Merge

Christos Froussios (4322754)

In this report, we present a parallel algorithm for merging two arrays A and B into an array AB. The arrays A and B are assumed to be sorted and their union is assumed to contain no duplicate elements. The resulting array AB will also be sorted and contain no duplicate elements.

Files and Usage

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

merge 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

```
merge(A[], B[], AB[]) {
    for(i=0 ; i<size(A) ; i++) pardo {
        rank_ab = i + 1 + binaryRank(B, 0, size(B), A[i]);
        AB[rank_ab - 1] = A[i];
    }

    for(i=0 ; i<size(B) ; i++) pardo {
        rank_ab = i + 1 + binaryRank(A, 0, size(A), B[i]);
        AB[rank_ab - 1] = B[i];
    }
}

binaryRank(a[], start, end, item) {
    if (start == end - 1)
        if (a[start] <= item)
            return start + 1
        else
            return start

    mid = (start + end) / 2;
    if (item >= a[mid])
        return binaryRank(a, mid, end, item);
    else
        return binaryRank(a, start, mid, item);
}
```

Explanation / Correctness

1) In a zero-indexed sorted array A with no duplicate elements, an item's A[i] rank is
rank(A[i]: A) = i + 1
because the number of items that are lesser is i (items A[0] to A[i-1]) and there is exactly one item equal to A[i].

2) To find *rank(b: A),* where b doesn't have a known position in A, we can use binary search to find the position.

3) Reversing 1), if an item A[j] exists in A and its rank is known, then we know its index:
j = rank(A[j]: A) - 1

4) Let *c* be an item from A. This means that c does not exist in B and exists exactly once in AB (as per the initial assumptions about the input). We can calculate its rank in AB as such:
rank(c: AB) = (# of items in AB that are less than c) + 1 =
(# of items in A that are less than c) + (# of items in B that are less than c) + 1 =
(rank(c: A) - 1) + (rank(c : B)) + 1 =
rank(c: A) + rank(c: B)
The above also holds if the item is from B.

In the algorithm, we use 1) and 2) to calculate rank of an item in A and in B. We used those two values and 4) to calculate the rank of the item in AB. We use 3) to position the item in the merged array, based on its rank.

Complexity

We will analyze the algorithm with the Worker-Time paradigm. Let n be the size of A and m be the size of B.

The complexity for sequential binary search is considered known and is O(log(n))

```
merge(A[], B[], AB[]) {
        for(i=0 ; i<size(A) ; i++) pardo {                      T = O(log(m))  W = O(n * log(m))
                rank_ab = i + 1 + binaryRank(B, 0, size(B), A[i]); T = W = O(log(m))
                AB[rank_ab - 1] = A[i];
        }

        for(i=0 ; i<size(B) ; i++) pardo {                      T = O(log(n))  W = O(m * log(n))
                rank_ab = i + 1 + binaryRank(A, 0, size(A), B[i]); T = W = O(log(n))
                AB[rank_ab - 1] = B[i];
        }
}
```

The total complexity for the algorithm is
T = O(log(n) + log(m))
W = O(n*log(m) + m*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 not counted towards the execution times.

Runtimes

NSize	#	Seq	1 thread	2 threads	4 threads	8 threads	16 thread
4096-1024	100	0.048	0.051	0.027	0.016	0.010	0.017
8192-2048	100	0.105	0.105	0.053	0.030	0.023	0.038
16384-4096	100	0.216	0.216	0.114	0.072	0.046	0.070
32768-8192	100	0.462	0.468	0.235	0.140	0.109	0.140
65536-16384	100	0.978	0.976	0.499	0.321	0.259	0.257
131072-32768	100	2.085	2.100	1.072	0.675	0.500	0.463
262144-65536	100	4.518	4.510	2.307	1.416	1.010	0.971

Speedup

NSize	#	Seq	1 thread	2 threads	4 threads	8 threads	16 thread
4096-1024	100	1	0.93	1.77	2.97	4.55	2.74
8192-2048	100	1	1.00	1.97	3.55	4.48	2.79
16384-4096	100	1	1.00	1.89	3.02	4.65	3.09
32768-8192	100	1	0.99	1.96	3.31	4.25	3.30
65536-16384	100	1	1.00	1.96	3.05	3.78	3.81
131072-32768	100	1	0.99	1.95	3.09	4.17	4.50
262144-65536	100	1	1.00	1.96	3.19	4.47	4.65

Efficiency

NSize	#	Seq	1 thread	2 threads	4 threads	8 threads	16 thread
4096-1024	100	1	0.93	0.88	0.74	0.57	0.17
8192-2048	100	1	1.00	0.99	0.89	0.56	0.17
16384-4096	100	1	1.00	0.95	0.75	0.58	0.19
32768-8192	100	1	0.99	0.98	0.83	0.53	0.21
65536-16384	100	1	1.00	0.98	0.76	0.47	0.24
131072-32768	100	1	0.99	0.97	0.77	0.52	0.28
262144-65536	100	1	1.00	0.98	0.80	0.56	0.29

We see that efficiency is approximately 1 for 2 threads, but falls when the number of threads is further increased. The decrease in efficiency is greatest when the threads are increased from 4 to 8. This is possibly because 8 cores is the maximum capacity for the system and at 8 threads, the experiment is competing with other processes present in the system, thus 8 cores aren't truly 100% available.

Increasing the number of threads to 16 offers no significant improvement since the system doesn't support it. However, there is a slight, yet noticeable trend, where 16 threads offer less speedup than 8 for lower input sizes and more for larger sizes. This persisted in our experiments. A possible explanation for this is that the presence of additional threads increases the percentage of resources that the program get when competing with other applications, but it takes input of a certain size for this effect to become more significant than the overhead of having more threads.

Finally, we can see that efficiency is unaffected by the size of the input, which shows that the overhead of managing threads is insignificant compared to the gains from parallel processing.

Alternative version

The algorithm, as it was presented, can take advantage of up to *max(size(A), size(B))* threads, with diminishing gains for threads over *min(size(A), size(B))*. This can be improved to take advantage of up to *size(A)* + *size(B)* threads by merging the two separate loops into one.

```
for(i=0 ; i<size(A) + size(B) ; i++) pardo {
        if (i<size(A)) {
                rank_ab = i + 1 + binaryRank(B, 0, size(B), A[i]);
                AB[rank_ab - 1] = A[i];
        }
        else {
                j = i - size(A)
                rank_ab = j + 1 + binaryRank(A, 0, size(A), B[j]);
                AB[rank_ab - 1] = B[j];
        }
}
```

This version was not prefered because the decision clause adds to the total amount of work, while the algorithm offers no actual benefit in the experiments, since the number of threads available is always much lower. We expect that the algorithm used is faster for the ranges in the experiment, but no tests were performed to verify this.