**//Parallel Quick sort Algorithm**

Code snippet shown below shows how QSORT is done

MPI\_Bcast(&s,1,MPI\_INT,0,MPI\_COMM\_WORLD);

/\* Broadcast size of array that each slave has to sort \*/

chunk = (int \*)malloc(s\*sizeof(int));

MPI\_Scatter(data,s,MPI\_INT,chunk,s,MPI\_INT,0,MPI\_COMM\_WORLD);

/\* Scatter the values from array to the slaves \*/

myqsort(chunk,0,s-1);

/\* --- call to quick sort function --- \*/

//Quicksort function:

myqsort( void \*a, int low, int high )

{

int pivot;

/\* Termination condition! \*/

if ( high > low )

{

pivot = partition( a, low, high );

quicksort( a, low, pivot-1 );

quicksort( a, pivot+1, high );

}

}

//Merge function:

while(i<n1 && j<n2)

if(v1[i]<v2[j])

{

result[k] = v1[i];

i++; k++;

}

else

{

result[k] = v2[j];

j++; k++;

}

if(i==n1)

while(j<n2)

{

result[k] = v2[j];

j++; k++;

}

else

while(i<n1)

{

result[k] = v1[i];

i++; k++;

}

Here the best case for merge would be when all values of one sequence are lower than all the vales of the other. The time complexity would be ?(n1 +1) or ?(n2+1) since after a pass through one array it will just copy the values of the other array and append to the first.

Performance:

Once partititon is done, different sections of the list can be sorted in parallel.

If we have p processors, we can divide a list of n elements into p sublists in O(n) average time, then sort each of these in O(n/p)log(n/p)) average time.

The disadvantages of this technique are as follows:

-Load balancing is not achieved since a particular process may finish its sorting process and wait for merge operation while other processes are still sorting their subsequences.

This leads to idle process time and is not efficient.

- Merge operations performed for each sorted subsequence is also computationally expensive