Parallel Bucket-Sort

Name: Adriano
Surname: Cardace
Date: 03/02/2017

Abstract:

This report compares two different implementation of a sorting algorithms called Bucket-Sort.
Firstly, an efficient sequential solution is provided, then a parallel version of the same algorithm will be explained in detail. In both cases there will be an evaluation of the performance.

Table of contents:

1)Introduction
2)Sequential bucket-sort
3)Testing
4)Analysis of the running time
5)Parallel bucket-sort
6)Scalability.
7)Analysis of the running time

1. Introduction

The purpose of this is assignment is to implement and test a sequential and a parallel algorithm for bucket-sort.
The presented solution are implemented in C ad parallelized using OpenMP; every running test is done on brake.ii.uib.no.
In order to get the best running time the programs are compiled with GCC using the -O3 flag (the initial time spent on memory allocation and on generating the data is not included).
Each run is done five times, and only the best time is reported.
The data used for testing the algorithms is generated randomly using the C function rand(), with every integers in the range from 0 up to 99 999.
As validation, both programs has a final test to assert that the data is sorted.

2. Sequential bucket-sort

2.1 **Steps of the sequential version:**

a)Variables definition and input
b)Count the number of elements for each bucket.
c)Store each element of the array to sort in the proper bucket
4)Sort local bucket

a)Variables definition and input

nd is given by the user.
ndomly generated integers will be stored.
ll the buckets will be stored, in fact there is no need to allocate more memory
 elements is *dim*.
 value is given by the user.

is possible that the last bucket has a greater width if *limit* is not evenly divided

truct bucket*.*
n_elem, *index and start*.
red in the i-th bucket, *start* is the starting position of the i-th bucket in B, this is
ents is *dim*, every item of each bucket will be stored directly in B.
t integer of the i-th bucket will be stored, so the range for *index* is [*start* : *start*

b) Counting the number of elements for each bucket

```
for (i=0; i<dim;i++){
        j = A[i]/w;
        if (j>n_buckets-1)
                j = n_buckets-1;
        buckets[j].n_elem++;
    }
```

The inner *if* is necessary to cover all the range, for instance if *limit*=100 and *n_buckets*=7, w=limit/n_buckets=14, but 14*7=98, so in this particular case, 98 and 99 will be stored in the last bucket.

c)Store each element of the array to sort in the proper bucket

Before starting to store the buckets in the temporary array B, is necessary to set up the right start, index values for each bucket.
The first one has start=0 and index=0, the second one has start=buckets[0].start + buckets[0].n_elem and index=buckets[0].index + buckets[0].n_elem and so on.

```
buckets[0].index=0;
buckets[0].start=0;
for (i=1; i<n_buckets;i++){
         buckets[i].index = buckets[i-1].index + buckets[i-1].n_elem;
         buckets[i].start = buckets[i-1].start + buckets[i-1].n_elem;
 }
```

After that, all items of each bucket are stored in B in the right position.

```
int b_index;
    for (i=0; i<dim;i++){
        j = A[i]/w;
        if (j > n_buckets -1)
                j = n_buckets-1;
        b_index = buckets[j].index++;
        B[b_index] = A[i];
    }
```

4)Sort local bucket

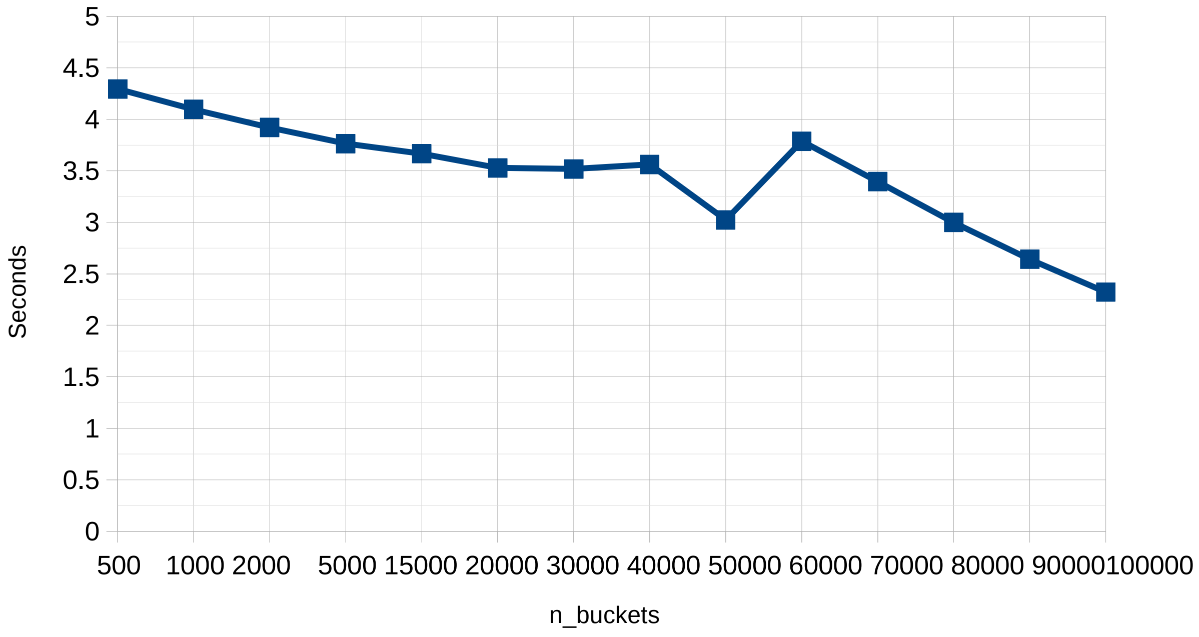
Finally all that’s left, is to sort the locally the buckets and then swap A and B.

```
for(i=0; i<n_buckets; i++)
        qsort(B+buckets[i].start, buckets[i].n_elem, sizeof(int), cmpfunc);

 temp = A;
 A = B;
 B = temp;
```

3) Testing

The test is done by fixing *n_buckets* = 1000 and searching for the biggest *dim* of the array that can be sorted in about 4 seconds.
With this restrictions, the maximum value of *dim* is 30 000 000.
Using this maximum value of *dim*, it could be interesting to show which value of *n_buckets* gives the smallest running time.
As showed in the graph above, the best value for n_buckets is about 100000.



Here there is representation as a table of the same data.

n_buckets	500	1000	2000	5000	10000	15000	20000	30000	40000	50000	60000	70000	80000	90000	100000
Seconds	4.295	4.096	3.920	3.764	3.666	3.755	3.475	3.532	3.562	3.030	3.787	3.395	2.999	2.641	2.322

4) Analysis of the running time

Assuming dim=n, n_buckets=b we want to create an expression for the running time of the algorithm.
In this sequential bucket-sort there are four main steps:

Step 0: Count the number of elements for each bucket. $t_0 \sim n$

Step 1: Initialize the buckets starting position. Time $t_1 \sim b$

o sort in the proper bucket. Time $t_2 \sim n$

k sort. Time $t_3 \sim n\log(\frac{n}{b})$

$nlog(\frac{n}{b})$.

al step dominates the others, and when n ~ b we get a linear time, this confirm
100000.

tSort.c, in order to test the program is sufficient to type ‘make’ and run

5)Parallel bucket-sort

The idea of the algorithm is as follows:
Each thread should first place n/p elements in each local buckets, then thread i gathers the contents of bucket i from each processor and sorts these.

4.1 **Steps of the parallel version:**

a)Variables definition and input
b)Divide data among the threads.
c)Set the right starting position for both local and global buckets in B.
4)Sort each global bucket

a)Variables definition and input

-*dim* is the size of the array to be sorted and is given by the user.
-*A*(size *dim*) is the array where all the randomly generated integers will be stored.
-*B*(size *dim*) is a temporary array.
-*n_buckets* is the number of buckets, this value is given by the user.
-*limit* is the range of the data(100 000).
-*w* is the width of each bucket.
-num_threads is number of threads (equal to n_buckets).
-*buckets (*size *n_buckets*num_threads)* is an array of struct bucke*t;* this is the array that contains all the local buckets.
-*workload is dim/num_threads,* this value represents the numbers of elements to be handled by each thread, note that the last one could have more integers, so that the algorithm will work for every *dim* and for every *n_buckets*.
-*global_n_elem*(size *n_buckets*) is an array that contains the number of elements for each global bucket.
-*global_starting_position*(size *n_buckets*) contains the starting position of the global buckets in B.

b)Divide data among the threads

With this loop each thread will handle roughly *dim/num_threads* integers of the array, assigning each element in the proper local buckets and at the same time update the right local buckets' counter.
To get the the index of a local bucket inside the array of *struct bucket*, one thread has first to multiply his id with the number of buckets, and then add the local index.

```
#pragma omp for private(i,local_index))
for (i=0; i< dim ;i++){
    local_index = A[i]/w;
    if (local_index > n_buckets-1)
            local_index = n_buckets-1;
    real_bucket_index = local_index + my_id*n_buckets;
    buckets[real_bucket_index].n_elem++;
}
```

c)Set the right starting position for both local and global buckets in B.

computes the number of elements for the global bucket i, and stores the result in

```
reads; j=j+num_threads){
```

After that, is possible to write the local buckets directly in B because we want to avoid dynamic memory allocation when the sorting is taking place , but in order to do that, is necessary to set up the right starting position for both local and global buckets.
This can be done with two loops, the first one done by the master thread, that initializes the starting position of the global buckets and the first *n_buckets* local buckets (note that is not necessary to initialize local bucket 0 since we used a calloc() before).
The second loop is done in parallel, since thread i can set up the right starting position in B for each local bucket i.

```
 #pragma omp barrier

    #pragma omp master
    {
     for (j=1; j<n_buckets; j++){
             global_starting_position[j] = global_starting_position[j-1] + global_n_elem[j-1];
             buckets[j].start = buckets[j-1].start + global_n_elem[j-1];
             buckets[j].index = buckets[j-1].index + global_n_elem[j-1];
        }
    }

    #pragma omp barrier
    for (j=my_id+n_buckets; j< n_buckets*num_threads; j=j+num_threads){
        int prevoius_index = j-n_buckets;
        buckets[j].start = buckets[prevoius_index].start + buckets[prevoius_index].n_elem;
        buckets[j].index = buckets[prevoius_index].index + buckets[prevoius_index].n_elem;
    }
```

Now thread i writes all the element of all the local bucket i directly in B, starting by the correct position and incrementing by one the index of the local bucket in each step.

```
    #pragma omp barrier

    #pragma omp for private(i, b_index)
    for (i=0; i< workload*num_threads ;i++){
        j = A[i]/w;
        if (j > n_buckets -1)
                j = n_buckets-1;
        k = j + my_id*n_buckets;
        b_index = buckets[k].index++;
        B[b_index] = A[i];
    }
```

4)Sort each global bucket

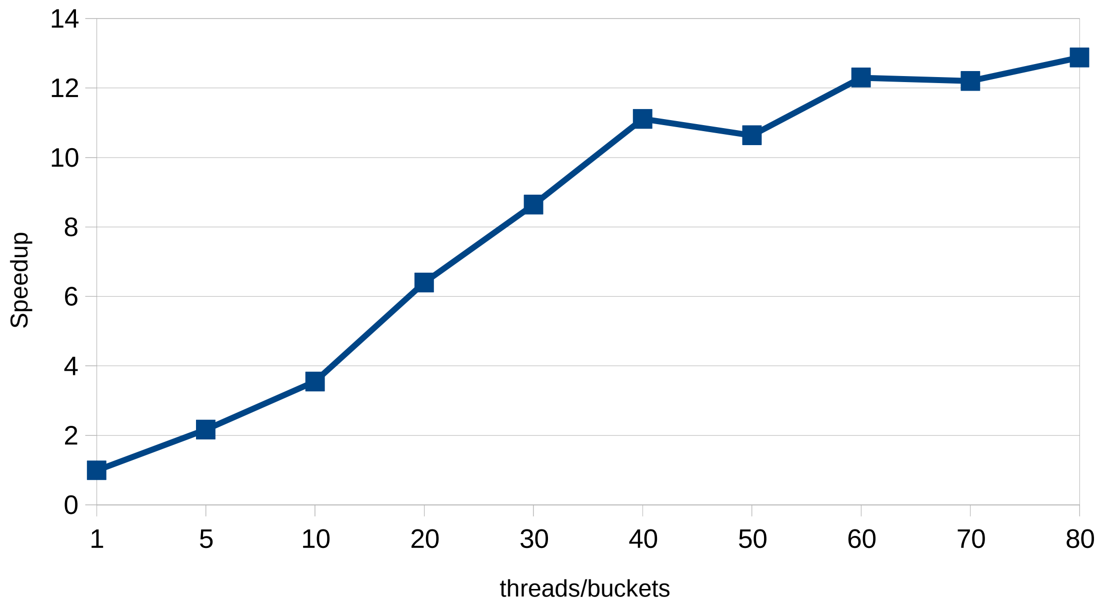
buckets, this is possible because we know where global bucket i starts and

, global_n_elem[i], sizeof(int), cmpfunc);

6. Scalability

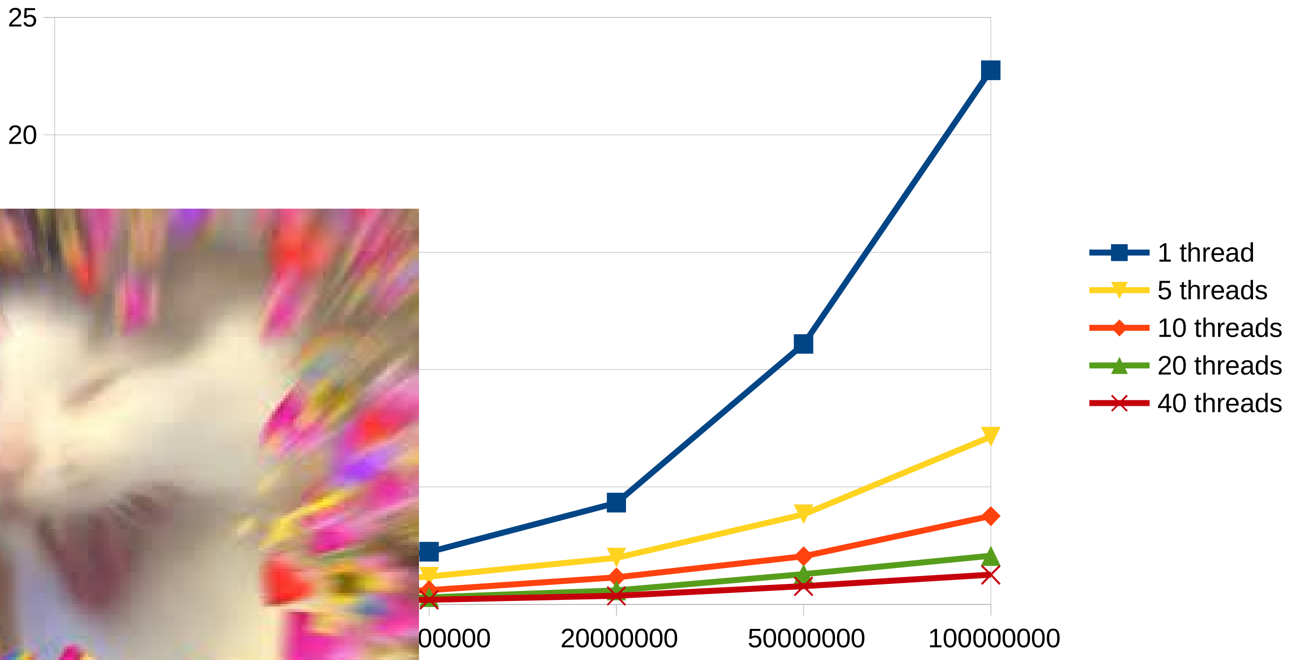
5.1 Strong scalability

The following chart shows the speedup with the size of the array fixed at 30000000:



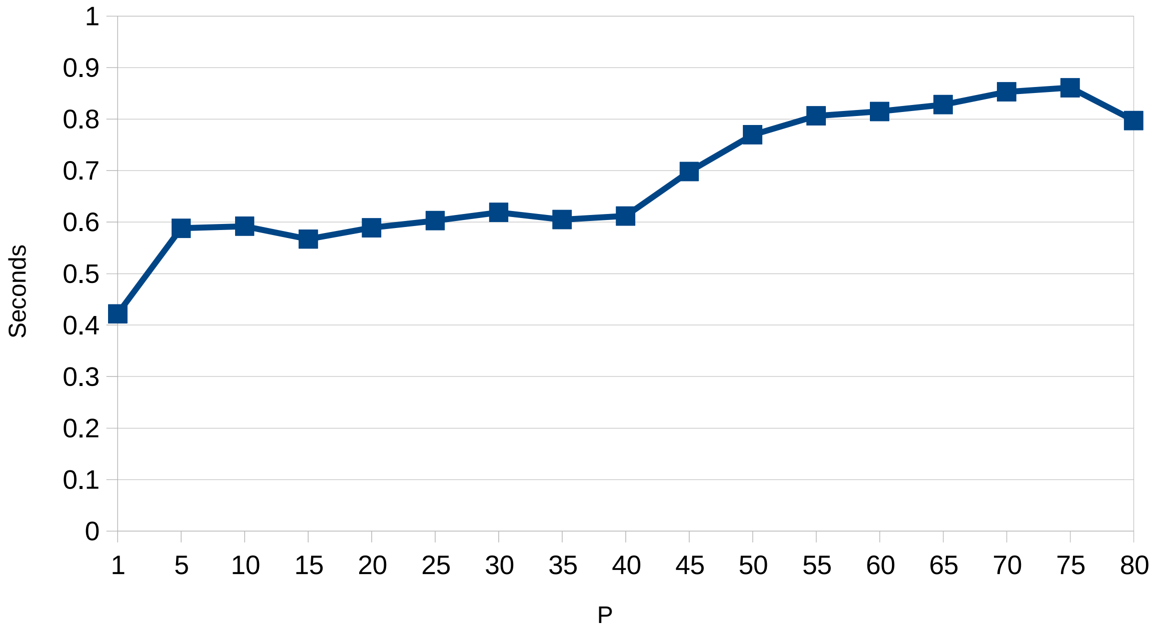
Buckets/threads	1	5	10	20	30	40	50	60	70	80
sequential	6.587	5.836	5.547	5.419	5.266	5.153	5.152	5.143	4.993	4.894
Parallel	6.596	2.685	1.562	0.846	0.609	0.461	0.484	0.418	0.409	0.380
Speedup	0.998	2.173	3.551	6.405	8.647	11.179	10.645	12.303	12.207	12.879

The following line graph shows instead a comparison between the program itself with 1 thread and many threads.



5.2 Weak scalability

For weak scalability when $P=1$ the dimension of the array is $1 \cdot 10^6$, therefore the amount of work for the first and the only one thread is $n \cdot \log(n)$, since we used quick sort and all the integers would go in one bucket. In order to keep constant the amount of work per thread, when $P=5$, n should be $5 \cdot 10^6$, since $\frac{5 \cdot n}{p} \cdot \log(\frac{5 \cdot n}{p})$ = $n \cdot \log(n)$,while when $P=10$ n should be $10 \cdot 10^6$ and so on.



As the line graph above shows, for $P \leqslant 40$ the running time remains fairly stable, while for higher value it increases. This is probably due to the fact that brake.ii.uib.no has only 40 real threads.

7. Analysis of the running time

The algorithm has four main sections:

step 0: Count how many elements each local bucket has.

This can be done in parallel, so $t_0 \sim \frac{n}{p}$ assuming an even distribution.

Step1:Set the right starting position for both local and global buckets in B.
In this section of the code there are three loops, the first one, done by each thread takes $t_{1,0} \sim b$, the second loop is

$\sim b-1 \sim b$, and the last one is done in parallel and time is

$\sim 3b$.

sort in the proper bucket.

lel, and assuming an even distribution time is $t_2 \sim \frac{n}{p}$.

.

s only $t_3 \sim \frac{n}{p} \log(\frac{n}{p})$.

rithm is $t_{tot} \sim 2\frac{n}{b} + 3b + \frac{n}{b} \log(\frac{n}{b})$ assuming the number of buckets b

In this expression the last term dominates the others, we can only consider $\frac{n}{p}\log(\frac{n}{p})$.

This means that we should get (at list on paper), a speedup equals to

$\frac{t_{seq}}{t_p} \sim \frac{(n \cdot \log(\frac{n}{b}))}{(\frac{n}{b} \cdot \log(\frac{n}{b}))} \sim b$, but as we have seen, we got a speedup about 13 in the best case when the number of buckets was 80, so the real speedup is not as good as the theoretical speedup.

The source file name is parallelBucketSort.c, in order to test the program is sufficient to type ‘make’ and run ./parallelBucketSort .