Parallel Computing 70 (2017) 22–26

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

ParConnect reproducibility report

Lei Yang, Yilong Li, Zhenxin Fu, Zhuohan Li, Wenbin Hou, Haoze Wu, Xiaolin Wang, Yun Liang*

Peking University, 5 Yiheyuan Road, Haidian, Beijing 100871, China

ARTICLE INFO

Article history:
Received 15 March 2017
Revised 26 June 2017
Accepted 16 July 2017
Available online 19 July 2017

Keywords:
De Bruijn graph
Connected component labeling
Distributed algorithm
Reproducibility

ABSTRACT

Flick et al. (2015) [1] proposed a novel distributed algorithm for connected component labeling on de Bruijn graphs. As a special activity of the Student Cluster Competition at SC16 (*The International Conference for High Performance Computing, Networking, Storage and Analysis*), we replicated the experiments in their paper. Though minor differences exist, experiment results show that, two optimizations, "load balancing" and "active partitions only", are both effective; the algorithm is scalable when processor cores are enough. We conclude that the correctness, performance and scalability of the algorithm are successfully reproduced.

1. Claims to reproduce

In bioinformatics, metagenomics study requires reconstruction and assembly of genomes from large and complex data sets, which contain gene extracts (or reads) from various species. De novo assembly method is used in the reconstruction process, where de Bruijn graphs are utilized to encode the overlap information [1]. However, because of the large size and complexity of the data set, ordinary de novo assembly tools won't work due to high memory demands [2]. Howe et al. discovered that the variety of species leads to a large number of disjoint connected components in the graph [3], which can be exploited to decompose the data set for parallel processing.

To solve the graph decomposition problem efficiently, Flick et al. [1] proposed a distributed memory algorithm based on connected component labeling. And they provided further optimizations and illustrated them with several experiments.

The original paper made three main claims, as listed below:

1. In respect of optimization: *Excluding completed partitions (AP)* eliminates unnecessary tuple traversal and inter-process communication; *Load balancing (LB)* dynamically adjusts workload on each process to avoid unnecessary waiting.
2. In respect of performance: Excluding completed partitions has better performance than naive algorithm (no optimization), and load balancing further improves the performance.
3. In respect of scalability: The strong scaling experiment demonstrates that the algorithm is highly scalable.

Also, the new algorithm is compared with a BFS based algorithm to further illustrate its advantage in performance and running time. As required by the competition, we reproduced all the three main claims listed above.

* Corresponding author.
E-mail addresses: yangl1996@pku.edu.cn (L. Yang), yilong@pku.edu.cn (Y. Li), fuzhenxin@pku.edu.cn (Z. Fu), zhuohan123@pku.edu.cn (Z. Li), houwenbin@pku.edu.cn (W. Hou), functioner@pku.edu.cn (H. Wu), wxl@pku.edu.cn (X. Wang), ericlyun@pku.edu.cn (Y. Liang).

http://dx.doi.org/10.1016/j.parco.2017.07.006
0167-8191/

Table 1
Configuration of our cluster.

IBM "Minsky" S822LC Server × 2, each node with		Vendor
CPU	2 × IBM POWER8+ Processor (8 cores 64 threads, 4GHz)	IBM
Memory	256GB DDR4 2133 (with L4 Cache)	Samsung
Accelerator	4 × NVIDIA P100 (NVLink)	NVIDIA
Network	100 Gbps Infiniband EDR	Mellanox
OS	Ubuntu Server 16.04 for POWER8	–

Table 2
Experiment result of Task 1.

Data set	# of reads	# of Edges	# of CC
Small	2,000,000	90,665,399	1,929,553
Large	10,000,000	442,461,650	9,006,212

2. Machine description

The configuration of our cluster is shown in Table 1.

Due to power limits and other requirements of the competition, our cluster varies from that in the original paper in several ways:

1. Number of nodes: Our cluster is a small one with only 2 nodes, compared to the much bigger *CyEnce* cluster used in the original paper.
2. Interconnection: Our cluster uses Infiniband EDR with 100 Gbps bandwidth, which is much faster than QDR (40 Gbps) used in the original paper.
3. L4 Cache: Our cluster has L4 cache chips on memory boards, which provide better memory performance when locality is good.
4. Simultaneous Multi-threading (SMT): Our cluster has only 4 CPUs with 32 physical cores in total, so SMT is necessary to run 128 concurrent processes.
5. CPU Architecture: Our cluster uses IBM POWER8+ architecture, which may behave differently from the x86-64 CPUs used in the original paper.

3. Machine and software setup

We installed gcc 5.4.0 (IBM/Ubuntu) compiler and OpenMPI 2.0.1 library on the Ubuntu 16.04 (ppc64le) system. And we configured NFS (Network File System) in order to share the input files between nodes.

In order to compile and run ParConnect in benchmark mode, some modifications were made to the original source code:

1. All the "`-march=native`" options in `CMakeLists.txt` were disabled since it is not supported by gcc on POWER platform.
2. The "`BENCHMARK_ENABLE_CONN`" option in `CMakeLists.txt` was enabled so that communication time can be collected for further analysis.
3. The "`OPTIMIZATION`" option in file "`src/coloring/labelProp.hpp`" was modified in order to run ParConnect in different algorithm variations.

4. Data and analysis

During the competition two data sets were provided, named "Large" and "Small" respectively. We were instructed to complete four tasks which resemble the experiments conducted in the original paper. The correctness, performance and scalability of the original algorithm can be examined from the results.

4.1. Task 1: number of edges and connected components

We ran ParConnect using different number of MPI processes (8, 16, 32 and 64) and different algorithm variations (Naive, AP, AP_LB) on the two data sets provided. Results are all same for one data set in different configurations, as shown in Table 2.

4.2. Task 2: program output timings

We collected the running and communication time from the outputs of the ParConnect program. Different algorithm variations were used to process the large data set. The results are shown in Fig. 1.

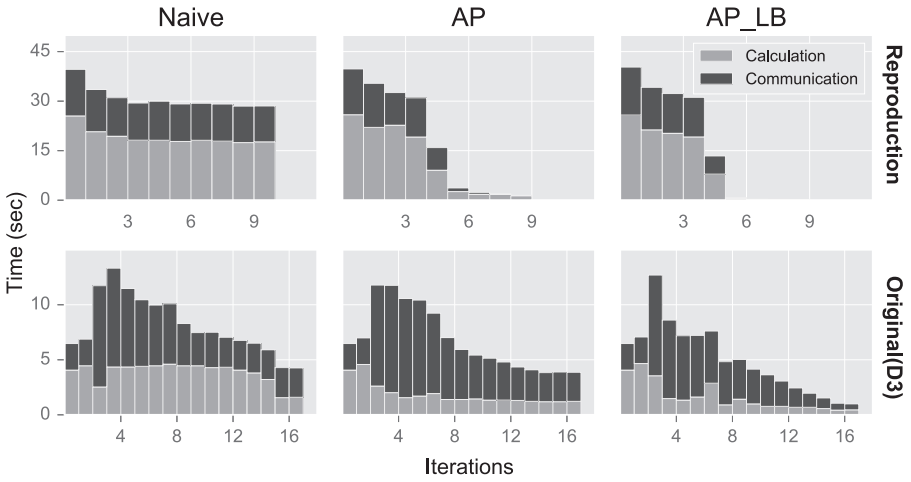


Fig. 1. Time spent on communication and computation in each iteration of the algorithm. Above are our results generated using the large data set on 32 processes; below are the original results using data set D3 on 1280 processes.

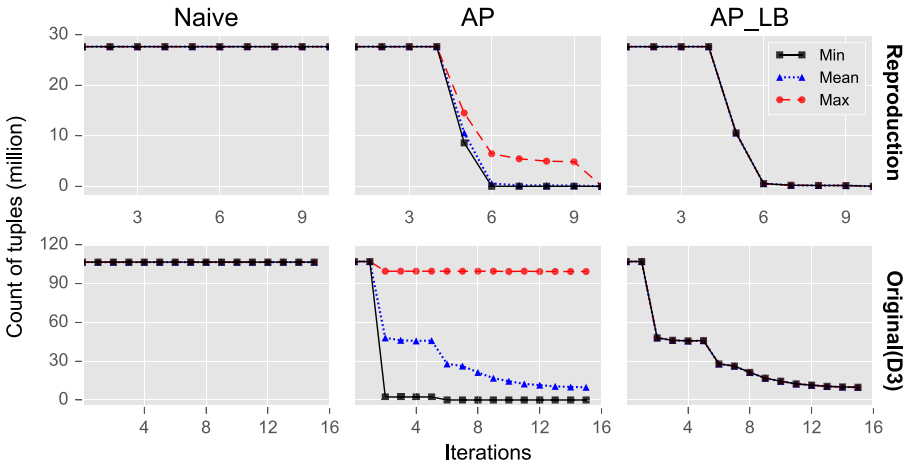


Fig. 2. Tuple load balance across processes during each iteration of the three algorithm variants. Above are our results generated along with Task 2; below are the original results using data set D2 on 60 processes.

Here the AP variant shows better performance (less computation time) than the Naive variant. The AP_LB variant further reduces communication and computation time on the basis of AP due to balanced distribution of workload. Those results correspond with the claims made in the original paper.

Slightly different from the results in the original paper, it is computation, rather than communication between nodes, that occupies the majority of running time. It is mainly because only two nodes were used so that most communications are conducted through shared memory, which has much higher bandwidth and lower latency than any network link. The high-speed Infiniband EDR interconnection is also responsible for the discrepancy.

4.3. Task 3: tuple analysis

During each iteration, the distribution of workload is examined. We did this by recording the numbers of working tuples for each process.

As shown in Fig. 2, for the naive variant, the maximum, mean and minimum tuple counts keep unchanged throughout the run, which perfectly corresponds with the results in the original paper. For the AP variant, the mean tuple count decreases sharply, indicating that there are a large number of completed partitions excluded from subsequent iterations. And for the AP_LB variant, the workload is evenly distributed at the beginning of each iteration, so that the maximum, mean and minimum tuple counts are equal.

A slight discrepancy between our results and the original results exists: In the original experiments, the AP variant started with a high max tuple count and this value remained high, which greatly affects the running time; but in our experiments, the max count of tuples decreases quickly. Two factors are responsible for this discrepancy: The first is the

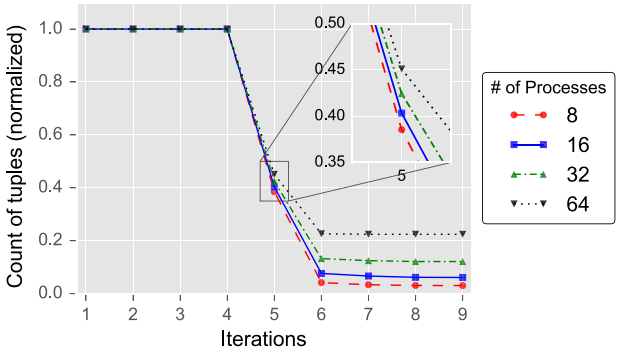
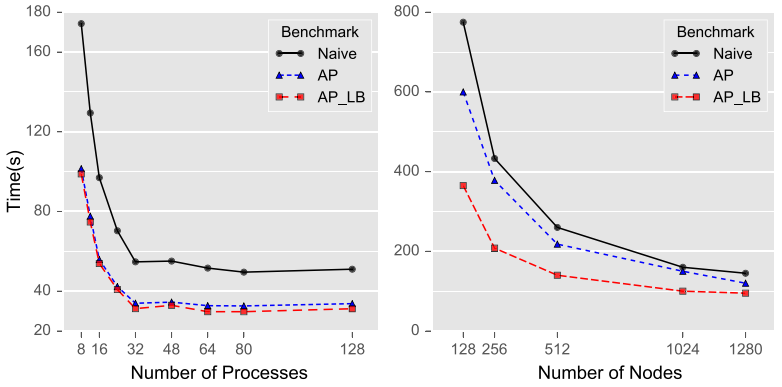


Fig. 3. The number of processes affects the max count of tuples. Max tuple count gets higher as the number of processes increases. Experiments were conducted using the small data set.



(a) Our reproduction (b) Original result

Fig. 4. Scalability of the three algorithm variants. Experiments are conducted using small data set in our reproduction (a) and using the D3 data set in the original paper (b).

number of processes. As the number of processes increases, less tuples are allocated to a single process. Processes with lower ranks will be less likely to have completed partitions. We analyzed the relationship between the number of tuples and processes to justify our hypothesis, of which the results are shown in Fig. 3; The second is the characteristics of the graph itself, since the max tuple counts are different when using different graph inputs.

4.4. Task 4: scalability analysis

A strong scalability analysis was also conducted. The small data set was used, as it is possible to run it on 8 processes in a reasonable time. The results are shown in Fig. 4.

When we run the benchmark on no more than 32 processes, where every process works on a single physical core, all the algorithm variants show good scalability. AP performs better than Naive, and AP_LB performs slightly better than AP.

Different from the original results, the performance difference between AP and AP_LB is slight. This is because the processing time of each iteration depends solely on the maximum tuple count. In our experiments, the maximum tuple counts of the AP and AP_LB algorithm variations are close, and so are the time spent on each iteration. This explains the difference between our results and the original results.

However, when the number of MPI processes exceeds 32, multiple processes are running on a single physical core using simultaneous multi-threading (SMT). SMT virtualizes a physical core into multiple SMT cores by allowing concurrent use of different functional parts on a physical core. But in our experiments all processes are doing similar tasks, and there is not enough hardware to serve all the SMT cores. As a result, the performance gain is little.

5. Discussion

As shown above, there are slight differences between our results and the paper's. But we consider all those differences explainable since our cluster varies greatly from the one used in the original paper, and the data sets are also different. For extra assurance, we analyzed the experimental data, which successfully confirmed our hypotheses.

For Task 2, we observe that AP greatly reduces computation and communication time compared to the naive algorithm, and LB further shortens the time, showing that AP and LB improves the efficiency of the algorithm, which tally with the performance claim in the original paper.

For Task 3, we observe that AP algorithm reduces tuple count, showing that AP optimization is effective. In AP_LB algorithm, maximum, mean and minimum tuple count are equal in each iteration, implying that the workload is evenly distributed across all processes. Both results conform to the effectiveness of the two optimizations claimed by the paper.

For Task 4, when the process number is below 32 where SMT is not active, the running time is almost inversely proportional to the process number, indicating excellent scalability. When the process number is above 32 where SMT is active, the running time decreases slower. This is because that SMT cores perform much worse than physical cores, and those data should not be taken into account when judging scalability. So the results is in consistency with the paper's scalability claim.

6. Conclusion

Although minor differences exist, our results firmly support the claims made in the original paper. By replicating the experiments on our cluster, we observe that AP and LB optimizations are effective in reducing unnecessary computation and waiting, and both greatly improve performance. Also, the scalability claimed by the paper is reproduced. So we conclude that the original paper is fully reproduced by us.

Acknowledgement

The authors would like to thank Beijing Rongtian Huihai Technology Co., Ltd. for providing us with supercomputing clusters, technical support and financial assistance.

References

[1] P. Flick, C. Jain, T. Pan, S. Aluru, A parallel connectivity algorithm for de bruijn graphs in metagenomic applications, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, 2015, p. 15.
[2] J. Pell, A. Hintze, R. Canino-Koning, A. Howe, J.M. Tiedje, C.T. Brown, Scaling metagenome sequence assembly with probabilistic de bruijn graphs, Proc. Natl. Acad. Sci. 109 (33) (2012) 13272–13277.
[3] A.C. Howe, J.K. Jansson, S.A. Malfatti, S.G. Tringe, J.M. Tiedje, C.T. Brown, Tackling soil diversity with the assembly of large, complex metagenomes, Proc. Natl. Acad. Sci. 111 (13) (2014) 4904–4909.