Scaling graph algorithm (All Pair Shortest Path)

Jatin Dev (18111027) Swapnil Raykar(18111078)

I. Problem Description

In computer science, the all pair shortest path algorithm is an algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles). Floyd−Warshall algorithnm is used to solve many other problems like Shortest paths in directed graphs, Transitive closure of directed graphs, Kleene's algorithm, Optimal routing problem, computing canonical form of difference bound matrices, computing the similarity between graphs. The FloydWarshall algorithm is an example of dynamic programming approach and have O(n^3) sequential time. As sequential algorithms for this problem yield long run-time, parallelization has shown to be beneficial in this field. Many optimizations has been done to reduce time of Floyd-Warshall algorithm which takes benefits of shared memory architecture. But for distributed algorithm as communication cost is generally high, it requires different solutions. In this project we studied and implemented different distributed algorithms using MPICH for All pair shortest path and tried to figure out the bottlenecks in communication and computation and improve them.

II. Related Work

The FloydWarshall algorithm is an example of dynamic programming, and was published in its currently recognized form by [1]Robert Floyd in 1962 which takes O(n^3) time where n is the number of vertices.

Some of the APSP algorithms improved the time complexity from$O(n^3)$to $O(n^3/logn)$ [2] or to$O(n^{2.4})$ [3]

Alternatively, we can find an APSP solution by applying single-source shortest-path (SSSP) algorithm from each of the vertices in the graph. There are two well-known algorithms for solving SSSP: the Bellman-Ford [4] and the Dijkstras [5] algorithms.

However,for this trivial parallelization $|V|$ is an upper bound for the number of processors,but there are variants which achieve parallelization for more than $|V|$ processors.

[3] Modify classic Dijkstra algorithm which utilize the previously-calculated results to accelerate the latter calculation and reduce time complexity to about $O(n^{2.4})$

[2]describe an $O(n^3/logn)$ time algorithm for the all-pairs-shortest-paths problem for a real-weighted directed graph with n vertices. This slightly improves a series of previous, slightly sub-cubic algorithms by Fredman.

[6]jing-Fu Jenq and Sartaj Sahni propose checkerboard version of parallel Floyd-warshall algorithm for hypercube multiprocessor.The cost matrix P is divided into equal parts of size $(n/\sqrt{p})X(n/\sqrt{p})$, and each is allocated to a different processor. Each processor has the responsibility to update its allocated part of the matrix in each iteration.

[7]Proposes a Blocked version of APSP which makes better utilization of the cache and achieved a speedup between 1.6 and 1.9 compared to original APSP algorithm.

[8]Paper compared different parallel versions of the APSP and concluded that Pipelined Checkerboard version of Floyds algorithm has better scalability than all other parallel algorithms for architectures such as Cube, Mesh, Mesh-CT, and Mesh-CT-MC.

III. Implementation

The basic idea of any parallel algorithm is to divide computation equally among nodes while keeping less communication overhead. There are various versions of the parallel all pair shortest path algorithm all of them adds minor optimizations to its previous variant. We studied and implemented following algorithms

- General Parallel algorithm
- Naive Checkerboard Version
- Naive Parallel APSP
- Blocked Parallel Version

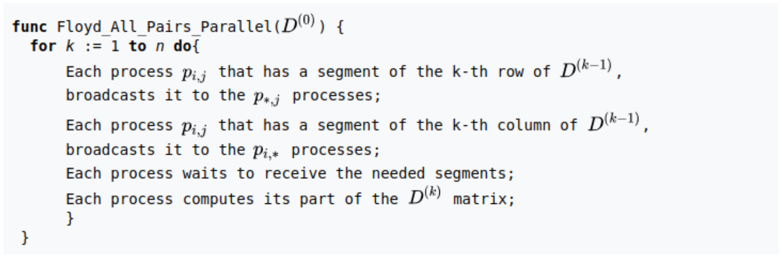
In starting we have done our analysis on naive checkerboard version and naive Parallel version. In both of these algorithms we need to have processors equals to the numbers of cells and number of rows respectively. Naive checkerboard shows very less computation time but have significant communication overhead. Naive parallel version is our baseline algorithm which we converted into Blocked parallel version by doing computations in blocks. We have presented our experiments and results for Blocked parallel APSP algorithm.

A. General Parallel algorithm

[8]General Algorithm parallel algorithm for APSP problem is given below for n nodes. The basic idea to parallelize the algorithm is to partition the matrix and split the computation between the processes. Each process is assigned to a specific part of the matrix. Each process that calculated a part of the k-th row in the D^{k-1} matrix has to send this part to all processes in its column. Each process that calculated a part of the k-th column in the D^{k-1}. D^{k-1} matrix has to send this part to all processes in its row.

B. Checkerboard Version

1) Cost matrix is divided into cells where each cell is of the size 1.
2) Each process read its own input data and store it.
3) There are n rounds and in each round the processes in ith row and ith column needs to send its data to all the processes in corresponding column or row.



```
func Floyd_All_Pairs_Parallel($D^{(0)}$) {
  for $k$ := 1 to $n$ do{
      Each process $p_{i,j}$ that has a segment of the k-th row of $D^{(k-1)}$,
      broadcasts it to the $p_{*,j}$ processes;
      Each process $p_{i,j}$ that has a segment of the k-th column of $D^{(k-1)}$,
      broadcasts it to the $p_{i,*}$ processes;
      Each process waits to receive the needed segments;
      Each process computes its part of the $D^{(k)}$ matrix;
      }
 }
```

Fig. 1. General Parallel Algorithm

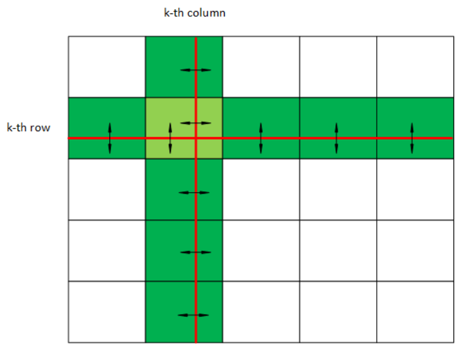


Fig. 2. Data dependencies in 2-D block mapping^{wiki}

4) After receiving the data each process start computes the distance.
5) After n iteration, each process had shortest path from itself to all other processes(SSSP).
6) Root Process will gather the data from all nodes and generate the final APSP.

C. Naive Parallel APSP

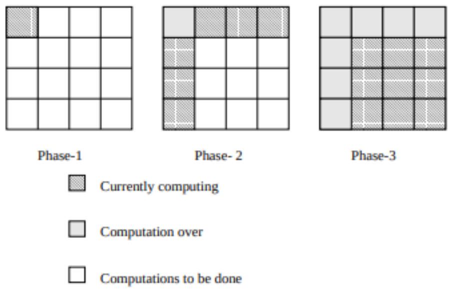
In the beginning we started with the basic Naive algorithm to analyze the time difference in serial and parallel APSP algorithm. The algorithm is as follows

1) Divide the graph into vertices and assign each vertex to one process(core).
2) Each process read its own input data(Vertex data) and store it.
3) There are n rounds with n processes and in each round the ith process needs to broadcast its data to all other process.
4) After receiving the data each process start computes the distance.
5) After n iteration, each process had shortest path from itself to all other processes(SSSP).
6) Root Process will gather the data from all nodes and generate the final APSP.

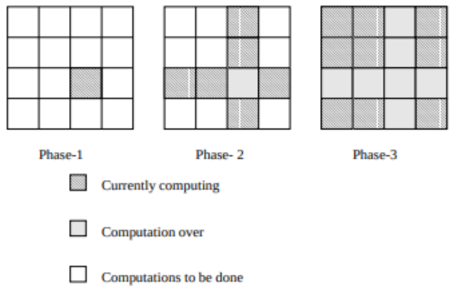
D. Blocked Parallel Version

1) Matrix is divided into blocks of the size $(n/p)^2$ and each process is given rows of blocks.
2) There are P iterations and each ith iteration proceed in three phases

- **Phase 1:** (i,i)th block is the pivot block for this iteration. This block first compute all-pair-shortest path and then broadcast updated blocks to all processes. Before broadcasting this process first update all blocks own by it using updated pivot block. There is no need to send to the corresponding row blocks because every process have its own row of blocks.
- **Phase 2:** After receiving updated pivot block, corresponding column blocks update their values using pivot block and itself by All pair shortest path.
- **Phase 3:** At last remaining blocks in every row update their values using updated block in corresponding pivot column, pivot row and itself by running All Pair Shortest Path algorithm.



(a) Phases when (1,1) is the self-dependent block



(b) Phases when block (t,t) is the self-dependent block

Fig. 3. [7]Phases of Blocked Parallel version

3) After P iteration, each process had shortest path from itself to all other processes(SSSP).
4) Root Process will gather the data from all nodes and generate the final APSP.

E. Random Graph Generation Algorithm

We generated our own input graphs with the following algorithm in python

1) Take the number of vertices(N) as user input
2) Generate the random number and store them in N*N array.
3) As we need the undirected graph, we make the matrix symmetric.

4) Make the diagonal elements as ’0’ as distance to self is zero.
5) Store these graph in a file in binary format

F. Graph Generation from Benchmark

We also written algorithm to generate the input graphs from SNAP (Stanford Network Analysis Platform) dataset particularly *soc-sign-bitcoin-otc*.

1) Take the name of data-set file, number of vertices(N) and Output file as user input.
2) Read the input data-set and tokenize according to the SNAP syntax.
3) Generate the adjacency matrix from the data.
4) Store these graph in a file in binary format.

IV. Experiments

We performed all our experiments on 2 different setups for Blocked Parallel APSP without barrier:

- 30 node cluster
- HPC 2010

Each of them is explained in details in next subsections A and B respectively.

A. 30 Node Cluster

In 30 nodes cluster, each node is powered with 12 cores intel i7 processors and 16 GB RAM. We used MPICH3 implementation of the MPI.

We generate the random input graphs of desired vertices using a python script.

We first used the naive Parallel APSP on graph of vertices from 4 to 1024. We will observe that time required in these algorithm for 1024 nodes is around 300 seconds which gets reduced to about 3 seconds in Blocked Parallel algorithm.

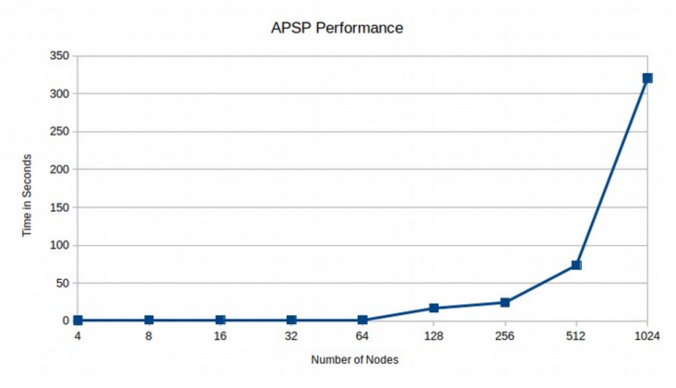


Fig. 4. Time for Naive APSP

We run the Blocked Parallel algorithm on different size graphs from 1024 to 16384 vertices and analyze them. Time for 1024 is very less (2 sec) as compared to 16384(7000 sec). So it is difficult to display all the results in 1 graph. Therefore, we show our results in different figures.

The below figures shows the time taken to run the Blocked Parallel algorithm on 1024, 2048, 4096, 8192 and 16384 vertices graphs respectively with 64, 128 and 256 processes with 12 ppn(process per node).

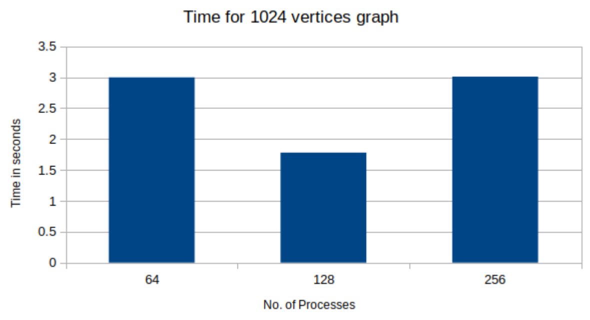


Fig. 5. Time for 1024 vertices graph

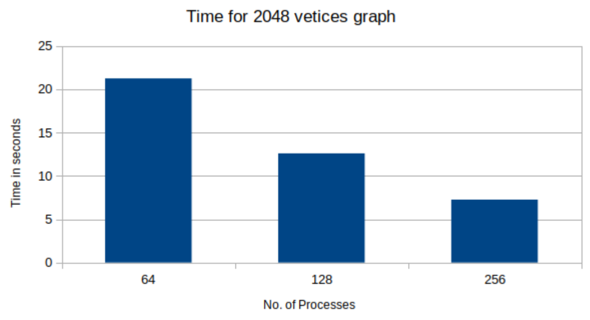


Fig. 6. Time for 2048 vertices graph

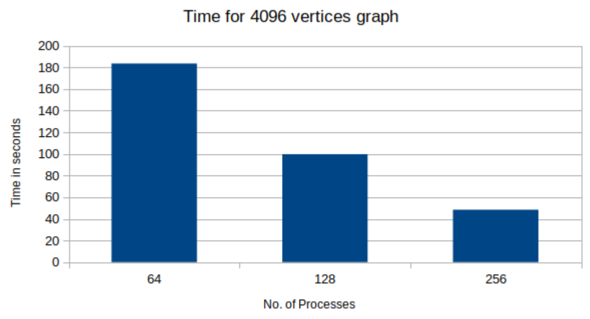


Fig. 7. Time for 4096 vertices graph

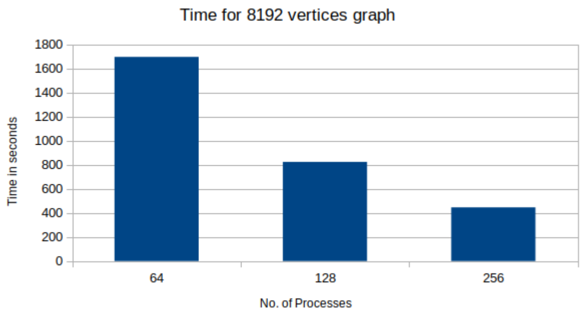


Fig. 8. Time for 8192 vertices graph

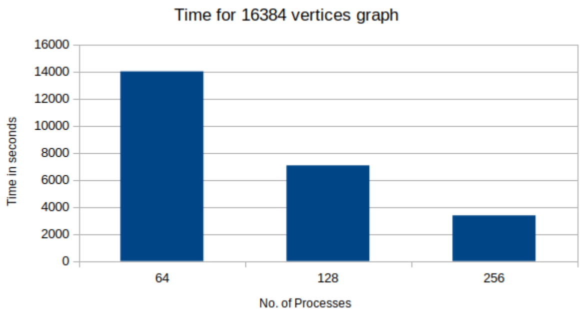


Fig. 9. Time for 16384 vertices graph

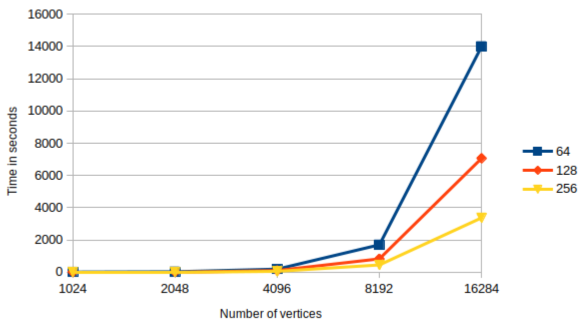


Fig. 10. Time on CSE cluster

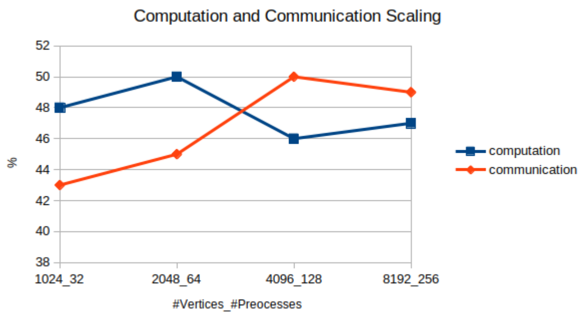


Fig. 11. Computation vs Communication

Observations:

- With increasing graph size, the difference in time becomes more significant.
- Running the algorithm with double number of processes for particular graph size decreases the time by approximately 50% . So, we can say that time required is inversely proportional with the number of processes.
- For 1024 vertices graph, the time increases when we increase the processes from 128 to 256, this is because the communication overhead is more than the computation in this case.
- Communication time increases if we increases number of vertices while keeping block size same and computation time almost same. This is because number of processes are increased.

B. HPC2010

We run the algorithm on HPC2010 also. It has 368 nodes, Intel Xeon X5570 2.93 GHz 2 CPU-Nehelam (8-cores per node. Each node has 48GB RAM.

We run the Blocked Parallel algorithm for 1024, 2048, 4096 and 8192 vertices graphs on 32, 64 and 128 processes with 8ppn. The time required is very less as compared to cluster.

We also tried running 16K and 32K nodes on HPC but we are not able to get the results for them.

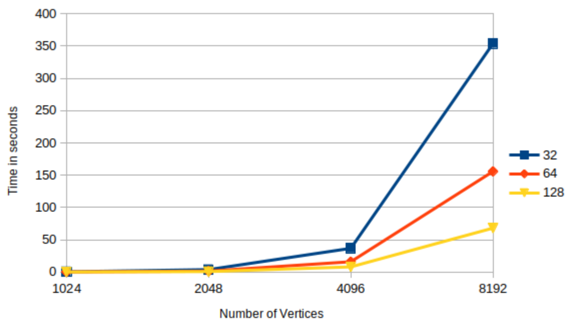


Fig. 12. Time on HPC

Observations:

- As we increase the number of processes for given No. of vertices the time decreases linearly.
- The difference is significant with increase in number of vertices in graph.

C. Profiling

We done the profiling of our algorithm using Tau. The profiling is generated with the same experiments.

The analysis shows that the time taken for the algorithm on large graphs is mainly

- Computation (upto 50 %)
- Communication (Broadcast) (upto 40-50 %)

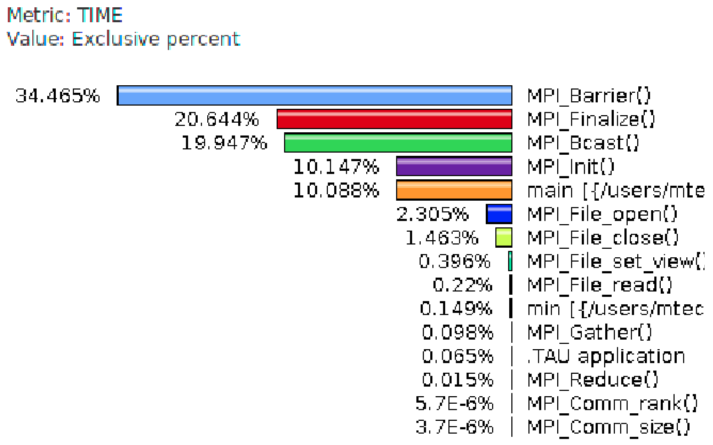


Fig. 13. Profiling of 1024 vertices graph with barrier

It is seen that Almost 34% spent in barrier which can be removes by minor changes because broadcast in next iteration implicitly work as barrier.

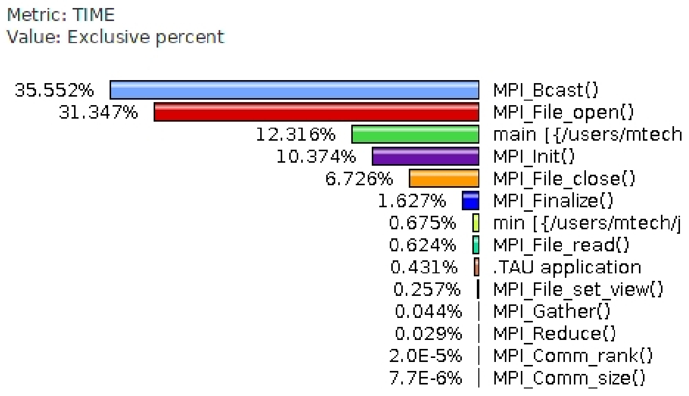


Fig. 14. Profiling of 1024 vertices with 256 processes

Above graph shows broadcast time increased because now processes wait for sender in broadcast instead of all the processes as in barrier.So,this implementation is similar to pipelined block parallel version.

We tried various ways to decrease the communication time, for example by removing the extra barriers without effecting correctness.

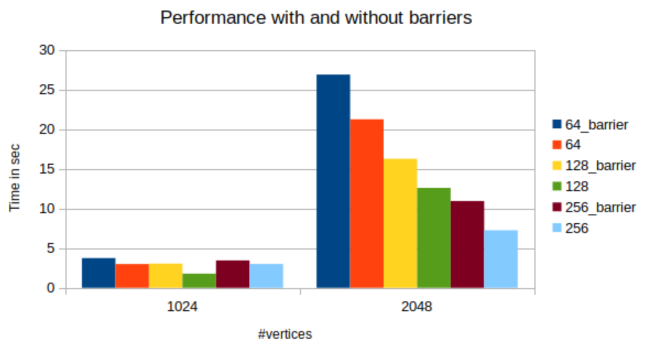


Fig. 15. Performance with and without barriers

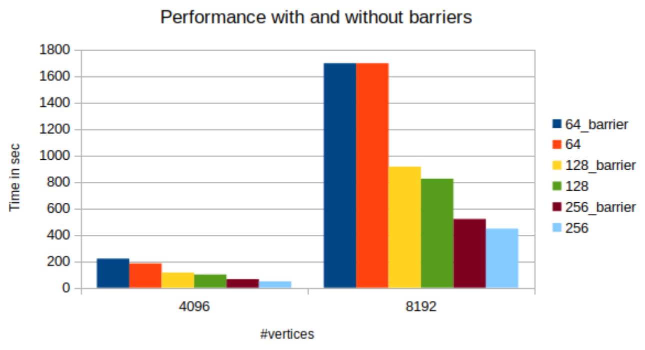


Fig. 16. Performance with and without barriers

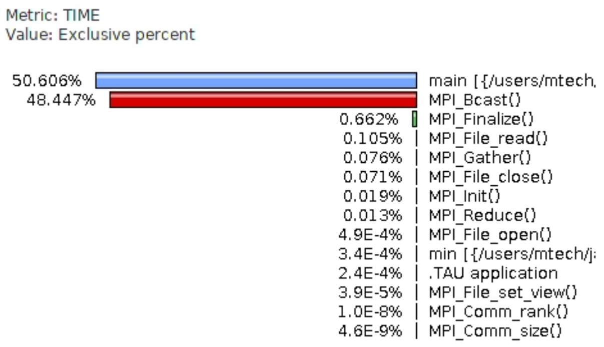


Fig. 17. Profiling of 16384 vertices 128 processes

As shown from above graph, When we run our algorithm on large dataset both computation and communication time contribute equally to total time.

It is clearly seen that our implementation without barrier greatly reduce time for all the cases.

V. Conclusion

There are various algorithm and optimizations for all pair shortest path.Some of the major optimizations are blocking,decomposition and pipelining in phases.We have seen that decomposition row wise decreases communication overhead.Further blocking improve computation time by making use of cache locality.We can overlap computation and communication time by avoiding synchronization at the end of each iteration and allowing processes to proceed as soon as they have their data to compute. [9]Discuss further optimization on performance which can be done by overlapping the phases in blocked parallel version. In blocked parallel version every node waits for pivot block to compute data and pivot block don't do anything in the remaining iteration after computing its value. Similarly, we can distribute load evenly amongst the nodes during a given iteration to the maximum extent possible by using increasing and decreasing phases.

References

[1] I. Robert W. Floyd Armour Research Foundation, Chicago, "Algorithm 97: Shortest path," 1962.

[2] T. M. Chan, "All-Pairs Shortest Paths with Real Weights in O($n^3/logn$)Time," 2008.

[3] F. Z. J. S. Wei Peng1, Xiaofeng Hu, "A Fast Algorithm to Find All-Pairs Shortest Paths in ComplexNetworks," 2012.

[4] R. Bellman, "On a routing problem," 1958.

[5] E. W. Dijkstra, "A note on two problems in connection with graphs," 1959.

[6] J.-F. JenqandSartajSahni, "Algorithm 97: Shortest path," 1987.

[7] S. S. G. Venkataraman and S. Mukhopadhyaya, "Blocked all pair shortest Path," 2003.

[8] V. Kumar, "Scalability of Parallel Algorithms for the All-Pairs Shortest-Path Problem," 1991.

[9] T. S. R. B. S. A. G. V. Hayawardh, "A scalable parallelization of all-pairs shortest path algorithm for a high performance cluster environment," 2007.

VI. APPENDIX

We both worked on project equally.We always discussed every point/steps and divided small amount of work rather than whole chunk.