Breadth First Search in Cilk++

by Vegar Engen (vegaen) and Arne Dahl Bjune (arnedab)

Project Introduction

For our project we did a bfs implementation in Cilk++ by using the graph500 specification. The breadth-first-search is an algorithm that explores the vertecies and edges of a graph, beginning from a particular starting vertex. To do BFS in a sequential algorithm is a remarkably simple operation, but to do it in parallel makes it challenging. The way we solved the problem was to split the nodes we wanted to traverse into a bag in each thread walk the nodes and add unvisited nodes to the bag. Then we needed to merge all our bags before doing the next level.

A nicer version of this file can be found on http://github.com/vegaen/cilk-bfs

Table of Contents

BFS

The breadth-first-search is an algorithm that explores the vertecies and edges of a graph, beginning from a particular starting vertex. After bfs is done it returns a tree of the graph, which tells you how far away any node is from the root node. To do this sequencial this is a rally simple problem since the algoritm is really easy:

```
procedure BFS(G,v):
  create a queue Q
  enqueue v onto Q
  mark v
  while Q is not empty:
    t ← Q.dequeue()
    if t is what we are looking for:
      return t
    for all edges e in G.adjacentEdges(t) do
      u ← G.adjacentVertex(t,e)
      if u is not marked:
      mark u
      enqueue u onto Q
```

When you want to make this problem a parallel problem the main issue is how you shall manage the queue. When you in a sequential program can add a new node to the back of the queue and keep going, you can't do the same in a parallel program since you have to keep track of which nodes that have been visited.
A high order approach to the problem is to split the queue between the processors and walk the neighbours of that vertex and put new obtained nodes into a shared next level queue and wait for every processor to be finished before you start on the next level.

Graph500

Graph500 is a rating of supercomputer system, with focus on Data intensive loads. Instead of counting double precision floating-point as the benchmark, Graph500 are using a breadth-first search as the benchmark. In the benchmark it is two computation kernels. The first kernel computes the time it takes to generate the graph and compress it into sparse structures CSR or CSC (Compressed Sparse Row/Column). The second kernel does a parallel BFS of 64 random vertecies per run.

Six possible scales, or sizes, are defined: * **Toy**: 2^26 vertices -> 17 GB of RAM * **Mini**: 2^29 -> 137 GB of RAM * **Small**: 2^32 -> 1,1 TB of RAM * **Medium**: 2^36 -> 17,6 TB of RAM) * **Large**: 2^39 -> 140 TB * **Huge**: 2^42; 1.1 PB of RAM

Implementation

Bag Structure

We used a long time before we was able to decide on the structure we wanted to use for our bag. After a while we figured out our bag needed certain properties: * No duplicates * Fast insert * Fast merge * Fast split

As far as we coudn't find any single data structur that fitted all of this properties, and most of them fit one or two properties, but are horrible at the others. For example a heap is really easy to insert into, but are not able to reject duplicates. While it is quite easy to secure a binary three against duplicates, but again a insert is really expencive. The solution we ended up with was quite clever, each processor gets it own bag which is a heap, but when they they merge the bag structure becomes a sorted list.

No duplicates

Since we mark a visited node, a single processor cannot try to add duplicates to the bag. The only way to get duplicates is it two (or more) different processors access read the same node at the same time and think it isn't visited earlier. By using this knowledge we do not need to handle duplicates for one single bag -> we can insert as a heap.

Fast insert

Since insert is an operation we do often, it is crucial that it is fast. Wheater or not the bag structure is a heap, it will always insert an element at the end, but if it is a heap it will min-heapify afterwards. So the insert operation will ether be O(1) or O(log n), which is reasonably fast.

Get min-value fast

Since we need the min-value of a bag in the merge it is crucial that getting the min-value is reasonably fast. When the bag is a heap, it is O(1) to peek at the min-value, but it is O(log n) to pop it. When the bag is a sorted list it is O(1) to peek and pop.

Fast merge

When we merge we do kind of a merge sort, we compare the two smallest elements in each of the bags and inserts the smallest element in a new bag (which is actually one of the old bags, to save initiation time of a new bag). If the smallest element of each bag is equal only one value is added, but we go further in both of the bags. When both of the bags are heaps a merge operation is O(n+m+log n + log m), and when both is sorted list a merge operation is O(n+m).

Fast split

Since we dont care how to split the vertecies between the processors, we do a really simple split operation. Processor 1 gets the first 1/nproc part of the vertecies and so on. Then the split operation is O(1)

Overall our bag structure is fast enough and we have a design without the need for reducers. Experience from earlier homework tells us that the reducers scale good but their performance is worse than what you can code yourself.

Build and Run

make

This will provide 2 files, bfs and bfs500.
bfs500 will output only which nodes it searches from and statistics when its finished running. Bfs will also gives a output containing the number of vertecies found on different levels and some other debuging info, but no statistics.

Usage: bfs [Options] BFS Search in Cilk

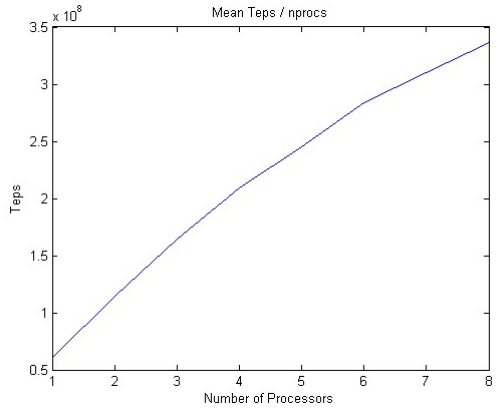
```
 -r    Reads from stdin
-g    Generates random edges (default mode)
-s    Scales, 2^Scales number of vertices (default 20)
-e    Edgefactor, average number of endges for each vertex
-n    The number of nodes to search from (default 64)
-h    Prints this helpfile
```

Results

Performance for graph search is mesured in Traversed Edges Per Second (TEPS)

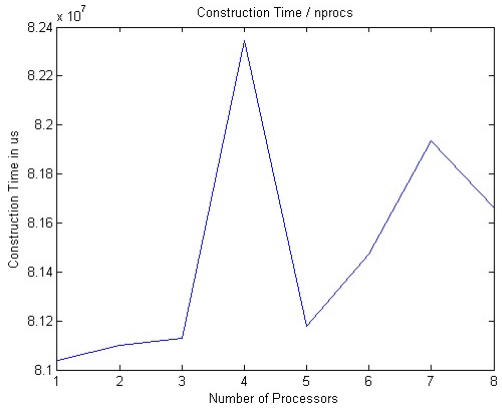
Teps vs Proccessor Count

This graph shows how TEPS scale as the number of cores is increased from 1 to 8 on the search. The values comes from an average of 64 runs.



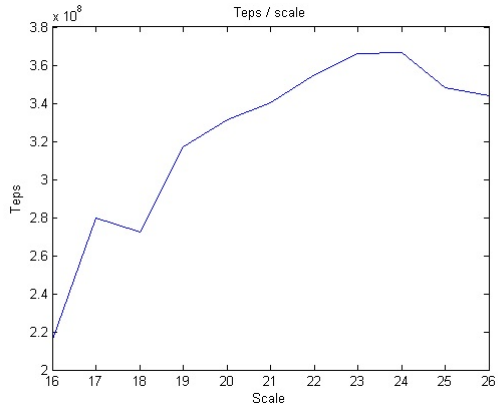
As we can se TEPS increases almost linear as the number of processors increas

Here we look at the time it takes to generate the graph from the edgelist.



We can clearly see that the graph generation algorith is not parallelized and has no performace gain from increasing the number of processors.

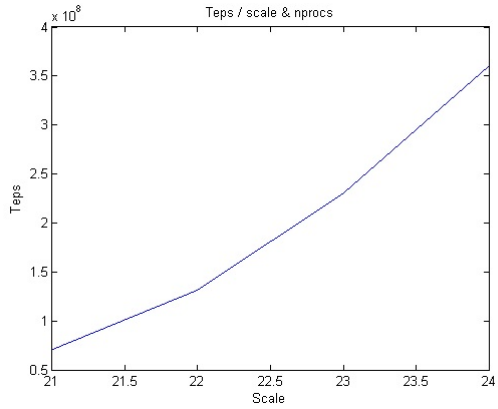
Teps vs Scale



We can se that the performace increases as the problem size increases except for a slight decrease for the two largest problem sizes. But that can also be because timing is inaqurate for small problem sizes. Unfortunetly we did not have time to run tests on multiple nodes. But we expect that performace would decrease due to reduced memory performance when running on multiple nodes.

Teps vs Scale and Proccessor Count

In this graph we look at how TEPS change as both problem size and number of processors double for each datapoint. Scale ranges from 21 to 24 and proccessors from 1 to 8.



Since both increasing scale and increasing processor count increases performace this graph shows as expected a better than linear increase in performance.

Example Output

This is a example output for run with scale 26, edgefactor 26 on a single core on Triton.

construction_time: 8.10384640000000000e+07us

min_time: 1.75544230000000000e+07us
firstquartile_time: 1.75830920000000000e+07us
median_time: 1.76162025000000000e+07us
lastquartile_time: 1.76431775000000000e+07us

max_time: 1.82889140000000000e+07us
mean_time: .png 1.76253703593750000e+07us

min_nedges: 1.07374182400000000e+09
firstquartile_nedges: 1.07374182400000000e+09
median_nedges: 1.07374182400000000e+09
lastquartile_nedges: 1.07374182400000000e+09
max_nedges: 1.07374182400000000e+09
mean_nedges: 1.07374182400000000e+09

min_TEPS: 5.87099826703761667e+07TEPS
firstquartile_TEPS: 6.08587563572206497e+07TEPS
median_TEPS: 6.09519458649718761e+07TEPS
lastquartile_TEPS: 6.10667234838191941e+07TEPS
max_TEPS: 6.11664540000000000e+07TEPS
mean_TEPS: 6.09218245382609293e+07TEPS

Problems

- We had a lot of problems when we tried to take the time of the graph search and since all the searches was sub-second, it was hard to get an accurate time.
- We also had problems getting time on the large memory nodes on Triton to run on lager scale problems.

Conclusion

We went through several different design choises before ending up with the final design. We are happy with our implementation and performacewise we are good enough to at least not be at the bottom of the graph500 list.

Future Work

Our current code does not verify that the result from the bfs search is correct. Since the graph problems is very large searching the graph to get an guaranteed verification. The suggested way to verify the graph search is to check that the BFS tree that what found c

Sources

- http://graph500.org
- http://en.wikipedia.org/wiki/Graph500