A Study of Internal Parallel Sorting Algorithms

Mohameed Kareemulla Dr. M. Punithavalli, Dr. Subramanian
Research Scholar, Manonmaniam Sundaranar University, Tamil Nadu, India.
Dean, Sri Ramakrishna College of Arts & Science for Women, Tamil Nadu, India.
Dean/Director, Indian Institute of Research & Management, Tamil Nadu, India.

Abstract - An ideal parallel sorting algorithm which uses P processors would reduce this time by at most a factor of P, simply because any deterministic parallel algorithm can be simulated by a single processor with a time cost of P. This leads us to the observation that an ideal comparison-based parallel sorting algorithm would take time NPlogN. Of course, this totally ignores the constant computational factors, communication costs and the memory requirements of the algorithm. All these factors are very important in the design of a parallel sorting algorithm.

Keywords: Parallel Processor, Parallel Shorting algorithm, Communication Cost, Memory requirement

II. INTRODUCTION

Parallel Sorting algorithms are focused to choose p-1 partitioning elements so that the final p sorted files are of roughly equal size, i.e., the load balance is good. We used two sampling techniques to select a sample of splitters: regular sampling and random sampling. Regular sampling selects splitters with equal intervals, while random sampling selects a certain number of pivots at random.

1. Main Algorithm

Main approach is derives from the work by Shi et al [14], which is an internal memory parallel sorting by regular sampling. researchers also applied the random sampling technique introduced by Quinn. In this thesis we consider the problem of external memory parallel sorting in a distributed memory system. In this multiprocessor architecture, each processor has its own memory and an independent disk (precisely, a big file), and all communications among processors must happen through an interconnection network.

1. Have a data file with N integer numbers created with a data generator. Each processor holds a disk file with N/p unsorted records. At the termination of the sorting algorithm, files have been partitioned, redistributed and merged into approximately equal sized non-overlapping sorted files, which must again be on disks, one at each processor. In more details, our algorithm can be described as follows:
2. /* Input: original file list F - f1, f2... fp (total size is N, p is the number of processors). Processor Pi holds N/p unsorted data items stored in file fi ($1 \sum i \sum p$). Output: sorted file list F'- f'1, f'2..., f'p, where all records in f'i are less than or equal to those in f'i+1($1 \leq I \leq p$ -1).*/
3. Each processor *Pi* samples its disk-resident file *fi*: it reads all data elements block by block ($B = 128K$) and selects $p_i 1$ pivots at equal intervals (or at random) from each block to form the set of splitters *Si*. So the size of *Si* is $N (p_i 1)/Bp$.
4. The coordinate processor $P1$ gathers all unsorted samples *Si* from all other processors. $P1$ then sorts the set of these samples to form a regular sample *S0* with size $N (p_i 1)/B$. Then the final sample *S* with *p*- 1 element at equal intervals are selected from the regular sample *S0*, and is broadcasted to all other processors.
5. Each processor *Pi* reads and sorts data items block by block from local file *fi*, and redistributes the records to the appropriate processors using the final sample *S*. When a processor's memory has been filled with incoming records, the processor sorts these records, writes the sorted run onto disk as a temporary file, and continues reading incoming records.
6. In parallel, the processors merge the sorted runs (precisely, temporary files) and back onto the disk as the final sorted file *f'i.* Note that during the second step, if we select pivots by regular sampling, then each data block has to be sorted first. This additional sorting time can be saved by an alternative way: write these sorted blocks into local temporary files, and in the third step, each processor reads these files (sorted blocks), not unsorted data block of the original data file. This means each data block of the original file has to be sorted only once anyway.

III. PARALLEL SORTING ALGORITHMS

Parallel programming is dealing with nondeterminism. For many computational problems, there is no inherent nondeterminism in the problem statement, and indeed a serial program would be deterministic—the nondeterminism arises

solely due to the parallel program and/or due to the parallel machine and its runtime environment. There is disagreement as to what degree of determinism is desired (worth paying for). Popular options include:

- **Data-race free:**

Which eliminate a particularly problematic type of nondeterminism: the data race. Synchronization Constructs such as locks or atomic transactions protect ordinary accesses to shared data, but nondeterminism among such constructs (e.g., the order of lock acquires) can lead to considerable nondeterminism in the execution.

- **Determinate(or External determinism):**

This requires that the program always produces the same output when run on the same input. Program executions for a given input may vary widely, as long as the program "converges" to the same output each time.

- **Internal determinism:**

In which key aspects of intermediate steps of the program are also deterministic, as discussed in this thesis.

Nested parallelism:

Nested-parallel computations achieve parallelism through the nested instantiation of fork-join constructs, such as parallel loops, parallel map, par begin/par end, parallel regions, and spawn/sync. More formally, nested parallel computations can be defined inductively in terms of the composition of sequential and parallel components. At the base case a strand is a sequential computation. A task is then a sequential composition of strands and parallel blocks, where a parallel block is a parallel composition of tasks starting with a fork and ending with a join.

Parallel programming is dealing with nondeterminism. For many computational problems, there is no inherent nondeterminism in the problem statement, and indeed a serial program would be deterministic—the nondeterminism arises solely due to the parallel program and/or due to the parallel machine and its runtime environment. There is disagreement as to what degree of determinism is desired (worth paying for). Popular options include:

- **Data-race free:**

Which eliminate a particularly problematic type of nondeterminism: the data race. Synchronization Constructs such as locks or atomic transactions protect ordinary accesses to shared data, but nondeterminism among such constructs (e.g., the order of lock acquires) can lead to considerable nondeterminism in the execution.

- **Determinate(or External determinism):**

This requires that the program always produces the same output when run on the same input. Program executions for a given input may vary widely, as long as the program "converges" to the same output each time.

- **Internal determinism:**

In which key aspects of intermediate steps of the program are also deterministic, as discussed in this thesis.

Nested parallelism:

Nested-parallel computations achieve parallelism through the nested instantiation of fork-join constructs, such as parallel loops, parallel map, par begin/par end, parallel regions, and spawn/sync. More formally, nested parallel computations can be defined inductively in terms of the composition of sequential and parallel components. At the base case a strand is a sequential computation. A task is then a sequential composition of strands and parallel blocks, where a parallel block is a parallel composition of tasks starting with a fork and ending with a join.

```
1   x:=0
2   in parallel do
3   {  r3:=Atomic Add(x,1) }
4   {  r4:= Atomic Add(x,10)
5   In parallel do
6   {  r6:= Atomic Add(x, 100)
    }
7   {  r7:= Atomic Add(x, 1000)
    }
```

Fig.1 A Sample Nested Parallel Program

Here, the in parallel keyword means that the following two f: : :g blocks of code may execute in parallel. Atomic Add(x; v) atomically updates x to x :=x + v and returns the new value of x.

The diamonds, squares, and circles denote forks, joins, and data operations, respectively. Nodes are numbered by line number, as a short hand for operations such as Atomic Add(x;

1). The left trace corresponds to the interleaving/schedule 1; 2; 3; 4; 5; 6; 7; 8, whereas the right trace corresponds to 1; 2; 4; 5; 7; 6; 3; 8. Because the intermediate return values differ, the program is not internally deterministic. It is, however, externally deterministic as the output is always the same. If Atomic Add did not return a value, however, then the program would be internally deterministic.

A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) which require input data to be in sorted lists; it is also often useful for canonicalizing data and for producing human-readable output. More formally, the output must satisfy two conditions:

1. The output is an increasing order (each element is no smaller than the previous element according to the desired total order)
2. The output is a permutation (reordering) of the input.

Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. For example, bubble sort was analyzed as early as 1956. Although many consider it a solved problem, useful new sorting algorithms are still being invented (for example, library sort was first published in 2006).

Sorting algorithms are prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts, such as big O notation, divide and conquer algorithms, data structures, randomized algorithms, best, worst and average case analysis, time-space tradeoffs, and upper and lower bounds.

IV. PERFORMANCE OF PARALLEL ALGORITHMS

Depending of the execution time of the parallel programs (measured with get time of day) only on the number of nodes along horizontal and vertical sides of a mesh and on the number of processors; the other parameters are fixed.

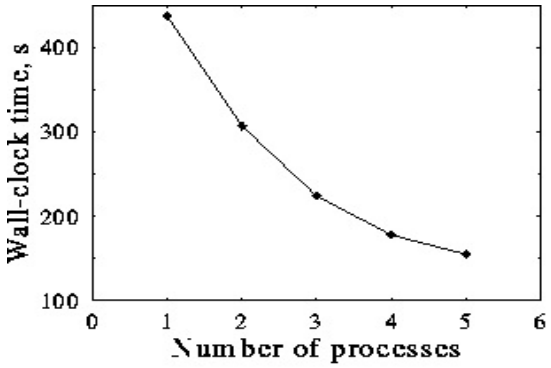

Fig.2 Execution time of sequential and parallel program on networked RS63; meshes with 2304 nodes; algorithm I.

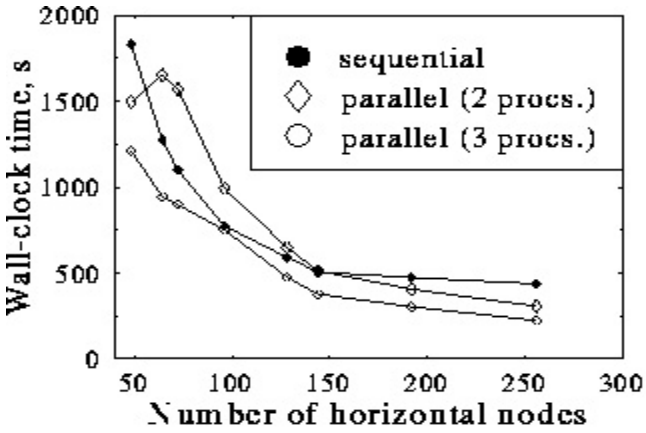

Fig.3 Execution time of parallel program on networked RS63 vs. number of processes; mesh: 256 x 9 nodes; algorithm I.

Estimating the Speedup

Speedup is defined by the following formula:

$$S_p = \frac{T_1}{T_p} \qquad (22)$$

Where:

- p is the number of processors
- T_1 is the execution time of the sequential algorithm
- T_p is the execution time of the parallel algorithm with p processors

Linear speedup or **ideal speedup** is obtained when $S_p = p$. When running an algorithm with linear speedup, doubling the number of processors doubles the speed. As this is ideal, it is considered very good scalability.

Efficiency is a performance metric defined as

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p.} \quad (23)$$

It is a value, typically between zero and one, estimating how well-utilized the processors are in solving the problem, compared to how much effort is wasted in communication and synchronization. Algorithms with linear speedup and algorithms running on a single processor have an efficiency of 1, while many difficult-to-parallelize algorithms have efficiency such as $\frac{1}{\ln p}$ that approaches zero as the number of processors increases.

In engineering contexts, efficiency is more often used for graphs than speedup, since

- all of the area in the graph is useful (whereas in a speedup curve 1/2 of the space is wasted)
- it is easy to see how well parallelization is working
- there is no need to plot a "perfect speedup" line

In marketing contexts, speedup curves are more often used largely because they go up and to the right and thus appear better to the less-informed. An important characteristic of any parallel algorithm is how much faster the algorithm performs than an algorithm on a serial machine.

The first choice gives that which is called the parallel efficiency of the algorithm. This is a measure of the degree to which the algorithm can take advantage of the parallel resources available to it. The second choice gives the fairest picture of the effectiveness of the algorithm itself. It measures the advantage to be gained by using a parallel approach to the problem. Ideally a parallel algorithm running on P nodes should complete a task P times faster than the best serial algorithm running on a single node of the same machine. It is even conceivable, and sometimes realizable, that caching effects could give a speedup of more than P.

V. CONCLUSION

It is described that the four-step approach to parallel algorithm design in which we start with a problem specification and proceed as follows:

- First partition a problem into many small pieces, or tasks. This partitioning can be achieved by using either domain or functional decomposition techniques.
- Next, to organize the communication required to obtain data required for task execution. It can distinguish between local and global, static and dynamic, structured and unstructured, and synchronous and asynchronous communication structures.
- Then, using agglomeration to decrease communication and development costs, while maintaining flexibility if possible.
- Finally, utilizing the map tasks to processors, typically with the goal of minimizing total execution time. Load balancing or task scheduling techniques can be used to improve mapping quality.

REFERENCES

[1] AGGARWAL, A. AND PLAXTON, C. 1993. Optimal parallel sorting in multi-level storage. Technical Report CS-TR-93-22, University of Texas at Austin. AJTAI, M., KOLMOS, J., AND SZERMEREDI, E. 1983. 3, 1 – 19.

[2] AKL, S. G. 1985. Parallel Sorting Algorithms. Academic Press, Toronto. (p. 47) BATCHER, K. E. 1968. Sorting networks and their applications. In Proc. AFIPS Spring Joint Computer Conference, Volume 32 (1968), pp. 307 – 314) BELL, T., CLEARY, J., AND WITTEN, I. 1990. Text Compression. Prentice Hall.

[3] BLELLOCH, G. E., LEISERSON, C. E., MAGGS, B. M., PLAXTON, C. G., SMITH, S. J., AND ZAGHA, M. 1991. A comparison of sorting algorithms for the connection machine CM-2. In Proc. Symposium on Parallel Algorithms and Architectures (Hilton Head, SC, July 1991).

[4] BURROWS, M. AND WHEELER, D. 1994. A block-sorting lossless data compression algorithm. Technical Report SRC Research Report 124 (May), Digital Systems Research Center.

[5] CALLAGHAN, B. 1998. Network file system version 4. http://www.ietf.org/html.charters/nfsv4-charter.html.

[6] ELLIS, J. AND MARKOV, M. 1998. A fast, in-place, stable merge algorithm. ftp://csr.uvic.ca/pub/jellis/.

[7] FENWICK, P. 1996. Block sorting text compression. In Proc. 19th Australasian Computer Science Conference, Melbourne, Australia (January 1996).

[8] FOX, G. C., JOHNSON, M. A., LYZENGA, G. A., OTTO, S. W., SALMON, J. K., AND WALKER, D. W. 1988. Solving Problems on Concurrent Processors, Volume 1. Prentice-Hall.

[9] GAILLY, J. AND ADLER, M. 1998. zlib. http://www.cdrom.com/pub/infozip/zlib/.

[10] GNU. 1998. The GNU thesis. http://www.gnu.org.

[11] GUTENBERG. 1998. Thesis gutenberg. http://www.gutenberg.net.

[12] HELMAN, D., BADER, D., AND J `AJ `A, J. 1996. A randomized parallel sorting algorithm with an experimental study. Technical Report CS-TR-3669, Institute for Advanced Computer Studies, University of Maryland.

[13] HOFF, A. AND PAYNE, J. 1997. Generic diff format specification. http://www.w3.org/TR/NOTE-gdiff-19970825.html.

[14] HUANG, B. AND LANGSTON, M. A. 1988. Practical in-place merging. Communications of the ACM 31, 348 – 352.

[15] ISHIHATA, H., HORIE, T., INANO, S., SHIMIZU, T., KATO, S., AND IKESAKA, M. 1991. Third generation message passing computer AP1000. In International Symposium on Supercomputing (November 1991), pp. 45 – 55.

[16] ISHIHATA, H., HORIE, T., AND SHIMIZU, T. 1993. Architecture for the AP1000 highly parallel computer. Fujitsu Sci. Tech. J. 29, 6 – 14.

[17] KARP, R. AND RABIN, M. 1987. Efficient randomized pattern-matching algorithms. IBM J. Research and Development 31, 249 – 260.