Optimizing Multi-Core Algorithms for Pattern Search

Veronica Gil-Costa[1,2], Cesar Ochoa[1] and Marcela Printista[1,2]

[1] LIDIC, Universidad Nacional de San Luis,
Ejercito de los Andes 950, San Luis, Argentina
[2] CONICET, Argentina

{gvcosta,mprinti}@unsl.edu.ar, elcessar@gmail.com

Abstract. The suffix array index is a data structure formed by sorting the suffixes of a string into lexicographic order. It is used for string matching, which is perhaps one of those tasks on which computers and servers spend quite a bit of time. Research in this area spans from genetics (finding DNA sequences), to keyword search in billions of web documents, to cyber-espionage. The attractiveness is that they are completely “array based” and have some benefits in terms of improving the locality of memory references. In this work we propose and evaluate the performance achieved by a static scheduling algorithm for suffix array query search. We focus on multi-core systems 32-core system. Results show that our proposed algorithm can dramatically reduce the cost of computing string matching.

Keywords: Pattern query search, multi-core, scheduling.

1 Introduction

New powerful processors and cheap storage allow to considerate alternative models for information retrieval other than the traditional one of a collection of documents indexed by keywords. One of these models is the full text model. In this model documents are represented by either their complete full text or extended abstracts. The user expresses his/her information need via words, phrases or patterns to be matched for and the information system retrieves those documents containing the user specified strings. While the cost of searching the full text is usually high, the model is powerful, requires no structure in the text, and is conceptually simple [9].

To reduce the cost of searching a full text, specialized indexing structures are used. Suffix arrays or pat arrays [9] are sophisticated indexing structures which take space close to the text size [10]. They efficiently perform phrases searching or complex queries such as regular expressions. In addition, suffix arrays can be used to index texts other than occidental natural languages, which have clearly separated words that follow some convenient statistical rules [9]. Examples of these applications include computational biology (ADN or protein strings), music retrieval (MIDI or audio files), oriental languages (Chinese, Korean, and others), and other multimedia data files.

In this work we evaluate the performance achieved by the suffix array index on a multi-core system, as multi-core systems have increasingly gained importance in high performance computers. Furthermore, we propose a scheduling algorithm devised to reduce access memory conflicts by increasing data locality and concurrency. We design our proposed scheduler to divide the pattern query search process into two steps. In the first step, all threads collaborate to classify queries into at most four groups. Each query is assigned to a particular group according to which part of the index is going to be required to process that query. In the second step, threads are assigned to process just one group of queries.

There are a number of methods proposed to schedule tasks over a set of cores. For instance, [8] shows how to reduce task pool overheads, [1] proposes a scheduler using information to determine the number of processors assigned to execute a job. The work presented in [7] allows stealing tasks from a queue using a specific data structure. The work in [15] presented a scheduling algorithm for metric space searches. The proposals are based on local or global index partitions. In the former, the database is split among threads and single index is build for each thread. Then all queries are processed by all threads. In the global index partition a single index is built and then the index is evenly distributed among threads. We did not find any related work in the literature for multi-core systems applied to problems with the features of pattern searches with suffix arrays.

The paper is organized as follows. The suffix array index and the parallel approach for multi-core systems are presented in Section 2. Our proposed scheduler algorithm is described in Section 3. Section 4 details the results obtained. Finally, conclusions are drawn in Section 5.

2. Suffix Array Data Structure

String matching is perhaps one of the most studied areas of computer science [4]. This is not surprising, given that there are multiple areas of application for string processing, being information retrieval and computational biology among the most notable nowadays. The string matching problem consists of finding some string (called the pattern) in some usually much larger string (called the text).

Many index structures have been designed to optimize text indexing [12,13]. In particular, suffix arrays [9] has already 20 years old and it is tending to be replaced by indices based on compressed suffix arrays or the Burrows-Wheeler transform [2,3], which require less memory space. However, these newer indexing structures are slower to operate. A suffix array is a data structure that is used for quickly searching for a keyword in a text database. Suffix arrays only provide efficient querying if T plus the index require less main memory than is available on the host computer, because random accesses are required to the index and the text

Essentially, the suffix array is a particular permutation on all the suffixes of a word. Given a text T[1..n] over an alphabet $\sum$, the corresponding suffix array SA[1..n] stores pointers to the initial positions of the text suffixes. The array is sorted in lexicographical order of the suffixes. As shown in Fig. 1 the text is “Performance$” and the symbol $\$ \notin \sum$ is a special text terminator, that acts as a sentinel. Given an

interval of the suffix array, notice that all the corresponding suffixes in that interval form a lexicographical subinterval of the text suffixes.

	1	2	3	4	5	6	7	8	9	10	11	12
T:	P	e	r	f	o	r	m	a	n	c	e	$

i	text suffix	SA[i]
1	$	12
2	ance$	8
3	ce$	10
4	e$	11
5	erformance$	2
6	formance$	4
7	mance$	7
8	nce$	9
9	ormance$	5
10	Performance$	1
11	rmance$	6
12	rformance$	3

Fig. 1: Suffix array for the example text “Performance$”.

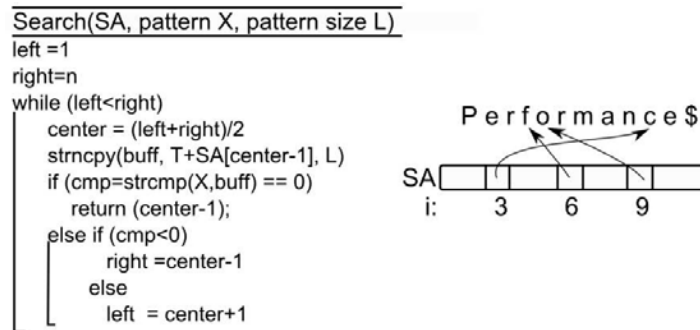

Fig.2: Search algorithm for a pattern query X of size L.

The suffix array stores information about the lexicographic order of the suffixes of the text T, but there is no information about the text itself. Therefore, to search for a pattern X[1..m], we have to access both the suffix array and the text T, with length |T|=n. Therefore, if we want to find all the text suffixes that have X as a prefix—i.e., the suffixes starting with X, and since the array is lexicographically sorted, the search for a pattern proceeds by performing two binary searches over the suffix array: one with the immediate predecessor of X, and other with the immediate successor. We obtain in this way an interval in the suffix array that contains the pattern occurrences. Fig. 2 illustrates how this search is carried out. Finally, Fig. 3 illustrates the code of the sequential search of NQ pattern queries. To this end, all patterns of length L are

loaded into main memory (to avoid the interference of disk access overheads). The algorithm scans sequentially the patterns array using the next() function, and for each pattern X the SA search algorithm is executed.

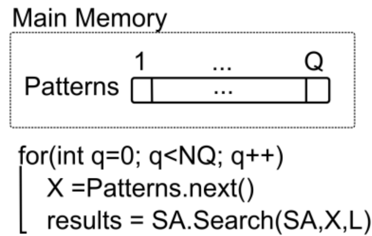

Fig.3: Sequential search algorithm

2.1 Multi-core Approach

Parallelization for shared memory parallel hardware is expected to be both, the most simple and less scalable parallel approach to a given code [13]. It is simple, due to the shared memory paradigm requires few changes in the sequential code and it is almost straightforward to understand for programmers and algorithm designers.

In this work we use the OpenMP[OMP] library, which allows for a very simple implementation of intra-node shared memory parallelism by only adding a few compiler directives. The strategy is to distribute the NQ pattern queries between execution cores. The OpenMP parallelization is made with some simple changes to the sequential code. To avoid read/write conflict, every thread should have a local *buff* variable which stores a suffix of length L, also local *left, right* and *cmp* variables which are used to decide the subsection of the SA where to continue the search (see Fig. 2 above). To avoid using a critical section which incurs into an overhead (delay in execution time), we replace the result variable of Fig. 3 by an array *results*[1..NQ]. Therefore, each thread stores the results for the pattern Xi into *results*[i]. The “for” instruction is divided among all threads by means of the “#pragma omp for” directive. Fig. 4 shows the threaded execution using the OpenMP terminology. Also, the sched_setaffinity function is used in this implementation to obtain performance benefits and to ensure that threads are allocated in cores belonging to the same sockets.

```
omp_set_num_threads(NT)  //Number of threads
#pragma omp parallel private(tid,X) shared(L,results)
  tid = omp_get_thread_num()
  #pragma omp for
    for (int q=0; q<NQ; q++)
        X = Patterns[q]
        results[q]  = SA.Search(SA,X,L)
```

Fig. 4: Parallel search algorithm.

3. Scheduling Pattern Queries

In this section we present a static scheduling algorithm devised to improve data locality for multi-core. To this end, we divide the pattern query processing operation into two steps. In the former, all threads work to classify queries into four groups. In the last step, threads search the queries using the Suffix Array index. Both steps are separated by a barrier synchronization to avoid data corruption.

Fig. 5 shows our proposed static scheduling algorithm which aims to prune those parts of the index which will not be acceded by any query. In this figure, we show four local queues assigned to different parts of the index. The partition criterion is set according to the SA centers. Then, all threads compare the incoming queries with the centers (three centers) of the suffix array. According to the comparison result, each query is assigned to a local queue. Pattern queries matching any of the three centers are considered solved. Therefore, they are not included in any local queue for processing in the second step.

After all incoming queries are pre-processed, we assign threads to each part of the index. The basic idea is to allocate threads according to the workload. To this end, we compute the minimum number of threads required by each partition:

```
minThreads[i] = (Local_queue[i].size()/totalQ)*NT

if ( tid < minThreads[i] )
    idGroup = i
```

where Local_queue[**i**].size() is the number of queries assigned to the index partition **i**, with **i**={1,2,3,4}. totalQ is the total number of queries that have to be processed in the second step, and NT is the number of threads. Then, each thread determines its index partition (idGroup) using its thread identifier (tid). Notice that in Fig. 5 the fourth index partition has no query assigned to the local queue. Therefore, no thread is allocated to this partition.

If the scheduling algorithm is executed with only two threads, we divide the index suffix array in two partitions (with two local queues) using the middle center as the partition criteria. With more threads we keep at most four partitions, due to increasing the number of partitions implies comparing the query with more elements of the suffix array which leads, for most of the queries, to finish their processing process before executing step 2.

In the second step, threads remove pattern queries from the local queue and search for them in the corresponding index partition. When more than one thread is allocated to the same index partition, they work in parallel by processing one query per thread.

Our hypothesis is that by dividing the pattern query search into two steps we increase data locality on the cache memories. In the first step threads compare the queries against three elements of the suffix array. Therefore, each thread will find these elements in cache avoiding accessing main memory. In the second step, threads access one part of the index, which also tends to increase the number of elements found in cache. The effect of our hypothesis has a direct effect on running time.

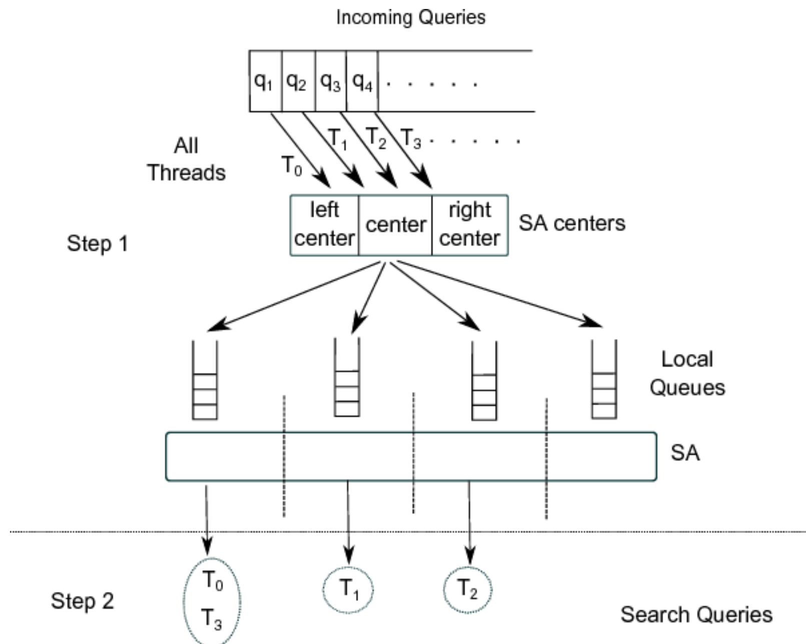

Fig. 5: Static Scheduling Algorithm.

4 Evaluation

4.1 Computing Hardware and Data Preparation

Experiments were performed on a 32-core platform with 64GB Memory (16x4GB), 1333MHz and a disk of 500GB. 2x AMD Opteron 6128, 2.0GHz, 8C, 4M L2/12M L3, 1333 Mhz Maximum Memory. Power Card, 250 volt, IRSM 2073to C13. The operating system is Linux Centos 6 supporting 64 bits. As shown in Fig. 7, we used the hwlock (Hardware Locality) tool [6] which collects all information from the operating system and builds an abstract and portable object tree: memory nodes (nodes and groups), caches, processor sockets, processor cores, hardware threads, etc. We used the OpenMP [11] library to implement the parallel codes.

To evaluate the performance of the SA index we use a 100-megabyte DNA text from the Pizza&Chili Corpus[5]. The resulting suffix array requires 801M. The text length is n=104857600. For the queries, we used 900000 random search patterns of length 10 and 30.

Fig. 7: Four sockets with two nodes. Each node has four cores.

4.2 Performance Evaluation

In this section we present experiment results for processing pattern queries of length L=10 and L=30 over the DNA text collection. We evaluate the baseline parallel suffix array algorithm as described in Section 2.1 and the parallel scheduler algorithm described in Section 3.

Fig. 8 and 9 show speed-up and Fig. 10 and 11 show efficiency measured as Speed-up/T, where T is the number of threads. Efficiency close to 1 indicates an optimal performance of the parallel algorithm. In all cases we use one thread per core and we execute the sched_setaffinity function.

With a small pattern length, the scheduler algorithm dramatically improves the performance of the suffix array search algorithm. The scheduler reports an improvement of 6,4% for T=2 and 77% better speed-up for T=32. Although, for T=32 the speed-up archived is 18,25 far from the optimal. These results can be

verified with efficiency reported in Fig. 10. In this last figure, the scheduler algorithm reports an efficiency close to 1 for T=2,4 and 8. Then with more threads, efficiency goes down to 60%. On the other hand, the baseline suffix array algorithm reports an efficiency close to 85% for T=2 and T=4, but it is drastically reduced to 10% for T=32.

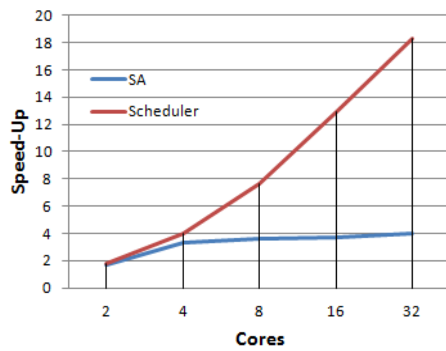

Fig. 8 Speed-Up achieved for pattern length L=10.

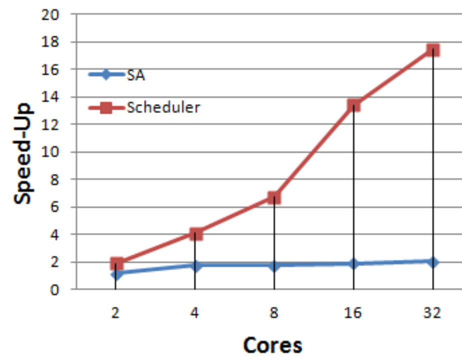

Fig. 9 Speed-Up achieved for pattern length L=30.

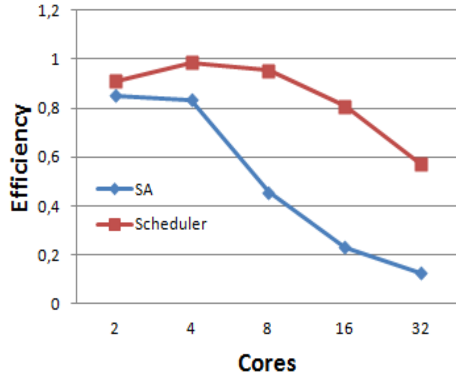

Fig.10 Efficiency for pattern length L=10.

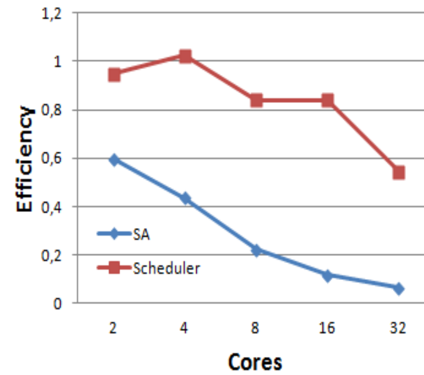

Fig. 11 Efficiency for pattern length L=30.

With a larger pattern length of L=30 (Fig. 9), the improvement reported by the scheduler algorithm over the baseline suffix array search algorithm is larger. In this case, the scheduler presents an improvement of 37% for T=2, and for T=32 the improvement is about 88%. Again, the efficiency showed in Fig. 11 confirms that the proposed scheduler algorithm reports better performance than the baseline.

For L=10, the first step consumes 15% of the total running time and the second step consumes 85% of the total running time. But with L=30, the first step consumes 12% of the total running time and the second step consumes 88% of the total running time.

In Table 1, we show the efficiency obtained by the scheduler algorithm in both processing steps. We measure the efficiency as the amount of comparisons performed by each thread (cmp) divided by the maximum number of comparisons (Max(cmp)). The formula is Σ cmp/Max(cmp)/T, where T is the number of threads. Efficiency close to one indicates that all threads perform the same amount of comparisons.

	Efficiency- Scheduler		
Threads	Step 1	Step 2	SA Baseline
2	0,99	0,751	0,99
4	0,99	0,750	0,99
8	0,99	0,748	0,99
16	0,99	0,748	0,99
32	0,99	0,748	0,99

Table 1: Efficiency reported by the Scheduling algorithm in each processing steps.

Table 1shows that the scheduler algorithm presents a balance workload in the first step, as all threads reports an efficiency of 99%. In the second step, efficiency goes down to 74% in average. Namely, some threads support a workload 30% higher.

The baseline suffix array parallel algorithm also presents an efficiency of 99%. In other words, all threads perform almost the same amount of comparisons. This is due to all queries are evenly processed by threads.

5 Conclusions

In this paper we presented a two step scheduler algorithm to improve suffix array pattern query searches. The design of the algorithm is devised to improve the data locality which has a direct effect on running time. It also prunes part of the index that will not be accessed by a given query. In the first step, the algorithm classifies and put pattern queries into at most four groups. Then, threads are assigned to process the queries of each group according to the workload.

Results show that our proposal improves the performance of the baseline suffix array index. But they also show that there is room for more research work to improve speed-up. Thus, as future work we plan to study dynamic scheduling algorithms. Namely, threads can be initially assigned to one group of queries and then during execution time, they can be moved to other groups. In this case, we must evaluate whether the cost of moving threads (cost of cache refreshing) is negligible. We also plan to study the effect of processing queries with different arrival rates.

References

1. K. Agrawal and Y. He and E. Leiserson. Adaptive work stealing with parallelism feedback. In Principles and Practice of Paralle Computing, pages 112-120. 2007.
2. Donald Adjeroh, Tim Bell, and Amar Mukherjee. The Burrows-Wheeler Transform: Data Compression, Suffx Arrays, and Pattern Matching. Springer, 2008.
3. Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Research Report 124, Digital Systems Research Center, Palo Alto California, May 1994.
4. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms (3rd ed.). MIT Press, 2009.
5. Ferragina, P. and Navarro, G. 2005. The Pizza&Chili corpus — compressed indexes and their testbeds.
6. The Portable Hardware Locality (hwloc): http://www. open-mpi. org/ projects/ hwloc/
7. D. Hendler and N. Shavit. Non-blocking steal-half work queues. In PODC, pages 280-289. 2002.
8. R. Hoffmann and M. Korch and T. Rauber. Performance Evaluation of Task Pools Based on Hardware Synchronization. In Supercomputing Conference. 2004.
9. Manber, U. and Myers, G., “Suffix arrays: A new method for on-line string searches”, SIAM J. Computation, vol.22, no.5, pp.935-948, 1993.
10. M. Marin and G. Navarro, "Distributed Query Processing using Suffix Arrays", In International Symposium on String Processing and Information Retrieval (SPIRE 2003), Manaus, Brazil, Oct. 8-10, Lecture Notes in Computer Science 2857, pp. 311-325, Springer, 2003.
11. OpenMP Architecture Review Board, OpenMP Application Program Interface - Version 3.1, July 2011.Available athttp://openmp.org/wp/
12. Jens Stoye. Suffix tree construction in ram. In Encyclopedia of Algorithms. Springer, 2008.
13. Fernando G. Tinetti, Sergio M. Martin. Sequential Optimization and Shared and Distributed Memory Optimization in Clusters: N-BODY/Particle Simulation. Parallel and Distributed Computing and Systems -PDCS. 2012
14. Peter Weiner. Linear pattern matching algorithms. In Proceedings of the 14th Annual Symposium on Switching and Automata Theory, pages 1-11, 1973.
15. Gil-Costa Veronica, Barrientos Ricardo, Marin Mauricio and Bonacic Carolina. Scheduling Metric-Space Queries Processing on Multi-Core Processors. Euro-PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing. IEEE. Pp. 187-194. Pisa, Italy February 17th to 19th, 2010. ISBN: 978-0-7695-3939-3