An Adaptive Environment To Evaluate The Performance Of Parallel Merge Sort

Husain Ullah Khan, Rajesh Tiwari

Abstract

Merging is very well understood in the sequential model of computation and a simple algorithm exists for its solution. Parallel computing on loosely coupled architecture has been evolved now days because of the availability of fast, inexpensive processors and advancements in communication technologies. The One common example of parallel processing is the implementation of the merge sort within a parallel processing environment. such as Open-MP and MPI, and run it on multi-core computers and multi-core clusters. The aim of this paper is to estimate the performance and speedup of parallel merge sort algorithm compare it with theoretical analysis .The parallel computational time complexity is O(p) using p processes and one element in each process. It has been found that there is no major difference between theoretical performance analysis and the actual result.

1. Introduction

Merge sort is an efficient divide-and-conquer sorting algorithm. Because merge-sort is easier to understand than other useful divide-and-conquer methods. One common example of parallel processing is the implementation of the merge sort within a parallel processing environment. In the fully parallel model, you repeatedly split the sub lists down to the point where you have single-element lists. [1] Intuitively, merge sort operates on an array of n objects as follows:
(1) if n > 1, divide the array into two sub-arrays of about half the size each;
(2) apply merge sort on each sub-array;
(3) merge the two sorted sub-arrays from step 2 into one sorted array.

Manuscript received Oct 18, 2012.

Husain ullah khan:- *M.E.(Pursuing) in Computer Technology & Application from Shri Shankaracharya College of Engineering & Technology, CSVTU Bhilai, India. Mob: +91-9907417003,(e-mail: khan.husain@gamil.com).*
Rajesh Tiwari*:- Dept. Of Computer Science & Engineering, Sr. Associate Professor in Department of computer Science in Shri Shankaracharya Group of institutions (Faculty of engineering), Mob: +91-9893411757, (e-mail:* raj_tiwari_in*@yahoo.com).*

2. Parallel Merge Sort

The merge sort algorithm uses a divide and conquer strategy to sort its elements [6].The list is divided into 2 equally sized lists and the generated sub-lists are further divided. The numbers are then merged together as pairs to form sorted lists .The lists are then merged subsequently until the whole list is constructed. This algorithm can parallelized by distributing n/p elements memory of n elements for its merge operation (the same as quick sort). The practical performance of merge sort is known to improve with recursion removal and cache memory utilization [4]. We use merge sort as a test bed to explore parallelization schemes that may possibly apply without significant changes to other divide-and conquer methods. Merge sort parallelization is well-studied in theory. Figure 2.1 shows the processing tree for the case in which you have a list of 2000 items to be sorted and have resources only sufficient for four parallel processes. The processes receiving the size 500 lists use some sequential sorting algorithm. Because of the implementation environment, it will be something in the C/C++ language and your favorite implementation of a fast sorting algorithm. Each leaf node (with a size 500 list) then provides the sorted result to the parent process within the processing tree that processes combines the two lists to generate a size 1000 list, and then sends that result upstream to its parent process. Finally, the root process in the processing tree merges the two lists to obtain a size 2000 list, fully sorted. If your environment supports more parallel processes, you might take the processing tree to four levels, so that eight processes do the sequential sorting of size 250 lists.

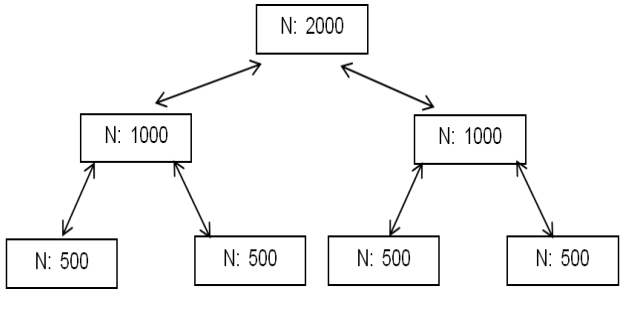

Figure:-2.1 Merge tree

3. Time Complexity of Parallel Merge Sort

Sequential merge sort time complexity is O (n log n). when parallelizing the merge sort algorithm the time

complexity reduces to O(n/p log n/p) as stated in [4].

4. Properties for Parallel Algorithm

Five important properties that we desire a parallel algorithm to possess. The first two properties concern the number of processors to be used by the algorithm. Let n be the size of the problem to be solved:
(i) p(n) must be smaller than n: No matter how inexpensive computers become, it is unrealistic when designing a parallel algorithm to assume that we have at our disposal more (or even as many) processors as there are items of data. This is particularly true when n is very large. It is therefore important that p(n) be expressible as a sub linear function of n, that is,
p(n) = n X ; 0 < x < 1.
(ii) p(n) must be adaptive: In computing in general, and in parallel computing in particular, "appetite comes with eating" . Algorithms using a number of processors that is a sub linear function of n [and hence satisfying property (i)], such as log n or n'/2, would not be acceptable either due to their inflexibility. What we need are algorithms that possess the "intelligence" to adapt to the actual number of processors available on the computer being used.

4.1 Running Time

The next two properties concern the worst-case running time of the parallel algorithm:
(i) t(n) must be small: Our primary motive for building parallel computers is to speed up the computation process. It is therefore important that the parallel algorithms we design be fast. To be useful, a parallel algorithm should be significantly faster than the best sequential algorithm for the problem at hand.
(ii) t(n) must be adaptive: Ideally, one hopes to have an algorithm whose running time decreases as more processes are used. In practice, it is usually the case that a limit is eventually reached beyond which no speedup possible regardless of the number of processors is used. Nevertheless, it is desirable that t(n) vary inversely with p(n) within the bounds set for p(n).

4.2 Cost

Ultimately, we wish to have parallel algorithms for which c(n) = p(n) x t(n) always matches a known lower bound on the number of sequential operations required in the worst case to solve the problem. In other words, a parallel algorithm should be cost optimal. In particular, when a set of processors are linked by an interconnection network, the geometry of the network often imposes limits on what can be accomplished by a parallel algorithm. It is a different story when the algorithm is to run on a shared-memory parallel computer. parallel algorithm for selecting the kth smallest element of a sequence S = {S, S2,..., s,}. The algorithm runs on an SM SIMD computer with N processors, where N < n. The algorithm enjoys all the desirable properties formulated in this section:

(i) It uses p(n) = n' -' processors, where 0 < x < 1. The value of x is obtained from N = n''. Thus p(n) is sub linear and adaptive.

(ii) It runs in t(n) = 0(nX) time, where x depends on the number of processors available on the parallel computer. The value of x is obtained in (i). Thus t(n) is smaller than the running time of the optimal sequential algorithm described in

5. An Algorithm For Parallel Selection

We are now ready to study an algorithm for parallel selection on an SM SIMD computer. The algorithm presented as procedure PARALLEL SELECT makes the following assumptions (some of these were stated earlier):

1. A sequence of integers S = {s1, s2, .. , s- and an integer k, I < k < n, are given, and it is required to determine the kth smallest element of S. This is the initial input to PARALLEL SELECT.
2. The parallel computer consists of N processors PI, ..., PN.
3. Each processor has received n and computed x from N = n', where O<x< 1.
4. Each of the n' processors is capable of storing a sequence of nx elements in its local memory.
5. Each processor can execute procedures SELECT, SEQUENTIAL, BROADCAST, and ALL SUMS.
6. M is an array in shared memory of length N whose ith position is M(i).

As usual, we denote by t(n) the time required by PARALLEL SELECT for an input of size n.
A function describing t(n) is now obtained by analyzing each step of the procedure.
Step 1: To perform this step, each processor needs the

beginning address A of sequence S in the shared memory, its size is ,and the value of k. These quantities can be broadcast to all processors using procedure BROADCAST: This requires O(log n' -x) time. If ISI < 4, then P1 returns the kth element in constant time. Otherwise, Pi computes the address of the first and last elements in Si from A + (i -1)nx and A + inX - 1, respectively; this can be done in constant time. Thus,

step 1 takes cllog n time units for some constant cl.
Step 2: SEQUENTIAL SELECT finds the median of a sequence of length nX in c2n' time units for some constant c2.
Step 3: Since PARALLEL SELECT is called with a sequence of length n'-x, this step requires t(n'-x) time.
Step 4: The sequence S can be subdivided into L, E, and G as follows:

(i) First m is broadcast to all the processors in O(logn1-x) time using procedure BROADCAST.

(ii) Each processor Pi now splits Si into three subsequences Li, E,, and Gi of elements smaller than, equal to, and larger than m, respectively. This can be done in time linear in the size of Si, that is, 0(nX) time.

(iii) The subsequences Li, E,, and Gi are now merged to form L, E, and G. We show how this can be done for the Li; An Algorithm for Parallel Selection running time can be derived for merging the Ei and G;, respectively. All these sums can be obtained by n`-x processors in O(log nl X) time using procedure ALLSUMS.

Step 5: The size of L needed in this step has already been obtained in step 4 through the computation . The same remark applies to the sizes of E and G. The preceding analysis yields the following recurrence for t(n): t(n) = c1log n + C2nX + t(n'X) + C3nX + t(3n/4), whose solution is t(n) = 0(nX) for n > 4.
Since p(n) = n', we have c(n) = p(n) x t(n) = n' -x x 0(nx) = O(n). Since N = n'-x and n/nX < n/log n, it follows that PARALLEL SELECT is cost optimal provided N < n/log n.

6. A Network for Merging

Special-purpose parallel architectures can be obtained in any one of the following ways:
(i) using specialized processors together with a conventional interconnection network,
(ii) using a custom-designed interconnection network to link standard processors,
Or
(iii) using a combination of (i) and (ii).

In this paper we shall take the third of these approaches. Merging will be accomplished by a collection of very simple processors communicating through a special-purpose network. This special-purpose parallel architecture is known as an (r, s)-merging network. All the processors to be used are identical and are called comparators. a comparator receives two inputs and produces two outputs. The only operation a comparator is capable of performing is to compare the values of its two inputs and then place the smaller and larger of the two on its top and bottom output lines, respectively

6.1 Analysis

Our analysis of odd-even merging will concentrate on the time, number of processors, and total number of operations required to merge. (i) Running Time. We begin by assuming that a comparator can read its input, perform a comparison, and produce its output all in one time unit. Now, let t(2n) denote the time required by an (n, n)-merging network to merge two sequences of length n each. The recursive nature of such a network yields the following recurrence for
t(2n): t(2) = 1 for n = I , t(2n) = t(n) + 1 for n > 1
whose solution is easily seen to be t(2n) = 1 + log n. This is significantly faster than the best, namely, 0(n), running time achievable on a sequential computer. (ii) Number of Processors. Here we are interested in counting the number of comparators required to odd-even merge.
merging network. Again, we have a recurrence:

p(2) = 1
for n = 1
p(2n) = 2p(n) + (n - 1) for n > I

whose solution p(2n) = I + n log n is also straightforward.
(iii) Cost.
Since t(2n) = 1 + log n and p(2n) = 1 + n log n, the total number
of comparisons performed by an (n, n)-merging network, that is, the network's cost, is c(2n) = p(2n) x t(2n)= O(n log2n).Our network is therefore not cost optimal as it performs more operations than the 0(n) sufficient to merge sequentially.

7. Choosing the Parallel Environment: MPI

There is an easily used parallel processing environment for you whether your target system is a single multiprocessor computer with shared memory or

a number of networked computers: the Message Passing Interface (MPI) [5] As its name implies, processing is performed through the exchange of messages among the processes that are cooperating in the computation. Central to computing within MPI is the concept of a "communicator". The MPI communicator specifies a group of processes inside which communication occurs. MPI_COMM_WORLD is the initial communicator, containing all processes involved in the computation. Each process communicates with the others through that communicator, and has the ability to find position within the communicator and also the total number of processes in the communicator. Through the communicator, processes have the ability to exchange messages with each other. The sender of the message specifies the process to receive the message. In addition, the sender attaches to the message something called a message tag, an indication of the kind of message it is. Since these tags are simply non-negative integers, a large number is available to the parallel programmer, since that is the person who decides what the tags are within the parallel problem solving system being developed. The process receiving a message specifies both from what process it is willing to receive a message and what the message tag is. In addition, however, the receiving process has the capability of using wild cards, one specifying that it will accept a message from any sender, the other specifying that it will accept a message with any message tag. When the receiving process uses wild card specifications, MPI provides a means by which the receiving process can determine the sending process and the tag used in sending the message. For the parallel sorting program, you can get by with just one kind of receive, the one that blocks execution until a message of the specified sender and tag is available. You need to initialize within the MPI environment. The presumption is that this one is called from the program's main, and so it sends pointers to the argc and argv that it received from the operating system. We choose MPI to implement message-passing merge sort on single and networked computers because

(i) MPI is implemented for a broad variety of architectures, including implementations that are freely available; (ii) MPI is well documented; (iii) MPI has grown much more popular than alternative platforms, such as PVM [3]. and, our preference for an implementation language is ANSI C because (i) C is fast and available on virtually any platform; (ii) C can be used to implement merge sort versions with various platform.

8. Merge Sort with MPI

```
void mergesort_serial(int a[], int size, int temp[]) {
if (size < SMALL) {
mergesort_serial(a, size/2, temp);
mergesort_serial(a + size/2, size - size/2, temp);
merge(a, size, temp);}
```

The MPI API [5] supports, on a variety of platforms, programming of message-based communication between processes and is typically used in distributed-memory systems, such as computer clusters. With MPI, programmers in a wide variety of languages use a set of library routines to implement communication and synchronization between processes. all MPI processes start at once at the very beginning of program execution, and all processes concurrently execute the same code the entire program. Consequently, the MPI program must permit each process to recognize its own place and role in the recursion tree. With MPI, processes need to be explicitly programmed to map themselves to nodes in the recursion tree from the recursion tree to threads. As MPI processes map themselves to nodes from the recursion tree, they form a virtual process tree. Process 0 is at the root of the tree, with the remaining processes appearing as nodes of the tree. The root process splits the data and sends half of it to a helper process which sorts the data and returns it to the root process (send operations are visualized as arrows in Figure:- 8.1. The other half of data is retained by the root process for further sorting by using this same procedure once sorted, the two halves of data are merged by the root process.

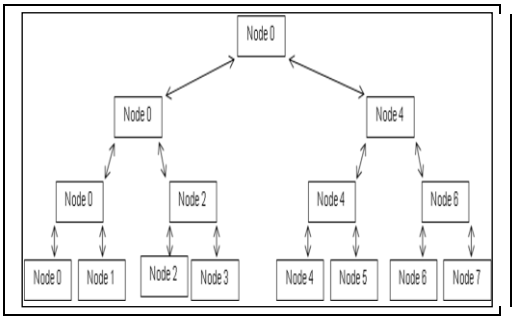

Figure:- 8.1 Root and helper processes merge sort.

MPI process tree for recursive merge-sort. Arrows visualize communications with helper processes; Note that the root process can further split its retained data and send half of it to yet another helper process. Helper

processes themselves can follow the same procedure as the root process. Splitting and sending data continues until each MPI process becomes a node in the virtual process tree, i.e. until all processes are sent some amount of data to sort. All MPI processes run the same main function which differentiates between the root process and helper processes. The root process prepares the array to sort and then invokes parallel merge sort while each helper process: (i) receives data from its parent process; (ii) invokes parallel merge sort; and (iii) sends sorted data back to parent (Figure:-8.1). Note that each helper process calculates the level of its top-most appearance in the process tree and passes it to the parallel merge sort function .

```
int main(…) {
// ask MPI for my_rank;
if (my_rank == 0) {
// allocate array to sort then run root to sort it:
run_root_mpi(a, size, temp, …);
} else {
run_helper_mpi(my_rank, …);
}
// array is sorted;
}
void run_root_mpi (int a[], int size, int temp[], …) {
int level = 0;
mergesort_parallel_mpi(a, size, temp, level,…);
}
void run_helper_mpi(int my_rank, …) {
// probe MPI for a message from parent process
// and identify message size and parent_rank;
// allocate int a[size], temp[size];
MPI_Recv(a, size, …, parent_rank, …);
int level=my_topmost_level(my_rank);
mergesort_parallel_mpi(a, size, temp, level, …);
// send sorted array to parent process:
MPI_Send(a, size,… , parent_rank, …);
}
int my_topmost_level_mpi(int my_rank) {
int level = 0;
while (pow(2, level) <= my_rank) level++;
return level;
}
```

Parallel merge sort is executed by various processes at various levels of the process tree, with the root being at level 0, its children at level 1, and so on (Fig.7.1). In that, the process's level and the MPI process rank are used to calculate a corresponding helper process's rank Then, merge sort communicates for further sorting half of the array with that helper process. Serial merge sort is invoked when no more MPI helper processes are available.

```
void mergesort_parallel_mpi
(int a[], int size, int temp[], int level, …) {
// my_rank is used to calculate helper rank:
int helper_rank = my_rank + pow(2, level);
if (helper_rank > max_rank) {
mergesort_serial(a, size, temp);
} else {
// send second half of array, asynchronous:
MPI_Isend(a+size/2, size-size/2, …, helper_rank,
…);
// sort first half:
mergesort_parallel_mpi(a, size/2, temp, level+1, …);
// receive second half sorted:
MPI_Recv(a+size/2, size-size/2, …, helper_rank,
…);
// merge the two sorted sub-arrays:
merge(a, size, temp);
}
}
```

The performance of the above message-passing (with MPI) implementation is evaluated in Section 5.

```
mergesort_parallel_omp(a, size, temp, threads);
} else {
MPI_Isend(a+size/2, size-size/2, …, helper_rank,
…);
mergesort_parallel_mpi_and_omp
(a, size/2, temp, level+1, threads, …);
MPI_Recv(a+size/2, size-size/2, …, helper_rank,
…);
merge(a, size, temp);
}
}
```

9. Mapping the Communications

You might initially think of letting each node in the processing tree be a separate process. That way you can simply borrow an idea from the binary heap when it is implemented in an array with the root at zero. For any in-use cell within the array with subscript k, the left child of that heap entry is at subscript 2*k+1, the right child is at subscript 2*k+2, and the parent is at (k–1)/2. This would also give the parent/child relationships within the complete binary tree that constitutes the processing tree. Thus an internal node would split the data in half and send the two halves to the child

processes for processing. Should an internal node have only one child process, it would have to sort its own right-hand side. Leaf nodes, of course, just do the sorting. The internal nodes then receive back the data, perform the merge of the two halves, and (for all but the root node itself) send the result to the parent.

The communication of sub problems is an overhead expense that you want to minimize. Also, there's no reason to allow an internal node process to sit idle, waiting to receive two results from its children. Instead, you want the parent to send half the work to the child process and then accomplish half of the work itself. It effectively becomes a node in the next level down in the sorting tree. Figure: 7.1 shows a full sorting tree in which all of the processes (represented by their ranks) compute at the sorting tree leaf level.

10. Performance Evaluation

We measured the performance of our shared memory, message-passing, and parallel merge sorts on 2, 4, 8 & 16 processes under Linux. Processors running under a 1.80 MHz clock. We executed our merge sorts with randomly generated arrays of 10^7 integer elements. No other applications were active on the cluster during our performance measurements. Shared memory merge sort was executed on 1, 2, 4, and 8 cores on the master node using all available cores for MPI processes. Our merge sorts process a single array that can be entirely held in RAM on a single node. This setup is advantageous for single node implementations centralized setup involves multiple MPI data transmissions that begin and end with the root node. in cluster nodes, and the cluster network; for example, a fast network can make a message-passing (with MPI) solution for some problem faster.

11. References

[1] Seyed H. Roosta, Parallel Processing and Parallel Algorithms (Springer-Verlag New York: 2000), pp. 397-98.
[2] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford. Introduction to Algorithms (3rd ed.), MIT Press, 2009.
[3] Geist, Al; Beguelin, Adam; Dongarra, Jack; Jiang, Weicheng; Manchek, Robert; Sunderam , Vaidy. PVM: Parallel Virtual Machine. MIT Press, 1994.
[4] LaMarca, Anthony; Ladner, Richard. The influence of caches on the performance of sorting. Proc. 8th Ann. ACM-SIAM Symposium on Discrete Algorithms (SODA97), 370–379.
[5] The Message Passing Interface (MPI) standard. Retrieved on March 1, 2011 from http://www.mcs.anl.gov/research/projects/mpi/.
[6]The OpenMP specification for parallel programming Retrieved on March 1, 2011 from http://openmp.org

Husain ullah khan

M.E.(Pursuing) in Computer Technology & Application from Shri Shankaracharya College of Engineering & Technology, CSVTU Bhilai, India. Research areas are Parallel Computing & its Enhancement.

Rajesh Tiwari

M.E. in Computer Technology & Application form SSCET, CSVTU Bhilai, India. Currently pursuing Ph.D. from CSVTU, Bhilai. He is working as Sr. Associate Professor in Department of computer Science in Shri Shankaracharya Group of institutions (Faculty of engineering).He is having long experience in the field of teaching & research. His research areas are Parallel Computing and its Enhancement, His research work has been published in many national and international journals.