# Parallel Quick sort algorithm

## With PVM optimization

Final project report
of CS62025

Parallel & Distributed Processing

For: Dr. Bharvsar

Due : *Dec 23, 1996*

Prepared by:
Zhao Chengyan 255459

University of New Brunswick
Faculty of Computer Science
Fredericton, NB
Dec 21, 1996

I hereby claim that this project is finished by myself only and not any portion of which is copied from any other people.

**Index**

## 1. Sequential quick sort algorithm

### *1). Sequential quick sort algorithm description:*

1>. Select one item (the left most item) as the pivot;
2>. Have the first partition, which will generate two groups:
Every element in the left group is less than or equal to the value of pivot.
Every element in the right group is greater than or equal to the value of pivot;
3>. Call the procedure recursively to make the whole left group in order.
4>. Call the procedure recursively to make the whole right group in order.

Note: I hereby give out the recursive sequential quick sort algorithm only, for the following reasons:
1>. The recursive algorithm could show the data parallelism clearly
2>. There are certain mechanisms to convert the recursive algorithm into non-recursive algorithm, with the help on stack or queue. But, when finished converting, the original clearly algorithm would merge into the stack operations and lose tractility.).

### *2) Sequential quick sort algorithm analysis:*

***1>. Main control algorithm:***

```
Procedure qksort(r:listtype;s,t: integer);
Begin
 If ( s < t ) then
  {
  Qkpass(r,s,t,k);
  Qksort(r,s,k-1);
  Qksort(r,k+1,t);
  }
End;
```

***2>. Auxiliary algorithm: partition.***

```
   Procedure Qkpass(r:listtype;s,t:integer; var i:integer);
Begin
   I = s; j = t; x = r[i].key;
   While ( I < j) do
   {
       While ( (i<j) and (r[j].key >= x)) do
        j = j - 1;
       X[i] <--> x[j];
       While ( (i<j) and (r[i].key <= x)) do
        I = I + 1;
       X[i] <--> x[j];
   }
End;
```

3). Sequential algorithm performance analysis:
1>. Average performance:

It is clear that if the load balancing is good (say, in every partition, the pivot would appear in the middle or near the middle of the new generated sequence), every partition would finish in o(n) time, while there would be o(log(n)) partitions altogether. So, the average time complexity is in order of n *log(n) ---> o(n * log(n)).

2>. Performance in worst case:

Let consider the worse case:

Say that the source sequence is in order or almost in order before sorting (it is seldom, but it does have a chance to happen), then:

1)>. In every partition, there would be a group to be empty , while at the same time, the other group would contain o(n-1) items.

2)>. It would require n time of partition, while every partition would have to finish in o(n) time. So, in the worst case, the time complexity would be : o(n * n).

We know that it is the same as the bubble sort. (In worst case, the quick sort algorithm would withdraw to bubble sort algorithm).

3). Idea of parallel:

In the execution of the sequential quick sort algorithm, I found that:

1>. The data in the two groups of one partition have no dependency.
2>. The data generated in the recursively called partition also have no dependency.

Thus, they could be dispatched into different processes and execute simultaneously. This is the original idea where the parallel quick sort algorithm is from.

## 2. Traditional parallel quick sort algorithm

1). Algorithm of the parallel quick sort algorithm:

1>. Algorithm executed in maser process:

```
Procedure maser_algo;
Begin
  Generate_source_array();
  Send Source data to slaves();
  Get results back();
End;
```

The main job the master process is going to do is :
the three self-executed partitions( I used 8 slave processes, so it is required to do log(8) = 3 times of partition).

2>. Algorithm execution in slave process:

```
Procedure pqksort(l,r,q);
Begin
 While ( l < r) do
 {

   if( I am slave 0) {     /* this is the first partition */
   choose pivot(l,r,q,v); /* select the pivot point */
   Partition(l,r,q,v);     /* execute the partition operation */
   }
   if( I am slave 0 or 1) { /* this is the second partition */
   choose pivot(l,r,q,v); /* select the pivot point */
   Partition(l,r,q,v);     /* execute the partition operation */
   }
   if( I am slave 0 or 1 or 2 or 3) { /* this is the third partition */
   choose pivot(l,r,q,v); /* select the pivot point */
   Partition(l,r,q,v);     /* execute the partition operation */
   }
   pqsort(l,v-1,q);      /* recursively calling, make the elements in the left group orderly */
   pqsort(v+1,r,q);     /* recursively calling , make the elements in the right group orderly*.
 }
 End;
```

The algorithm executed in the slave processes is quite similar to the sequential one. In fact, it is just the sequential quick sort algorithm, only with the difference that the source data is from the master and generated after the n = log(8) = 3 times of partition, not generated by itself directly.

**2). Performance analysis:**

1>. Analysis in mathematics:

Let consider the average load balancing case:

In this case, the full load would be divided into eight parts and every process would afford one part only.

1)>. Computational time:

So, the computational time would be : o( N/p * log(N/p)), where p is the number of processes.

When the number of processes(p) is far below the job size, we know that the computational time is: o(N *log(N/p)).

2)>. Message transferring time:

From the diagram above, we know that there are a lot of messages to transfer:

They are:

A. The messages transferred from the master to slave.(send source data)

B. The messages transferred from the slave to master.(receive result data)

C. The messages transferred from the slave to other slaves with having the partition.

In fact, the message transferring time takes a great part of the total time.

3)>. Total time:

In general, we have the following time:

**Total time = o(N* log(N/p)) + Message time**.

2>. Analysis in Diagram:

1)>. Execution time in multi-host PVM system:

From the graph, we know that with the increase of the number of PVM host, the execution time would go to the min at the number of **three.** After that, the execution time would increase again. There are several possible reasons as following :

A. The load unbalancing:

Because the source data numbers of every execution time is generated purely randomly, there might be some job unbalancing in after the three partition in the beginning.

B. Different PVM performance of host.

The host computers have different performance in computation. While at the time of process spawning, the system would have no idea on what performance every host has. If it is unluckily to spawn a big loaded process to a poor performance host, the time would be longer.

C. Unexpected users:

While in PVM testing, there are some other users who use the PVM system. They would spawn the processes at any time they like, which cause the system performance unstable.

2)>. Speedup and efficiency

It is clear that the speedup and efficiency are generated from the execution time directly, as shown in the following graph.

3)>. Load balancing:

Well, when I analysis the load balancing results, I find that the traditional parallel quick sort algorithm has a very bad load balance. Please see the following graph for the load unbalancing on the execution time.

/* please note: all the results show on the graph are generated from the arithmetic average of 5 times testing, it is fair. */

3). un-satisfaction:

With the program developed for the traditional parallel quick sort algorithm, it does work, but has a very low speedup(large number of idle processes show in graph above) and a lot of time is spent in message transferring, no in computation.

Thus, I have a idea to optimize it in PVM programming environment.

The improvement would be made mainly on message transferring:

1>. Increase the message size;

2>. Decrease the message number;

To make a system has a better granularity.

## 3. PVM optimized algorithm

1). Algorithm description

1>. Algorithm executed in master:

```
Procedure pvm_pksort(l,r,q);
   Begin
     Send source data q to all slaves;
     Choose_pivot_array(l,r,q,parray);
     Send parray to correspond slaves;
     Receive result;
   End;
```

an assumption: all the source data are in pure random status.

```
Procedure CHOOSE_PIVOT_ARRAY(l,r,q,parray);
 begin
     For I= 0 to (p-1) do
   {
      parray[i] = SOURCE[(N/P)*I];
   }
 end;
```

It is clear that the master algorithm is quite different, comparing with those executed in traditional parallel quick sort algorithm.

The mainly difference are:

1>. The master would have responsibility to send **all** the source data to every slaves (broadcast)

2>. The master would have to scan the source data sequence at least once to decide the source data distribution, generated the partition boundaries and send the boundaries to every slaves.

Well, on the other hand, the similarity is :

1>. Both of them must send source data.

2>. Both of them must receive result data and store in source buffer.

2). Algorithm executed in slave program:

```
  Procedure pvm_qksort_slave;
 Begin
  B1 <-- Receive left boundary;
  B2 <--- receive right boundary;
 Data < -- Get first partition;
 qksort(l,r,Data); /* the qksort is a general
            sequential quick sort */
Send result back;
End;
```

**Get first partition:**

```
procedure get_first_partition;
begin
  Count = 0;
  For I = 0 to (N-1) do
  {
  If (SOURCE[i] >= B1) and(SOURCE[i] <=B2) then
    {
   Data[counter]= SOURCE[i]
   Counter = counter + 1;
   }
}
end;
```

well, the slave algorithm is also quite different comparing with the slave ones in traditional algorithm.

The main difference are:

1>. Every slave would have to receive the whole source data, coming from the master.

2>. Every slave would have to receive the whole boundaries array coming from the master and pick out the couple that it requires.
3>. After it gets the couple of boundaries, it would do a special partition procedure: pick out the elements that it requires and sorts them only.

2). Optimization mechanism for PVM:
We know that the PVM is a message transfer based must-host system. My target is to decrease the time used in message transferring, thus will have a better computational performance in PVM system.

1>. Decrease the number of message in system : between the master and the slave.

1)>. In traditional parallel quick sort algorithm:
The number of message is :
0>>. Master will distribute the whole original data to slave1;
1>>. Slave1 has a self partition and distribute to slave 1 and slave2: N/2 * 2;
2>>. Slave1 and slave2 have a self partition and distribute to slave..slave4:
N/4 * 4;
3>>. Slave1..slave4 have self partition and distribute the partition to slave1..slave8:
N/8 * 8;
4>>. All the slave processes will return the ordered results to the master :
N/8 * 8;
So, all the message required in the system is : 1 +2 + 4 + 8 + 8 = ***23;***

2)>. In PVM optimized parallel quick sort algorithm:
1>>. The master will broadcast the whole message to every of the slaves:
N * 1;
2>>. The mater will distribute the boundaries to each slave:
1 * p;
3>>. The master will receive the ordered results from each of the slaves:
N/8 * 8;
So, the whole message number is : N * 1 + p * 1 + N/8 * 8,
When the number of slaves(p) is far below the number of un-ordered data, the p could be ignored, thus, the final message would be : 1 + 8 = 9 ;

2>. Decrease the size of message:

1)>. In general parallel quick sort algorithm:
1>>. Message send from master to slave1: N * 1;
2>>. Message send from slave1 to slave1..slave2: N/2 * 2 = N;
3>>. Message send from slave1..slave2 to slave1..slave4: N/4* 4 = N;
4>>. Message send from slave1..slave4 to slave1.. Slave8: N/8 * 8 = N;
5>>. Message send back from every slave to the master is : N /8 * 8 = N;
So, the total message size passed is : N + N + N + N + N= 5 * N;

2)>. In PVM optimized parallel quick sort algorithm:

1>>. Message broadcast from master to all slaves: N * 1;
2>>. Message send back to the master is : N/8 * 8 = N;
So, the total message transferring is : 2 *N;
(From this view of point, the message transferring time is greatly reduced )

3>. Reason & assumptions to optimize the algorithm under PVM:

1. We have only the PVM parallel programming environment, the performance of the algorithm could be only evaluated under such an environment;
2. In PVM programming, if the program does not have a very good granularity, most of the time would be spent in message transferring, instead of computation
so, message transferring optimization is the most important part of our parallel algorithm implementation programming.

3). Pvm optimized algorithm performance analysis:
1>. Analysis in theoretical:

1)>. The computational time is the same as the traditional parallel quick sort algorithm.
In general, the whole algorithm could expect to finish in :o(N * log(N/p)) time
2)>. The message transferring time:
In my optimization, the message transferring time is greatly reduced.
3)>. Total time:
The total execution time = o(N * log(N/p)) + Message time
2>. Analysis in diagram:
1)>. Execution time comparison:

From the graph above, it is clear that the message reduce mechanism takes great effects on the whole system performance (the two jobs have the same size).

*Note: former is the traditional parallel algorithm, while the later one is the PVM optimized algorithm .*

2)>. Speedup & efficiency comparison:

From the picture shown, we know that the speedup and efficiency are almost the same. This limitation is caused by the quick sort algorithm itself and it would have nothing to do to improve it.

3)>. Load balancing on PVM optimized algorithm:

From the graph show below, we know that the PVM optimized algorithm has a very good load balance. This is the result of the original o(N) scan in the source master program. It is also the main reason why the PVM optimized algorithm would have a performance improvement of over 100 perfect.

4). Conclusion:

Comparing with the different execution time of the same parallel quick sort algorithm, we found that the PVM optimized one could execute almost as half time as the traditional one.

It means that my PVM optimization get a complete success, the message transferring time is greatly reduced.

## 4. Reference

1). Data structure , 1981, TsengHua University press
2). Parallel & Distributed computing, Akl, Prentice Hall
3). Journal of Parallel & Distributed Computing, 1993, A parallel quick algorithm

## 5. Appendix

1). A-1: traditional parallel quick sort master algorithm: source code with comments
Programming in C code, cc compiler under Unix..
2).A-2: traditional parallel quick sort slave algorithm: source code with comments
Programming in C code, cc compiler under Unix.
3) B-1: PVM optimized parallel quick sort master algorithm: source code with comments
Programming in C code, cc compiler under Unix.
4).B-2: PVM optimized parallel quick sort slave algorithm: source code with comments
Programming in C code, cc compiler under Unix.