CSE 6220, Spring 2015

Programming Assignment 3 - Report

Ankit Srivastava & Cansu Tetik

February 15, 2017

1 Problem Description

Given a list of n integers, block distributed on p processors, we want to sort the integers using a parallel version of the sequential quicksort algorithm.

2 Algorithm Design Description

In this section, we give a brief description of the algorithms that we implemented for solving the given problem.

2.1 Sequential Algorithm

Since the problem description said that any algorithm can be used for sorting sequentially, we used C++ *algorithm* library's *std::sort* function for sorting when $p = 1$.

2.2 Parallel Algorithm

For sorting the given n elements in parallel on p processors, using quicksort, we partition the given array into two parts using a randomly chosen pivot: one consisting of elements less than or equal to the pivot and the other consisting of elements greater than the pivot. Then, based on the sizes of the two partitions, we split the p processors into two parts and block distribute the elements in both the partitions on the corresponding processors. We then sort the two partitions recursively and later redistribute the two sorted partitions on p processors.

We briefly explain our implementation of the given function, and a few other utility functions that we added, below.

- `parallel_sort`: This is the function that is called by the framework for sorting in parallel. We implemented the above defined algorithm primarily in this function. The function first checks if the communicator size is 1 and sorts sequentially if that is the case. If the communicator size is more than 1, it chooses a random pivot and partitions the array into two parts, using `partition` function described later, and splits the communicator based on the partition sizes. The partitioned data is then distributed on the corresponding processors, using `distribute_data` function. We then call `parallel_sort` again, for sorting the partitioned array recursively. Once the two partitions are sorted, we redistribute the sorted elements in the two partitions as per the original sizes, using `collect_data` function.

- `partition`: This function works like a sequential partition function for quicksort and is used by all the processors to partition their local data by the pivot, in place.

- `distribute_data`: This function moves data in a particular partition on all the processors to the processors assigned to sort elements in that partition. The function tries to minimize the number of required communications by only moving data whose movement is required i.e. data in this partition that resides on processors assigned to the other partition. It calls `MPI_Alltoallv` once for distributing the data.

- `collect_data`: This function collects sorted data in the processors assigned to the two partitions and block distributes it in the original configuration passed to `parallel_sort` that called this function. Again, it calls `MPI_Alltoallv` once for collecting the data.

Pseudocode for the parallel algorithm is shown in Algorithm 1.

Algorithm 1 Parallel

procedure LocalPartition($begin$, end, $pivot$)
    $left \leftarrow 0$
    $right \leftarrow (end - begin) - 1$
    while $left < right$ **do**
        if $pivot \in [a[right], a[left])$ **then**
            Swap $a[left]$ and $a[right]$
        if $a[left] \leq pivot$ **then**
            $left \leftarrow left + 1$
        if $a[right] > pivot$ **then**
            $right \leftarrow right - 1$
    return $left$
procedure DistributeData($sbuf$, $current$, $rbuf$, $final$, $communicator$)
    $q \leftarrow$ Size of $communicator$
    $r \leftarrow$ Rank in $communicator$
    $extra \leftarrow$ Vector of size q
    $need \leftarrow$ Vector of size q
    for $p \in [0, q)$ **do**
        if $final[p] < current[p]$ **then**
            $extra[p] \leftarrow current[p] - final[p]$
        else
            $need[p] \leftarrow final[p] - current[p]$
    $cumulativeNeed \leftarrow$ Prefix sum of $need$
    $cumulativeExtra \leftarrow$ Prefix sum of $extra$
    $scounts \leftarrow$ Vector of size q
    $rcounts \leftarrow$ Vector of size q
    $extraMin \leftarrow cumulativeExtra[r] - extra[r]$
    $extraMax \leftarrow cumulativeExtra[r] - 1$
    $needMin \leftarrow cumulativeNeed[r] - need[r]$
    $needMax \leftarrow cumulativeNeed[r] - 1$
    for $p \in [0, q)$ **do**
        if r is p **then**
            continue
        if $cumulativeExtra[p] \in (needMin, needMax + extra[p]]$ **then**
            $procMin \leftarrow \max(cumulativeExtra[p] - extra[p], needMin)$
            $procMax \leftarrow \min(cumulativeExtra[p] - 1, needMax) + 1$
            $rcounts[p] \leftarrow procMax - procMin$
        if $cumulativeNeed[p] \in [extraMin, extraMax + need[p])$ **then**
            $procMin \leftarrow \max(extraMin, cumulativeNeed[p] - need[p])$
            $procMax \leftarrow \min(extraMax, cumulativeNeed[p] - 1) + 1$
            $scounts[p] \leftarrow procMax - procMin$

```
    if extra[r] > 0 then
        scounts[r] ← final[r]
        rcounts[r] ← final[r]
    else
        scounts[r] ← current[r]
        rcounts[r] ← current[r]
    sdispls ← Offsets corresponding to scounts
    rdispls ← Offsets corresponding to rcounts
    MPI_Alltoallv(sbuf, scounts, sdispls, rbuf, rcounts, rdispls)
procedure CollectData(sbuf, current, rbuf, final, communicator)
    q ← Size of communicator
    r ← Rank in communicator
    cumulativeCur ← Prefix sum of current
    cumulativeFin ← Prefix sum of final
    scounts ← Vector of size q
    rcounts ← Vector of size q
    currentMin ← cumulativeCur[r] − current[r]
    currentMax ← cumulativeCur[r] − 1
    finalMin ← cumulativeFin[r] − final[r]
    finalMax ← cumulativeFin[r] − 1
    for p ∈ [0, q) do
        if cumulativeCur[p] ∈ (finalMin, finalMax + current[p]] then
            procMin ← max(cumulativeCur[p] − current[p], finalMin)
            procMax ← min(cumulativeCur[p] − 1, finalMax) + 1
            rcounts[p] ← procMax − procMin
        if cumulativeFin[p] ∈ [currentMin, currentMax + final[p]) then
            procMin ← max(extraMin, cumulativeNeed[p] − need[p])
            procMax ← min(currentMax, cumulativeFin[p] − 1) + 1
            scounts[p] ← procMax − procMin
    sdispls ← Offsets corresponding to scounts
    rdispls ← Offsets corresponding to rcounts
    MPI_Alltoallv(sbuf, scounts, sdispls, rbuf, rcounts, rdispls)
procedure ParallelSort(begin, end, communicator)
    q ← Size of communicator
    r ← Rank in communicator
    localSize ← end − begin
    m ← Sum of localSize across all the processors in communicator
    if q is 1 then
        std::sort(begin, end)
    minIndex ← Global minimum index in this processor
```

```
80:   Seed the random number generator with the level of recursion
81:   pivot ← Random pivot element
82:   boundary ← LocalPartition(begin, end, pivot)
83:   paritionSizes ← {boundary + 1, localSize − (boundary + 1)}
84:   currentSizes ← Gather partitionSizes from all the processors
85:   sumSizes ← Sum of both partitionSizes across all the processors
86:   numProcessors ← Split q into two based on sumSizes
87:   color ← 0 if r < numProcessors[0] else 1
88:   newCommunicator ← MPI_Comm_Split(communicator, color, r)
89:   finalSizes ← Block distribution of sumSizes on numProcessors
90:   finalSize ← Block distribute sumSizes on numProcessors
91:   finalBuf ← Temporary buffer of finalSize
92:   dataOffset ← 0
93:   for i ∈ {0, 1} do
94:       current ← currentSizes[color]
95:       final ← finalSizes[color]
96:       sbuf ← begin[dataOffset]
97:       rbuf ← finalBuf if i == color else NULL
98:       DistributeData(sbuf, current, rbuf, final, newCommunicator)
99:       dataOffset ← (boundary + 1)
100:  newBegin ← Initial pointer of finalBuf
101:  newEnd ← Final pointer of finalBuf
102:  ParallelSort(newBegin, newEnd, newCommunicator)
103:  original ← Block decompose m on q processors
104:  CollectData(newBegin, current, begin, original, communicator)
```

3 Plots and Observations

For running the experiments, we had only two parameters that we could vary: number of elements to be sorted (n) and the number of processors used for sorting (p). We obtained timings for different combinations of the two parameters and plotted and analyzed the run times so obtained. In the following sections, we discuss how we varied the parameters and the effect that the variation had on the performance.

We used *jinx4*, which has 4×6 cores, for running the experiments up to $p = 24$ and used 5 *sixcore* nodes for running the experiments for $p = \{26, 28, 30\}$.

3.1 Array Size (n)

First of all, we had to choose a suitable problem size for running our experiments. During some preliminary experimentation, we observed that the implementation started running out of memory while sorting 1 billion elements on higher number of processors. Since *jinx* cluster has low memory per node (maximum of 24 GB on one node) and 1 billion integers means 4 GB of memory, the memory could quickly run out if it requires a lot of recursions to solve the problem. Therefore, this was expected.

The implementation was able to sort arrays of size up to 900 million, up to $p = 24$ without any memory issues. However, for $p \geq 26$ the 900 million array size also starts failing because of insufficient memory. Therefore, we chose to vary the array size in the set
$\{500000000, 600000000, 700000000, 800000000, 900000000\}$. The run times so obtained, for different problem sizes, are plotted in Figure 1 and the corresponding speedups, with respect to the run time of $p = 1$, are plotted in Figure 2.

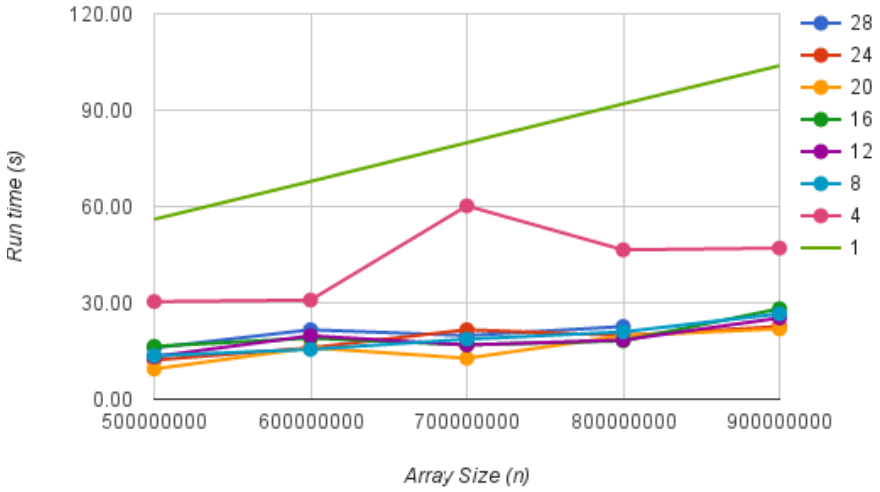

Figure 1: Run time v/s Array Size

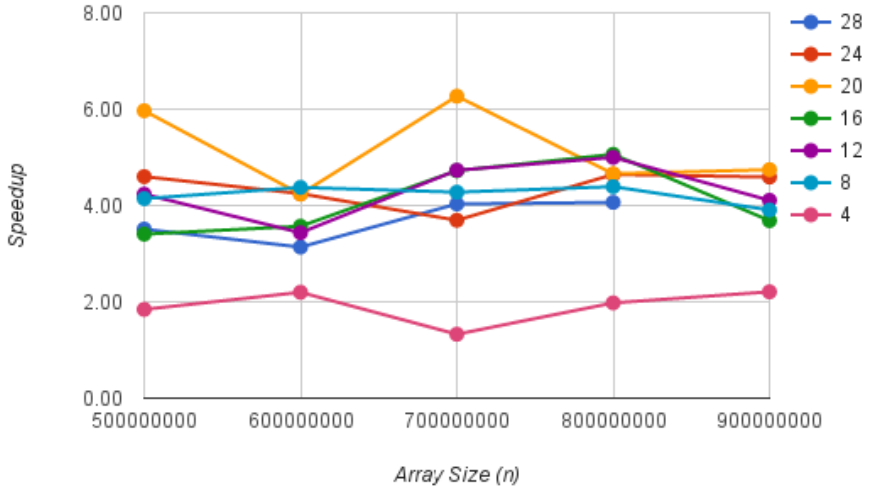

Figure 2: Speedup v/s Array Size

It can be seen from the run time plot that the run time generally increases on increasing the problem size and is maximum for $p = 1$, followed by $p = 4$ case. However, the distinction between run times on higher number of prcocessors is less clear. Therefore, we refer to the speedup plot where we can see that the speedup remains approximately the same on increasing the problem size. This was expected since the speedup depends on the number of processors used. We

did observe that the maximum speedup was obtained for $n = 700000000$, using $p = 20$.

3.2 Number of Processors (p)

We varied number of processors in the range [1, 30], in jumps of 2, and obtained the run times for all the problem sizes discussed in Section 3.1. The run time and speedup plots so obtained are plotted in Figure 3 and Figure 4. As discussed in Section 3.1, we didn't have any data for $p \geq 26$ and $n = 900000000$ and therefore the corresponding data points are missing from the plots.

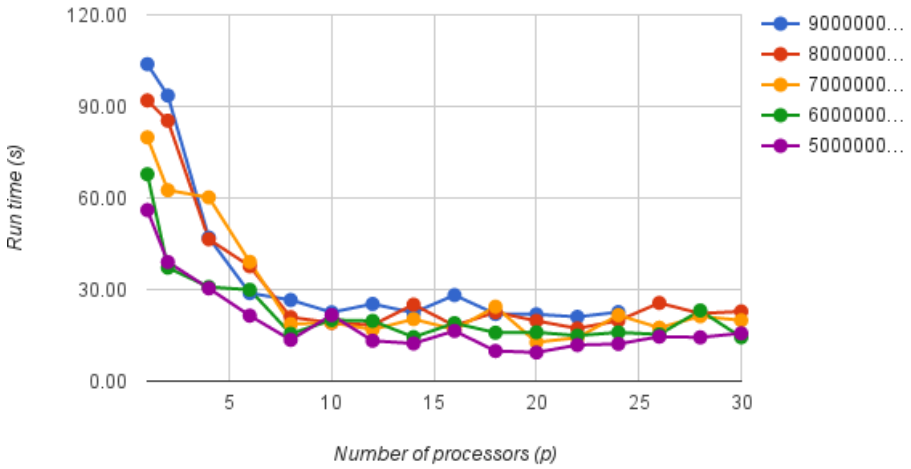

Figure 3: Run time v/s Number of Processors

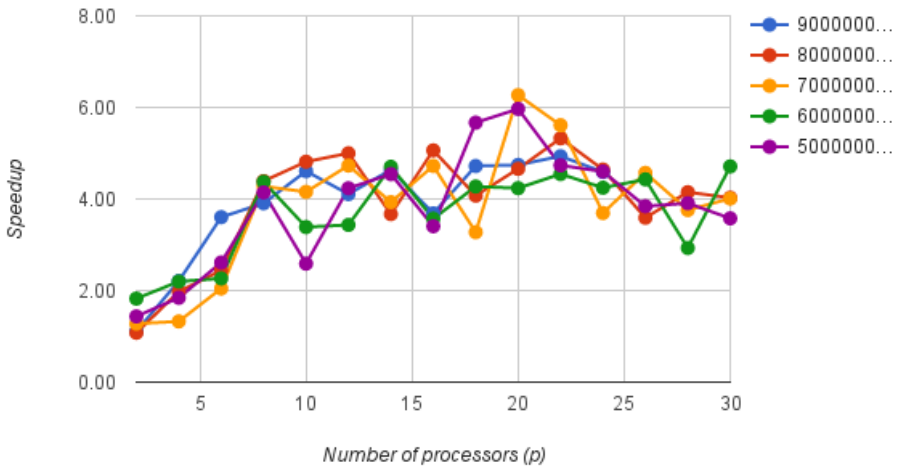

Figure 4: Speedup v/s Number of Processors

From the above plots, we can see that on increasing the number of processors the runtime decreases up to $p = 20$ and then starts increasing. Thus the observed speedup is highest, for most problem sizes, for $p = 20$. This wasn't expected and could be because of the choice of pivot or because of the low memory on the nodes.