Suffix Array Performance Analysis for Multi-Core Platforms

Verónica Gil-Costa[1,2], Cesar Ochoa[1], and A. Marcela Printista[1,2]

[1]LIDIC, University of San Luis,
Argentina

[2]CONICET,
Argentina

gvcosta@unsl.edu.ar

Abstract. Performance analysis helps to understand how a particular invocation of an algorithm executes. Using the information provided by specific tools like the profiler tool Perf or the Performance Application Programming Interface (PAPI), the performance analysis process provides a bridging relationship between the algorithm execution and processor events according to the metrics defined by the developer. It is also useful to find performance limitations which depend exclusively on the code. Furthermore, to change an algorithm in order to optimize the code requires more than understanding of the obtained performance. It requires understanding the problem being solved. In this work we evaluate the performance achieved by a suffix array over a 32-core platform. Suffix arrays are efficient data structures for solving complex queries in a number of applications related to text databases, for instance, biological databases. We perform experiments to evaluate hardware features directly aimed to parallelize computation. Moreover, according to the results obtained by the performance evaluation tools, we propose an optimization technique to improve the use of the cache memory. In particular, we aim to reduce the number of cache memory replacement performed each time a new query is processed.

Keywords. Multi-core, suffix array.

Análisis de performance para el arreglo de sufijos sobre plataformas multi-core

Resumen. El análisis de performance es utilizado para entender cómo se ejecuta una invocación particular de un algoritmo. Al utilizar la información provista por las herramientas específicas como Perf o “Performance Application Programming Interface” (PAPI), el proceso de análisis de performance provee un puente entre la ejecución del algoritmo y los eventos de los procesadores de acuerdo a las métricas definidas por el desarrollador. También es útil para encontrar las limitaciones del rendimiento del algoritmo, las cuales dependen del código. Además, para modificar un algoritmo de forma tal de optimizar el código, es necesario no sólo entender el rendimiento obtenido, sino que requiere entender el problema que se quiere resolver. En este trabajo, evaluamos el rendimiento obtenido por el arreglo de sufijos en un procesador de 32 cores. Los arreglos de sufijos son estructuras de datos eficientes para resolver consultas complejas en aplicaciones relacionadas con bases de datos textuales, por ejemplo bases de datos biológicas. Ejecutamos experimentos para evaluar las características del hardware con el objetivo de mejorar el cómputo paralelo. Además, de acuerdo a los resultados obtenidos a través de las herramientas de evaluación de performance, proponemos una técnica de optimización para mejorar el uso de la memoria cache. En particular, nuestro objetivo es reducir el número de reemplazos realizados en las memorias caches.

Palabras clave. Multi-core, arreglo de sufijos.

1 Introduction

Several microprocessor design techniques have been used to exploit the parallelism inside a single processor. Among the most relevant techniques we can mention bit-level parallelism, pipelining techniques and multiple functional units [4]. These three techniques assume a single sequential flow of control which is provided by the compiler and which determines the order of execution in case there are dependencies between instructions. For a programmer, this kind of internal techniques has an advantage of allowing parallel execution of instructions by a sequential programming language. However, the

degree of parallelism and pipelining obtained by multiple functional units is limited.

Recently, the work in [8, 9] showed that increasing the number of transistors on a chip leads to improvements in the architecture of a processor to reduce the average time of execution of an instruction. Therefore, an alternative approach to take advantage of the increasing number of transistors on a chip is to place multiple cores on a single processor chip. This approach has been used in desktop processors since 2005 and is known as multi-core processors. These multi-core chips add new levels in the memory (cache) hierarchy [10]. There are significant differences in how the cores on a chip or socket may be arranged. Memory is divided up into pages and pages can be variable in size (4K, 64K, 4M). Each core has an L1 cache capable of storing a few Kb. A second level called L2 cache is shared by all cores grouped on the same chip and has a storage capacity of a few Mb. An example is the Intel machine Sandy Bridge-E which has up to 8 cores on a single chip, 32K L1 cache, 256 K L2 cache, and up to 20M L3 cache shared by all cores on the same chip.

To take advantage of current commodity architectures, it is necessary to devise algorithms that exploit (or at least do not get hindered by) these architectures. There are several libraries for multi-core systems as OpenMP [12] which has a high level of abstraction, TBB [13] that uses cycles to describe the data parallelism, and others like IBM X10 [4] and Fortess [1] focusing on data parallelism but also providing task parallelism. A multi-core system offers all cores a quick access to a single shared memory, but the amount of available memory is limited. The challenge in this type of architecture is to reduce the execution time of programs.

In this work, we propose to analyze the performance of a string matching algorithm in a multi-core environment, where communication is realized only through cache and memory, there are no dedicated instructions to do either synchronization or communication, and dispatch must be done by (slow) software. In particular, we propose to use the Performance Application Programming Interface (PAPI) tool and the Perf tool to perform performance analysis (which helps to improve applications by revealing their drawbacks) over the suffix array index [11] and then, guided by the results obtained, we propose an optimization technique to increase the number of cache hits. The suffix array index is used for string matching, which is perhaps one of the 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, and even to cyber-espionage. In fact, several problems are also interpreted as a string matching problem to be tractable.

The remainder of this article is organized as follows. In Section 2, we describe the suffix array index and the search algorithm. In Section 3, we present the parallel algorithm and the proposed optimization. In Section 4 we show experimental results. Conclusions are given in Section 5.

2 Suffix Array

String matching is perhaps one of the most studied areas of computer science [5]. This is not surprising given that there are multiple areas of application for string processing, information retrieval and computational biology being among the most notable nowadays. The string matching problem consists in finding a particular string (called the pattern) in some usually much larger string (called the text).

Many index structures have been designed to optimize text indexing [14, 16]. In particular, suffix arrays [11] have already been in use for 20 years and they tend 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 used for quick searching for a keyword in a text database.

Conceptually, a 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 an example, Figure 1 shows the text “Performance$” in which the symbol $\$ \notin \sum$ is a special text terminator which acts as a sentinel. Given an

	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$"

```
Search(SA, pattern X, pattern size L)
left =1
right=n
while (left<right)
    center = (left+right)/2
    strncpy(buff, T+SA[center-1], L)
    if (cmp=strcmp(X,buff) == 0)
       return (center-1);
    else if (cmp<0)
           right =center-1
        else
           left  = center+1
```

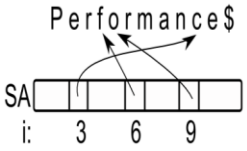
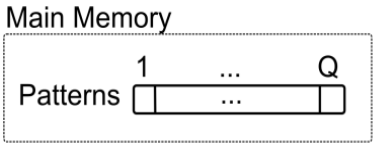


Fig. 2. Search algorithm for a pattern X of size L

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

The suffix array stores information about the lexicographic order of the suffixes of the text T, but does not include 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




```
for(int q=0; q<NQ; q++)
    X =Patterns.next()
    results = SA.Search(SA,X,L)
```

Fig. 3. Sequential search algorithm

array is lexicographically sorted, the search for the pattern proceeds by performing two binary searches over the suffix array: one with the immediate predecessor of X, and the other with the immediate successor. This way we obtain an interval in the suffix array that contains the pattern occurrences. Figure 2 illustrates how this search is carried out. Finally, Figure 3 illustrates the code of the sequential search of NQ patterns. To this end, all patterns of length L are loaded into main memory (to avoid the interference of disk access overheads). The algorithm sequentially scans the patterns in the array using the next() function, and for each pattern X the SA search algorithm is executed.

3 Shared Memory Parallelization

Parallelization for shared memory parallel hardware is expected to be both the most simple and less scalable parallel approach to a given code [15]. It is simple due to the fact that the shared memory paradigm requires few changes in the sequential code and it can be understood in a straightforward manner by programmers and algorithm designers.

In this work we use the OpenMP [14] library which allows a very simple implementation of intra-node shared memory parallelism only by adding a few compiler directives. The OpenMP parallelization is achieved with some simple changes in the sequential code. To avoid read/write conflict, every thread should have a local buff variable which stores a suffix of a length L, also local left, right and cmp variables used to decide in which subsection of the SA the search must be continued (see Figure 2 above). To

```
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

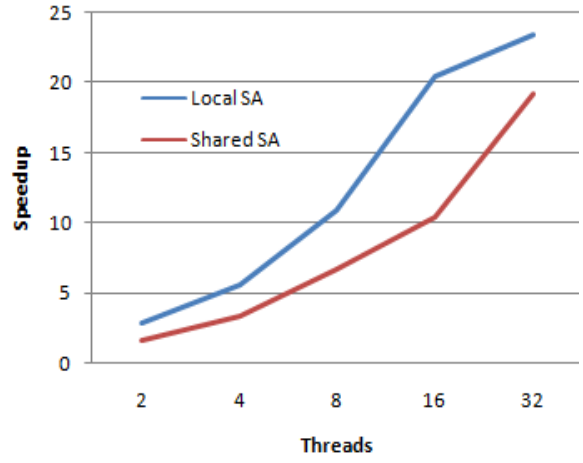
avoid using a critical section which incurs an overhead (delay in execution time), we replace the result variable of Figure 3 by an array of results [1..NQ]. Therefore, each thread stores the results for the pattern Xi into results[i].

The "for" instruction is divided among all threads by means of the "#pragma omp for" directive. Figure 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.

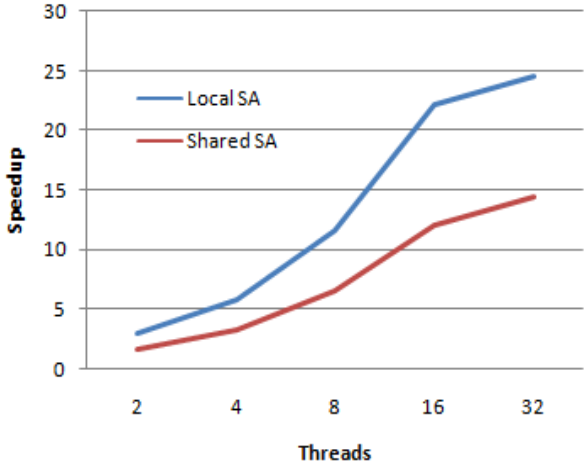
We further investigate the effect of using a local SA index per thread or a global shared SA index. To this end, we performed experiments to measure the speedup (measured as the sequential execution time divided by the parallel execution time) achieved with both approaches. Figure 5 shows the results obtained with up to 32 threads, an index size of 801M and pattern length of L=10 and L=20. In both cases, efficiency is improved by using a local data structure (17% with L=10 and 41% with L=20). Hence, despite performing read-only access to the SA index, there is an overhead associated with the search operation over the shared data structure when many threads process the search patterns. This overhead includes the administration (creation/deletion) of the local variables such as buff, cmp, left, right each time we call the Search function. But as the local approach requires more memory, it can be used only when

NT*sizeof(SA index) < Memory size,

where NT is the number of threads.



(a)

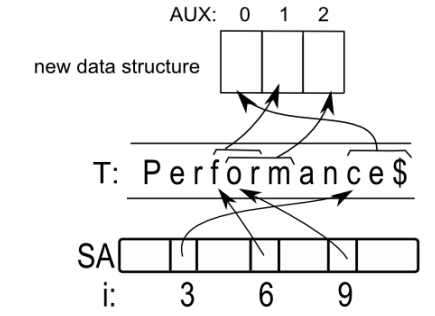


(b)

Fig. 5. Speedup achieved when each thread uses a local copy of the index (Local SA) and when all threads share the same SA index. (a) Results obtained with L=10 and (b) L=20

3.1 Optimization

The suffix array search algorithm presents features that make it suitable to improve its efficiency in multi-core processors. In particular, we propose to take advantage of the binary search algorithm which makes some elements of the suffix array more feasible to be accessed than others, namely, the center element of the suffix array and the left and right centers as shown in




```
Search(SA, pattern X, pattern size L)
left =1;  right=n;  step = -1;  it =0;
while (left<right)
   center =(left+right)/2
   if (step==-1)
    strncpy(buff, T+AUX[1], L)
    else if ( step == 1 && it==1 )
     strncpy( buff, AUX[0], L ); //left
     else if ( step == 2  && it==1 )
        strncpy( buff, AUX[2], L ); //right
        else
          strncpy( buff, T+SA[center-1], L );
   if (cmp=strcmp(X,buff) == 0)
         return (center-1);
   it++;
   if (cmp<0)
      right =center-1
      step = 1
   else
      left  = center+1
       step = 2
```

Fig. 6. Cache optimization

Figure 6 for a pattern length L=3.This effect is known as temporal locality.

Therefore, we propose to introduce an extra data structure storing the most frequently referenced elements of the suffix array. This new data structure is small enough to fit into the low level cache memory and it is used to prune the search of patterns. The proposed algorithm is detailed in Figure 6. To begin searching for a new pattern (iter=1), we access the second position of the AUX data structure. If we have to continue the search at the left side of the text stored into AUX[1], the next iteration (iter=2) compares the first element of AUX with the search pattern X, otherwise the third element of AUX and the pattern X are compared.

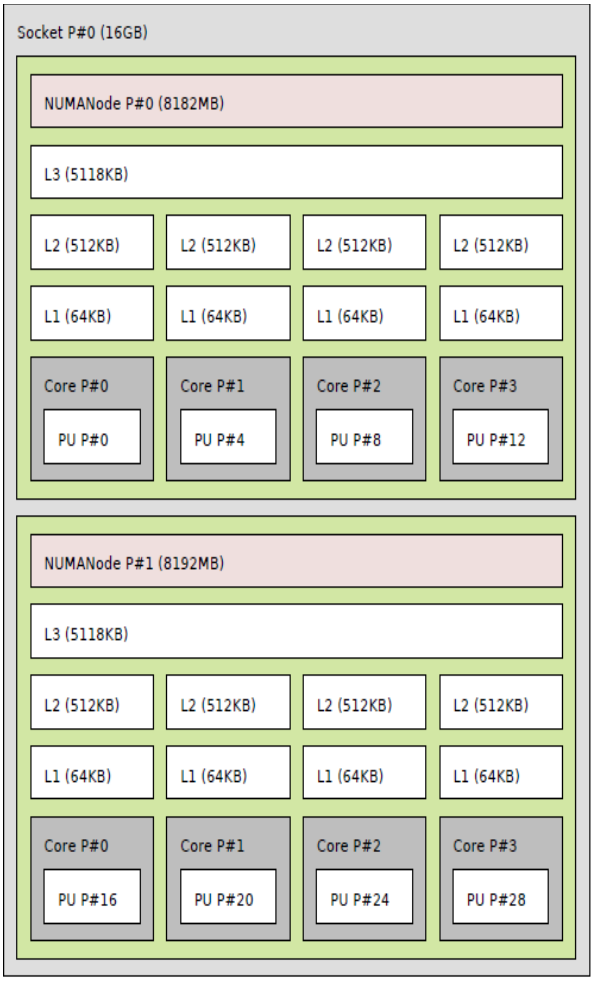


Fig. 7. Four sockets with two nodes. Each node has four cores

4 Evaluation

4.1 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 Cord, 250 volt, IRSM 2073to C13. The operating system is Linux Centos 6 supporting 64 bits. As shown in Figure 7, we used the hwlock (Hardware Locality) tool [7] which

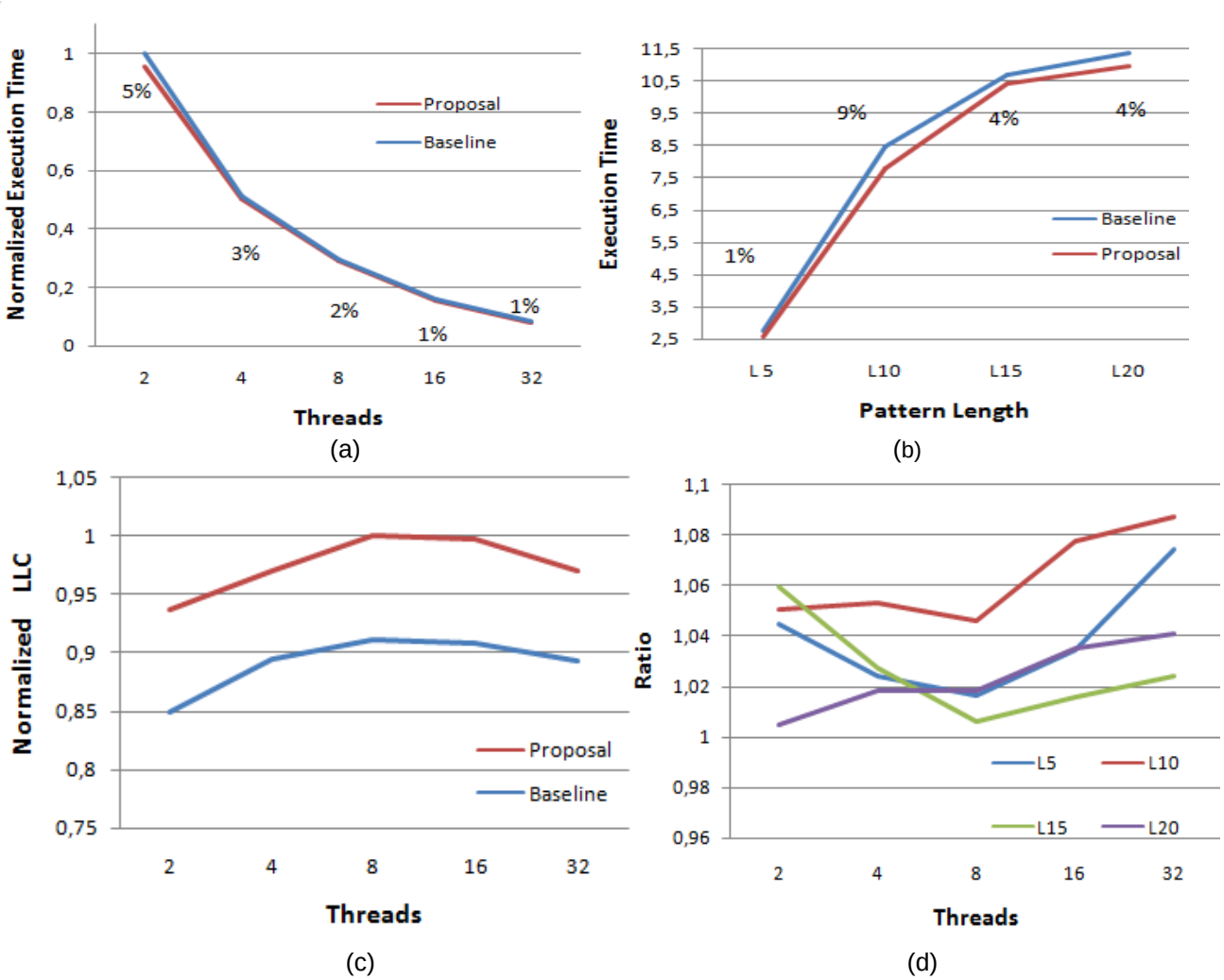


Fig. 8. Results obtained with a small index size. (a) Execution time reported as the baseline and the proposed approach for a pattern length L=5; (b) Execution time for different pattern length and 32 threads; (c) LLC hits for L=5 and (d) Ratio of the baseline execution time to the proposed approach execution time with 32 threads

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.

To evaluate the performance of the SA index, we use a 3-megabyte and 100-megabyte DNA text from the Pizza&Chili Corpus [6]. The resulting suffix array requires 25M and 801M, respectively. The text length in each case is n=3145728 and n=104857600. For the queries, we used 1000000 random search patterns of length 5, 10, 15 and 20.

4.2 Performance Evaluation

In this section we evaluate the performance of the SA index using the PAPI tool version 5.0.1. In particular, we measure the cache hit ratio, as well as the execution time. We use the PAPI_thread_init function which initializes thread

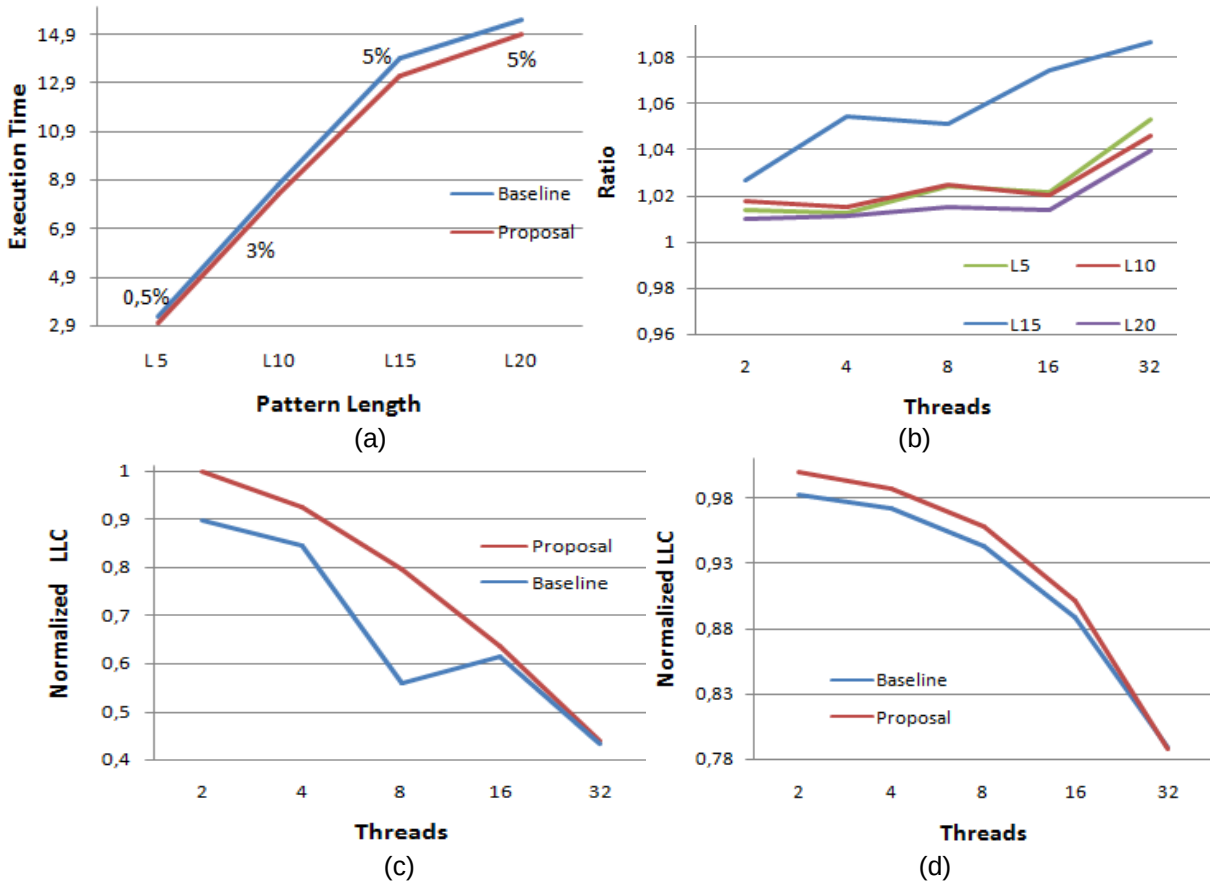


Fig. 9. Results obtained with a larger index size. (a) Execution time for different pattern length and 32 threads; (b) Ratio of the baseline execution time to the proposed approach execution time. (c) LLC hits for L=5 and (d) LLC hits for L=20

support in the PAPI library. When all threads finish their work, we execute the PAPI_unregister_thread function. For the Perf performance tool we use the version 3.2.30. For the Perf tool we count the L1 data cache hits and the Last level cache (LLC) hits.

In Figure 8 we show results obtained for a small index of 25M and different pattern lengths. Figure 8(a) demonstrates that the normalized running times obtained with both the baseline and our proposed optimization algorithm for a pattern length of L=5 are almost the same. We show results normalized to 1 in order to better illustrate the comparative performance. To this end, we divide all quantities by the observed maximum in each case. Figure 8(c) shows the LLC hits reported for the same experiment. Our proposal reports an improvement of almost 20% in average which remains constant with various numbers of threads although this benefit is not reflected in the running time. Figure 8(b) shows that the best performance is achieved by our proposal for a pattern length of L=10. This last result is confirmed by Figure 8(e) where the best ratio (running time reported as the baseline divided by the running time shown by the proposal) is obtained with a pattern length L=10.

Figure 9 shows results obtained with a large index of size 801M. In this case, the best performance of our optimization is reported, in Figure 9(a) for a pattern of length L=15 and L=20. This result is confirmed in Figure 9(b) where the

best ratio of the running time reported as the baseline to the running time shown by our proposal is given for a search pattern L=15.

Figure 9(c) and (d) show the LLC hit for L=5 and L=20. In both figures, the LLC hits tend to go down as we increase the number of threads. In the former, our proposal obtains a gain of 15% in average but in the last figure the gain is reduced to 4%.

5 Conclusions

In this work we analyzed the performance of the suffix array index by means of the Perf and PAPI tool on a multi-core environment. Parallel codes were implemented with the OpenMP library. Experiments were performed with various index sizes and various patterns length over a 32-core platform. We ran experiments to evaluate hardware features directly aimed to parallelize computation. Results show that read-only operations performed over shared data structures affect performance. We also proposed an optimization scheme which allows increasing the number of hit cache and tends to improve the running time.

Acknowledgements

This work has been funded by CONICET, Argentina.

References

1. **Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.W., Ryu, S., Steele Jr., G.L., & Tobin-Hochstadt, S. (2007).** *The Fortress Language Specification, version 1.0 beta*, Sun Microsystems, Inc.
2. **Adjeroh, D., Bell, T., & Mukherjee, A. (2008).** *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. New York: Springer.
3. **Burrows, M. & Wheeler, D.J. (1994).** *A block-sorting lossless data compression algorithm* (Research Report 124), Palo Alto California: Digital Systems Research Center.
4. **Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., & Sarkar, V. (2005).** X10: an object oriented approach to non-uniform cluster computing. *20th annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA '05).* San Diego, CA, 519–538.
5. **Cormen, T.H., Leiserson, C.E., Rivest, R.L. & Stein, C. (2009).** *Introduction to Algorithms* (3rd ed.). Cambridge, Mass.: MIT Press.
6. Ferragina, P. & Navarro, G. (s.f.). The Pizza&Chili corpus — compressed indexes and their testbeds. Retrieved from http://pizzachili.dcc.uchile.cl/index.html.
7. The Portable Hardware Locality (hwloc). Retrieved from http://www. open-mpi.org/ projects/ hwloc/.
8. **Hennesy, J.L. & Patterson, D.A. (2007).** *Computer Architecture - A Quantitative Approach* (4th ed.). Amsterdam; Boston: Morgan Kaufmann.
9. **Patterson, D.A. & Hennesy, J.L. (2009).** *Computer Organization and Design, The Hardware/Software Interface* (4th edition). Burlington, MA: Morgan Kaufmann.
10. **Hager, G. & Wellein, G. (2011).** *Introduction to High Performance Computing for Scientists and Engineers*. Boca Raton, FL: CRC Press.
11. **Manber, U. & Myers, G. (1993).** Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5), 935–948.
12. OpenMP Application Program Interface - Version 3.1, (2011). Retrieved from http://www.openmp.org/mp-documents/OpenMP3.1.pdf.
13. **Reinders, J. (2007).** *Intel Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism*. Beijing; Sebastopol, CA: O'Reilly.
14. **Stoye, J. (2007).** Suffix tree construction in ram. *Encyclopedia of Algorithms* (925–928). New York; London: Springer.
15. **Tinetti, F.G., Martin, S.M. (2012).** Sequential Optimization and Shared and Distributed Memory Optimization in Clusters: N-BODY/Particle Simulation. *Parallel and Distributed Computing and Systems (PDCS 2012)*, Las Vegas, USA.
16. **Weiner, P. (1973).** Linear pattern matching algorithms*. IEEE Conference Record of 14th Annual Symposium on Switching and Automata Theory (SWAT'08),* 1–11.

Verónica Gil-Costa received her M.S. (2006) and Ph.D. (2009) in Computer Science both from Universidad Nacional de San Luis (UNSL), Argentina. She is currently a professor at the University of San Luis, an Assistant Researcher at the National Research Council (CONICET) of Argentina and a researcher of Yahoo! Labs Santiago (YLS). Her research interests are in the field of performance evaluation, similarity search and distributed computing.

Cesar Ochoa is an undergraduate student at the Universidad National de San Luis (UNSL), Argentina. He has worked at the Estirenos (ARCOR) Company since 2005.

Marcela Printista received her Ph.D. in Computer Science from the University of San Luis, Argentina, in 2004. She was co-dean of the College of Physics, Mathematics and Natural Sciences from 2007 to 2013. She has been an associate professor from to 1998. Her main research interests are parallel computation model, performance evaluation techniques and high performance simulation.

Article received on 01/02/2013; accepted on 30/07/2013.

.