Project 1, Part 2 Parallel Systems
DVA314 Report

Tamara Dancheva, tda14001@student.mdh.se
Marin Tomić, mtc14002@student.mdh.se

1.Detailed description of the problem

For this task, we have chosen to implement bucket sort in combination with sample sort. The sorting of the buckets internally is done using the quicksort sorting algorithm.

The main reason why bucket sort is used is break apart the array into multiple smaller arrays that can be sorted more quickly, thereby achieving a speed up since it takes much less time when the right ratio of elements to buckets is chosen and especially if the sorting is also done in parallel. Bucket sort can be used recursively (very resource consuming and not recommended) or it can be used in combination with another sorting algorithm (in this case we decided to use quicksort). Bucket sort reaches its expected theoretical complexity when the the distribution of the sample to be sorted is uniform. Its average complexity is O(n+k), while its worst complexity is up to O(n^2). In order to avoid this last scenario, we have to make sure that the range of the buckets fits the distribution of the sample that is to be sorted. This can be done using statistical methods of approximating the distribution of the sample by sampling it in regular intervals often enough to detect at least the major curves/changes. Once this is done, the taken subset of the original sample is sorted and splitter values (the number of splitter values=number of buckets -1) are picked out from in regular intervals. These splitter values are then used to initialize the range of the buckets. In this way we assure that each bucket will contain approximately the same number of elements, which is the case when the original sample is generated using uniform distribution. We have measured the performance of our solution using normal distribution and uniform distribution over a discrete interval. The results for both the sequential and the parallel solution will be presented in the following sections.

2. A detailed description of the sequential solution

Firstly a subset of size Size of the array/ Number of buckets (in our sequential solution) is selected from the initial array always using the same offset between the picked elements from the array. Then splitter values are defined (equal to Number of buckets-1) and used to initialize the range of the buckets. The bucket structure contains an array, start index, end index and count (number of elements in the array) since we do not know the exact number in advance we need to make a guess. Then for each element in the array we check if its value is between the start and the end index of each bucket until we find in which bucket the element belongs. After this each bucket is sorted using the quicksort algorithm and finally the resulting array is formed by iterating the buckets in order and concatenating them. This is all possible having the range of the original sample as an input.

3. A motivation to why you find it suitable to solve the problem using parallel programming

The independency of the buckets is the most promising part of the definition of the problem since it is basically a guarantee that a parallel solution is feasible. The most convenient thing to do is to make the number of processes is equal to the number of buckets. In this case each process can independently sort its allotted part of the array into buckets and then sort its own single bucket after obtaining all the elements from the whole array that belong to it. Also the sampling can be sped up by having multiple processes pick samples from different not overlapping parts of the array.

4.A description of your expectations on the parallel solution to the problem

Due to the time overhead cause by the synchronization and communication between the processes, the additional code needed to determine the logic that is applied for each of the threads, the time needed to initialize the structures that we use to run the processes in parallel (the processes themselves with all the dependencies) for a small number of elements in the array, the sequential solution will prove to be faster, however as the array size increases , the parallel solution will prove much more efficient. As the number of buckets increases after some point the speed up will decrease because the number of processes executing independently (literally in parallel) on the blade server is 4, with hyper-threading 8.

5. Description of possible sequential optimizations (e.g. looporder ing, cache-blocking, memory usage etc.)

-Compiler options O3 optimization

-Concerning the size of the array, using as exact generation method as possible to generate a sample of a certain distribution and using an optimal number of sample values to determine the buckets' range is necessary in order to not have to make a too wild memory consuming guess of the number of elements in the bucket. The more equal the number of elements is in each bucket the more faster our solution is. Also we don't have to allocate too much extra memory to the bucket to make assure it is not going to overflow.

6. A description of the parallel computer architecture to be used in your solution. You should also include a decision on which parallel software formalism to use.

Each process has its own separate memory space and it can communicate with the other processes. This model is called distributed memory/message passing model demanding from the processes to cooperate with each other in order to be able to exchange messages between each other.

As for programming models, we are using a hybrid model, partly peer, partly master/slave for the parts that need to be coordinated by a single process because data has to be gathered from all the processes and/or same data has to be shared between all the processes.

7. A detailed description of your parallel solution to the problem

Firstly, each process (equal to the number of buckets) is alloted part of the array. Then each process samples its sub-array and sends the sample to the master process that sorts the sample, creates the splitters. Then each process initializes its own list of buckets (size:number of buckets) using the same splitters, sorts each element of its array putting it in the appropriate bucket, sends its number of elements in each bucket to the master process. Then the master process collects all the information about the number of elements in each bucket in each process and broadcasts it to all the other processes. Using this information, each process sends all the buckets except the bucket they are responsible for to the process that is responsible for that bucket. After that each process sorts its own bucket using quicksort. The final step is gathering all the sorted buckets in order into one resulting array (the sorted array).

8.A detailed description of your implementation.

There are two ways for generating the array, one is using random and simulating an uniformly distributed array and the other one is the C library for generating normally distributed samples. Then we define the size of the sample (SIZE of the array/ number of buckets, came to the formula after experimenting with different values) and we initialize/allocate space for some of the variables that we are going to need. For the sample we allocate an new array called mySample that is used by the slave processes while the master process populates the sample array directly (peer mode). When every process is finished, the master process gathers all the sub-arrays of the sample, then creates the splitters (the number of splitters=number of buckets -1 , because each bucket is

initialized with an start index and end index, the first bucket has 0 as starting index, the last one has range as the last elements, therefore we only need number of buckets -1 splitter values). The first process may have to handle more elements than the rest of the processes if SIZE %NUM_BUCKETS!=0 so we allocate space for the extra elements, iterate over them and put them in the appropriate bucket. We do the same with slave process (they all have same number of elements). Each bucket additionally puts the number of elements in each of its buckets in an array. These arrays are reduced using the sum operator into an array in the master process that holds the total number of elements in each bucket as if we only have one process that iterates over the array. The sum of these elements is the SIZE of the array. Since every process should send NUM_BUCKETS -1 arrays to other processes since one process is responsible only for its bucket whose index corresponds with the rank of the process to make it easier, we need to have a matrix with number of elements in each bucket in every process. This is done by gathering the bucketSizes into one array of length (NUM_BUCKETS*NUM_BUCKETS). This array is used to initialize the size of the bucket array(each process responsible for one bucket) and gather it (determine the index range). Then each process sends the buckets using a nonblocking send so that the program does not stop running and waiting for the receive call immediately. The receive for each process is done after the sending(non blocking one) to avoid a deadlock. IndCount and rindCount(reverse indCount) are used to translate the one dimensional array into a two dimensional matrix since it is more convenient to work that way and have separate variables for the sender(i dimension) and the receiver(rank).The reverse is used to match the send call tag with the receive call tag. Every process assigns the elements of his allotted bucket into a new array in which all the elements from the whole array that belong to that bucket are being put. After each process receives the elements that it needs, each process ends up with elements that belong to one bucket only and sorts then. Then finally, we just gather them all in a resulting array in the master process.

9. A discussion of the parallel overheads in your solution (e.g. algorithmic complexity,task management, communication, load imbalance, true/false sharing etc.).

The main overhead in this solution is caused by the communication between the processes, since firstly the data has to be distributed among the process, then returned to the master process that combines it and additionally there is the info (an array of size NUM_) about the number of elements in each bucket that needs to be broadcasted to each process.

We cannot directly tackle the load imbalance problem since it is the job of the schedule. The only thing we can to is to make sure that the processes do approximately the same amount of work. In our case the master process (the 0 process) does more work than the other processes, because it coordinates them and in order to cut the amount of work does some calculations and broadcasts the result to each of the process in order for them to be able to do their work.

The complexity of the algorithm lies in the logic to extra structures needed in order to be able to distribute the work among the process, the memory needed to enable this kind . Otherwise it is quite straight-forward, there are no nested loops , the reduce operator is used wherever needed to decrease the complexity and each task is executed by all the processes except for the few coordination tasks that only the master process does.

The memory used by each process since they all have their own private memory and communicate by exchanging messages, is problematic especially as the array size increases. Not only the size of the messages increases but also more memory is needed, and more cahce misses occur slowing down the program.

10.Build and execute code

We are using a C library that generates a normally distributed sample. [1] This library requires the -lm flag

BucketSampleSort.c (sequential algorithm) and Quicksort.c

Compile command: gcc -Wall -c "%f" -lm

Build command: gcc -Wall -o "%e" "%f" -lm

BucketSampleSortMPI.c

! Modify Num_Buckets to be equal to the number of processes you are going to use before running the solution in order to get correct results

Makefile

```
MPICC= mpicc
CC= gcc


PRGS=bucketSampleSortMPI

all: $(PRGS)

time.o: time.c
	$(CC) -O3 -c time.c -o time.o


%: %.c time.o
	$(MPICC) -o $@ $@.c time.o -lm


clean:
	rm -f *.o
	rm -f $(PRGS)

distclean: clean
	rm -f *~
```

11.Some evaluation of the correctness of your solution and implementation.

Concerning the result, the solution produces the correct result. There are two important parameters that greatly affect the the correctness of the result: the size of the sample and the number of buckets. We have chosen the number of buckets to be the same as the number of buckets.
Implementation-wise this can be done differently and each process can sort more than one bucket. This would however increase the algorithmic complexity, but might result in a better performance due to a better work distribution (more coarse grained parallelism).
As for the size of the sample, the goal using the sample sort algorithm is to make the number of elements in each buckets approximately the same. There is always a factor of uncertainty since statistical methods are used in order to generate the sample. The same applied for the way we determine the range of each bucket since we use the sampling technique and it is very hard to detect all the minor changes and produce the perfect distribution among the buckets. We use the formula: SIZE_SAMPLE=SIZE/NUM_BUCKETS which produces pretty good approximation. It can of course be improved upon with some more research using more advanced statistical methods to analyze the curve and determine the sample size.

12. Some illustration of the performance; e.g. the speed-ups (scalability)

or
throughput compared to the original (possibly sequential) solution, if one exists. Use tables and graphs to show the results of executing your code using, at least, [1; 8] threads, if applicable.

Quicksort

Here we can observe that quicksort has much better performance when the distribution is uniform since the number of elements in each bucket is approximately the same. As the size of the array increases the effect is more and more visible.

```
/*RESULT EXECUTION TIME: NORMAL DISTRIBUTION a=70, b=10
 * SIZE=10; 0.000022;
 * SIZE=100; 0.000053;
 * SIZE=1000; 0.000376;
 * SIZE=10000; 0.011826;
 * SIZE=20000; 0.043048;
 * SIZE=30000; 0.115734;
 * SIZE=40000; 0.115432;
 * SIZE=50000; 0.174121;
 * SIZE=60000; 0.249102;
 * SIZE=70000; 0.298845;
 * SIZE=80000; 0.427522;
 * SIZE=90000; 0.483830;
 * SIZE=100000; 0.598888;
 * SIZE=110000; 0.705203;
 * SIZE=120000; 0.895879;
 * SIZE=130000; 1.036586;
 * SIZE=140000; 1.137492;
 * SIZE=150000; 1.322866;
 * SIZE=160000; 1.473868;
 * SIZE=170000; 1.713851;
 * SIZE=180000; 1.926754;
 * SIZE=190000; 2.063295;
 * SIZE=200000; 2.320106;
 * SIZE=250000; 3.591560;
 * SIZE=300000; 5.155354;
 * SIZE=350000; 6.924571;
 * SIZE=400000; 9.002567;
 * SIZE=450000; 11.482232;
 * SIZE=500000; 14.069999;
 * SIZE=550000; 16.996690;
 * SIZE=600000; 20.179871;
 *
 * RESULT EXECUTION TIME WITH UNIFORM DISTRIBUTION
 * SIZE=10; 0.000003;0.000003
 * SIZE=100; 0.000013;0.000242
 * SIZE=1000; 0.000149;0.000227
 * SIZE=10000; 0.003187;0.004901
 * SIZE=20000; 0.011451; 0.010554
 * SIZE=30000; 0.043213;
```

```
 * SIZE=40000; 0.060761;
 * SIZE=50000; 0.073121;
 * SIZE=60000; 0.078017;
 * SIZE=70000; 0.110439;
 * SIZE=80000; 0.131061;
 * SIZE=90000; 0.167746;
 * SIZE=100000; 0.255511;0.189425
 * SIZE=110000; 0.240379; 0.226070
 * SIZE=120000; 0.292963;0.268522
 * SIZE=130000; 0.332249;0.313948
 * SIZE=140000; 0.387530;
 * SIZE=150000; 0.453347; 0.411841
 * SIZE=160000; 0.497361;0.476609
 * SIZE=170000; 0.546070;
 * SIZE=180000; 0.584870;
 * SIZE=190000; 0.692192;
 * SIZE=200000; 0.747082;0.733892
 * SIZE=250000; 1.136697; 1.124118
 * SIZE=300000; 1.603325;
 * SIZE=350000; 2.176600;2.179614
 * SIZE=400000; 2.798710;
 * SIZE=450000; 3.554562;3.584586
 * SIZE=500000; 4.387099;
 * SIZE=550000; 5.279040;
 * SIZE=600000; 6.299424;
 *
 * */
```

Sequential solution

As it can be seen the execution times for both distributions is almost the same. This means that the solution is correct and the sampling is successful.

Normal distribution a=1000,b=400, Range=2000

N=100 0.000016
N=1000 0.000191
N=10000 0.003712
N=20000 0.007293
N=30000 0.008322
N=40000 0.011775
N=50000 0.015575
N=60000 0.224366
N=70000 0.299301
N=80000 0.389705
N=90000 0.491970
N=100000 0.634794
N=150000 1.390067
N=200000 2.476932
N=250000 3.853016
N=300000 5.542123
N=350000 7.585515
N=400000 9.843156

N=450000 12.465545
N=500000 15.460339
N=550000 18.668377
N=600000 22.118212
N=650000 26.288943
N=700000 30.181689

Uniform distribution, Range=150

N=10 0.000002
N=100 0.000026
N=1000 0.000422
N=10000 0.006623
N=20000 0.019345
N=30000 0.046122
N=40000 0.131939
N=50000 0.247801
N=60000 0.251204
N=70000 0.302546
N=80000 0.384603
N=90000 0.452631
N=100000 0.743287
N=150000 1.462433
N=200000 2.354208
N=250000 3.956372
N=300000 5.496573
N=350000 7.635241
N=400000 9.754522
N=450000 12.469953
N=500000 15.472488
N=550000 18.665542
N=600000 22.163425
N=650000 26.302626
N=700000 30.204525

Parallel solution

Finally the parallel solution performance results confirm that for a smaller number of elements it is not affordable to use this solution because of the overhead. However, for more than about 10000 we can see that the parallel solution yields much better results.

//NORMAL 4 BUCKETS
N (size array) execution time, a=1000, b=400, RANGE=2000
//100 0.006795 seconds
//1000 0.003986 seconds
//10000 0.010741 seconds
//20000 0.023911 seconds
//30000 0.034233 seconds
//40000 0.044709 seconds
//50000 0.050988 seconds
//60000 0.069799 seconds

//70000 0.082035 seconds
//80000 0.094025 seconds
//90000 0.105220 seconds
//100000 0.124452 seconds
//150000 0.203834 seconds
//200000 0.296855 seconds
//250000 0.409109 seconds
//300000 0.543571 seconds
//350000 0.675193 seconds
//400000 0.842150 seconds
//450000 1.004756 seconds
//500000 1.189591 seconds
//550000 1.390809 seconds
//600000 1.600214 seconds
//650000 1.833183 seconds
//700000 2.099296 seconds

//UNIFORMM 4 BUCKETS RANGE=150
//10 0.248607
//100 0.236084
//1000 0.259914
//10000 0.251962
//20000 0.278182
//30000 0.268879
//40000 0.259212
//50000 0.283983
//60000 0.245250
//70000 0.331961
//80000 0.263934
//90000 0.365239
//100000 0.325433
//110000   0.368437
//150000 0.384665
//200000 0.408849
//250000 0.402261
//300000 0.504242
//350000 0.468067
//400000 0.608201
//450000 0.575140
//500000 0.619874
//600000 0.674000
//700000 0.871993

13.Conclusions on your parallel solution regarding your expectations, the performance,open problems, current difficulties and achievement, etc.

The yielded results met our expectations considering the achieved performance.
One of the most difficult things to implement was the part where each bucket sends its buckets to the appropriate process so that each process ends up with one buckets and sorts it. We tried

implementing this part firstly with a synchronous send and receive but each time ended up in a deadlock no matter how we scheduled the send and the receive operation. We realized this is impossible with a synchronous send and receive since we cannot know the order in which the process will execute and therefore cannot predict the order of send and receive considering that each process sends data to all the other processes. Consequently, we used the MPI non-blocking send and used the two “dimensions” : rank and number of the bucket to create a unique tag for the send operation.
The formula used to determine the size of the initial buckets supposes that the number of elements in each bucket will be the same. However this is not always the case since the approximation cannot be 100% accurate(also depends on the sample size, which in turn depends on the number of buckets and the range of the numbers). Therefore a more complex formula should be found for the size of the sample and the size of the initial buckets. Now it is just a guess and we allocate a lot more space in order to assure that we will not have problems with the size being too small in case of a very non equal distribution of the elements when sorted into the buckets.
As the range increases , the solution yields better and better results, meaning that the distribution of the elements improves when we have a bigger range. The worst case is when the range is small and the number of elements is big. The problem with this solution is that for the normal distribution , we don't have the minimum and the maximum, also another problem when evaluating our solutions is that the result is too dependent on the ratio between the number of elements and the range making it hard to compare the sequential and the parallel solution. What we have been able to observe is a general pattern that is explained in the previous sections.