15-418 Assignment 3

Haibin Lin, Yiming Wu
haibinl, wyiming
26 Feb 2016

Parallel Graph: Writeup

Dense and sparse representation

In our implementation, there're two representations of the same vertex set - sparse and dense. These two representation directly affects whether to take top down approach or bottom up approach in edgeMap.

For a sparse vertex set, the vertices in the set are represented in an array with their vertex id's. The valid length of the array is equal to the number of vertices in the set. For example, an array [1, 4, 5, 10] represents an array with vertices 1, 4, 5 and 10. To remove a vertex in the sparse vertex set, mark the original vertex with -1 is sufficient, since no vertex has negative id.

For a dense vertex set, the vertices in the set are represented by a bitmap - where 0 denotes an absent vertex, 1 denotes a present vertex. The length of the bitmap is always the number of all vertices in the graph. For a graph of 12 vertices, a dense vertex set with bitmap [0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0] represents a vertex set with vertices 1, 2, 3, 4, 6, 7, 8, 10, 11. We favour this kind of representation when the number of vertices present is large.

Implementation of vertexMap

If the input vertex set is sparse, the output vertex set will be sparse, too. We check all present vertices in parallel and apply f() to them. If that is successful, we update the vertex id at the same index in the result set. Otherwise, the vertex id is marked as -1 (absent) is result vertex set. The size is calculated with openmp reduction.

If the input vertex set is dense, we check all present vertices in the bitmap and apply f(). The bitmap in the output set is updated to either 0 or 1 depending the result of f(). The size is also accumulated via reduction. Either way we don't have any critical section in the parallel for loop.

During the parallel mapping step, we make the scheduling policy as dynamic, with chunk size 512. We chose dynamic because the workload across different threads could be imbalanced, dynamic scheduling will adjust the workload in run time to make sure each thread have similar amount of work. The chunk size is set to 512, so that different threads won't access the same cache line causing cache invalidation when the number of threads is large.

Implementation of edgeMap

If the input vertex set is sparse, we take the top down approach in edgeMap. The reason is with a sparse set, there're fewer outgoing edges and vertices to check. In the top down approach, we create a bitmap to store temporary result, and check the present vertices in parallel. For each present vertex, we get all the outgoing vertices and check cond() and apply update() to the outgoing vertex. If these are successful, the corresponding id in bitmap is set to 1 to indicate its presence.

If the input vertex set is dense, we take the bottom up approach. Now we check all the outgoing endpoint in parallel. For each endpoint, we first check if cond() is true, if not there's no need to do anything else further. Otherwise, we check if the incoming vertex is included in the input vertex

set, and apply update() accordingly. The qualified vertices are marked with 1 in the output bitmap, 0 otherwise.

Both top down and bottom up approaches store the result in a temporary bitmap. We tried several threshold to determine the criteria/threshold to convert the type back to sparse. In general, if the number of vertex in the set is small, it's better to convert the type back to sparse. Currently we set the threshold to number of vertices in the set / number of nodes in graph = 0.1. We also tried to include the number of outgoing edges of the vertex set, but the performance we get is not significantly different. Therefore we chose the simpler threshold at the end.

Dense to sparse conversion

Regarding transform a bitmap back to array, prefix sum is used to collect the vertices present in the bitmap. Different from what we did in the previous assignment, we implemented the "scan on larger array" method mentioned in the lecture notes "Parallel Programming Case Studies".

In this implementation, the bitmap is divided into chunks. The number of chunk is equal to the number of threads available. For each chunk, we first do a sequential scan on the chunk to gather a local prefix sum. After this, local prefix sum results are combined to get the global prefix sum base for each chunk. Finally the we adjust all the local prefix sum according the to base. With this, we can easily collect all the vertex which appears in the set.

Synchronization

In our implementation, little synchronization is needed to achieve vertexMap and edgeMap in parallel. Within the parallelized for loop, there's no critical section. There's some synchronization to get the accumulated size with openmp reduction(i.e. reduce the result once all threads are done), but the synchronization overhead is not significant.

On the other hand, we chose dynamic scheduling policy to perform the parallel for loop, which involves some synchronization among all the threads to achieve work stealing. But again, the synchronization overhead is not significant, and works better compared to static scheduling, where workload imbalance may often occur.

Data movement and memory

Perfect speed up is not achieved partially due to cache invalidation. In the top down approach, it's possible that two threads are updating the status of same endpoint vertex, which will cause one of the thread to flush its write and drop its cache line. Also, one thing we noticed is that memset is essentially linear, which doesn't scale with the number of threads. Therefore, we implemented a pmemset() function to do memset in parallel threads.

On the other hand, we try to avoid the number of times of invoking malloc(), which may trigger a syscall to acquire more memory space when needed. Instead of malloc() and free() all the time, we get and reuse arrays/bitmaps from a memory pool, and put it back when done.