A Recursive Parallel Quick Sort Algorithm on a Message Passing Machine

Quicksort is one of the most popular algorithms for sequential computers because of its simplicity, low overhead and optimal average complexity. Quicksort algorithm is an ideal candidate for recursive implementation as it is based upon the recursive decomposition of the input data. In this report, I discuss the various issues involved in parallelizing the Quicksort algorithm on a Message Passing Machine as well as give an algorithm that is aimed at solving these issues.

1 The issues involved in Parallelizing Quicksort effectively.

There can be a number of ways in which the quick sort algorithm can be parallelized, one of which comes directly to mind when we look at the linear recursive quick sort algorithm and can’t help feeling that it is inherently parallel, what we have to do is just let each process deal with the each new recursive step. But this trivial approach suffers from performance degradation on two grounds. One, if we let one process deal exclusively with the whole of the dataset initially, then all of the other potential processes will be sitting idle at that time. Second, the load balancing is poor in this case as each new process has to deal with almost half of the problem size as compared to its parent. Moreover, creating processes only at the recursive steps affects the maximum degree of concurrency badly and also since the root process has to partition the whole data set serially, it is a big blow to the parallel algorithm performance. Therefore, any parallel implementation of Quicksort will have to start with all the potential processes (say P) alive at the time of the start and each one dealing with its own N/P elements of the dataset where N is the total number of elements in the dataset. Since the output of the parallel Quicksort is defined to be that all the elements within each process are sorted and all the elements in Process Pi are greater than or equal to the largest number in process Pi-1 and smaller than or equal to the smallest number in process Pi+1 for all processes.

I will now discuss various steps of the Quicksort algorithm and the issues involved with them in a message passing paradigm using MPI.

2 The Pivot Selection

First I’ll explain why the pivot selection strategy affects the overall performance of the Quicksort so heavily. The reason is simple; Quicksort for its optimal performance requires the pivot element to be the ‘middle one’ of the output sorted dataset i.e the pivot should lie in the centre of the particular dataset elements’ distribution. If it does not happen to be the case and the pivot element comes out to be the extreme left or right of the distribution then Quicksort’s performance degrades from O (nlogn) to O(n^2) which shows exactly how important is pivot selection criteria for efficient Quicksort implementation.

There are a lot of approaches to pivot selection while implementing serial Quicksort. These include and are not limited to picking up first, last or a random element of the dataset as a pivot. It has been well known that any of the deterministic pivot selection mechanism will suffer heavily from performance degradation when the distribution of the data is not suitable to that particular criterion. For example, let’s suppose we choose the first element to be our pivot then if we encounter data that is already ordered in decreasing fashion we’ll end up paying O (n^2) cost for Quicksort implementation as its effects are not limited to just one partition of the dataset, in fact it affects each new partition step as badly as the first one. That is why picking up a random pivot has almost become a standard practice in implementing Quicksort (at least on
serial machines) as it averages out the effects of picking up a bad pivot with those of good pivots. Good pivots here refer to those pivots which when selected take the performance towards the optimal one and the bad pivots are those which do exactly the opposite.

Now the selection of random pivot might result in better performance on serial machines but it usually is not the case on a parallel machine as it can lead to unbalanced load. As opposed to the serial machine, on a parallel machine there is no way to return once a bad selection of the load per process is made. Therefore another approach which involves selection of the median of the dataset as the pivot is considered much useful in a balanced-load efficient parallel Quicksort implementation. In this approach, each process has to sort the elements first to calculate median, this is one potential blow to this algorithm as we are paying $O(n/p \log n/p)$ cost just to achieve load balancing. To start with I’ve implemented a random pivot selection approach adopted for what I think considerable amount of load balancing without having to sort each individual process’ dataset first. I was looking forward to compare it with the median based approach but due to lack of time, I think, I wouldn’t be able to do that.

3 Selection of the Global Pivot and Partitioning of the data

To obtain the desired output we must select an overall pivot for all processes so that data can be partitioned based upon this global pivot giving way to the required organization of the sorted data. Once each process has selected its own pivot, they communicate with each other to decide upon one global pivot. I’ve used `MPI_Allgather` to collect all the selected pivots at each process. Each process selects a random pivot which is taken as the global pivot. Now to make sure that each process picks the same random pivot, I’ve seeded the random number generator at each process with the same number to generate the same random pivot index at each process. In fact, I’ve used the recursive step number of the Quicksort as the seed. That means two different steps of the Quicksort will have different random pivots but within the same step, each process will have the same random index to select global pivot from. This also saves extra communication as we don’t have to broadcast one global pivot to each process. (As per requirement of using `MPI_Bcast`, yes I had actually used it in my initial implementation but then dropped due to the reason mentioned above. You may find it in some commented portion of the code).

After each process has determined the global pivot, it then partitions its local data in two sets one which contains all elements less or equal to the global pivot and the other with all elements greater than the global pivot. At this point we are faced with the selection of optimal number of processes for each task of dealing with each set. This selection criterion is discussed next.

4 Task to process mapping

As is clear from above, after partitioning the local data according to the global pivot; each process knows exactly how many elements within its local dataset are less or equal to the global pivot and how many of them are greater than the global pivot. In order to achieve optimal task to process mapping, we first determine the total number of all such elements over all processes and distribute it to all processes. This all is done in one step using `MPI_Allreduce`. After this each process knows exactly how many of the elements in the global data set are less than/equal to the global pivot and how many of them are greater than it. Based upon this knowledge, each process can determine how many processes will be required for each set of elements in the next phase. We take the ratio of these total numbers and based upon this ratio, set aside specific number of processes for each set.

5 Creating new communicators for the new mapping

Since we are aiming at a recursive implementation of the Quicksort, that means we will look to apply same steps as we have mentioned in the previous sections on the newly organized data sets. We have seen in the previous sections that there were collective operations performed by the processes which involve all other processes in the domain (we’ll call it by MPI specific word communicator from now on). Now one of the properties associated with the new data sets is that they are totally independent of each other. That means the operations performed over these data sets and the processes working with these data sets are to work independent of each other. In fact interaction with processes working on the other data set can lead up to erroneous results. Therefore, the recursive implementation of the Quicksort not only demands the recursive
decomposition of the data but also of the communicator. MPI provides a number of ways to define new communicators, `MPI_Comm_split` and `MPI_Comm_create` are one of them. A good thing about `MPI_Comm_split` is that it saves a lot of development time by abstracting lot of things from the developer while creating new communicator from an existing communicator as opposed to `MPI_Comm_create` which also can be reckoned as its drawback as it does not give the flexibility that `MPI_Comm_create` gives. In order to achieve full flexibility while creating new communicators from the existing one, I’ve used `MPI_Comm_create`. The process of creating a new communicator using `MPI_Comm_create` consists of following steps.

1. Determine the `MPI_Group` that consists of all the processes in the parent communicator using `MPI_Comm_group`.
2. Make two new `MPI_Groups` of processes from the parent group using `MPI_Group_incl` which requires that the ranks of the processes that are to be included in the new `MPI_Group` be given explicitly in the form of an array. This array contains ranks in the old communicator of those processes which are to be included in the new `MPI_Group`. Note that the processes have different ranks in the new groups than the parent group. For example in the definition of the new group, the process at the index ‘0’ in the ranks array (which is passed as a parameter to the new group constructor) has a rank ‘0’ in the new group.
3. After the new `MPI_Groups` are created, they can be passed as a parameter to `MPI_Comm_create` to create a communicator based upon a specific group.

The result of the above mentioned steps result in a new configuration which is given below.

Figure 1: Illustration of the Communication World after two new communicators are created. Here ns = number of processes fixed for smaller than global pivots elements and ng = number of processes fixed for greater than global pivot elements

6 Distributing data to respective groups

Before we go on a recursive call to the Quicksort algorithm based upon new communicators, we need to distribute data from each process to the respective group process. For example the processes reserved for less-than/equal-to global pivot data elements contains elements greater than global pivot in a separate set which is to be sent to the respective process in the group fixed for greater-than global pivot data elements using the parent communicator.

There can be different scenarios based upon the number of processes fixed for each kind of set. Following is the complete illustration of each of these scenarios using total number of processes P = 4.
Figure 2: When processes working on the two data sets are same in number. Each arrow represents the transfer of elements to the corresponding group process. The sending process first sends the size of the elements to be sent then the elements themselves. The scheme is a simple one and is prone to a little bit of load unbalancing. The load unbalancing occurs when the two processes on the left side have to send drastically different amounts of data to the right side. The remedy can be the distribution of elements from the process in the left side (shown by green color circles) to all the processes on the right side (processes shown by red color circles).

Figure 3: The processes allocated for smaller than global pivot elements (shown by green circles) are more than the processes fixed for larger than global pivot elements. As P3 has already sent its local set of smaller elements to P0. It just has to receive P1 and P2’s set of larger elements. The technique used is that whenever a process finds out that it does not have corresponding process in the other group, it simply falls back to the last process of the other group (P3 in the illustration).
Figure 4: Processes allocated for smaller numbers are smaller than the processes allocated for larger numbers. Since P0 has sent its local set of larger elements to the P1, it only receives smaller elements’ sets from P2 and P3.

7 Recursive Call to Quicksort

After all the data movement steps are over, each process makes a recursive call to the Quicksort with one of the new communicators (that this process is a member of) as a parameter. The recursive calls reach the base case when there is only one process left in the communicator in which case a linear quick sort is applied on the local data set elements. Each process may have different number of elements in the final output. All of the processes have elements organized in the required format.

8 Pseudo code for Recursive Message Passing Parallel Quicksort algorithm

Procedure Quicksort(in_array[],first,last,num_procs,comm)
Begin
If num_procs := 1 then
    Linear_quicksort(in_array,first,last);
Return
Else
    PivIndex := partition(in_array,first,last,num_procs,comm);
    Num_s := pivIndex – first+1;
    Num_g := last – pivIndex;
    For i:= 0 to pivIndex do
        S_array[i] := in_array[i]
    End for
    For i:=0 to Num_g do
        G_array[i] = in_array(pivIndex+i+1);
    end for
    MPI_Allreduce(&Num_s,&total_s,1,MPI_INT,MPI_SUM,comm);
    MPI_Allreduce(&Num_g,&total_g,1,MPI_INT,MPI_SUM,comm);
    If total_g > 0 then
        Begin ratio = total_s/total_g
        Num_s_procs = ratio*num_procs/2;
        If Num_s_procs >= num_procs then
            Num_s_procs = Num_s_procs -1
        end if
    end if
End

Endif
If Num_s_procs <= 0 then
    Num_s_procs := 1
End if
Num_g_procs := num_procs – num_s_procs;
Else
    Num_g_procs := 1
    Num_s_procs = num_procs – num_g_procs;
End if
For i=0 to num_S_procs do
    ranks1[i] = i;
end for
For i=0 to num_G_procs do
    ranks2[i] = i+num_S_procs;
end for
MPI_Group_incl(orig_group,num_S_procs,ranks1,&new_groupS);
MPI_Group_incl(orig_group,num_G_procs,ranks2,&new_groupG);
MPI_Comm_create(comm,new_groupS,&new_commS);
MPI_Comm_create(comm,new_groupG,&new_commG);
If new_commS != MPI_COMM_NULL then
    MPI_Comm_size(new_commS,&comm_s_size)
End if
If new_commG != MPI_COMM_NULL then
    MPI_Comm_size(new_commG,&comm_g_size)
End if
If  rank >= num_S_procs then
    /* Broad cast the lower numbers than the pivot to the lower communicator processes*/
    /*First send the size of the s_array*/
    MPI_Send(&num_s,1,MPI_INT,rank%num_S_procs,tag,comm);
    MPI_Send(&s_array,num_s,MPI_INT,rank%num_S_procs,tag+1,comm);
End if
if rank < num_S_procs then
    for i =0 to num_s do
        in_array[i] = s_array[i];
    end for
    total_S_size = num_s;
    for i = num_S_procs to num_procs do
        if((rank+i)%num_S_procs == rank && (rank+i)<num_procs)
            source =;
            MPI_Recv(recv_S_size,1,MPI_INT,source,tag,comm,&status);
            MPI_Recv(recv_S_array,recv_S_size,MPI_INT,source,tag+1,comm,&status);
            For i =total_S_size to total_S_size+recv_S_size then
                in_array[i] = recv_S_array[i-total_S_size];
            end for
            total_S_size = total_S_size+ recv_S_size;
        end if
    end if
    if num_S_procs > 1 then
        quicksort(in_array,0,total_S_size-1,num_S_procs,new_commS);
    end if
else
    linear_quicksort(in_array,0,total_S_size-1);
end if
    MPI_Send(&num_g,1,MPI_INT,rank+num_S_procs-1,tag,comm);
MPI_Send(&g_array,num_g,MPI_INT,rank+num_S_procs-1,tag+1,comm );

int recv_G_size;
int recv_G_array[num_per_proc];

if rank < num_S_procs then
    if (rank+num_S_procs) < num_procs then
        /* Broadcast the higher numbers than the pivot to the higher communicator processes*/
        /*First send the size of the g_array*/
        MPI_Send(&num_g,1,MPI_INT,rank+num_S_procs,tag,comm);
        MPI_Send(&g_array,num_g,MPI_INT,rank+num_S_procs,tag+1,comm );
    End if
Else
    int dif = rank+num_S_procs-(num_procs-1);
    MPI_Send(&num_g,1,MPI_INT,rank+num_S_procs-dif,tag,comm);
    MPI_Send(&g_array,num_g,MPI_INT,rank+num_S_procs-dif,tag+1,comm );
/* Receive the Larger numbers arry and its size from rank%num_S_procs if in the sammler numbers group*/
    if rank >= num_S_procs then
        for i =0 to num_g do
            in_array[i] = g_array[i];
        end for
        total_G_size = num_g;
        MPI_Recv(&recv_G_size,1,MPI_INT,rank%num_S_procs,tag,comm,&status);
        MPI_Recv(&recv_G_array,recv_G_size,MPI_INT,rank%num_S_procs,tag+1,comm,&status);
        For i := total_G_size to total_G_size+recv_G_size do
            in_array[i] = recv_G_array[i-total_G_size];
        end for
        total_G_size = total_G_size+recv_G_size;
    if num_G_procs < num_S_procs then
        int offset = num_S_procs-num_G_procs;
        if rank := num_procs-1 then
            for i= 1 to offset then
                MPI_Recv(&recv_G_size,1,MPI_INT,rank-num_S_procs+i,tag,comm,&status);
                MPI_Recv(&recv_G_array,recv_G_size,MPI_INT,ranknum_S_procs+i,tag+1,comm,&status);
            For j =total_G_size to total_G_size+recv_G_size do
                in_array[j] = recv_G_array[j-total_G_size];
            end for
            total_G_size = total_G_size+recv_G_size;
        end if
    if(num_G_procs > 1){
        quicksort(in_array,0,total_G_size-1,num_G_procs,new_commG);
    end if
else
        linear_quicksort(in_array,0,total_G_size-1);
    end else
if rank<num_S_procs then
    MPI_BARRIER(new_commS);
Else
    if num_S_procs > 1 then
        quicksort(in_array,0,total_S_size-1,num_S_procs,new_commS);
    end if
else
    linear_quicksort(in_array,0,total_S_size-1);
end else
end if
else
    MPI_BARRIER(new_commG)
If num_G_procs > 1 then
    quicksort(in_array,0,total_G_size-1,num_G_procs,new_commG);
end if
else
    linear_quicksort(in_array,0,total_G_size-1);
end else
end if
end if

9 Complexity Analysis using Bulk Synchronous Parallel (BSP) model

Consider a message passing computer with P processes each storing N/P data elements in its local memory. The recursive parallel Quick sort algorithm proposed above would involve following computations in each superstep. Since each superstep can comprise only of the computations done on the local data and not the message communication, we’ll have to break apart our algorithm into recursive calls to a serial flow of supersteps. In each call to the Quick sort routine, following will be the computation steps (super steps) performed by each process.

1. Selection of the random pivot – constant time
2. Selection of the global pivot from received array of pivots. And partitioning (splitting) of the local data based upon the global pivot. – Constant time for the selection of the global pivot and O (N/P) for splitting the locally assigned portion of array.
3. At the base case of the algorithm, each process will linearly sort its local data set taking O (N/P log N/P) time.

The communication steps during each call to the Quick sort algorithm are

1. After superstep1, MPI_Allgather is performed to collect pivots at each process. In MPI_Allgather, each process receives data from all other processes and sends its local dataset to all other processes. If P processes are involved in MPI_Allgather then each process will send P-1 messages and will receive P-1 messages. Thus there will be total \( P^2-P \) messages being sent in the system and \( P^2-P \) messages being received. Therefore the cost \( h \) associated with this communication step would be \( P^2-P \), while \( g \) is \( 1*tw \) (ignoring the setup cost).
2. After the global pivot is selected and data is partitioned in the superstep2, the MPI_Allreduce is performed twice to let each process know how many of the data elements are smaller or equal to the global pivot and how many of them are actually greater than the global pivot. Therefore the cost factor \( h \) for this communication part would be \( 2* ( P^2-P) \) and \( g \) is the same as mentioned above.
3. The cost of performing MPI_Comm_create on a group of P processes is \( P^2-1 \).
4. The cost of distributing smaller and larger data sets to their respective groups is to be derived next. On the average the message size will be \( N/(2*P) \). That implies the cost \( g \) associated with each message will \( N/(2*P)*tw \) (ignoring the setup cost.). Moreover there will be \( 2*P \) messages in the system for a communicator of size \( P \).
So the total cost associated with the process of distributing the data sets is \( t_2 = 2*P \ (N/2*P*tw) \).

Now we’ll compile all of the costs for each superstep.

\[
T_{superstep1} = O(1) + (P^2-P)*1*tw + L
\]
Where \( L \) = synchronization cost. Now since the computation cost associated with the superstep is constant, there should not be a significant difference between the individual processes in reaching out of the first superstep.

\[
T_{superstep2} = O \left( \frac{N}{P} \right) + [2*(P^2-P)+(k^2-1)+N*P^2]*tw + L.
\]

\[
T_{superstep3} = T_{basecase} = T(\text{linear Quick sort}) = O(\frac{N}{P} \log \frac{N}{P})
\]
Combining the first two supersteps gives the cost of a typical Quicksort call while the Tsuperstep3 alone gives the base case cost.

10 Results

I have run the algorithm on Multimedia Lab Cluster of Microsoft Windows PC’s connected through 100Mb Ethernet. The cluster has the latest MPICH version installed on it. The plan was to make a comparison between this low profile cluster and the Borg, but while attempting to login to Borg, I found out that my account was temporarily unavailable. (A password change related security issue.) The comparison therefore will be done when the Borg account will be available again.

The following are the run times for the algorithm when running on two systems (Silicon and Awaz) on the above mentioned cluster. The input data size 5, intentionally kept small so that the output does not look chaotic.

11 OUTPUT

Input Array for awaz.
41
18467
6334
26500
19169
Input Array for Silicon.
38
7719
21238
2437
8855
Partition time =0.0604074
Time for all reduction =0.000739759
QuickSort:num_S_procs =1
QuickSort:num_G_procs =1
QuickSort:num_S_procs =1
QuickSort:num_G_procs =1
New Communicators creation time=0.078703
Sorted Array of Silicon.
P:1 Receiving from P0
P:1 Data distribution time =0.00398179
Sorted Array of awaz.
19169
21238
26500
38
41
2437
6334
7719
8855
18467
Total time taken 0.215059

One good thing about the MPIRUN facility that I’m using, it gives the rank based color output which significantly eases the debugging task. For example, in the above output all the AWAZ (Process Rank 1) activity is shown in dirty green color and the Silicon (Process Rank 0) activity is shown in Black color. Up till now, I’ve configured five of Multimedia Lab PCs as a part of the cluster. Presently this cluster is in its
infancy and goes down time and again. But bringing it up is not difficult task as most of the required tools are provided with the MPICH distribution.

Here verifying the output of the program, the input array to silicon was

Input Array for Silicon.
38
7719
21238
2437
8855
and that for Awaz was

Input Array for awaz.
41
18467
6334
26500
19169

These arrays are generated randomly. The output on P0(Silicon) is given below

Sorted Array of Silicon
38
41
2437
6334
7719
8855
18467

similarly for AWAZ

Sorted Array of awaz.
19169
21238
26500

which match the required format of the output.

12 Analysis:

<table>
<thead>
<tr>
<th>Number of Processes</th>
<th>Total Execution time</th>
<th>Max Partitioning time</th>
<th>Reduction Time</th>
<th>New Communicator Creation time</th>
<th>Data Distribution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0336043</td>
<td>N.A</td>
<td>N.A</td>
<td>N.A</td>
<td>N.A</td>
</tr>
<tr>
<td>2</td>
<td>0.0909266</td>
<td>0.029487</td>
<td>0.000590298</td>
<td>0.00662822</td>
<td>0.029249</td>
</tr>
<tr>
<td>3</td>
<td>0.167143</td>
<td>0.047042</td>
<td>0.00170189</td>
<td>0.0129706</td>
<td>0.0136886</td>
</tr>
<tr>
<td>4</td>
<td>0.373953</td>
<td>0.169717</td>
<td>0.00274923</td>
<td>0.0205367</td>
<td>0.0173989</td>
</tr>
</tbody>
</table>

N.A = Not Applicable
The Performance Graph For Parallel Quicksort Algorithm

Figure 5: The performance graph for parallel Quicksort algorithm. The increasing total execution time is due to the fact that with each increase in the processes size, the data to be sorted also increases. This graph is aimed to give the proportion of the time each part of the algorithm takes to get its work done rather than the actual performance of the algorithm with the same amount of data to be sorted.

Note: Since the code was compiled for Windows platform. In order to run it on Borg, please compile it first.

13 References