List distances

Christos Froussios (4322754)

Files and Usage

The are 2 source files.
openmp.cpp contains the implementation in OpenPM
distance.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. *distances*: 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.

distance 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.

Algorithm

Unlike the problem description, the algorithm here uses 0-based indexing (-1 denoting NULL), to be more like the C language.

```
distances(list, n) {
    for (i = 0; i < n; i++) pardo {
        distance[i] = distanceNew[i] = (list[i] == -1) ? 0 : 1;
        jumplist[i] = jumplistNew[i] = list[i];
    }

    for (int y = 0; y < logn ; y++) {
        for (i = 0; i < n; i++) pardo {
            if (jumplist[i] != -1) {
                distanceNew[i] = distance[i] + distance[jumplist[i]];
                jumplistNew[i] = jumplist[jumplist[i]];
            }
            else {
                distanceNew[i] = distance[i];
                jumplistNew[i] = jumplist[i];
            }
        }
        swap(jumplist, jumplistNew);
        swap(distance, distanceNew);
    }

    return distances;
}
```

Explanation

The algorithm performs pointer jumping. *jumplist[]* contains each node's next node. *distance[]* contains the distance from the node to its current next node. In each round of pointer jumping, the new values are written to a new container to avoid race conditions, where a node and its next are handled by different threads.

Correctness

The algorithm is correct if

1. After each round, distance[] contains the distance from the node to its next node.
2. After the last round, every node points to the end of the list (-1).

Proof for 1:
Initialisation: Every node's next is its actual next in the original list, which is 1 jump away. The last node is initialised to 0, because its 0 steps away from the end.
Step: The distance from a node to its next's next is the sum of the distance from the node to its next and from its next to the next's next.

Proof for 2:
The node that needs the most jumps to reach the end is the beginning of the list. That node's distance from the end is equal the length of the list *n*. In every subsequent round, the distance covered by a jump doubles. Therefore, it take log(n) to pointer jumps for the first node to point to the end of the list.

Complexity

We will analyze the algorithm with the Worker-Time paradigm. Let n be the size of the list. Swapping is done by reference and has complexity O(1).

```
distances(list, n)                                          T = O(log(n))  W = O(n+n*lon(n))
    for (i = 0; i < n; i++) pardo {                         T = O(1)  W = O(n)
        distance[i] = distanceNew[i] = (list[i] == -1) ? 0 : 1;
        jumplist[i] = jumplistNew[i] = list[i];
    }

    for (int y = 0; y < logn ; y++) {                       T = O(log(n))  W = O(n*lon(n))
        for (i = 0; i < n; i++) pardo {                     T = O(1)  W = O(n)
            if (jumplist[i] != -1) {
                distanceNew[i] = distance[i] + distance[jumplist[i]];
                jumplistNew[i] = jumplist[jumplist[i]];
            }
            else {
                distanceNew[i] = distance[i];
                jumplistNew[i] = jumplist[i];
            }
        }
        swap(jumplist, jumplistNew);
        swap(distance, distanceNew);
    }

    return distances;
}
```

The total complexity for the algorithm is
T = O(log(n))
W = O(n + 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. This means that and Time and Work complexity in the experiments is:
T = O(n + log(n))
W = O(n + n*log(n))

Runtimes

NSize	#	Seq	1 thread	2 threads	4 threads	8 threads	16 thread
4096	100	0.044	0.045	0.032	0.020	0.023	0.076
8192	100	0.097	0.098	0.062	0.045	0.038	0.101
16384	100	0.213	0.220	0.121	0.093	0.071	0.132
32768	100	0.515	0.535	0.300	0.228	0.150	0.255
65536	100	1.245	1.269	0.728	0.536	0.337	0.538
131072	100	2.899	2.907	1.631	0.895	0.744	1.116
262144	100	6.775	6.801	3.640	2.351	1.637	2.312

Speedup

NSize	#	Seq	1 thread	2 threads	4 threads	8 threads	16 thread
4096	100	1	0.98	1.38	2.20	1.91	0.58
8192	100	1	0.99	1.56	2.16	2.55	0.96
16384	100	1	0.97	1.76	2.29	3.00	1.61
32768	100	1	0.96	1.72	2.26	3.43	2.02
65536	100	1	0.98	1.71	2.32	3.69	2.31
131072	100	1	1.00	1.78	3.24	3.90	2.60
262144	100	1	1.00	1.86	2.88	4.14	2.93

Efficiency

NSize	#	Seq	1 thread	2 threads	4 threads	8 threads	16 thread
4096	100	1	0.98	0.69	0.55	0.24	0.04
8192	100	1	0.99	0.78	0.54	0.32	0.06
16384	100	1	0.97	0.88	0.57	0.38	0.10
32768	100	1	0.96	0.86	0.56	0.43	0.13
65536	100	1	0.98	0.86	0.58	0.46	0.14
131072	100	1	1.00	0.89	0.81	0.49	0.16
262144	100	1	1.00	0.93	0.72	0.52	0.18

Doubling the size of the input leads to more than double the execution time, which is expected, since the Work complexity has a *n*log(n)* part along with a linear part.

We can see that doubling the number of threads available for processing does not double the speed. This happens for two reasons. First is the fact that the O(n) initialisation of the input in each iteration was counted towards the time. This initialisation is sequential and unaffected by the degree of parallelism. The second reason is the cost of synchronising the threads: every round of pointer-jumping has an implicit barrier before the next. The impact of the barrier is most visible when doubling the threads from 8 to 16, where we are not adding to the processing capacity (the system supports up to 8 threads), but still need to synchronise the extra threads.

The significant overhead cost imposed by thread management can be seen in the efficiency table. As input size increases, so does the efficiency. The number of barriers executed in the program is O(log(n)), while the amount of work is O(n + n*log(n)).

Lessons learned

This algorithm needs internal barrier to eliminate race conditions. When doing the first experiments, the barriers were, leading to “cleaner” execution speedup, but erroneous input. After correcting the mistake, runtimes and speedup suffered noticeably. This demonstrated to me that the cost synchronising threads can be significant, even though it is usually not discussed when analysing parallel implementations.