Suffix/Prefix Minima

Christos Froussios (4322754)

Files and Usage

The are 2 source files.
openmp.cpp contains the implementation in OpenPM
minima.cpp is based on the template and contains

1. A sequential implementation
2. A implementation based on pthreads

The provided makefile produces:

1. *openmp*: The openmp executable
2. *openmpseq:* The openmp program compiled without OpenMP enabled
3. *minima*: which executes both the sequential and the PThreads implementations

All programs execute the algorithm on randomised input, with different input sizes and different number of threads, and print the runtimes as seen in the experiments.

minima can take the command “correctness”. In that case, it will perform one small test and print the input and the output, so that the user can verify correct execution.

Note that the OpenMP implementation was very inefficient when run in Ubuntu Linux, but normal when run in Windows.

The algorithm

The algorithm expects input sizes in powers of 2. However, any other size can be put on an array of the expected size and filled with the maximum integer value. The algorithm also requires that there are no duplicate items in the input.

It uses two balanced binary trees: *expand* and *reduce* (not corresponding to a reduce).
Expand's leafs is the input array. The rest of the tree is constructed such that every node is contains the minimum value of all its children.
Reduce contains in each level the preffix/suffix minima for the corresponding *expand* level. The final level (leafs) is the solution, since the corresponding *expand* level was the input.

The suffix and prefix problems are symmetrical. If the nodes on each level of the binary trees are traversed in reverse order, an algorithm that gives the suffix minima will produce the prefix minima and vise versa. Therefore, the same algorithm is run twice, where the tree we called "reduce" is either *suffix* or *prefix*.

The binary trees are implemented using arrays. Depth 0 in the tree is the first item, depth 1 is the next 2, depth 3 is the next 4 etc. Accessing the n-th item on the h-th level of the binary tree is accomplished in O(1) time. The function that calculates the appropriate index, mapping the tree onto the array, is *index(depth, i)*. A function *index_rev* will traverse the same level on the same tree in reverse order.

```
for (h=logn-1 ; h>=0 ; h--) {                                   T = O(log(n))   W = (n*log(n))
        leveln /= 2;
        for (int i=0 ; i<leveln ; i++) pardo {                  T = O(1)    W = (n)
                expand[index(h, i)] = min(expand[index(h+1, 2*i)], expand[index(h+1, 2*i+1)]);
        }
}

leveln = 1; // The number of elements in this depth (i.e. h+1^2)
for (h=0 ; h<=logn ; h++) {                                     T = O(log(n))   W = (n*log(n))
        for (i=0 ; i<leveln ; i++) pardo {                      T = O(1)    W = (n)
                if (i == 0)
                        prefix[index(h, i)] = expand[index(h, i)];
                else if (i % 2 == 1)
                        prefix[index(h, i)] = prefix[index(h-1, i/2)];
                else {
                        if (expand[index(h, i+1)] != prefix[index(h-1, i/2)])
                                prefix[index(h, i)] = prefix[index(h-1, i/2)];
                        else
                                prefix[index(h, i)] = prefix[index(h, i-1)];
                }
        }
        leveln *= 2;
}

leveln = 1;
for (h=0 ; h<=logn ; h++) {                                     T = O(log(n))   W = (n*log(n))
        for (i=0 ; i<leveln ; i++) pardo {                      T = O(1)    W = (n)
                if (i == 0)
                        suffix[index_rev(h, i)] = expand[index_rev(h, i)];
                else if (i % 2 == 1)
                        suffix[index_rev(h, i)] = suffix[index_rev(h-1, i/2)];
                else {
                        if (expand[index_rev(h, i+1)] != suffix[index_rev(h-1, i/2)])
                                suffix[index_rev(h, i)] = suffix[index_rev(h-1, i/2)];
                        else
                                suffix[index_rev(h, i)] = suffix[index_rev(h, i-1)];
                }
        }
        leveln *= 2;
}
```

The total complexity of the algorithm is *T = O(log(n)), W = O(n*log(n))*

Experiments

The testing system was an 8-core system (Intel Core i7-4710HQ) with 8GB RAM running Ubuntu Linux. The algorithm was run 100 times on each input and the numbers presented are the sum. The O(n) construction of a randomized input was counted towards the execution times, because it needs to be copied to a specific location in the *expand* array, for the algorithm to applicable. Therefore, the complexity for the algorithm in the experiments is $T = O(\log(n)), W = O(n*\log(n))$.

Runtimes

NSize	#	Seq	1 thread	2 threads	4 threads	8 threads	16 thread
4096	100	0.024	0.024	0.026	0.043	0.103	0.215
8192	100	0.054	0.046	0.036	0.048	0.111	0.234
16384	100	0.091	0.087	0.060	0.069	0.129	0.265
32768	100	0.183	0.172	0.102	0.101	0.163	0.313
65536	100	0.360	0.342	0.190	0.174	0.215	0.391
131072	100	0.716	0.681	0.371	0.344	0.270	0.556
262144	100	1.479	1.391	0.755	0.624	0.475	0.878

Speedup

NSize	#	Seq	1 thread	2 threads	4 threads	8 threads	16 thread
4096	100	1	1.01	0.93	0.55	0.23	0.11
8192	100	1	1.20	1.53	1.13	0.49	0.23
16384	100	1	1.05	1.52	1.33	0.70	0.34
32768	100	1	1.06	1.79	1.81	1.12	0.58
65536	100	1	1.05	1.89	2.07	1.67	0.92
131072	100	1	1.05	1.93	2.08	2.65	1.29
262144	100	1	1.06	1.96	2.37	3.11	1.68

Efficiency

NSize	#	Seq	1 thread	2 threads	4 threads	8 threads	16 thread
4096	100	1	1.01	0.46	0.14	0.03	0.01
8192	100	1	1.20	0.76	0.28	0.06	0.01
16384	100	1	1.05	0.76	0.33	0.09	0.02
32768	100	1	1.06	0.89	0.45	0.14	0.04
65536	100	1	1.05	0.94	0.52	0.21	0.06
131072	100	1	1.05	0.97	0.52	0.33	0.08
262144	100	1	1.06	0.98	0.59	0.39	0.11

The most noticeable result is that, for small inputs, introducing more workers slows down the execution. The cost of managing even just two threads is greater than the benefit processing the data in double the speed. To better understand that, one needs to consider that a barrier exists at the end of each level in the tree. While 4096 would seem enough that parallelism would be a benefit, the level in the tree are of much smaller size. The input need to be more than 8K for 2 threads to be of benefit, and over 32K for 8 threads to be of benefit. For 2 threads and more, efficiency increases with the size of the input.

A noticeable result is that the parallel version with 1 thread is slightly, but consistently, more efficient that the sequential. While the code executed is theoretically the same, with the parallel version suffering from unnecessary barriers, the parallel version was somehow able to better use the system's resources. We have no explanation for that difference in speed.