High Performance Programming

Parallel Bucket Sorting

Linus Lindell

March 2023

UPPSALA
UNIVERSITET

Contents

1 Introduction

For the project in High Performance Programming I chose to implement parallel bucket sorting in c. I did this using `OpenMP` since it is more straightforward than `Pthreads`, and since micromanaging threads will not be necessary for this problem.

2 Problem

Bucket sort is a sorting method that works by dividing the list into multiple sublists or "buckets". The division of the original list is done according to element size, which means the list is partially sorted after the division is done. Afterwards each bucket is sorted locally using a sorting method of choice. At last the elements in the buckets are put back into the original list in order, resulting in a sorted list.

The method consists of five main steps:

- Bucket creation.
- Bucket distribution.
- Local bucket sorting.
- List redistributing.

The most time-consuming step is the local bucket sorting, which is why this is the main step to parallelize. The other parts are less computationally demanding but can also be coded to run in parallel to gain some performance.

The project also includes implementing three different types of value distribution for the list; uniform, normal and exponential.

3 Solution

3.1 Algorithm

The algorithm follows the main steps of the sorting method:

3.1.1 Bucket creation

The buckets were implemented using a 2D array. First memory was allocated for an `int**`-array of length equal to number of buckets, this will act as a cointainer for all the buckets. For each element in the bucket container, memory was then allocated for an array of integers. These arrays will act as the actual buckets. The length of the allocated buckets was set equal to the size of the original list, this is however a subject of optimization.

3.1.2 Bucket distribution

In order to sort the elements into its correct bucket, some algorithm was needed that takes the element value and gives a bucket index (0 to N_buckets-1). There are several ways to do this algorithm, which is a key piece in managing the work load balance for the parallelization. Ideally every bucket recieves the same number of elements, however the distribution of list values plays a big role in this. An algorithm that works well for uniformly distributed values might not work well for normal or exponentially distributed values and vice versa.

The chosen algorithm works by assigning each bucket a value interval and then distributing list elements accordingly. This works best for uniformly distributed lists. First the range of values in the list is found by taking the difference between the minimun value and the maximum value. The bucket index for a list element is then calculated by the following formula:

$$\text{Bucket Index} = \texttt{floor}((\text{element value} - \text{min value}) \cdot \frac{\text{Number of buckets}}{\text{list range} + 1}) \tag{1}$$

This is done in a `for`-loop for every list element. To avoid repeated `double` division, which is computationally intensive, the fraction part is calculated beforehand and thus only subtraction and multiplication is done inside the loop.

3.1.3 Local bucket sorting

After each bucket is filled with elements in their size range, it is time to sort them locally. This can be done using any sorting method, including recursive bucket sort. Since this is the part that takes the most CPU time, the choice of sorting method here is essential for overall performance. The easiest choice is insertion sort, which is efficient for short lists but is very time-consuming for large lists as it scales with $\mathcal{O}(n^2)$ on average. For longer lists it would be better to use something like quicksort, however quicksort would be slower than insertion sort for small lists as it has more overhead code. In this implementation insertion sort was chosen as method, but combining this with quicksort for longer lists would likely improve performance and is an option for optimisation.

3.1.4 List redistributing

This part works by nesting two `for`-loops, one for looping through the buckets in order and a second one for looping through the bucket elements. Each element is inserted into the original list at a position determined by an independent variable that is incremented with each insertion.

3.2 List generating

3.2.1 Uniform distribution

To achive uniform distribution each element in the list was set to a random value between 0 and N, where N is the length of the list. This range was chosen to avoid an abundance of repeated numbers in the list.

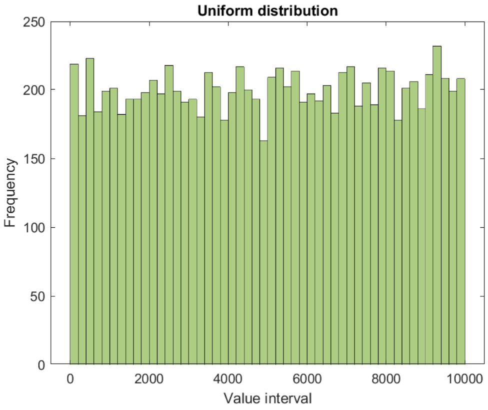


Figure 1: Uniform distribution in a list with N = 10 000

3.2.2 Normal distribution

To achive normal distribution each element in the list was set to a a value generated by a Box-Muller transform. Firstly two variables, u and v, are created and given random values between 0 and 1. Another variable z is then calculated using the following formula:

$$z = \sqrt{-2 \cdot \ln(u)} \cdot \cos(2\pi \cdot v) \tag{2}$$

This variable can be interpreted as the amount of standard deviations σ the number will deviate from the mean μ. Finally the list element is set to $z \cdot \sigma + \mu$, and the process is repeated for all elements. The shape of the generated bell curve will depend on the choice of μ and σ. In this project the values were set to $\mu = N/2$ and $\sigma = N/12$ for a list of length N, which resulted in a good looking curve.

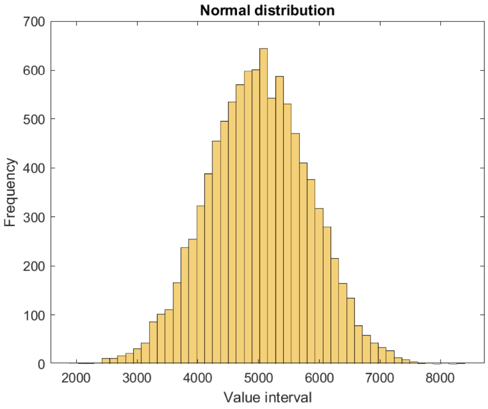


Figure 2: Normal distribution in a list with N = 10 000, $\mu = N/2$, $\sigma = N/12$

3.2.3 Exponential distribution

To achive exponential distribution each element in the list was set to a value given by the inverse transformation method. The method work by calculating the inverse of the cumulative probability function, and then submitting a uniformly random number between 0 and 1 as parameter.

$$F(x) = 1 - e^{-\lambda x} \rightarrow F^{-1}(x) = -\frac{ln(1-x)}{\lambda} \rightarrow \text{value} = F^{-1}(u), u \in [0,1] \tag{3}$$

The range of the distribution curve is decided by the value of λ, which in this case is set to 0.001.

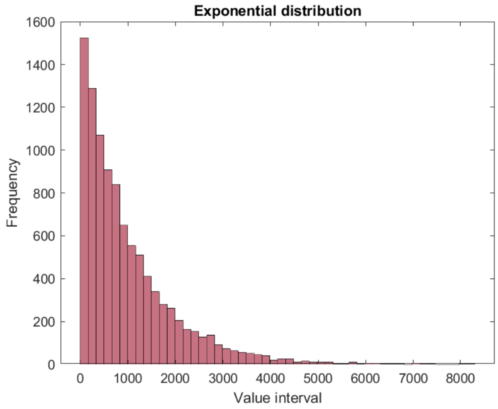


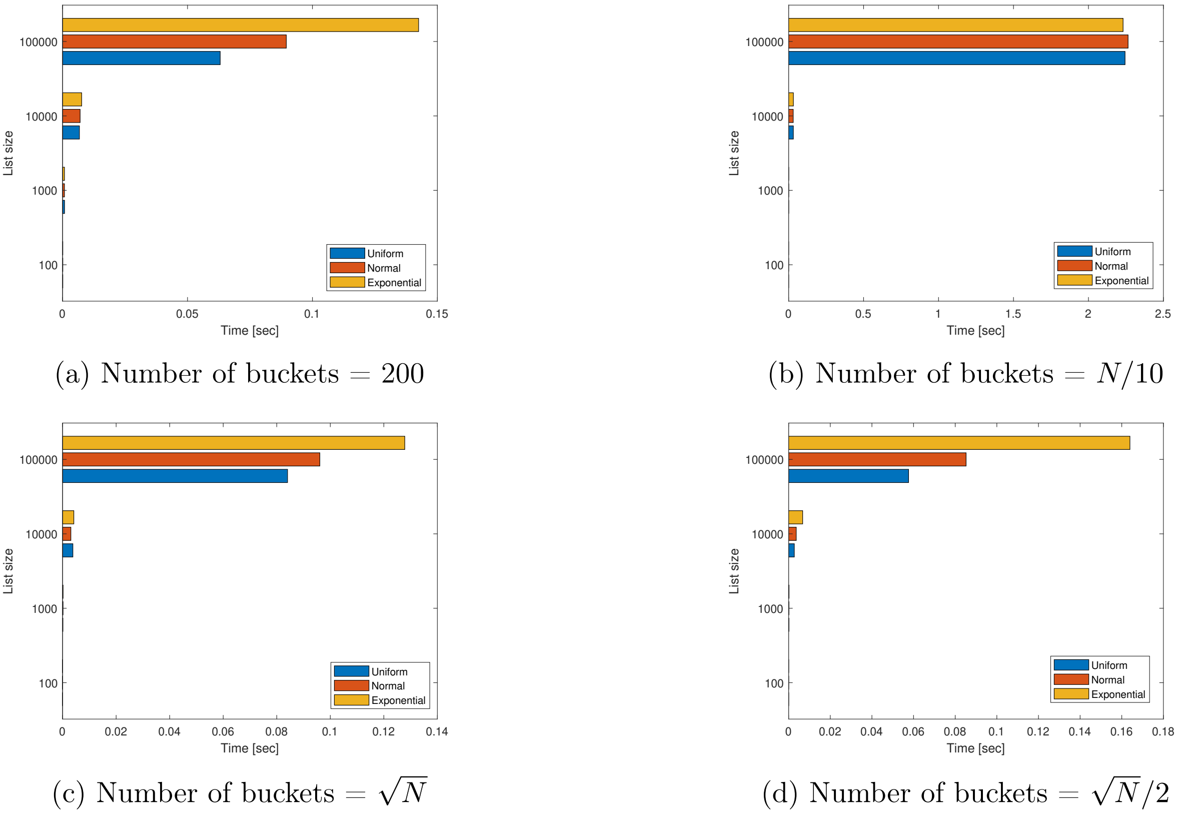
Figure 3: Exponential distribution in a list with N = 10 000, $\lambda = 0.001$

3.3 Optimization

To begin optimisation all execution for time comparison were done in serial.

3.3.1 Number of buckets

Presumeably, the number of buckets should depend on the size of the input list.



(a) Number of buckets = 200

(b) Number of buckets = $N/10$

(c) Number of buckets = $\sqrt{N}$

(d) Number of buckets = $\sqrt{N}/2$

Figure 4: Execution time for lists of different sizes N

It is visible in Figure 6 that the overall best performance was achieved with number of buckets being set to $\frac{1}{2}\sqrt{N}$, and the worst performance with number of buckets being a tenth of the list size. From this it can be deducted that a relatively small amount of buckets in comparison to the list size is preferable. It is also a clear trend that the exponential distribution results in worse performance. This is expected since the work load between buckets wont be even, and insertion sort will be used to sort bigger lists than desired.

3.3.2 Work load balance

The part that is the most time-consuming is the local sorting of the buckets, which is why it is desirable to do so in parallel. The parallelization will however be held back by the number of elements in the buckets. If one bucket contains half the lists elements the parallelization will not improve performance as much as it could since the work balance is uneven.

The first implementation of the distribution algorithm works by finding even size intervals from the range of list element values. This results in good distribution and work balance if the list elements are uniformly distributed, since every bucket will contain roughly the same amount of elements. If the list elements are subject to normal or exponetial distribution this is not the case, and finding a way to achive this would improve overall performance greatly.

Ideally there would exist a hashing function that evenly distributes the datapoint into the buckets, no matter the distributes of the datapoint values. However no such function exists, and changing the hashing function according to the distribution of data given by the user is unrealistic. Instead the code can be altered to better handle the bucket with many elements. One way of accomplishing is implementing quicksort and using this instead of insertion sort for bigger buckets. Insertion sort is still faster than quicksort for some small lists sinice it requires less overhead, hence the bucket length needs to be compared to some factor in order to decide which sorting method to use. After testing it was discovered that insertion sort performed better on lists under the length of 10, which was then chosen as factor.

After implementing quicksort the performance improved as follows:

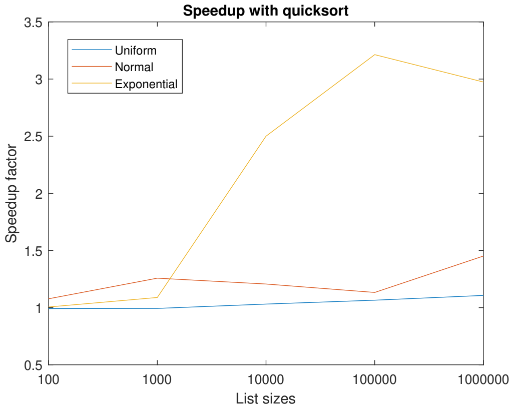


Figure 5: Speedup for list sizes N after implementing quicksort. Buckets = $\sqrt{N}/2$

As expected the exponentially distributed lists were affected to most by the implementation of quicksort, with a speedup factor over 3.0 for longer lists. Lists with uniform and normal distributed were affected less but still gained time, and there is a visible trend that normal distribution lists will gain more speedup for even longer lists.

3.3.3 Bucket creation

After implementing quicksort the local bucket sorting is no longer the most time-consuming step. With a list of size 1000000 almost all time was spent on creating buckets and allocating memory:

	Uniform	Normal	Exponential
Total time [s]:	1.124530	1.138399	1.471471
Bucket creation [s]:	1.099310	1.102726	1.102399
Percentage:	97.7%	96.8%	74.9%

Table 1: Time for memory allocation using only `malloc()`. List size N = 1 000 000

Optimising this part is a ket step in improving performance. Before improvement the code function as follows:

```c
int **buckets = malloc(N_buckets*sizeof(*buckets));
    int *bucket_count = malloc(N_buckets*sizeof(*bucket_count));
    for(int i = 0; i < N_buckets; i++){
        buckets[i] = malloc(N_list*sizeof(*list));
        for (int j = 0; j < N_list; j++){
            buckets[i][j] = 0;
        }
        bucket_count[i] = 0;
    }
```

Firstly, memory for each list is allocated using `malloc()`. Then each element in the 2D `buckets`-array and the regular `bucket_count`-array is initialized to 0 using a nested `for`-loop. The two `for`-loops are likely the time-consuming part, and they can be replaced to improve performance. Instead of allocating space using `malloc()`, `calloc()` can be used to allocate memory and initialize to 0 at once. This means the nested loops can be replaced with only one loop over each bucket. Additionally, the individual bucket elements do not need to be intialised at all:

```c
int **buckets = malloc(N_buckets*sizeof(*buckets));
    int *bucket_count = calloc(N_buckets, sizeof(*bucket_count));
    for(int i = 0; i < N_buckets; i++){
        buckets[i] = malloc(N_list*sizeof(*list));
    }
```

	Uniform	Normal	Exponential
Total time [s]:	0.026911	0.038219	0.379130
Bucket creation [s]:	0.002262	0.002288	0.002251
Percentage:	8.4%	6.0%	0.6%

Table 2: Time for memory allocation using `calloc()`. List size N = 1 000 000

It is visible in Table 2 that memory allocation is now only a small part of the entire process. This optimisation was clearly a huge improvement for performance.

3.3.4 Memory optmisation

As seen in the previous optimisation, memory is allocated in each bucket to fit the entire list. this is in case of the worst case scenario. Obviously it is impossible for the entire list to be put into every bucket, meaning this is a huge waste of memory. Instead the array `bucket_count` can be used to determine how many elements are going into each bucket before allocating the necessary memory. This means a new `for`-loop need to be added which will increase time, however sacrificing some time to gain a big improvement in memory efficiency is a good trade.

Total memory allocation can be displayed using `valgrind`. For a list of size N = 1 000 000, the following improvement was recorded:

	Before:	After:
Time [s]:	0.028510	0.024869
Total allocation:	2 Gb	8 Mb

Table 3: Time and total allocation, list size N = 1 000 000, Number of buckets = $\sqrt{N}/2$.

In Table 3 it is clear that the change resulted in a huge improvement regarding memory allocation. This was expected and makes sense when looking at the math. Before optimisation each bucket was given memory for the entire list. With 500 buckets and a list of one million integers the memory allocation can be calculated as:

$$\text{Memory} = 4 \text{ bytes} \cdot 1000000 \cdot 500 = 2000000000 \text{ bytes} = 2 \text{ Gb} \tag{4}$$

With the new algorithm the total memory for all buckets will equal the list, so only twice the list size will be allocated in total:

$$\text{Memory} = 4 \text{ bytes} \cdot 1000000 \cdot 2 = 8000000 \text{ bytes} = 8 \text{ Mb} \tag{5}$$

This matches the information in Table 3. Hower, Table 3 also shows that the total time is lower, which means the time for the additional `for`-loop is less than allocating all the extra memory.

3.4 Parallelization

The main step to parallelize is the sorting step. As it is a `for`-loop that iterates over the buckets and calls a sorting function for the bucket, it can easily be parallelized using `#pragma omp parallel for`. If the list is uniformly distributed, the work load will be even between threads without having to change the `schedule()` function. Since every bucket will be of almost equal length the default `static` scheduling will give each thread almost equal work. However with normal or exponential distribution the work load will be very unbalanced. With normal distribution the middle threads will have to sort the majority of the list, for example, while the beginning and end threads will have very little work. Because of this setting `schedule(dynamic)` makes sense. This will assign the threads more work as they finish their iteration, ensuring better work balance than before. The parallelization of the local bucket sorting resulted in the following improvement:

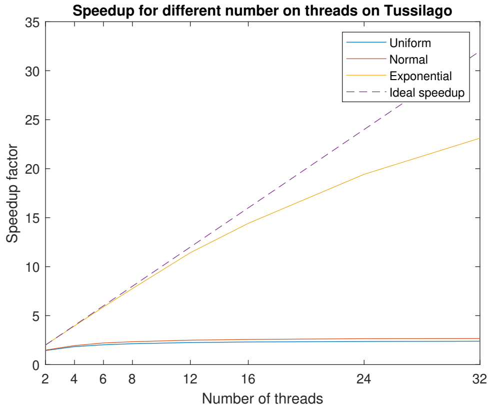


Figure 6: Speedup for different number of threads. N = 10^7, Buckets = $\sqrt{N}/2$

The speedup it not close to ideal, except for with exponential distribution. This is because only the local bucket sorting is parallelized, the other parts of the method are still run in serial. The time consumption for these parts is independent of distribution type, and since exponential list take significantly longer to sort they appear to follow the ideal speed up since the local sorting time dominates. To observe the effect of parallelization the time for the local bucket sorting can be plotted instead:

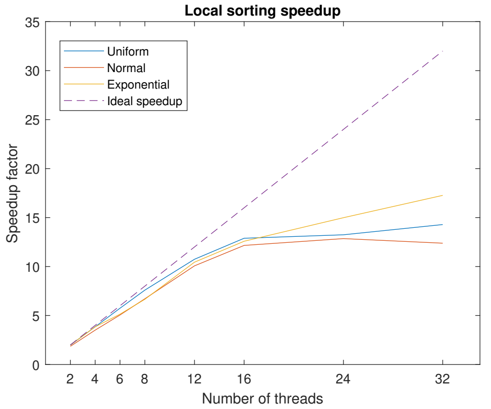


Figure 7: Speedup for local sort, N = 10^7

It is visible in Figure 7 that the sorting for all three distribution roughly follow the ideal speedup up to 16 threads, afterward, after which the speedup curves flatten out. To improve total speedup the other parts need to be parallelized as well.

3.4.1 Bucket creation and distribution

To parallelize the bucket creation and distribution it has to be rewritten. As described in Section 3.3.3, the algorithm functions by first looping through the list to find some necessary values:

```c
    int max_element = list[0], min_element = list[0];
    for(int i = 1; i < N_list; i++){
        if(list[i] > max_element) max_element = list[i];
        else if(list[i] < min_element) min_element = list[i];
    }
    int range = (max_element - min_element);
    double temp = (double)N_buckets/(range+1);
```

Afterwards a second loop for memory optmisation, and some allocation:

```c
    for (int i = 0; i < N_list; i++){
        int bucket_index = (int)((list[i]-min_element)*temp);
        bucket_count[bucket_index]++;
    }
    for(int i = 0; i < N_buckets; i++){
        buckets[i] = malloc(bucket_count[i]*sizeof(*list));
        bucket_count[i] = 0; //reset bucket count
    }
```

Finally the elements are distributed with a third iteration:

```c
    for (int i = 0; i < N_list; i++){
        int bucket_index = (int)((list[i]-min_element)*temp);
        buckets[bucket_index][bucket_count[bucket_index]] = list[i];
        bucket_count[bucket_index]++;
    }
```

To find the arrays maximum and minimum in parallel, a `reduction` clause in OpenMP can be used. For this only one line needs to be added above the `for`-loop:

```c
    #pragma omp parallel for reduction(max:max_element) reduction(min:
  min_element)
```

The rest of the loops do not have independent iterations as the position of each element in its bucket is dependent on a variable which is incremented each iteration. In order to parallelize these, the code has to be rewritten. First an additional array is created of the same length as the list. This are for storing each list elements corresponding position in the bucket, which was previously done by iterating the `bucket_count`-array.

```c
    int *list_element_bucket_count = malloc(N_list * sizeof(*list));
```

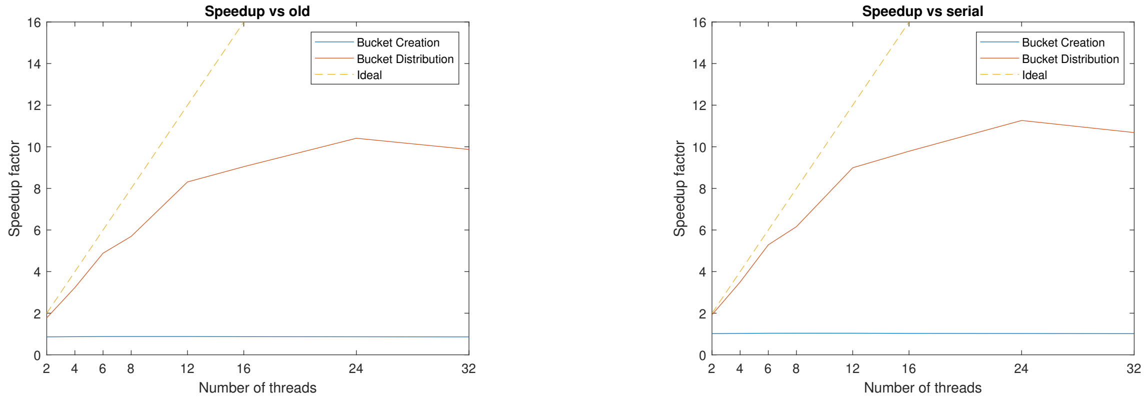
Now an iteration over the list is required to fill the new arrays with their values:

```c
    int bucket_index;
    for (int i = 0; i < N_list; i++){
        bucket_index = (int)((list[i]-min_element)*temp);
        list_element_bucket_count[i] = bucket_count[bucket_index];
        bucket_count[bucket_index]++;
    }
```

The addition of this arrray means that each list elements position in the bucket does not depend on a critical variable but is already stored, meaning the loop iterations are now independent and can be parallelized:

```c
    #pragma omp parallel for
    for(int i = 0; i < N_buckets; i++){
        buckets[i] = malloc(bucket_count[i]*sizeof(*list));
    }
     #pragma omp parallel for
    for (int i = 0; i < N_list; i++){
        bucket_index = (int)((list[i]-min_element)*temp);
        buckets[bucket_index][list_element_bucket_count[i]] = list[i];
    }
    //this is no longer needed
    free(list_element_bucket_count);
```

This solution is a bit more memory expensive, as additional memory equal to the list size is allocated. However this is significantly less than was saved with the memory optimisation in Section 3.3.3. After this implementation the following time improvement for was recorded:



(a) Speedup compared to old algorithm

(b) Speedup compared to serial

Figure 8: Speedup with new bucket algorithm, N $= 10^7$

As seen in figure 10 the bucket distribution is up to ten times faster than the old algorithm and up to 12 times faster than serial, when using 24 threads. It also follows ideal speedup behaviour fairly well up until 12 threads. In contrast the bucket creation does not not speed up which is expected since it only consists of allocating memory.

3.4.2 Bucket putback

The last part to parallelize is putting back the elements from each bucket into the list. This is not very time-consuming but could be still be improved slightly. It was prevously done by two nested for-loops iterating through each bucket and each elements. In the inner loops a varaible keeping track of the index in the list is incremented.

```
    int list_index = 0;
    for (int i = 0; i < N_buckets; i++){
        for (int j = 0; j < bucket_count[i]; j++){
            list[list_index] = buckets[i][j];
            list_index++;
        }
```

To parallelize this the `list_index` variable needs to be removed since it would make the iteration independent of each other. Each elements posisition in the list can instead be calculated using the cumulative sum of the bucket_count-array with the following formula:

$$\texttt{index} = \texttt{bucket\_count}[0] + \texttt{bucket\_count}[1] + \dots + \texttt{bucket\_count}[i-1] + j \tag{6}$$

This would give the correct index in the list for an element at position j in bucket i. To implement this another array is required of the same size as bucket_count, which will hold the cumulative sum of all previous buckets:

```c
    int *bucket_accum = malloc((N_buckets)*sizeof(*bucket_accum));
    bucket_accum[0] = 0;
    for(int i = 1; i<N_buckets; i++){
        bucket_accum[i] = bucket_count[i-1] + bucket_accum[i-1];
    }
```

If the list index is calculated this way the for-loop should be perfectly parallelizable giving the following results:

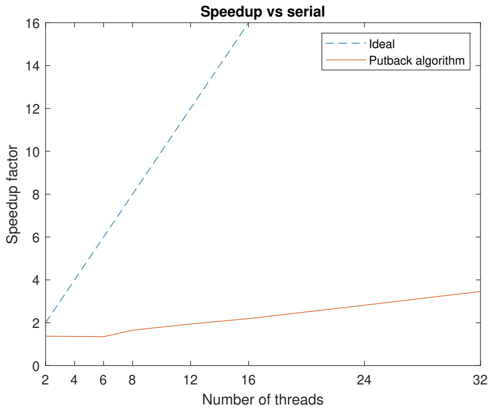


Figure 9: Speedup for putback using cumulative sum, N $= 10^7$

It is clear in Figure 9 that the change did not have a big impact, as the time consumption was already small and the speedup factor only reaches four.

4 Experiments

4.1 Timing for different N's

The following timings were made on the linux server vitsippa.it.uu.se using 32 threads. Number of buckets used was $\frac{1}{2}\sqrt{N}$ for each list size N.

List size:	1 000 000	10 000 000	100 000 000	1 000 000 000
Total time [s]:	0.025	0.230	2.447	32.522
Bucket counting time [s]:	0.020	0.178	1.837	23.808

Table 4: Timings for uniform distribution

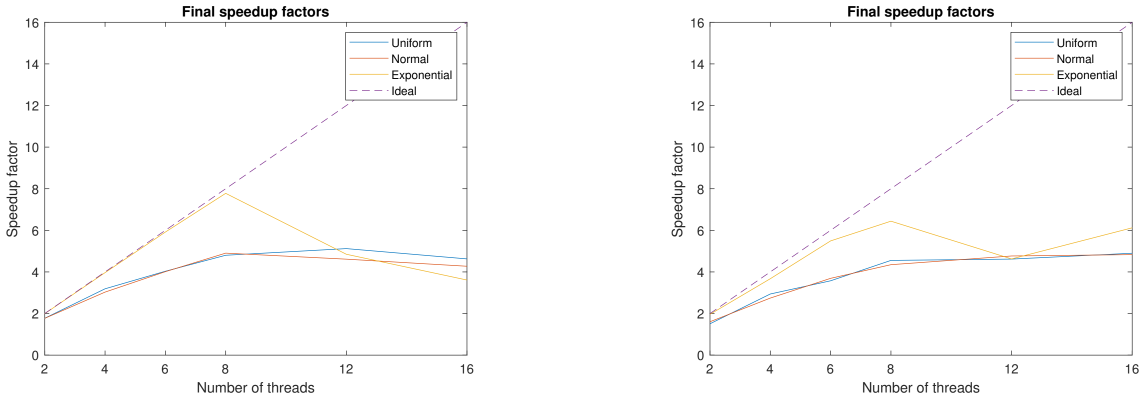
List size:	1 000 000	10 000 000	100 000 000	1 000 000 000
Total time [s]:	0.028	0.236	2.550	31.274
Bucket counting time [s]:	0.020	0.175	1.769	20.077

Table 5: Timings for normal distribution

List size:	1 000 000	10 000 000	100 000 000	1 000 000 000
Total time [s]:	0.069	4.178	330.506	-
Bucket counting time [s]:	0.019	0.174	2.253	-

Table 6: Timings for exponential distribution

Final speedup factors, run on arrhenius.it.uu.se which has 16 available threads:



(a) $N = 10$ million, Buckets $= \sqrt{N}/2$

(b) $N = 2$ million, Buckets $= \sqrt{N}/2$

Figure 10: Final speedup factors on arrhenius serever

5 Discussion and conclusions

After running tests for different large list sizes, as seen in Section 4, it is clear that the loop that is introduced in the optimisation in Section 3.4.1 is a bottleneck for performance. On average 73% of the time was spent on this loop when sorting lists with uniform or normal distribution (as seen in table 4, 5). If this loop was optimised further or parallelized the performance of the sorting method could be greatly improved.

When sorting lists of exponential distribution only about 2% was on average spent on the bucket counting loop, and the percentage drops as the list size grows. This is of course because the sorting algorithm struggles to balance the work load of exponentially distributed lists, and thus sorting takes a very long time compared to sorting equally big lists of other distribtutions. This can be explained by the fact that an exponential distribtution is close the worst case scenario for parallelised bucket sort, as most of the elements are placed in few of the buckets leaving many buckets almost empty in comparison.

To further improve performance it was attemped to fully implement pointer swapping instead of writing new values to the array. This did however not give the excpected result as it slowed down the program. There are some possible explanations for this, for example the program relies hevily on fetching values from arrays that are stored in order in memory. Writing new values to allocated arrays makes sure the data is stored in the relevant vicinity. Using pointer swapping will perhaps instead result in the memory being pointed to being further apart in the memory, which could lead to possible cache misses. It is also likely that the compiler generates optimised code for writing new values to memory that is more efficient than the implmented pointer swapping.

In the program there is a distinct function for choosing the pivot element for quicksort, which for the tests have given the last element in the list. Different functions such as returning the mean or median element were also tried in order to improve performance. This did however not make the code faster and is perhaps due to the fact that the buckets are already partially sorted, meaning the range of values is reduced. Also the function that chooses the pivot element is recursively called many times, meaning a small additional computational cost could have a bigger impact than expected.

The reason the parallelization speedup does not follow the ideal speedup is because not every part of the code is able to run i parallel. Besides all the overhead for the main thread there are several spots in the code that is always run in serial, such as the bucket creation, the bucket count loop, and the cumulative sum calculation. In order to fully parallelize every part of the code, the entre process need to be run i parallel such that each thread does bucket sort on their own subarray instead the parallelization being inside the bucket sort. This would however mean that after merging the sorted subarrays the entire list would need to be sorted again. This would probably perform worse than the parallelisation implemented here.

6 References