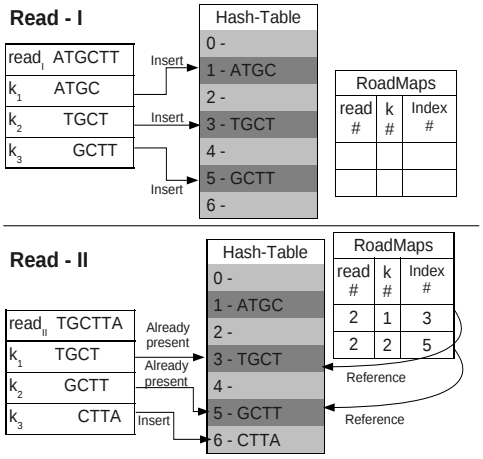

Fig. 2. *Serial-VelvetH: Insertion of k-mers into hash-table.* Read I - ATGCTT generates three k-mers (k_1, k_2, k_3), since this is the first occurrence of these k-mers, these are inserted into hash-table sequentially. Then Read II - TGCTTA generates three k-mers (k_1, k_2, k_3). k_1 and k_2 are already present in the hash-table, so a reference of these k-mers is stored in the *Roadmaps*. k_3 is inserted into hash-table.

Both the above issues can be resolved using a locking mechanism. However, this creates sequentiality, and reduces the potential speedup. Also, the memory problem is unresolved. In the next section, we present our solution, which deliver a speedup as well as reduces the memory requirements on individual rank(a MPI process).

III. Our approach to parallelization of VelvetH

Processing input files and converting data to the internal format takes about 40% of the run time, and can easily be done in parallel, since there are no dependencies. All files are evenly distributed across ranks, that independently carryout the data conversion.

In order to address the run time and memory issue with the rest of VelvetH, we process as follows. The hash-table, which has n entries, is distributed among ranks, so that the first rank owns the first n/r entries, the second the next n/r entries and so forth. Since the input data is processed in parallel, each rank already has the data it has to analyze. For each read, all k-mers are generated and the hash-keys for each k-mer immediately tells us which part of the hash-table this k-mer is to be inserted into (or checked if a prior identical k-mer has been encountered before). This can be mapped to the corresponding rank, which owns that portion of the hash-table. If the index of the k-mer, being processed by

rank r_1, falls in the range of the hash-table that is owned by rank r_1, then processing is straight forward. If not, then a communication is initiated with the rank, say r_2, that owns the section of the hash-table of interest. Rank r_2 can then either insert the k-mer into the hash-table or determine that this had been inserted before, and extract the necessary reference information and store it in the *Roadmaps* file. As can be easily seen, as the number of ranks increases, this approach becomes infeasible, since the communication overheads increases drastically. For 2 ranks, about half the k-mer that a rank analyzes are processed locally and half are communicated. This fraction drops to $1/n$ for n ranks, and becomes no better than sequential processing. However, the memory problem has been addressed. Each rank can run on a separate node of a cluster (for r ranks), and requires $1/r$ of the memory, since the hash-table and its associated data is now distributed across r ranks.

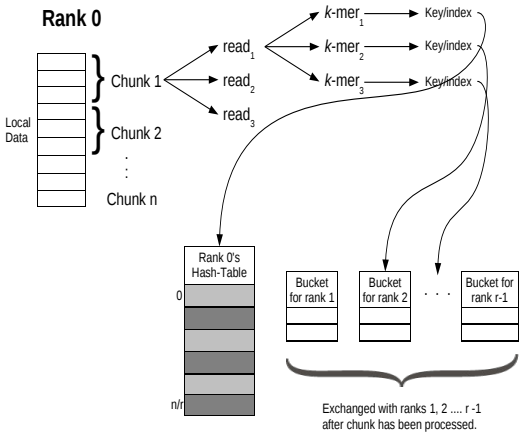

Fig. 3. *Parallel-VelvetH: Insertion of k-mers into distributed hash-table.*

We now address the speedup issues. The approach presented above is highly communication intensive. In addition, ranks spend more time communicating with each other rather than processing data. We overcome this limitation as follows. We define a *chunk* as a certain number of k-mers. Each rank process this number of k-mers. Every rank also maintains *buckets* for every other rank as in Figure 3. If a k-mer on rank i has to be sent to a different rank, say rank j, it is put into rank j's bucket. After processing all the k-mers in a chunk, rank i will have processed its own k-mers in the usual manner and will have $r-1$ buckets of k-mers that belong to other ranks. This is true for all ranks. We then do an all-to-all communication, where data is exchanged and the communicated k-mers can now be processed locally. We note that while the all-to-all communication is being carried out, processing of the next chunk can begin. In this manner, we separate out the issue with simultaneous insertion of k-mers into the hash-table (even in the case of collision) and communication and synchronization needed between ranks. This approach provides the expected speedup when concurrency is being put to use. As mentioned before, since the hash-table and associated data is distributed across ranks, we address the memory problem as well.

The size of a chunk is a parameter that can be adjusted, and it can have a significant impact on performance. A chunk size of 1 is no different than our initial approach presented previously. Setting the chunk size to the largest value, which would be number of k-mers that a rank has to process would not deliver the benefits of parallelism, and raises another serious problem - in worst case, all k-mers would be sent to another single rank, which means we need to allocate buckets of size k, thereby leading to large memory requirements.

For intermediate values of chunk size, performance is obtained by achieving a balance between how much communication there is in absolute terms, how much of this communication can be overlapped with the computation of the next chunk, the memory overhead of maintaining multiple buckets per rank, and the number of k-mers that are eventually processed locally, versus the number that are sent for processing to and from other ranks. We present some experiments with chunk size and its impact on performance in the following section.

IV. Results and discussion

In this section we present performance results of both, serial VelvetH and parallel-VelvetH on EKA(CRL's public cloud facility), a 1800 node infiniband cluster. Each node is equipped with 16 GiB RAM and with dual quad core Intel Xeon processors. Every node is connected to each other with the central storage(80 TB) through a 20 gigabit Infiniband network. We ran our experiments with a pre-released version of Velvet 1.0.19 for serial VelvetH and for our parallel VelvetH we used hp-MPI(mpirun: HP MPI 02.02.05.01 Linux x86-64 major version 202 minor version 5).

In order to test for correctness and measure benefits of parallel-VelvetH with respect to serial VelvetH, we selected three genomes - *Bacteria, Yeast, Worm*. The sizes of the genome and those of the input data are as shown in Table I. Note that the input data of *Bacteria* is much higher than the size of genome would suggest. This is because the coverage of the genome in this data set is higher than the other two. The run time and memory requirements of serial VelvetH on these input is a function of both the genome size and input data size. The last two columns of Table I show the run time and memory when serial VelvetH is used to process these three genomes.

TABLE I
SERIAL-VELVETH: STATISTICS FOR RUN TIME AND MAXIMUM MEMORY OCCUPANCY OF 36 LENGTH SHORT READS ASSEMBLY PROCEDURE. *SERIAL VELVETH CANNOT HANDLE THIS INPUT.

Genome	Size Of Genome(MB)	Size Of Input(MB)	No. Of reads	Time (sec)	Memory (MB)
Yeast	16	5000	0.27×10^8	244	3925
Bacteria	4	6600	0.35×10^8	295	14443
Worm	80	8000	1.15×10^8	1183	15047
Human	3300	239000	20.2×10^8	*	*

As can be seen in Table I, the time and memory requirement grows exponentially for these genomes. The human genome is approximately 40 times larger than *Worm*, and the corresponding input data is 30 times larger. Of course, running this on a machine with 16 GB of memory is infeasible. We tried processing parts of input data on a computer with 256 GB of memory (Table II). We were able to process upto 900 million reads at the most− with 1 billion reads, the program ran out of memory. Extrapolating this data, we estimate that for 2.02 billion reads, serial VelvetH would take 500 GB of memory and 2 days to complete.

TABLE II
SERIAL-VELVETH: STATISTICS OF RUN TIME AND MAXIMUM MEMORY OCCUPANCY OF 36 LENGTH SHORT READS ON PARTS OF HUMAN GENOME PERFORMED ON A MACHINE WITH 256 GB OF MEMORY. *SERIAL VELVETH COULD NOT HANDLE THIS INPUT.

Number Of Reads (Billions)	Time	Memory (GB)
0.3	$5hr11min$	65.6
0.6	$10hr21min$	124
0.9	$16hr43min$	166
1	*	*

In the remainder of this section, we present the results of parallel-VelvetH run on the *Bacteria, Yeast, Worm* genomes, followed by the results on the human genome and finally an analysis of the performance and approach towards tuning the same.

A. Experiments with smaller genomes

Figure 4 and 5 present the speedup and memory improvement that is obtained by running parallel-VelvetH on the small genomes for different number of ranks. With the exception of 2 ranks, significant improvement with respect to memory requirement per rank and speed was achieved with parallel-VelvetH.

The poor performance for 2 ranks is due to communication overhead. This is discussed in details towards the end of this section. Note that the total memory used by all ranks in a parallel-VelvetH run is actually greater than that used by serial VelvetH, due to the book keeping and communication buffer overheads. However, this is not an issue as each rank individually can be run on a computer with a reasonable amount of memory.

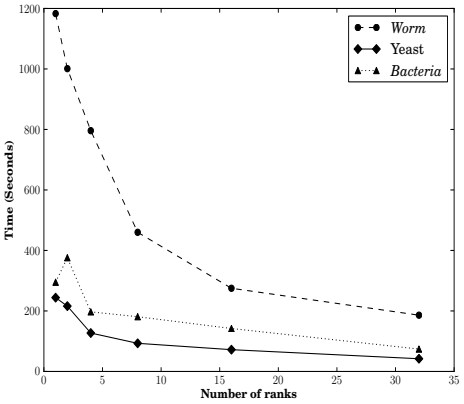

Fig. 4. *VelvetH: Run time of* 36 *length short reads assembly procedure for different data sets. On x-axis data points are drawn at* 1 *for serial VelvetH and at* $2, 4, 8, 16$ *and* 32 *ranks for parallel-VelvetH.*

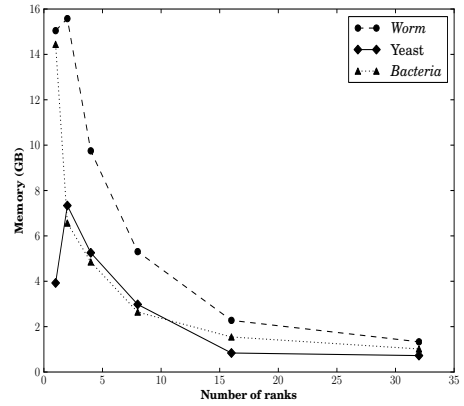

Fig. 5. *VelvetH: Maximum memory occupancy of 36 length short reads assembly procedure for different data sets. On x-axis data points are drawn at* 1 *for serial VelvetH and at* $2, 4, 8, 16$ *and* 32 *for parallel-VelvetH.*

B. Experiment for human genome

We have used parallel-VelvetH to process the human genome starting with 128 ranks and going all the way to 512 ranks. The run times vary from 2 hours to 1.5 hours respectively. This is the key contribution of this work. We can process the human genome in a couple of hours, rather than the previous couple of days. This brings the heuristic "instant decoding of an individual genome" closer to reality.

C. Analysis of run times

In this section, we experimented with the chunk size, described in section III to get better performance. we also decompose the run time of parallel-VelvetH to understand its behavior.

1) Experiment for optimal Chunk Size: As described earlier, parallel-VelvetH splits input data to be processed at each rank into *chunks*. The size of a chunk can vary from 1 (minimum) to the entire input data set being processed by that rank (maximum). An intermediate values of chunk size balances computational and parallel efficiency with communication and memory overheads. There are many parameters that can influence the impact of chunk size on performance, and we are in the process of developing an analytical model for the same.

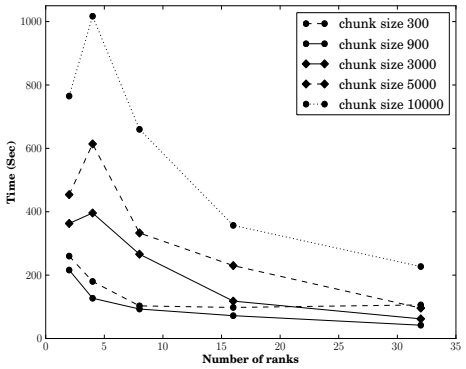

Fig. 6. *Parallel-VelvetH: Chunk size experiment for Worm. On x-axis data points are drawn at 2, 4, 8, 16 & 32 ranks*

In our experiments, we tried various chunk sizes, to determine the optimal value. The results of this are shown in Figure 6. For a given number of ranks, say 16, the run time with chunk sizes of $300 - 10000$ can vary by 4 times. For most ranks, a chunk size of 900 is the optimal. This means that each rank process 900 reads at a time, adding these to either its internal processing queue or to other rank's buckets for communication. A smaller chunk size results in more communication time while a longer size needs more memory and does not have much run time benefit.

2) Communication and computation among ranks: Figure 7 breaks out the time spent by parallel-VelvetH when processing *Worm* into two components– the time spent during processing and that spent for communication, for different ranks. As it is obvious that the computation time reduces as the number of ranks increase. This is expected, since the amount of data being processed by each rank is $1/r$ the total data, when 'r' ranks are being used. The communication time is roughly constant across different ranks. The implication of this is the cross-over point at 16 ranks, after which every rank is spending more time in communication than in computation but still the total time is getting reduced. This inhibits the speedups that can be obtained by increasing the number of ranks.

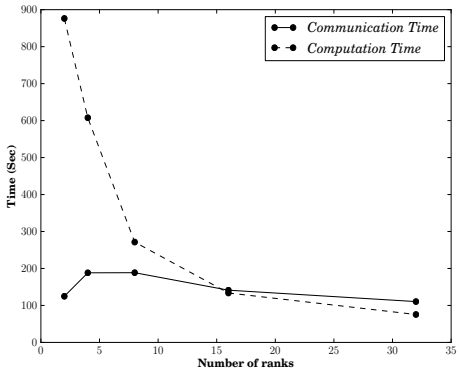

Fig. 7. *Parallel-VelvetH: Communication and Computation for Worm.*

V. Conclusions and future work

Genome sequencing technology has grown at an outstanding rate over the past few years. However, while the technology to generate short reads has progressed, assembling these reads to produce the complete genome is a bottle neck in *de novo* sequencing. The work presented in this paper is a major step forward towards near-real time genome assembly on commodity hardware. We have demonstrated significant speedups over the sequential version for smaller genomes (*Bacteria, Yeast, Worm*), and the human genome was processed in 1 hour 30 minutes on a small cluster.

While this requires $128 - 512$ ranks, the cost of this cluster is much less than the cost of High Throughput Sequencing Machine. An equivalent single computer with terabytes of memory would be prohibitively expensive thus, this work is a quantum leap towards near real time genome sequencing.

Most computers today have multiple cores. We are also working on a hybrid model of using threads on a compute node with our MPI based approach across nodes in order to obtain greater speedups.

References

[1] Genome sequence assembly primer [http://www.cbcb.umd.edu/research/assembly_primer.shtml].

[2] Jonathan Butler, Iain MacCallum, Michael Kleber, Ilya A Shlyakhter, Matthew K Belmonte, Eric S Lander, Chad Nusbaum, and David B Jaffe. Allpaths: de novo assembly of whole-genome shotgun microreads. *Genome Res*, 18(5):810–20, May 2008.

[3] Ruiqiang Li, Hongmei Zhu, Jue Ruan, Wubin Qian, Xiaodong Fang, Zhongbin Shi, Yingrui Li, Shengting Li, Gao Shan, Karsten Kristiansen, Songgang Li, Huanming Yang, Jian Wang, and Jun Wang. De novo assembly of human genomes with massively parallel short read sequencing. *Genome Res*, 20(2):265–72, February 2010.

[4] Jared T Simpson, Kim Wong, Shaun D Jackman, Jacqueline E Schein, Steven J M Jones, and InanÃ Birol. Abyss: a parallel assembler for short read sequence data. *Genome Res*, 19(6):1117–23, June 2009.

[5] Daniel R Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome Res*, 18(5):821–9, May 2008.