Journal of Computer Sciences

Original Research Paper

DPSA: Deterministic Parallel Search Algorithm in Large Database

Sami Qawasmeh and Arwa Zabian

Jadara University, Irbid-Jordan

Article history
Received: 10-10-2016
Revised: 31-01-2017
Accepted: 19-07-2017

Corresponding Author:
Sami Qawasmeh
Jadara University, Irbid-Jordan
Email: sqawasmeh@jadara.edu.jo

Abstract: The main goal of parallel processing is to reduce the complexity of finding a solution for a problem. In this study, we consider the problem of searching for multiple items at the same time in a large database. We propose a parallel search algorithm that reduces the searching time in comparison to binary search algorithm saving in that the time needed for sorting. Our algorithm works well for any data that can be represented in binary and it converts the searching of two items to search for a composed key that is the AND-combination of the two searched items. DPSA our proposed algorithm outperforms Binary search algorithm in searching for two items at the same time where the binary search will search them sequentially. The running time of our algorithm in the worst case is $O(n)$ for searching two items in a data input of size n.

Keywords: Distributed Database, Parallel Searching, Parallel Processing, Multithreading Programming

Introduction

The searching problem is a simple problem that can be expressed as follows given an array of integers A of size n and an element x, we must define with good precision that x is an element of the array A or not. Generally, using sequential search it requires linear time with n, its worst case cost is proportional to the number of elements in the list.

The binary search algorithm performs well in solving the searching problem. But, it works only under a sorted data. For that, the time saved in searching is spent in sorting. The searching problem is faced in different fields for that we must find a solution with the best performance. Searching database means issue a query to locate a record that has a specific field (or key) equal to a specific value. Another type of database query may require finding the smallest (or largest) key value and in some cases, more than one query can be directed to the database simultaneously, or sometimes searching an item can be done by composite fields. In all these cases, we need a good and fast response time taking in consideration that the size of the database is increased continuously and the number of queries also increased with the time. For that, we need always an effective and fast search mechanism that satisfies the user requirements.

Information Retrieval Model (IRM) defines the interaction between a user and information retrieval system and consists of three parts are: Document representation, user need and a matching function. Both document representation and matching functions must be defined in a manner to satisfy the user need. This means, finding the requested object in small time. For that, request-response time is an important factor in information retrieval system and searching time is one issue that influences in the request-response time. The motivation of our work is to minimize the cost of searching multiple items when searching in the database or in an information retrieval system (data stored on the server, on the proxy, on the cloud). Our work consists of reducing the searching time without ordering the data. Our algorithm works well for any data represented by a key that is an integer for any length. The main important result of our proposed algorithm is that the searching time is increased in a moderate ratio when the size of input data is increasing rapidly. Parallel processing is used to reduce the searching time. Parallel processing means to enable the microprocessor to perform the same operation (logical, arithmetic, or other) in parallel on several independent data sources. It is possible to divide the

data into smaller units and perform operations on that unit in parallel. Parallel execution reduces response time for the data-intensive operation on the large database. The most common use of parallel execution is in DSS and data warehouse environments, complex queries such as those involving joins of the several tables or search of very large tables are often best executed in parallel.

The idea of our work is to use the parallel search to improve the results of a multi-query for a large database. Consider we have a large database and more than one user sends concurrently a query to this database. The performance of the database is seen in the request-response time. The sequential search will execute each query in a time proportional to the database size. The binary search algorithm will sort the data, then process each query sequentially. Our proposed algorithm will divide the database into two parts and will search about two queries in parallel on the two parts. The worst case of searching one or more than one item in our proposed system will be equal to the half of the size of the database. In addition, when the size of the data is increased the searching time will be always proportional to the half of the data size.

This paper is organized as follows: Section 2, will present some of the related work in parallel database and information retrieval. Our proposed algorithm will be presented in section 3. Simulation results are presented in section 4. In section.5 is analyzed our results and the final conclusion and future works.

Related Work

The basic idea behind the parallel algorithm is to reduce the problem complexity and the time needed to solve it. So, if we have a problem, we can solve it quickly and in an easy manner if we divide it into sub-problems and solving them at the same time in parallel (if possible). Most of the NP-complete problems are solved in this manner, assigning a processor to each sub-problem. The running time of the algorithm is then the longest running time of any of these processors. A parallel algorithm is optimal in the running time if its upper bound is the best known sequential algorithm for the problem. For that, parallelism is used in different computing area to reduce the complexity and running time but it suffers from overhead.

Parallel search is used to solve optimization problems such that scheduling, robotic, game playing. In such problem the most important thing is when x the input size is large, in this case finding a near optimal solution is not an easy task. Grama and Kumar (1995) have introduced a list of parallel search algorithms for solving an optimization problem. The Parallel Depth First Search algorithm (PDFS) according to (Kumar and Rao, 1987; 1990) is based on the idea to partition the tree into smaller parts, these parts require no or minimal communication. The idea of tree partitioning can be applied using stack splitting or node splitting. In parallel DFS using stack splitting each processor searches a disjoint part of the tree in a depth-first fashion. When a goal is found, all of them quit. If the tree is finite and has no solution, then eventually all processors would run out of work and then the parallel search will terminate. When a processor runs out of work it selects a target processor of addressing a work request. On receiving a work request, a processor either responds with a part of its work, or a reject message that it does not have any work. This process continues until all processors go idle or a solution is found. Finkel and Manber (1987) have presented the performance results of PDFS for a number of problems such as traveling salesman problem and other problems. Monien and Vornberger (1987) showed that a linear speedup can be obtained to solve combinatorial problems using parallelism. Acar *et al*. (2015) have proposed a parallel algorithm for unordered depth first search on a graph, in PDFS each processor maintain a data structure in which is stored the visited vertices and each processor works locally on its data structure. When a new visited vertex is discovered, it is visited by comparing and swap mechanisms, if it is successes, this vertex is added to the data structure. To minimize the running time on a parallel machine, PDFS performs load balancing to keep all processors busy. The limitation of this algorithm is that the cost of creating each thread must not overweight the benefits of parallelism and that the amount of work in each thread is proportional to the total vertices reachable from the vertex. The total work performed by PDFS is bounded by $O(n + m)$ where n, m are the number of vertices and edges respectively. Jeon *et al*. (2013) have proposed an adaptive parallelization strategy that dynamically selects the degree of parallelism on a query-by-query basis in web search. The paper introduces a dynamic fine grain sharing technique that parallelizes each individual request with preserving the sequential order of execution. The idea is to index the web page related to some topics, then the index data is partitioned and each part is assigned to a thread that searches about the requested document, the threads communicate with each other to merge the top results they have found. When better results have obtained the threads stop processing the query, reducing in that the computation overhead. The results show that the proposed technique outperforms the linear search in

term of delay time and overhead on the system. The main advantage of the proposed work is that dynamically decide the degree of parallelism based on the job under consideration that reduces the partitioning load and overhead.

Multiple search problem is the problem to search for more than one element at the same time, this problem can be solved in parallel or in binary search and generally is solved in time O (n log n) performing n binary search operations according to (Akl and Meijer, 1990). Chen (1990) has proposed a parallel binary algorithm to solve the multiple search algorithm on two sorted arrays of different size in time O(log m) using O(n) processor, where m, n are the size of the two arrays. The algorithm starts by merging the two arrays into one array of length m, considering that n is too small with respect to m and then it searches in parallel in the resultant array. The parallel search is used by (Kaldewey *et al*., 2010) to improve response time for massively parallel architecture like Graphic Processor Architecture (GPU). P-ary algorithm outperforms its previous in term of throughput and response time. In parallel binary search, we searched for four different keys using multithreading, if three keys were found the corresponding thread must be idle until the last thread finishes that influence the response time. In P-ary proposed by (Kaldewey *et al*., 2010), a domain decomposition strategy is applied to search, in which all threads search about the same key in parallel in each time and the searching operation is done by dividing the data sets into zones and each time the key is compared with its two boundary values, then is assigned the disjoint subsets and continuing searching the key. If more than one thread finds the key the searching process is stopped and the results are considered correct. This searching process reduces the searched range about 1/p where p is the number of threads in each iteration. The worst case execution time is $\log_p n$ and the response time is significantly lower than other parallel search algorithms for the same context. The results show that throughput of P-ary algorithm is 30% better than binary search algorithm over sorted data. Aboutabl (2013) has presented a model for parallel query processing in web search, in the web, the interactive response time in searching a document is becoming a challenge due to the tremendous increase in the size of information available. The results show that parallel query processing outperforms cluster based architecture in term of average response time, speed and efficiency.

Any search engine has three main components are the web crawler, indexer and searcher. Aboutabl (2013) has used parallelism in document indexing and in query processing. The most important approaches for parallelism in query processing are replication and index partitioning. In replication approach, consider we have n indexer node the same index is assigned to each of the n nodes and for searching for a term (index) a parallel query is sent to n nodes but each request is processed sequentially. In index partitioning approach, the index or term is divided into parts and the parts are assigned to different nodes and each node is responsible for a subset of the index. For searching a document (term) the query is sent in parallel to different nodes that mean is searched in parallel. The index partitioning mechanism is considered as throughput oriented. The results show that the efficiency of using parallelism in web search about 97.5% for 4 processors, however, it is 91.9% for clustered system with 4 processors. Varsamis *et al*. (2012) have proposed a parallel search algorithm that can scale easily to large input size for searching in large geographical data sets. The idea is that when designing or storing a map a set of insets placed on the map. For that, it is necessary to store these insets and their position. In very large geographic data sets the searching process requires high running time depending on the data set size. For that, it is used parallelism to reduce the running time of searching algorithm. The idea is to transform the geographic data sets (map) to a matrix m*n where n is the length and m is the width of the data set in pixels. In a sequential search, the number of iterations depends on the matrix dimension and is in quadratic order (where it is used two iterations, each iteration defines an index that indicates the data set correspond to sea or land). Using parallel search in this context means parallelize one of the two loops that can reduce the searching algorithm complexity, but still depends on the number of processors. Theoretical speedup of the parallel algorithm is depending on the number of processor and the time of communication between processors. However, the efficiency depends only on the number of processors. The algorithm is implemented using Matlab and the results show that the execution time of the searching algorithm is reduced using parallel processing.

Deterministic Parallel Search Algorithm (DPSA)

DPSA is a deterministic parallel search algorithm that uses divide and conquers technique to divide the array A into two subarrays A_1 to the left and A_2 to the right.

Algorithm Description

Input an array of size n. A [1……n], two items x, y;

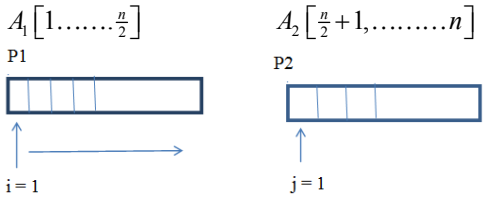
Output x or y is in A, x and y are in A, or x and y are not in A

Processing: divide the array into two subarrays A_1, A_2, If the size of A is an even number, then the two subarrays (A_1 and A_2) are of an equal size which is equal $n/2$ for each. If the size of A is odd, then the size of the first subarray (A_1) will be equal $\frac{n}{2}$ and the size of the second subarray (A_2) will be $n/2 + 1$. Two threads called P_1, P_2 are working in parallel on A_1, A_2 searching for item x and/or item y. The algorithm works recursively in phases, in the first phase, it uses two threads P_1 and P_2 to search for the requested items (x and y) in parallel in the two subarrays A_1, A_2. If one item is found, P_1 and $P2$ will stop and the algorithm passes to the second phase. In the second phase, another two threads called P_3, P_4 continue searching from the location where P_1 and P_2 are stopped in the first phase; it searches sequentially about the missing item only in both A_1 and A_2. The main goal of our algorithm is to reduce the number of comparisons needed to find two searched items. In our algorithm, the worst case in finding an item is $O\left(\frac{n}{2}\right)$ whereas a sequential search needs $O(n)$.

First Phase

DPSA composes a key k as follows: $k = (x \parallel y)$ in binary and in each iteration the two threads work in parallel on A_1 and A_2 from left to right comparing the (key && A_1 [i]) with the values of the array $A_1[i]$, $i = 0..\ \frac{n}{2}$ And the (key && $A_2[i]$) with the values of the array $A_2[i]$, $i = \frac{n}{2}+1\ ..\ n$. When there is a match that means either x or y has been found.

Phase 1	*Pseudocode*
P_1 is a thread that represents a process that searches for the requested items in the left subarray. *P_2 is another thread it works as P_1 on A_2* *Lower indicates the first location in the array* *Upper indicates the last location in the array* *i,j are two indices that indicate the location where the threads are searching.*	*Search_Thread(1)(Left[1],int(n/2)], x, y,Key)* *Thread$_1$_begin = Start timer* *Lower = 1* *Upper = int(n/2)* *Found_Loc1= 0* *While (not (Found x_Left OR Found y_Left) AND (Lower <= Upper))* *Begin* *if ((Left(Lower) And Key) = Left(Lower)) Then* *Begin* *//Candidate found either x or y* *if Left(Lower) = x Then* *Found x_Left = true* *Else if Left(Lower) = y Then* *Found y_Left = true* *Exit loop* *End* *Lower = lower + 1* *End* *Found_Loc1 = Lower* *loop* *Thread1_end = end_ timer*



First DSPA composes its key as follows:

Key = ($x \parallel y$) in binary, where, in each iteration, the threads P_1 and P_2 make the following comparisons:

If (A_1 [i] && key = = A_1 [i]) (1) or

If (A_2 [j] && key = = A_2 [j]) (2)

i takes value from 1 to $\frac{n}{2}$ and j takes values from $\frac{n}{2}+1$ to n

If the either of the previous comparisons is true, that means one of the two items has been found, on the other hand, if the previous two comparisons are true, that means both items have been found.

Second Phase

In the second phase, two threads P_3, P_4 search sequentially about the missing item in the rest of the array as follows: If P_3 reaches $n/2$ and P_4 reaches n without finding the missing item that means the missed

item is not in the array. Otherwise, the missing item has been found and the two threads will stop the search.

Pseudo code of the second phase:

Thread (3)	Thread(4)
Search sequential_missing_item	Search sequential_missing_item
*(Left[*Found_Loc1 + *1], int(n/2], Missing_item)*	(Right[Found_Loc2 + 1], n, Missing_item)
Thread$_3$_begin = Start timer	Thread4_begin = Start timer
FOUND = False	FOUND = False
i = Found_Loc1 + *1*	j = Found_Loc2 + 1
Do While ((Not FOUND) And (i <= n/2))	Do While ((Not FOUND) And (j <= n))
If *Left*(i) = *Missing_item* Then	If Left(j) = Missing_item Then
FOUND = True	FOUND = True
Stop Thread (3)	Stop Thread(3)
Stop loop	Stop loop
Else	Else
i = i + 1	j = j + 1
Loop	Loop
Thread(3)_end = stop timer	Thread(4)_end = stop_timer

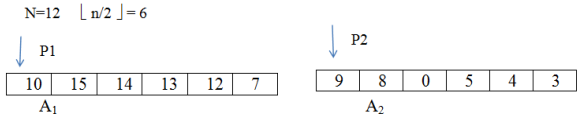
Cases:

- One item is found at $A_1[1]$ and the second is found at $A_2[n/2]$ and that is the best case and the running time, in this case, is $O(1)$
- Duplicate items are found (duplicate x or duplicate y), in phase one, after the if statement (if $A[i]$ && key =A[i]) which means one item is found, it compares A[i] with x if $A[i] = x$ that means x is found in $A_1[1]$ and x can be found also in $A_2[n/2]$. In this case, P_1 and P_2 will halt and P_3 and P_4 continue working sequentially searching for y only. The running time to find x is O (1) and the running time to find y, in this case, will be O (n/2) if y is in the array. The same procedure is used if y is found first
- The average case is that one item is found in A_1, or in A_2 in some location < Upper. Then P_1 and P_2 will halt and P_3 and P_4 continue working sequentially from where P_1 and P_2 are stopped searching for the missing item
- The worst case, in this case, P_1 and P_2, continue working recursively in the first phase until reaching the end of the array without finding any of the two items and that means both items are not in the array. In this case, the running time is O (n/2) for each item

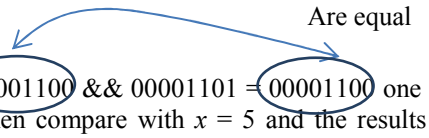
Our algorithm is a scalable algorithm, when it searches about two or more elements at the same time (multiple of 2) and it can search for any item that can be represented in binary (names, words…letters).

How does DPSA Algorithm Work?

Example: Consider the following array
A = [10, 15,14,13,12,7,9,8,0,5,4,3] and we want to search for 12 and 5 in the array in parallel.



Key = (12 or 5) correspond in binary x = 12 and y = 5
Key = ((00001100) || (00000101)) = 00001101
In the first phase: i = 1, j = 7
For i = 1: 00001010 && 00001101 = 00001000 items not found
Are equal
For i = 2 ….i = 3 ……continue



For i = 5 00001100 && 00001101 = 00001100 one item is found. Then compare with x = 5 and the results x is found P_1 stop working.
In the same manner and in parallel P_2 works
In j = 1 00001001 && 00001101 = 00001001 this item is a candidate to be one of the searched items. It is compared with x and y if not equal to both the procedure continues testing $A[j]$ until finding one of the two items or until arriving at $n = j$ without finding any of the items. Figure 1, show a graphical representation of the steps of our algorithm in comparison to binary search algorithm.

Complexity Analysis

The best case of running time for DPSA is $O(1)$, where the two items were found in the first location of A_1 and A_2.

The worst case for searching an item using DPSA is $\frac{n}{2}$ where it is the size of each partition. Our algorithm performs better than binary search because it saves the time to sort the array. The main advantage of our algorithm that it can search for anything can be represented in binary and can search about 2 items in each iteration.

The running time of DPSA for searching n items is $n^2/2$ and in that it outperforms binary search wherein binary search to search n items from the unsorted array will cost as follows:

- Sorting phase vary from n log n $\rightarrow n^2$
- Searching for an element in the binary search algorithm requires log n in the worst case. So, searching n items in binary search requires n log n. That means the total cost for searching n items in binary search varies from (2 n log n) to (n^2+ n log n); however, in DPSA:
- Searching one item in the worst case is $\frac{n}{2}$
- Searching n items require $\frac{n^2}{2}$

Our simulation results show mathematically the difference between DPSA and binary search in searching two items. Then, we have calculated the search for n items. In addition, a mathematical calculation has been done for the number of iterations needed for each algorithm.

Simulation Results

Our proposed algorithm has been implemented from scratch in the C++ programming language. The simulator runs on a 2.20 GHz dual-core machine with 4 GB memory and 64 bits Windows platform operating system.

To evaluate the performance of our algorithm we implement both DPSA and binary search algorithms in the same environment and we made different tests with different input sizes. Then, we compare our results with that obtained from binary search algorithm under the same conditions. In our algorithm, we save the time needed for sorting in searching in parallel in the unsorted array. The running time of the worst case of DPSA (item not found) is invariant for any case, where always the algorithm must search in all the element and is O(n/2). Our results show that DPSA outperforms binary search algorithm in the worst case for searching the first item. In the average case, (where the two elements are found in the array) our algorithm performs similarly to the binary search algorithm. In the worst case, for searching n items, DPSA requires slightly more time than a binary search algorithm, but the binary search algorithm requires more overhead because it searches it sequentially. For that, if the binary search algorithm runs n times to search for items, our algorithm runs n/2 times to search the same number of items. Table 1 shows the results obtained from DPSA for searching in parallel for two items from different input sizes in all cases (best, average and worst).

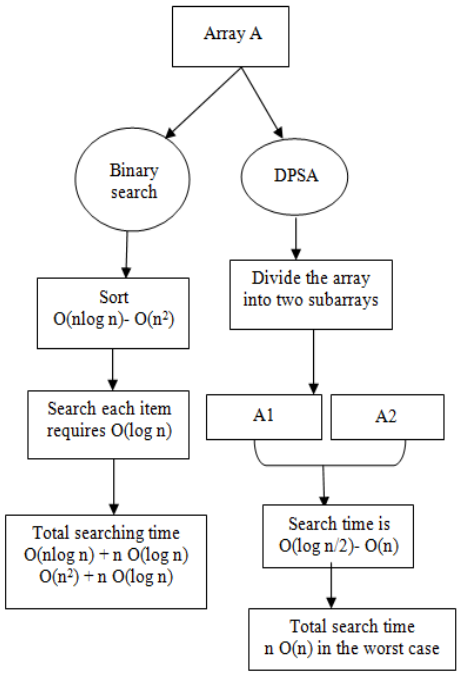


Fig. 1. Graphical representation of DPSA

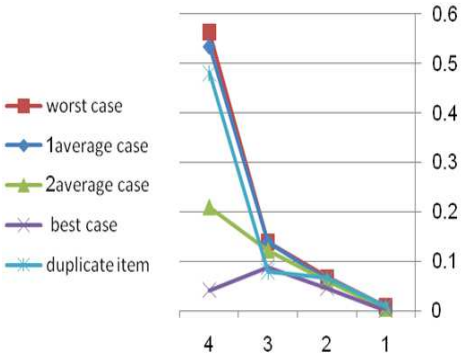


Fig. 2. The running time of DPSA for different input sizes

Table 1. The running time of DPSA for different cases and different input sizes

	Running time (ms)			
Cases	50.000	500.000	1.000.000	5.000.000
Worst case (not found)	0.0080	0.0670	0.1380	0.5630
Average case (find one item randomly)	0.0060	0.0620	0.1380	0.5350
Average case (finding two items randomly)	0.0040	0.0590	0.1220	0.2090
Best case (finding the two items in the first location)	0.0010	0.0460	0.0880	0.0410
Finding one item duplicated	0.0078	0.0650	0.0790	0.4815

Figure 2 shows that the running time of DPSA is increased linearly with the input size. In addition, the running time with a variation of input size differs slightly, which ensures the robustness and scalability of our proposed algorithm.

Table 2 shows the comparison of running time of DPSA and binary search algorithm for different input size. In the table, column 2 indicates the running time of DPSA for different items size column 3 indicates the running time of binary search algorithm for searching the first two items that include the sorting cost. Column (3.2) shows the running time of binary search algorithm for the next two items after having the array sorted and the column (3.3), shows the average running of binary search algorithm considering that the sorting cost will be distributed on the next searches. Comparing column 2 with column (3.3) in Table 2 it is clear that the DPSA performs better than binary search algorithm under the same input size.

Results Discussion

From Fig. 3, it is clear that when the input size is increased the running time of binary search will be higher than DPSA and following the analysis presented in the previous section, when the searching space is increased, if we search for 10 items, DPSA will search it in 5 iterations searching two items at a time and the binary search will search it in 10 iterations. The main advantage of our algorithm is that it searches multiple items at a time and without the need of any kind of sorting.

Analyzing Table 1, confirms that our system is scalable one where if the size of the input is increased n times, the running time will increase less than n times, (Table 3).

From Table 3, it can be concluded that our best results are obtained when the two items were found randomly in the search space (average case). But also, the scalability in the worst case is acceptable.

Table 2. The comparison between the running time of DPSA and the average running time of binary search algorithm for different input size

		Running time (ms) Binary search algorithm		
Input size	Running time (ms) DPSA (Col 2)	First search	Second search (Col 3.2)	Average (col 3.3)
50.000	0.0080	0.2045	0.0035	0.1040
500.000	0.0670	1.7077	0.0037	0.8557
1.000.000	0.1380	3.576	0.0040	1.7900
5.000.000	0.5630	13.9885	0.0065	6.9975

Table 3. The increased in performance of DPSA

Cases	Increase percentage in running time when the input size is increased from 1.000.000-5.000.000 (5 times)	Increase percentage in running time when the input size is increased from 500.000-5.000.000 (10 times)
Worst case (not found)	4.07	8.40
Average case (find one item randomly)	3.87	8.60
Average case (finding two items randomly)	1.71	3.54
Best case (finding the two items in the first location)	4.65	0.89
Finding one item duplicated	6.09	7.40

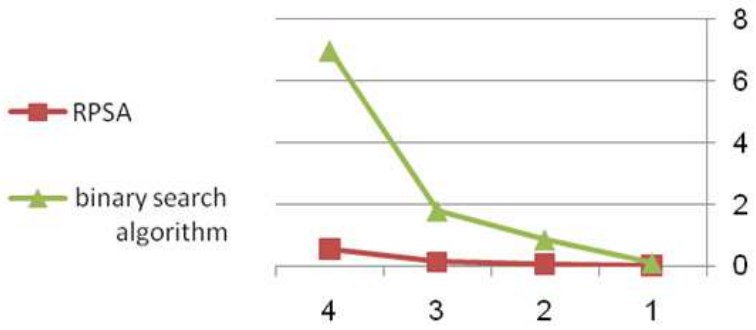


Fig. 3. Comparison between DPSA and binary search algorithm running time

Conclusion and Future Works

In this study, we proposed a parallel search algorithm that performs better than the binary searching algorithm, gaining the time spent in sorting in the binary search algorithm. The main purpose of this paper was to develop an efficient search algorithm that scales well when the searching space is increased. For that, we have used the parallel processing to distribute the load on more than one process. Our results show, that working in parallel and without sorting the input items can give an advantage to the system in reducing the searching time. In many applications the request- response time is important to evaluate the performance of the application and in most cases, the request-response time depends on the searching operation. Reducing the searching time can reduce the request-response time that improves the performance of the system. Our results show that our proposed algorithm is a scalable one and it works well when the search space is increased in searching for n items in parallel searching for 2 items at a time. Our future work is to use the genetic algorithm to search for multiple items in the unsorted array.

Author's Contributions

Sami Qawasmeh: Algorithm design, implementalion and resul analysis and discusion.

Arwa Zabian: Related works, data analyzing and results analysis and discussion.

Ethics

This article is original and contains unpublished material. The corresponding author confirms that all of the other authors have read and approved the manuscript and there are no ethical issues involved.

References

Aboutabl, A.E., 2013. Exploiting parallelism in query processing for web document search using shared-memory and cluster-based architectures. Comput. Inform. Sci., 6: 125-137. DOI: 10.5539/cis.v6n3p125

Acar, U.A., A. Charguéraud and M. Rainey, 2015. A work-efficient algorithm for parallel unordered depth-first search. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Nov. 15-20, ACM., Austin, Texas. DOI: 10.1145/2807591.2807651

Akl, S. and H. Meijer, 1990. Parallel binary search. IEEE Trans. Parallel Distributed Syst., 1: 247-250. DOI: 10.1109/71.80139

Chen, D.Z., 1990. Efficient parallel binary search on sorted arrays.

Finkel, R. and U. Manber, 1987. DIB-a distributed implementation of backtracking. ACM Trans. Programm. Lang. Syst., 9: 235-256. DOI: 10.1145/22719.24067

Grama, A. and V. Kumar, 1995. Parallel search algorithms for discrete optimization problems. ORSA J. Comput., 7: 365-385. DOI: 10.1287/ijoc.7.4.365

Jeon, M., Y. He, S. Elnikety, A.L. Cox and S. Rixner, 2013. Adaptive parallelism for web search. Proceedings of the 8th ACM European Conference on Computer Systems, Apr. 15-17, ACM., Prague, Czech Republic, pp: 155-168. DOI: 10.1145/2465351.2465367

Kaldewey, T., J. Hagen, A. Di Blas and E. Sedlar, 2009. Parallel search on video cards. Proceedings of the 1st USENIX Conference on Hot Topics in Parallelism, Mar. 30-31, USENIX Association, Berkeley California USA., pp: 9-9.

Kumar, V. and V.N. Rao, 1987. Parallel depth first search. Part II. Analysis. Int. J. Parallel Programm., 16: 501-519. DOI: 10.1007/bf01389001

Kumar, V. and V.N. Rao, 1990. Scalable Parallel Formulations of Depth-First Search. In: Parallel Algorithms for Machine Intelligence and Vision, Kumar, V., P.S. Gopalakrishnan and L.N. Kanal (Eds.), Springer-Verlag, New York, pp: 1-41.

Monien, B. and O. Vornberger, 1987. Parallel processing of combinatorial search trees. Parallel Algorithms Architectures. DOI: 10.1007/3-540-18099-0_29

Varsamis, D., P. Mastorocostas, A. Papakonstantinou and N. Karampetakis, 2012. A parallel searching algorithm for the insetting procedure in Matlab Parallel Toolbox. Proceedings of the Federated Conference on Computer Science and Information System, Sept. 9-12, IEEE Xplore Press, pp: 587-593.