Fast parallel Quicksort Project

HAND-IN

Martin Klampfer, 01526110 & Calvin Claus, 01429731
TU WIEN | LVA: 184.710 PARALLEL COMPUTING EINFÜHRUNG PARALLELES RECHNEN

Index

1. Problem statement:

1.1. Problem definition:

Hypothesis: An array of C integers can be sorted in parallel achieving almost linear speedup.

Thus, the parallel algorithms should not suffer from the O(n) time partition bottleneck. The input is given in an array of some C basetype (int or double, …) with "<" as the comparison function

.

1.2. Task:

Implement the specified problem in all three frameworks: OpenMP, Cilk and MPI.

1.3. General Notes:

According to the task, we have to take a close look on the partition function because this is the time bottleneck. Our straightforward and sequential approach to the partition function looks like this:

```
1 void partition(int a[], int start, int end, struct partitionResult * result,
                 int pivotValue) {
2   int aa, i, j;
3   i = start-1; j = end+1;
4
5   for (;;) {
6     while (++i<j&&a[i] < pivotValue);
7     while (a[--j] > pivotValue && j>=start);
8     if (i>=j) break;
9     aa = a[i]; a[i] = a[j]; a[j] = aa;
10  }
11
12  result->smaller = j-start+1;
13  result->larger = ((end - start) + 1) - result->smaller;
14}
```

In this algorithm, start and end are the starting and end indices to partition in the array a. Before the end of the method, two integer values are set in the struct partitionResult: smaller contains the number of elements smaller than the pivot value whereas larger contains the values larger than the pivot value. This method is used in all our implementations (sequential, OpenMP, Cilk and MPI) but we use the values in the struct differently. To evade the time bottleneck here, we use different approaches in the implementations. These approaches are explained further down in this document.

1.4. Referential, sequential implementation:

As sequential reference implementation, we used a straightforward approach, which looks like this in pseudocode:

```
1 void quicksortS(int arr[], int low, int high) {
2   if (low < high) {
3     pivotIndex = get random value between low and high
4     pivotValue = value at position pivotIndex
5     //switch pivot to first element
6     swap(pivotIndex, low)
7
8     partition(arr, low+1, high, &result, pivotValue);
9     pi = new pivot Value index
10    swap(low, pi)
11
12    quicksortS(arr, low, pi - 1);
13    quicksortS(arr, pi + 1, high);
14  }
15}
```

The worst-case space complexity of this algorithm is O(n).
The worst-case performance is $O(n^2)$, the average performance is O(n log(n)).

2. Experimental set up

To test our solutions, we ran them on different systems. During the implementation phase, we tested them on our own machines. The final benchmarking values were then gathered from the machines of the Research Group for Parallel Computing, TU Wien. We started the algorithms in a benchmarking loop with 25 iterations to gather more than one time value and our system outputs the best, worst and average runtime of the algorithm(s).

2.1. The machines:

2.1.1. Shared memory machine

To benchmark the implementations of Cilk and OpenMP we used the machine *saturn.par.tuwien.ac.at*. This is a shared memory machine with four sockets: four AMD Opteron 6168 (12 Cores, 1.9 GHz, 12 MB cache) and 128GB DDR3 memory and a Debian System is running on this machine.

2.1.2. Computer cluster

To benchmark the implementation of MPI we used the machine *jupiter.par.tuwien.ac.at*. This is a computer cluster consisting of 35 compute nodes where each node has two AMD Opteron 6134 (eight cores, 2.3 GHz, 12 MB cache) and 32 GB DDR3 memory per node. The network between the nodes is a QDR InfiniBand and Gigabit Ethernet and CentOS 6 is running on this cluster.

2.1.3. Test data

The algorithms each were tested using three equivalence classes:

0. Periodic
1. Equal
2. Ascending
3. Descending
4. Random

The values in the enumeration correspond to the possible arguments for the –s option in our implementations.

Each implementation was also tested against an empty array and an array with exactly one value.

3. General notes

- OpenMP and Cilk implementations use a constant called *UNIT*. This constant is defined in every c file and is used, to check if a recursive call to the quicksort method is done with an array, which has less than *UNIT* elements. If so, the algorithm switches to the sequential implementation described in 1.4.
- To find a pivot value, all our implementations use the same method: we use a random pivot element. This works fine in most cases but can be a huge performance killer in some cases. We could improve this, so that some elements are taken from the array and the average value is chosen as pivot.
  In our MPI implementation, we use the simple approach, that the process with rank 0 chooses a random pivot from its values and the pivot value is broadcasted afterwards. This could also be improved so that all processes choose a pivot value and they exchange them and agree on the "best" one.
- To generate the main file for OpenMP or Cilk use:

```
make main
```

This program can be run with the following arguments:

- n: array size
- s: type of array (see 0 for the options)
- t: number of threads
- S: seed to generate random numbers
- c: number of repetitions
- a: implementation (o=OpenMP1, O=OpenMP2, c=Cilk1, C=Cilk2)
- A: second implementation for comparison (same options as for a)

To generate the file for MPI use:

```
make mpi
```

This program can be run with the subset of arguments: n, c, s, S. All of them have the same meaning as described above for the main program.

4. OpenMP

4.1. Implementation

We implemented two different versions: one version, where the partition part is not parallelized, and one version with a parallelized partition part. With this approach, we can compare the two different implementations. In theory, the second implementation should not suffer from the O(n) time bottleneck in the partition part.

4.1.1. Without parallelized partition

This implementation differs only a little bit from our sequential implementation as described in 1.4. The difference is that we call the recursive part inside an omp statement:

```
1    #pragma omp parallel
2    #pragma omp for
3    for(int k = 0; k < 2; k++) {
4      if(k == 0) {
5        quicksortOImpl(a, pi, maxThreads);
6      }else {
7        quicksortOImpl(a+pi+1, n-pi-1, maxThreads);
8      }
9    }
```

With this approach, we tried to mimic the behavior of a task parallel quicksort (like Cilk) to spawn two tasks that each sort a part of the array.
We did not intend to use this as our final solution for the OpenMP task but just to see the speedup or possible overhead of the OpenMP calls. Furthermore, we can compare this minimal approach to a minimal Cilk approach that just changes the loop to two cilk_spawn statements and the rest of the algorithm stays the same.

4.1.2. With parallelized partition

This implementation tries to circumvent the O(n) time bottleneck in the partition part. To achieve this, we split the array in parts, where each part is n/threads elements large. N is the array size of this recursive call and threads is the number of processes. Each process independently starts to partition its part of the array. Afterwards a single process builds a prefix sum overall all processes. In this step, the sums of smaller and larger values are built. The next step is parallelized again: every process writes its partitioned array back into the main array to the desired position and the pivot value is written into the correct position. After this step, the pivot value is in the correct position and we can recursively advance. This is done in the same way as described in 4.1.1.
To be able to write back in parallel we need a second array. In our implementation, there is just one helper array, which has the same size as the input array. Every process copies the part he partitioned to the helper array in every recursive call. Then he can write from the helper array to the correct index into the input array.

4.2. How to run

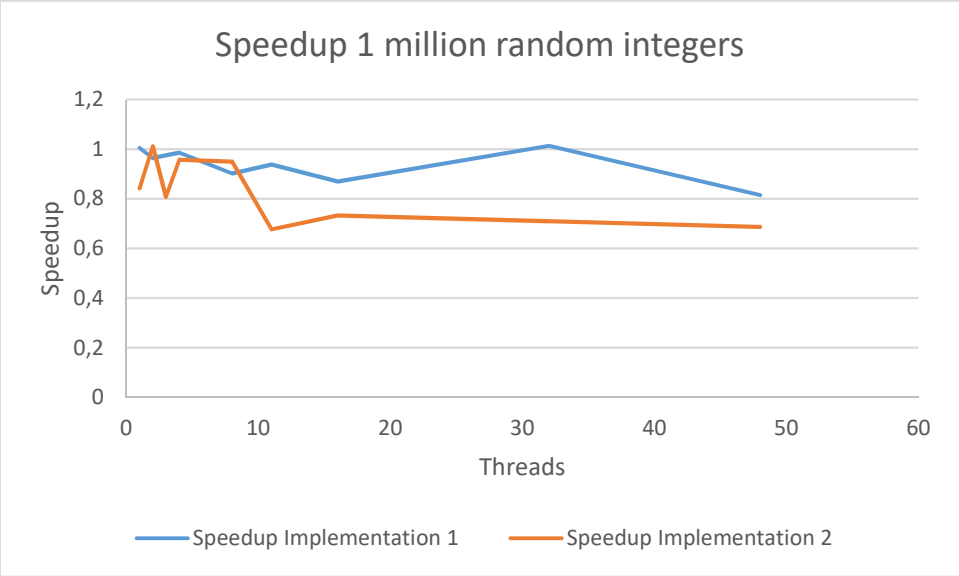
Compile using `make`, then execute `./main`.

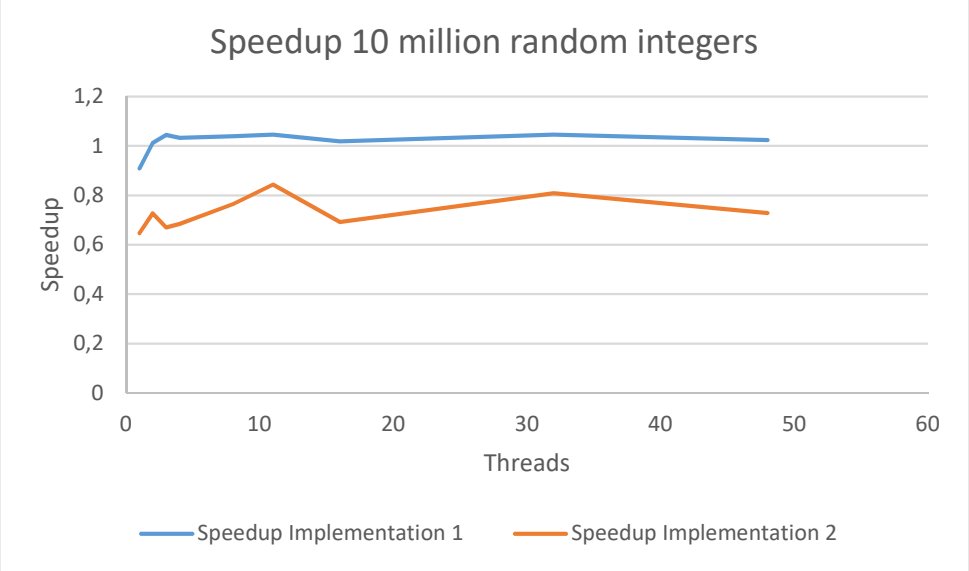
```
make clean && make main && ./main -a (o|O) -n data-size -s kind-of-generated-data
[-A (o|O) -c repetitions]
```

To run the OpenMP implementations, specify –a o for the implementation 1 and –a O for implementation 2. –A can be used for comparisons with when –a has another argument for comparison.

4.3. Experimental results

As you can see in the graphs below, our implementations do not benefit from an increasing number of threads available to them on the Saturn machine. The speedup is, if at all, only very marginal. Worth mentioning here is, that the implementation without parallelized partition (Implementation 1) has a slightly better performance than the one with parallelized partition.





5. Cilk

5.1. Implementation

Similar to our method applied in OpenMP we implemented two algorithms in Cilk. Both use cilk_spawn for each recursive call of quicksort but only one parallelizes the partitioning. We refer to the algorithm without parallel partitioning as A to the other as B.

A works by swapping the pivot element to the beginning of the array and letting the partition function run on indexes 1 to n-1. The partition function sets the number of elements smaller

than the pivot in a struct. We can use this result to swap the pivot currently at [0] with the last element smaller than the pivot value.
We then spawn two cilk threads. One on the data to the left side of the pivot and another on the data to the right side of the pivot.
When n is smaller than the unit value we sort the remaining data sequentially.

B's needs a helper array of size n. This initialization task looks like this:

```
void publicB(int a[], int n, int maxThreads ) {
  int * helperArray = malloc(sizeof(int) * n);
  B(a, n, helperArray);
  free(helperArray);
}
```

A call/recursion of B works like A as long as n is smaller than 10.000. This number has no particular significance and was chosen arbitrarily.
If n is larger than 10.000 a parallel partitioning algorithm is used in which two threads are spawned that each partition half of the data. First each half is partitioned in place using the aforementioned partitioning algorithm. The result of the partitioning process is copied into the helper array.

```
partition(a, low, high, result, pivotValue);
memcpy(helperArray+low, a+low, (sizeof(int) * (high-low+1)));
```

Each invocation of quicksort uses the same indexes in the helper array and a itself. As no thread is working on the same part of a, conflicts are avoided.
The helper array is necessary to allow for the parallel write back of the partition results of the two threads.
To do this we spawn to threads. One is responsible for writing back the values smaller than the pivot from the beginning of a until overallSmaller, the other for the larger values starting from overallSmaller+1 until n-1.

```
cilk_spawn writeBack(helperArray, a, n, &partitionResult1, &partionResult2,
writeSmaller: true);
cilk_spawn writeBack(helperArray, a, n, &partitionResult1, &partionResult2,
writeLarger: false);
```

We then swap the pivot with the element with the largest index of values smaller than the pivot.

```
a[overallSaller] = pivotValue;
```

We compute the number of elements smaller and larger than the pivot by using the output of the partition functions.

```
int overallSmaller = res1.smaller + res2.smaller;
int overallLarger = res1.larger + res2.larger;
```

Finally, we spawn threads for each recursive call.

```
cilk_spawn quicksort2(a, overallSmaller, helperArray);
cilk_spawn quicksort2(a+overallSmaller+1, overallLarger,
helperArray+overallSmaller+1);
```

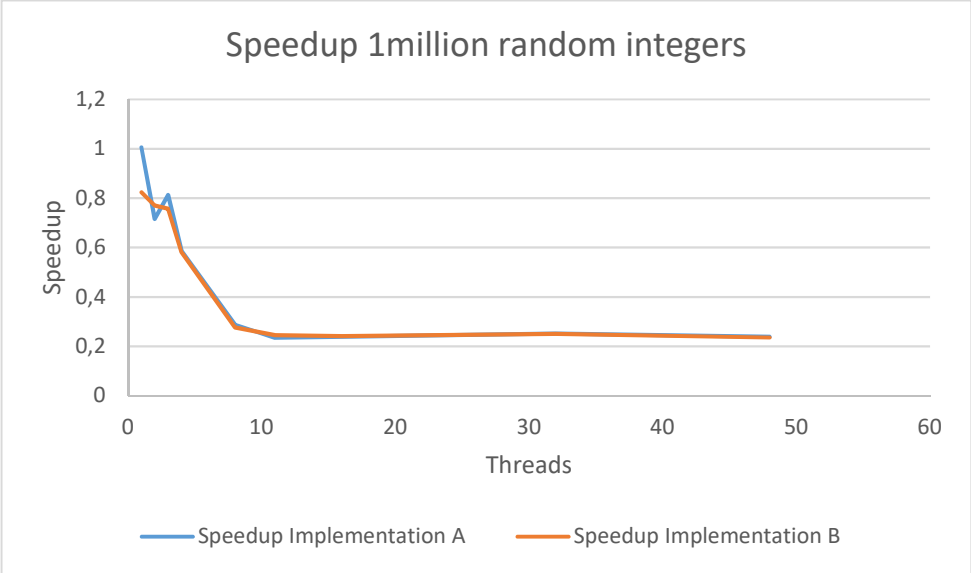
5.2. How to run

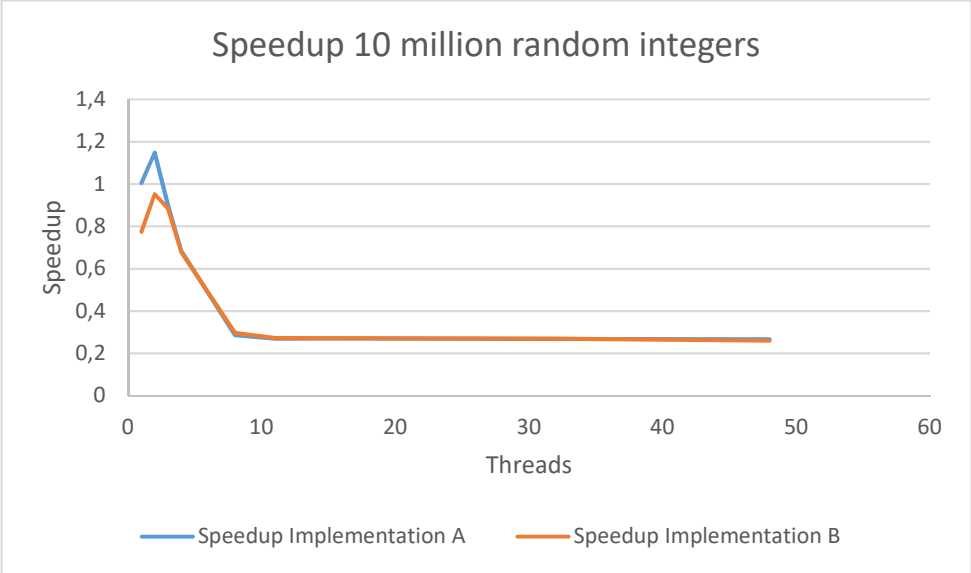
```
make clean && make main && ./main -a (c|C) -n data-size -s kind-of-generated-data
[-A (o|O) -c repetitions]
```

Supply ‘c’ to –a for implementation A and ‘C’ for implementation B.

5.3. Experimental results

As you can see in the graph below, the described algorithms both become slower the more threads are available to them. The speedup is smaller than 1.0 and there is no visible trend of speedup increasing with p.
This shows the unsuitability of our approach to outperform the sequential algorithm. The implementation with parallel partitioning runs just as slow as the simple implementation simply spawning two threads for each recursion.





6. MPI

6.1. Implementation

After the input array is generated, MPI_Scatter is used to assign a part of it to each available processor. Each processor writes its share of the work into a helper array (partialArray).

```
int * a = generateFullArray();
int * partialArray = (int*)malloc(sizeof(int)*(n/size));
MPI_Scatter(a, n/size, MPI_INT, partialArray, n/size, MPI_INT, 0, MPI_COMM_WORLD);
```

We then commence the quicksort on each processor.

```
int newSize;
partialArray = quicksort(partialArray, n/size, MPI_COMM_WORLD, &newSize);
```

quicksort returns the pointer to sorted partialArray. Since processors trade values during the algorithm the returned partialArray does not contain the same elements of the input partialArray. quicksort also uses the pointer newSize to return the potentially changed size of the partialArray.

Each processor calls quicksort. The processor with rank number zero chooses the pivot and broadcasts it to all other processors.

```
MPI_Bcast(&pivotValue, 1, MPI_INT, 0, comm);
```

After the processors all share a common pivot value, they each commence the partitioning process as usual.
The partitioned data is distributed among the processes such that processes with a rank number in the lower half of its group receive the values smaller than the pivot from the processor with `rank+(group_size/2)`. Processes with a rank number larger than or equal to group_size/2 receive the values larger than the pivot from the processor with rank `rank-(group_size/2)`.
To achieve this, each processor sends the size of its result to its partner process first and simultaneously receives this information from its partner process. E.g.:

```
    int partner_process = rank+(group_size/2);
    MPI_Sendrecv(
        &partitionResult.larger, 1, MPI_INT, partner_process, 1,
        &numSmallerFromOtherProcess, 1, MPI_INT, partner_process, 1,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

After the partialArray has been resized and tempPartialArray created to hold the data the processor needs to send to its partner, the following exchanges data between processors:

```
    MPI_Sendrecv(
        tempPartialArray, partitionResult.larger, MPI_INT, partner_process, 2,
        partialArray + partitionResult.smaller, numSmallerFromOtherProcess
        MPI_INT, partner_process, 2,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

The communicator is then split via `rank_in_communicator_for_group<(group_size/2)`. This way the broadcasting of the pivot element in the next recursion is easily implemented.
This process is continued recursively until a process finds itself to be the only processor remaining in a communicator. In this case, the remaining array is sorted sequentially.

Finally, the partial arrays returned by each processor are written back into the original array in the right order.
First, the process with rank zero gathers the size of each processors resulting partial array.

```
int * elementsPerProcess = NULL;
int * displacementPerProcess = NULL;
if (rank == 0) {
  elementsPerProcess = (int*) malloc(size*sizeof(int));
  displacementPerProcess = (int*) malloc(size*sizeof(int));
}
MPI_Gather(&newSize, 1, MPI_INT, elementsPerProcess, 1, MPI_INT, 0
       MPI_COMM_WORLD);
```

We then create the displacementperProcess to be able to effectively use Gatherv.

```
  if (rank == 0) {
    int numValues = 0;
    for (int i = 0; i < size; i++) {
      displacementPerProcess[i] = numValues;
      numValues += elementsPerProcess[i];
    }
  }
  MPI_Gatherv(partialArray, newSize, MPI_INT, a, elementsPerProcess,
       displacementPerProcess, MPI_INT, 0, MPI_COMM_WORLD);
```

6.2. How to run

The number of processors must be a power of two.
The size of the input data must be divisible by the number of processors.

```
make clean && make mpi && srun -N num_cores -p q_student --ntasks-per-
node=tasks_per_node ./mpi -n data_size -s kind_of_data -S seed -c repetitions
```

E.g.:

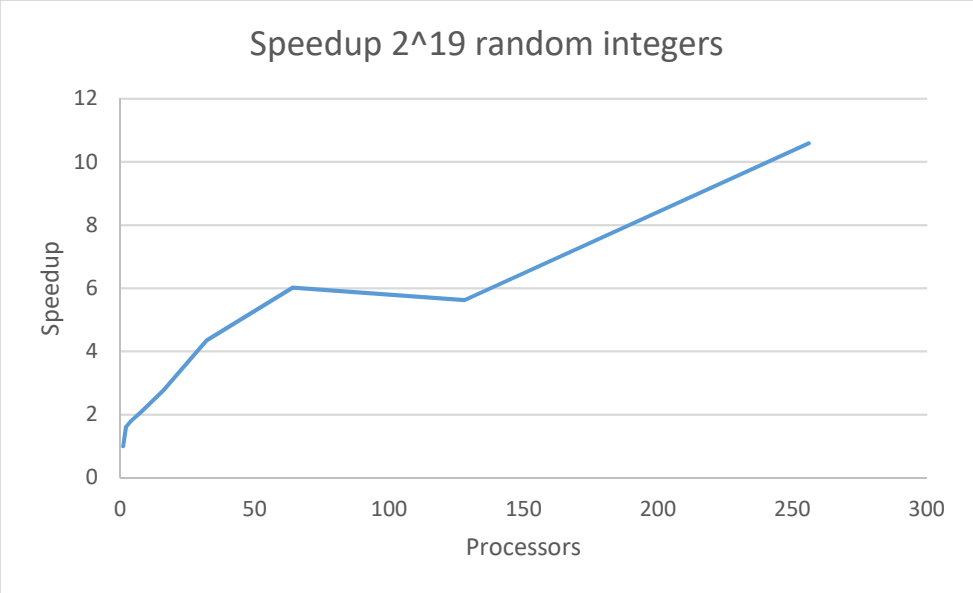
```
make clean && make mpi && srun -N 32 -p q_student --ntasks-per-node=8 ./mpi -n
1048576 -s 4 -S 987654321 -c 10
```

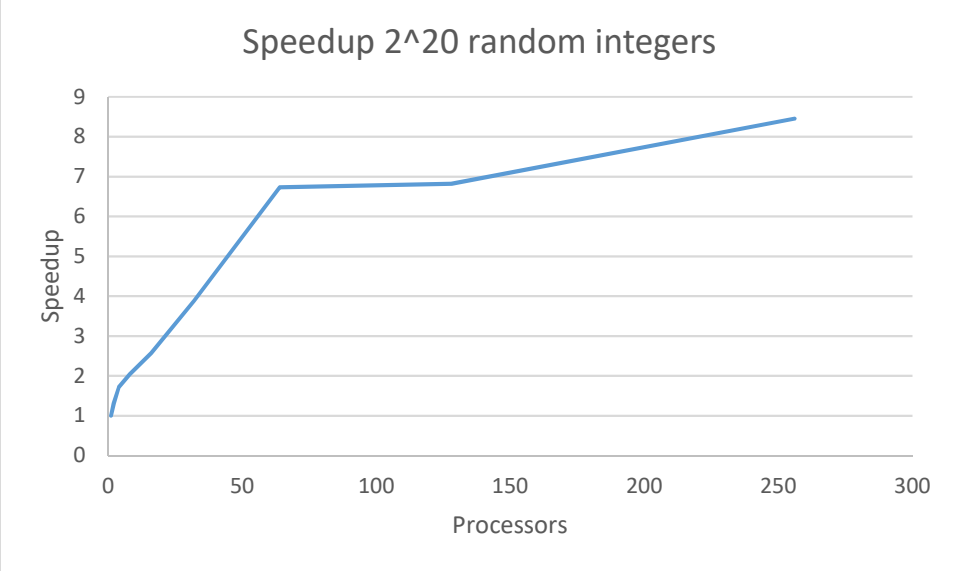
6.3. Experimental results

We ran the application with 1 to 256 processors in steps of 2^i. The inputs for `-N` and `--ntasks-per-node` for each trial are displayed below.

Processors	-N	--ntasks-per-node
1	1	1
2	2	1
4	2	2
8	4	2
16	8	2
32	8	4
64	16	4
128	16	8
256	32	8

The following figures show the speedup for 2^{19} and 2^{20} elements:





7. Summary

Looking back at the implementation and testing process, the thing that surprised us most is the difference that CPU clock speeds or different system configurations can make. We implemented and tested each implementation locally on our machines and even got some decent looking speedup values for 2-4 threads on our own machines with both Cilk Implementations and both OpenMP Implementations. Then we uploaded them to Saturn and there we could not even get any speedup or only a marginal one. To show this, we have one example from OpenMP:

> With the sequence parameter –s 0 (periodic sequence) and using only 2 threads the mean runtime on a system with 64 bit Windows 10 (Build 16299.192) running on an Intel Core i5-4670k (3.4 GHz overclocked to 4.2 GHz) with 16 GB DDR3 memory is 0.492 seconds with 10 million elements to sort. On comparison, the runtime with the same parameters on Saturn is 1.427 seconds. For sure, we have to take into account that the pivot value is chosen randomly but we ran this test multiple times and the Saturn machine is slower with a factor around 3.

When trying to compare the different frameworks we used, we would say that Cilk is the easiest to use and it is a good idea to start with Cilk. With that framework, everyone can simple start new threads on every recursive call and see the improvement of the algorithms. However, it is not easy to parallelize the partition step with Cilk, which is much easier in OpenMP. There you can use the keywords to parallelize the loops needed in the partition step. One drawback is the missing cilk_spawn counterpart in OpenMP (we were not allowed to use OpenMP tasks) that makes the parallelization of the recursive calls difficult.

MPI on the other hand forced us to completely think in another way and to focus on the messages and data that needs to be sent. This was strange in the beginning but we managed to get the code up and running pretty fast after an initial thinking phase. The problems arose then, when we had some errors in our implementations and even got segfaults. With the parallel execution, it was really hard debugging and finding the errors.