High-Performance Priority Queues for Parallel Crawlers

Mauricio Marin[1] Rodrigo Paredes[1] Carolina Bonacic[2]

[1] Yahoo! Research Latin America, Santiago of Chile
[2] ArTeCS, Complutense University of Madrid, Spain
Contact-author: mmarin@yahoo-inc.com

ABSTRACT

Large scale data centers for crawlers are able to maintain a very large number of active http connections in order to download as fast as possible the usually huge number of web pages from given sections of the WWW. This generates a continuous stream of new URLs of documents to be downloaded and it is clear that the associated work-load can only be served efficiently with proper parallel computing techniques. The incoming new URLs have to be organized by a priority measure in order to download the most relevant documents first. Efficiently managing them along with other synchronization issues such as URLs downloaded by different processing nodes forming a cluster of computers are the matters of this paper. We propose efficient and scalable strategies which consider *intra*-node multi-core multi-threading on an *inter*-nodes distributed memory environment, including efficient use of secondary memory.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Search process*

General Terms

Algorithms, Performance

Keywords

Priority Queues, Parallel and Distributed Computing

1. INTRODUCTION

The basic architecture of a Web search engine is composed of (**i**) a *crawler* which is in charge of retrieving and storing the documents to be indexed, (**ii**) an *indexer* that builds an index data structure necessary to reduce the response time to user queries, and (**iii**) the search engine itself which performs query processing upon the index structure to quickly present users the top-k documents per query. A crawler is composed of a *scheduler* that decides which documents are to be retrieved first from the Web, and a large set of so-called *robots* which make http connections to Web servers to obtain the documents. The scheduler maintains an URLs priority queue to determine the order in which documents are retrieved from the Web. Priority is defined by a composition of various importance metrics for Web sites.

Current search engines perform centralized crawling using a number of clusters of computers composed of several thousand processors or processing nodes, all of them organized to focus on different segments of the Web. These systems are fully asynchronous and their large sizes are explained by the huge size and highly dynamic nature of the Web and the requirement of retrieving web documents in the least possible time. Retrieving a given document i from a web site takes a latency time ℓ_i and a transfer time $t_i(d)$ proportional to the document length d where both values ℓ_i and $t_i(d)$ are comparatively very large with respect to the values of response times of the cluster processors, inter-processors communication network, and the bandwidth seen from the data center to the WWW. That is, connection and document retrieval time is affected by the large latencies of the Internet and therefore the available bandwidth can only be exploited by increasing the number of concurrent robots. This in turn is possible by executing a large number of asynchronous threads running on a given set of processors. Thus at any time these robots are at different stages of execution and the whole system operates at a certain rate λ of finished documents per unit time (throughput).

After a document has been downloaded a number of operations take place on it. The main ones are extraction of links, text and relevant terms, together with collection of information for ranking and indexing purposes. The particular details are out of the scope of this paper as they depend on the method of crawling and indexing/ranking used by the search engine. The relevant fact for this paper is that the fully asynchronous activity of robots generates a constant stream of operations that compete for the use of resources such as processors, inter-processors communication hardware and disk arrays. The arrival time of these on-line jobs and the amount of work demanded on the resources are in general unpredictable.

Current cluster realizations of crawlers are implemented using the message passing approach to parallel computing. Robots are implemented using threads and communication is effected by point-to-point individual messages among processors. For example a feasible scheme is to have a pair (scheduler, set of r robots) in each processor and distribute links or pointers to document uniformly at random by using MD5 hashing on their URLs. Or even better is to distribute Web site domains onto processors rather than individual URLs. In this case the MD5 function maps a given URL onto the processor in charge of its particular Web site domain. A communication action is triggered by a robot when it discovers URLs that MD5 maps to other processors. This reduces communication significantly as most web pages of a given site are expected to point to pages in the same site.

An excellent discussion on parallel asynchronous crawlers can be found in [7] and in the references there mentioned. More recent work can be found in [4, 6, 8, 10].

This paper focuses on an entirely different problem whose efficient solution (to the best of our knowledge) has not been investigated so far (or at least no much technical details have been revealed apart from what one can infer from the typical, and in our opinion less efficient, standard multi-heavy-threaded asynchronous message passing methods of parallelization). We focus on the efficient implementation of an important component of parallel crawlers devised to run on cluster of computers. We believe our approach is novel to the field because of the unique type of hybrid parallelization method we promote and the data structures and parallel algorithms that have devised and specially tailored to our method of parallel computing.

In this paper we are specifically interested in obtaining the best performance of hardware in charge of processing URLs in order to efficiently feed up the pairs (scheduler, r-robots) distributed onto a set of P processors. We show that by properly decoupling the inherently asynchronous nature of the work performed by each pair (scheduler, r-robots) from the computations related to URL administration it is possible to achieve efficient and scalable performance. In particular we perform these computations in a bulk-synchronous manner that allows the following optimizations (which are the main contributions of the paper).

Inter-nodes parallelism: Overall parallel computations are performed in blocks of R URLs in each processor (R being an average value) and messages among them are also sent in blocks. This prevents concurrency control conflicts since grouped URLs are processed sequentially at each processor which reduces overheads significantly. The value of R is related to the rate at which Web pages are downloaded and processed by each pair (scheduler, r-robots). In practice $r \gg R$ because of the relative difference in the speed between both tasks, and in an actual setting the hardware is scaled down to achieve the proper average R with the less possible amount of resources.

Associated with each pair (scheduler, r-robots) there is a queue Q which contains the very next URLs to be downloaded and the bulk-synchronous processes feed up the queues Q in their respective processors at regular intervals. At the same time the new URLs discovered by the robots are stored in an input queue associated with each bulk-synchronous process. These processes pick up these URLs at the instant in which they store their R-sized packages in the queues Q. A fraction of these robot's URLs are sent to other processors and the remaining ones are stored locally in the processor. This sets the communication in both directions between the asynchronous and bulk-synchronous tasks.

Overall we need to ensure (with relatively small variations) that at any instant the whole set of $r \cdot P$ robots are downloading the most relevant URLs discovered until that moment in the Web sample known (stored) up to this point by the P processors. This defines the problem as a parallel priority queue one in which we extract the top-(rP) URLs, we download them, store the newly discovered URLs in the queue, then pick-up the next globally top-(rP) and so on. Internet and communication hardware latencies make unfeasible this ideal scenario. At the inter-nodes level we reduce this gap by being efficient in communication since the bulk-synchronism allows us to send messages in blocks among processors and the retrieval of the top-r in each processor can also be made in bulk. Our experiments with actual samples of the Web tell us that error is quite below 1% with respect to an optimal "communication instantaneous" priority queue.

We also provide an algorithm to control the amount of error incurred when processors work independently downloading a given amount of web pages without performing communication among them. Sending messages grouped into single blocks is more efficient than sending individual point to point messages. This is not only because of the cumulative overheads of individual messages but also because of the associated of cost of heavy-threads management (e.g. Possix threads) that it is necessary to perform in order to handle messages in an asynchronous manner. In our case the number of these heavy threads can be reduced to just one and replaced by light threads (as we explain below) administered by hardware as understood in multi-core processors programed with the openMP library.

Intra-node parallelism: In each cycle of the bulk synchronous parallel computations, each processor has to deal with a set of URLs to be extracted from its local priority queue and a set of URLs to be inserted in the queue, and yet another set of URLs to be sent to other processors. In this paper we propose two highly optimized data structures and algorithms to perform these tasks. Both strategies are friendly to secondary memory and multi-core parallel processing and can be used alternatively. One ensures logarithmic worst case whereas the another has better average performance. The key to efficient performance in these strategies comes from the fact that they work in chunks of R URLs rather than inserting/removing individual URLs from the data structure.

In addition, our bulk-synchronous scheme allows each processor to work independently from the others during fairly regular intervals of time. During each of these intervals each processor essentially executes an *insert-many*(M) operation followed by an *extract-top*(R) operation to store M newly discovered URLs and get the R ones with the highest priorities from the queue respectively. This bulk-synchronism enables us to achieve near-optimal inter-node parallelism in a multi-core processor supporting the efficient use of T light threads. Namely, the running time costs of the operations *insert-many*(M) and *extract-top*(R) operations can be reduced to a fraction very close to $1/T$ by keeping T independent queues and executing in parallel T *insert-many*(M/T) operations upon them and T operations *extract-top*(R/T) performed also in parallel. When all data fits into main memory our experiments show that this approach achieves at least 95% of the $1/T$ optimal. The same sort of performance is expected in processors containing arrays of disks where each thread can write/read different sections of the array in parallel.

Our data structures are also efficient in secondary memory because they work with blocks of URLs. For example, one of the data structures works with chunks of size R each of which is stored in contiguous blocks of disk and it ensures at most a logarithmic number of disk accesses per operation.

2. PARALLEL CRAWLING

We first describe the overall process of performing crawling in parallel on a cluster of computers. We simplify the discussion by assuming the use of a standard priority queue per processor and assuming that the Web graph is distributed on the processors by websites. We also assume that the asynchronous and synchronous parts of the parallel crawler are both hosted by the same set of processors (which may not be the case in a production system). Each part lives as an independent set of threads in each processor.

The cluster has P processors where each one maintains a priority queue storing the URLs of the web-sites assigned to the respective processor. Each URL is assigned a priority value which depends of the application (in our experiments we use OPIC [1]). Each processor maintains a scheduler and r robots. The asynchronous component of the crawler is composed of a set of Possix threads. The main one of these threads in each processor executes the tasks of the scheduler and there are r additional threads for executing the robots (one thread per robot).

The overall process of crawling works in cycles composed of three main steps:

1. Each scheduler performs $R \leq r$ operations *extract-min* on its priority queue to obtain the R URLs with the highest priority and assigns them to the $R \leq r$ robots waiting for work to do.

2. Each robot downloads the document associated with the URL and may buffer new URLs obtained from the links in the document.

3. Once a sufficient number of robots have finished their tasks[1], each processor performs an *insert* operation onto its priority queue to insert the URLs stored in the buffer. This for the URLs belonging to the web-sites assigned to the processor. The URLs that belongs to sites assigned to other processors are packed together into a single message for each processor and sent to their respective destinations.

The processes executing the synchronous machine are treated as a bulk-synchronous parallel (BSP) computer [15]. In BSP the parallel computation is organized as a sequence of *supersteps*. During a superstep, the processors may perform computations on local data and/or send messages to other processors. The messages are available for processing at their destinations by the next superstep, and each superstep is ended with the barrier synchronization of the processors. The underlying communication library ensures that all messages are available at their destinations before starting the next superstep.

The main BSP thread of the crawler performs an infinite loop where in each cycle it executes a sequence of operations given by the functions *receiveMessages*(), *run*(), *sendMessages*() and *bsp_sync*() which indicates the synchronization point and delivery of messages. The function *run*() is in charge of processing all messages. In particular messages sent to the main thread by the robots. Upon reception of a message of type "URL_Retrieval" a new URL is extracted from the priority queue and the URL is assigned to an idle robot. Synchronization in this case is made by POSIX pthread variables and functions. When the *run*() function does not find idle robots it places the operation in a pending jobs queue Q. Robots that finish current jobs get new ones from this queue Q or if the queue is empty they sleep themselves on condition variables. Processing a document involves mainly (**i**) performing a parsing to extract links to other documents (which can produce messages to other processors), and (**ii**) extract and store the text and ranking information for the search engine. Messages are sent to the main thread of other processors to update their respective URL priority queues.

The crawler must be able to automatically determine the maximum number R_x of documents that are allowed to be downloaded before sending messages containing packages URLs to other processors. The average value of R_x can be determined *off-line* from a previous crawling of the same Web as follows. The downloaded sample can be represented as a graph where nodes are web pages and arcs links. For nodes we determine the pageRank value using the standard iterative algorithm [5]. A priority queue Q_x is created using the pageRank values of the nodes as priority values. The algorithm that we propose for this task performs the following steps:

1. Set the superstep counter S= 0, the array of counters for downloaded nodes $C[i]$= 0 and buffered messages $M[i]$= 0 for each processor i, and the set W to empty.

2. Get the next node n from the priority queue Q_x.

3. Set p = processor hosting site of node n and $C[p] = C[p] + 1$.

4. For each processor i check if any $M[i] > \beta$ where β is a tolerance parameter. If true, then set $S = S + 1$ and $M[j] = 0$ for all processors j, calculate error (see below), and empty W.

5. For all processors i containing a site for one of the links k of node n, set $M[i] = M[i] + 1$ and insert k in the set W.

6. Repeat from step 2 until Q_x becomes empty.

The tolerance β indicates the number of messages that are retained in each processor before sending the URLs to their respective processors. This parameter can be set by performing a binary search through several executions of the algorithm with increasing values of β. The binary seach determines the largest β value that keeps the average error below a given threshold value. The error in each superstep is computed as the ratio A/B where A is the number of nodes that were extracted from Q_x and stored in W during the superstep, and B is the total number of nodes (documents ids) extracted from Q_x during the superstep. Notice that in the simulated parallel computer the nodes stored in W are not downloaded in the same superstep and thereby we

[1]In practice it is not necessary to wait for idle robots since the connection between the asynchronous and synchronous processes can be made by the queue Q discussed in Sec. 1.

signal them as erroneously missed. The value of R_x is calculated as $\max\{C\}/S$. The cost of this algorithms is similar to the cost of the pageRank algorithm. Notice that OPIC can also be used in this case which would speed up noticeably the execution of the whole process. A comparison of the effectiveness of this variant is out of scope in this paper.

3. PRIORITY QUEUES

3.1 A logarithmic worst case approach

Our first PQ is based on the idea of binary tournaments upon a Complete Binary Tree (CBT) [11]. Each item stored in the PQ consists of a priority value and an identifier. We associate each leaf of the CBT with one item, and use the internal nodes to maintain a continuous binary tournament among the items. A match, at internal node i, consists of determining the item with higher priority between the two children of i and writing the identifier of the winner in i. The tournament is made up of a set of matches played in every internal node located in every path from the leaves to the root. Every time we change the priority associated with a leaf k, the tournament is updated by performing matches along the unique path between k and the root of the tree. We call this last operation *update-cbt*. The operations *extract-top*, and *insert* are implemented using *update-cbt* as the basic primitive. This structure is different from the standard binary heap (BH) priority queue and we have shown it is more efficient than BH in several settings and applications [11].

A PQ with N items is implemented using: (**i**) an array CBT$[1 \ldots 2N-1]$ of integers to maintain results of matches among items, (**ii**) an array Prio$[1..N]$ of priority values, and (**iii**) an array Leaf$[1..N]$ of integers to map between items and leaves. A node at position k in the array CBT has its children at positions $2k$ and $2k+1$. The parent of a node at k is at position $\lfloor \frac{k}{2} \rfloor$. All internal nodes are stored between positions 1 and $N-1$ of the CBT. The highest priority in the PQ is given by Prio[CBT[1]], its identifier is i=CBT[1], and its associated leaf is at position k=Leaf[i] of the CBT. [Note that it is not necessary to explicitly maintain the leaves of the tree in the array CBT since by using simple integer arithmetic on the array Prio we can calculate the priority associated with a given leaf]. Finally, to enable a dynamic reusing of item identifiers in the PQ, the array Leaf is also used to maintain a single linked list of available item identifiers.

Deletions in the CBT are performed by removing the child with lower priority between the children of the parent of the rightmost leaf, and exchanging it with the target leaf to be deleted. On the other hand, insertions are performed by appending a new rightmost leaf and updating the CBT. This is done by expanding in two leaves the first leaf of the tree. The cost of every step of the *extract-top* and *insert* operations is constant except by the cost of the *update-cbt* operation which in the worst case is obviously $O(\lg N)$. This cost can be further reduced using parallel algorithms.

We maintain an independent CBT array for each one of the T multi-core threads held in *each* processor. One can think of it as a single global CBT stored in the processor but symmetrically partitioned into T slices. The *insert-many*(M) operation, executed as a sequence of *insert-many*(R) operations in each processor, distributes uniformly at random M/T URLs priorities (we call them *keys*) in each CBT. The *extract-top*(R) extracts R/T keys from each CBT

```
procedure insertion-update-cbt($i, \mathcal{S}, k$)
    $h$:= $\lfloor \lg k \rfloor$;
    for $j := 1$ to $h$ do $I_y[j]$:= CBT[$k$ div $2^{h-j+1}$];
    Build up array $D_y$ from $I_y$ without duplicates;
    for $j := 1$ to $|D_y|$ do
        $a$:= $D_y[j]$;
        $e$:= SELECT(Prio[$a$] $\cup$ $\mathcal{S}$, $n$);
        Prio[$a$]:= $\{ x \mid x \in$ (Prio[$a$] $\cup$ $\mathcal{S}$) and $x \geq e \}$;
        $\mathcal{S}$:= $\{ x \mid x \in$ (Prio[$a$] $\cup$ $\mathcal{S}$) and $x < e \}$;
    endfor
    Prio[$i$]:= $\mathcal{S}$;
end
```

Figure 1: Insertion update.

```
procedure extraction-update-cbt($k$)
    $h$:= $\lfloor \lg k \rfloor$;
    for $j := h$ downto $1$ do
        $a$:= $2\,(k$ div $2^{h-j+1})$;
        $b$:= $a+1$;
        $x$:= MIN(Prio[CBT[$a$]]);
        $y$:= MIN(Prio[CBT[$b$]]);
        if ( $x < y$ ) then swap($a, b$);
        CBT[$k$ div $2^{h-j+1}$]:= $a$;
        $e$:= SELECT(Prio[$a$] $\cup$ Prio[$b$], $n$);
        Prio[$a$]:= $\{ x \mid x \in$ (Prio[$a$] $\cup$ Prio[$b$]) and $x \geq e \}$;
        Prio[$b$]:= $\{ x \mid x \in$ (Prio[$a$] $\cup$ Prio[$b$]) and $x < e \}$;
    endfor
end
```

Figure 2: Extraction update.

to get the top-R priority keys.

Each node Prio[k] of the global CBT represents a set of R keys such that the identifier a at any given node Prio[a] of the global CBT, namely a = CBT[i], which is the winner against another node Prio[b], holds the invariant that its R keys are all of better priority than the corresponding R keys of node b. We use a *quick-sort* like SELECT operation (see next subsection) to redistribute the contents of global nodes Prio[i] associated with every item i located along the path from a leaf k to the root of the global CBT. The SELECT operation returns the R-th priority value from a set of $2R$ keys stored in two nodes of the global CBT. We also need to determine the minimum key in any global Prio[k] node.

The insertion of a set $\mathcal{S}$ of $n = R$ priorities associated with a new item i is made by transforming into an internal node with two leaf children the leaf located at position $N = m/n$ of the global CBT where m is the total number of keys. During an *insert* and after setting CBT[$2\,N$]=CBT[N], CBT[$2\,N+1$]=i, and Leaf[i]=$2\,N+1$ the operation *insertion-update-cbt*$(i, \mathcal{S},$ Leaf$[i])$ is executed as shown in Figure 1 (larger numerical values indicate higher priority values). The construction of arrays D_y and I_y, and the update of array Prio[k] takes time $O(\log N)$.

The extraction of the set $\mathcal{S}$ containing the $n = R$ highest priorities is performed as follows. Let us assume that k is the position of the leaf that holds the item which contains $\mathcal{S}$ and i is the new item selected to be stored in k. During an *extract-top*(n) and after setting Leaf[i]=k and CBT[k]=i, the *extraction-update-cbt*(k) operation executes the steps shown in Figure 2.

```
IQS (Set $A$, Index $idx$, Stack $S$)
    If $idx = S$.top() Then $S$.pop()
        Return $A[idx]$
    $pidx \leftarrow$ random[$idx$, $S$.top()$-1$]
    $pidx \leftarrow$ partition($A$, $A[pidx]$, $idx$, $S$.top()$-1$)
    $S$.push($pidx$)
    Return IQS($A$, $idx$, $S$)
```

Figure 3: IncrementalQuicksort.

For secondary memory management, the same strategy of using R-sized nodes can be applied. Here each node is stored in R/b blocks of contiguous blocks of disk of size b. To reduce the number of accesses to disk, a RAM cache of blocks of size R can be maintained using the LRU replacement heuristic.

3.2 An amortized cost approach

Let us now switch to the incremental sorting problem, which can be stated as follows: Given a set A of m numbers, output the elements of A from smallest to largest, so that the process can be stopped after k elements have been output, for any k that is unknown to the algorithm.

We will explain how to obtain the keys in increasing order. To obtain them in the reverse order it is enough with multiply each priority value by -1. So, when extracting, we multiply again by -1 in order to restore the priority (if it is necessary).

To output the k smallest elements, **IQS** [14] calls Quickselect [9] to find the smallest element of arrays $A[0, m-1]$, $A[1, m-1]$, $\ldots$, $A[k-1, m-1]$. This leaves the k smallest elements sorted in $A[0, k-1]$. **IQS** avoids the $O(kn)$ complexity by reusing the work across calls to Quickselect.

When we call Quickselect on $A[1, m-1]$, a decreasing sequence of pivots has already been used to partially sort A in the previous call on $A[0, m-1]$. Therefore, **IQS** stores these pivots within a stack S, as they are relevant for the next calls to Quickselect. To find the next minimum, we first check whether p, the top value in S, is the index of the element sought, in which case we pop it and return $A[p]$. Otherwise, because of previous partitionings, it holds that elements in $A[1, p-1]$ are smaller than all the rest, so we run Quickselect on that portion of the array, pushing new pivots into S. Figure 3 shows algorithm **IQS**, which solves the incremental solving problem in optimal $O(m + k \log k)$ optimal expected time.

By virtue of **IQS** invariant, we see the following structure in the array. If we read it from right to left, we start with a pivot and at its left side there is a chunk of elements smaller than it. Next, we have another pivot and another chunk, and so on, until we reach the last pivot and a last chunk.

This resembles a heap structure, as the elements in the array are semi-ordered. In the following, we exploit this property to implement a priority queue over an array processed with algorithm **IQS**. We call this **IQS**-based priority queue *Quickheap* (**QH**). From now on we briefly explain how to obtain a min-order quickheap. For further details, please refer to Paredes' PhD thesis [13]. To implement a quickheap we need the following structures:

1. An array *heap*, which we use to store the elements.

2. A stack S to store the positions of pivots partitioning *heap*. Recall that the bottom pivot index indicates the fictitious pivot ∞, and the top one the smallest pivot.

3. An integer idx to indicate the first cell of the quickheap. Note that it is not necessary to maintain a variable to indicate the last cell of the quickheap (the position of the fictitious pivot ∞), as we have this information in $S[0]$.

4. An integer *capacity* to indicate the size of *heap*. We can store up to *capacity* -1 elements in the quickheap (as we need a cell for the fictitious pivot ∞). Note that if we use *heap* as a circular array, we can handle arbitrarily long sequences of insertions and deletions as long as we maintain no more than *capacity*-1 elements simultaneously in the quickheap.

In the case of circular arrays, we have to take into account that an object whose position is *pos* is actually located in the cell *pos* mod *capacity* of the circular array *heap*.

We add elements at the tail of the quickheap (the cell *heap*[$S[0]$ mod *capacity*]), and perform min-extractions from the head of the quickheap (the cell *heap*[idx mod *capacity*]). So, the quickheap *slides* from left to right over the circular array *heap* as the operation progresses.

To construct a quickheap, we create the array *heap* of size *capacity* with no elements, and initialize both $S = \{0\}$ and $idx = 0$. The value of *capacity* must be sufficient to store simultaneously all the elements we need in the array.

We note that idx indicates the first cell used by the quickheap allocated over the array *heap*, and the pivots stored in S delimit chunks of semi-ordered elements. Thus, to find the minimum of the heap, it is enough to focus on the first chunk, that is, the chunk delimited by the cells idx and S.**top**$() - 1$. For this sake, we just call **IQS**$(heap, idx, S)$ and then return the element *heap*[idx]. However, in this case **IQS** does not pop the pivot on top of S. (Remember that an element whose position is *pos* is located at cell *pos* mod *capacity*, thus we have to slightly change algorithm **IQS** to manage the positions in the circular array.)

To extract the minimum, we first make sure that it is located in the cell *heap*[idx]. (Once again, in this case **IQS** does not pop the pivot on top of S.) Next, we increase idx and pop S. Finally, we return the element *heap*[$idx - 1$].

To insert a new element x into the quickheap we need to find the chunk where we can insert x in fulfillment of the pivot invariant. Thus, we need to create an empty cell within this chunk in the array *heap*. To do that, it is enough to move some pivots and elements to create an empty cell in the appropriate chunk. We first move the fictitious pivot, updating its position in S, without comparing it with the new element x, so we have a free cell in the last chunk. Next, we compare x with the pivot at cell $S[1]$. If the pivot is smaller than or equal to x we place x in the free place left by pivot $S[0]$. Otherwise, we move the first element at the right of pivot $S[1]$ to the free place left by pivot $S[0]$, and move the pivot $S[1]$ one place to the right, updating its position in S. We repeat the process with the pivot at $S[2]$, and so on until we find the place where x has to be inserted, or we reach the first chunk. Figure 4 shows the algorithm.

In order to cope with the crawling application, we modified the *extract-min* operation to support efficiently our *extract-top*(R). Insertions can be made key by key and the effect is by design the same than inserting a set of n in bulk.

Since idx signals the first cell of the quickheap allocated on the array *heap* and pivots stored in S delimit chunks of

```
insert(Elem $x$)
  $pidx \leftarrow 0$ // moving pivots, starting from pivot $S[pidx]$
  While TRUE Do
    $heap[(S[pidx]+1) \bmod capacity] \leftarrow$
        $heap[S[pidx] \bmod capacity]$
    $S[pidx] \leftarrow S[pidx]+1$
    If $(|S| = pidx+1)$ OR
        $(heap[S[pidx+1] \bmod capacity] \leq x)$ Then
        $heap[(S[pidx]-1) \bmod capacity] \leftarrow x$
        Return // we found the chunk
    Else
        $heap[(S[pidx]-1) \bmod capacity] \leftarrow$
            $heap[(S[pidx+1]+1) \bmod capacity]$
        $pidx \leftarrow pidx+1$ // go to next chunk
```

Figure 4: Insertions on the quickheap.

```
extractR(int $R$)
  $finalPos \leftarrow idx+R-1$, $top \leftarrow S.\mathbf{top}()$
  While $finalPos \geq top$ Do
    While $idx \leq top$ Do Report $heap[idx], idx \leftarrow idx+1$
    $S.\mathbf{pop}()$, $top \leftarrow S.\mathbf{top}()$ // we consumed this chunk
  If $idx = finalPos+1$ Then Return // we are done
  // else, we have to find $finalPos$. We use quickselect and
  $first \leftarrow idx$, $last \leftarrow top-1$ // push on $S$ pivot positions
  While TRUE Do // greater than or equal to $finalPos$
    $pidx \leftarrow \mathbf{random}[first, last]$
    $pidx \leftarrow \mathbf{partition}(heap, heap[pidx], first, last)$
    If $pidx < finalPos$ Then $first \leftarrow pidx+1$
    Else
      $S.\mathbf{push}(pidx)$
      If $pidx = finalPos$ Then $top = pidx$, Break
      Else $last \leftarrow pidx-1$
  While $idx \leq top$ Do Report $heap[idx], idx \leftarrow idx+1$
  $S.\mathbf{pop}()$ // we have consumed this chunk
```

Figure 5: Extraction of R minima.

semi-ordered elements, we can perform a multi extraction of R minima following this algorithm, depicted in Figure 5. We compute the position $finalPos \leftarrow idx+R-1$, which is the R-th cell of the current extraction of R-elements. Next, we traverse the stack S. All the pivots placed before $finalPos$ belong to the R-element set of minima we have to answer, so we simply report all of them and their respective chunks of elements (at their left), and update idx accordingly. If idx reaches cell $finalPos+1$ we are done. Otherwise, using the classic quickselect algorithm [9] we look for the $finalPos$-th element in the chunk delimited by idx and $S.\mathbf{top}()-1$. All the pivots in positions greater than or equal to $finalPos$ are pushed in stack S. Finally, as the pivot on top of S is $finalPos$ we report all the elements from idx to $finalPos$ and extract the pivot on top of S.

Notice that Quickheaps exhibit a local access pattern, which makes them excellent candidates to reside on secondary memory. First, the stack S is small and accessed sequentially. Second, each pivot in S points to a position in the array *heap*. Array *heap* is only modified at those positions, and the positions themselves increase at most by one at each insertion. Third, **IQS** sequentially accesses the elements of the first chunk. Thus we can consider that our page replacement strategy keeps in main memory: (**i**) the stack S and integers *idx* and *capacity*; (**ii**) for each pivot in S, the disk block containing its current position in *heap*; and (**iii**) the longest possible prefix of $heap[idx, N]$.

4. EXPERIMENTAL RESULTS

We worked with two datasets that correspond to pages under the `.cl` (Chile) and `.gr` (Greek) top-level domains. We downloaded pages using the WIRE crawler [2, 3] in breadth-first mode, including both static and dynamic pages. While following links, we stopped at depth 5 for dynamic pages and 15 for static pages, and we downloaded up to 25,000 pages from each Web site. We use two samples of each domain taken during different months. Both datasets are comparable in terms of the number of Web pages, but there are wide differences in terms of geography, language, demographics, culture, etc..

The efficiency goal of the crawler is to retrieve the documents with the highest pageRank first [5]. However, the pageRank value of a document is unknown during the crawling as it has to be calculated considering the whole collection of documents. Instead crawlers can use heuristics to increase the probability of retrieving the best documents in terms of pageRank. A well-known heuristic is the so-called OPIC (*On-line Page Importance Computation*) [1]. In OPIC, the documents are given priority values as follows. The OPIC value of a given document i is given by

$$\text{opic}(i) = \sum_k \text{opic}(k) \,/\, \text{outDegree}(k)$$

where k are the documents that points to document i and *outDegree(k)* is the number of documents pointed to by document k. Initially the *home pages* are given OPIC values 1.0, and these values are spread out to all descendant documents. A given document can receive OPIC contributions from several pages which can belong to many other sites located in different processors.

4.1 Previous experiments

In [12] we have verified that the site distribution approach is very effective in reducing the total amount of communication among processors. In particular we have found that even in small Web samples as the ones we use in this paper the amount of available parallelism for robots can be huge which ensures a constant stream of R documents per unit time onto the bulk-synchronous crawler.

Figure 6 shows for each superstep the total number of "failures" that occurred by adding over all processors and taking average every 10 supersteps with R= 12,000 and P= 4, 8, 16 and 32 processors. We define "failures" as the number of times in which a processor receives a message asking to update an URL priority value for a document and the document has already been retrieved by the respective robot. Such a message can only increase the priority of the document and thereby this does not necessarily implies an error in the sequence of retrieved documents with respect to a sequential crawler. However, the timely arrival of such message could have put the document above other documents that were retrieved before it. The results show that the number of failures is larger for larger P values at earlier supersteps. This is because for smaller P values the

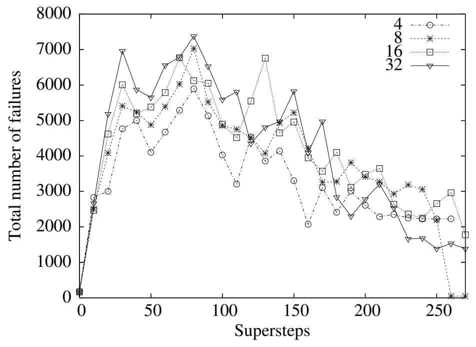

Figure 6: The effect of late arrival of messages for R= 12,000 and P= 4, 8, 16 and 32 processors.

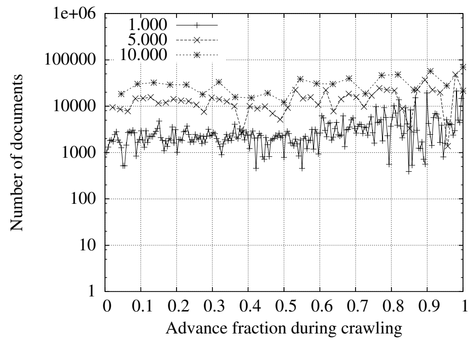

Figure 7: Predicting the number of crawled documents per processor per superstep for an error below 5% using 4 processors on a 2 millions pages sample of the CL web.

processors keep a larger number of sites. Nevertheless results obtained with an equivalent sequential crawler shows that failures are much more frequent in sequential crawling. Failures caused for messages from other processors are below 10% of the failures caused by URL updates belonging to co-resident sites. Thus this effect is negligible.

From a Web sample crawled just before the current crawling it is possible to predict the size R of the batch of documents that can be downloaded asynchronously. In Figure 7 we show results for the prediction algorithm proposed in section 2 for tolerances of 1,000 to 10,000 messages per processor for P= 4. These results were obtained for an error below %5 and show predictions fairly similar to R values set manually to reduce the error. In particular, the average values for tolerance of 1,000 (which leads to errors quite below %1) are similar to the values R/P used in those experiments.

4.2 Sequential experiments

To show the advantage of working with chunks of URLs in each priority queue rather than individual URLs, we present experiments that compare our queues with the standard binary heap (BH) approach. Figure 8 shows results that compare BH with our quickheap approach (QH). The performance metric in this case is the number of comparisons among keys. These results show that the QH outperforms BH for a wide range as R scales up. The same trend is observed in the I/O disk operations performed by the QH as R grows. See Figure 9. In this case the BH strategy cannot compete since performs too many random accesses to disk. The same trend is observed for the different Web samples we used in our experiments. We obtained similar performance with the CBT based queue though it is less efficient than the QH strategy with respect to disk access operations (about 20% on the average).

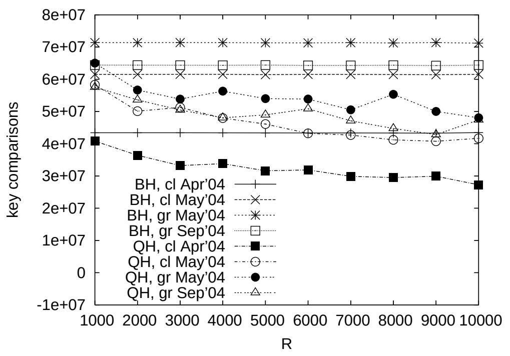

Figure 8: Number of key comparisons for different web samples.

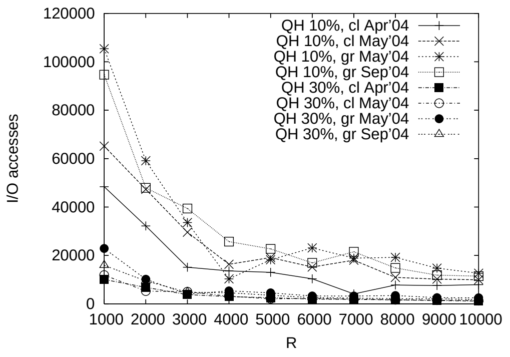

Figure 9: Number of I/O operations for different web samples, and a ram size which is 10 % and 30% of the queue size.

4.3 Multi-threading experiments

We made experiments using openMP as implemented in g++}version 4.1.2 and using T openMP threads on an *Intel's Quad-Xeon* multi-core processor with 8 CPUs. We run a benchmark program that repeatedly executed an *extract-top*(R/T) operation immediately followed by a corresponding *insert-many*(R/T) operation on the CBT queue. First we initialize each CBT with A/T keys where A is 80 millions.

In Figure 10 we report speed-ups results defined as the ratio running-time$(T = 1)$/running-time(T), namely the time with 1 thread to the time obtained with T threads. These results show that it is feasible to achieve near optimal performance for 8 threads which matches the number of CPUs. This is because the CBT queue is able to achieve very good

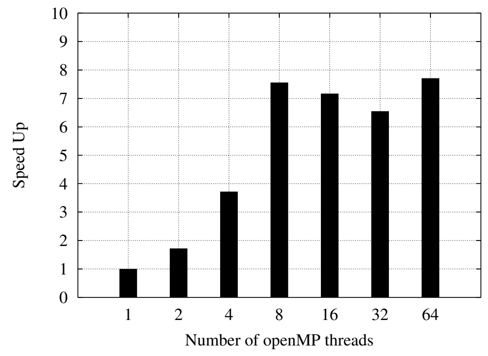

Figure 10: Speedups for T= 1, 2, 4, 8, 16, 32 and 64 light threads.

T	R/T	Extract	Insert
1	8,000	1.00	1.00
2	4,000	0.99	0.98
4	2,000	0.98	0.97
8	1,000	0.95	0.93

Table 1: Efficiencies of extract and insert operations.

load balance, namely on the average all the computations executed in each CBT by each thread are fairly similar. In table 1 we show a validation of this claim. In this table we show the efficiency defined as X/Y where X is the average amount of computations performed in each CBT and Y is the average maximum performed in any CBT. Optimal balance is achieved when efficiency is equal to 1, and the results of the table show values very close to 1 for both operations.

We could not achieve similar performance with the quickheap strategy since it presented high imbalance in the *extract-top*(R/T) operation due to the its amortized cost strategy. We plan in the near future to explore a different way to use the T threads on this data structure. Clearly the approach of using T independent quickheaps does not work properly in this case.

5. CONCLUSIONS

We have presented two alternative priority queues for handling the priorities of URLs in large scale parallel crawlers. The first one is based on a Complete Binary Tree in which nodes are associated with chunks of R priority values. This strategy achieves near optimal performance in multi-core processors since it presents an almost perfect load balance when inserting and extracting URLs. We believe this is the perfect choice when most of the queue can be kept in main memory. For instance in cases in which the total number of processors P is very large and the target section of the WWW to be crawled can be evenly distributed on the P processors. On the other hand, the second queue we propose in this paper is more suitable for cases in which secondary memory efficiency is of paramount importance. In any case, it is clear that using standard heaps to organize URLs can be very detrimental to performance.

Acknowledgements: This work has been partially funded by Yahoo! Research Latin America grant "Compact Data Structures" and AECI Project A/8065/07, Spain.

6. REFERENCES

[1] S. Abiteboul, M. Preda, and G. Cobena. Adaptive on-line page importance computation. In *2003 WWW Conf.*, pages 280–290. ACM Press, 2003.

[2] R. Baeza-Yates and C. Castillo. Balancing volume, quality and freshness in web crawling. In *Soft Computing Systems - Design, Management and Applications*, pages 565–572. IOS Press, 2002.

[3] R. Baeza-Yates, C. Castillo, M. Marin, and A. Rodriguez. Crawling a country: Better strategies than breadth-first for web page ordering. In *2005 WWW Conf.* ACM Press, 2005.

[4] P. Boldi, B. Codenotti, M. Santini, and V. Sebastiano. Ubicrawler: a scalable fully distributed web crawler. *Software, Practice and Experience*, 34(8):711–726, 2004.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, 1998.

[6] D.H. Chau, S. Pandit, S. Wang, and C. Faloutsos. Parallel crawling for online social networks. In *2007 WWW Conf.*, pages 1283–1284. ACM Press, 2007.

[7] J. Cho and H. Garcia-Molina. Parallel crawlers. In *2002 WWW Conf.*, pages 124–135. ACM Press, 2002.

[8] S. Dong, X. Lu, L. Zhang, and K. He. An efficient parallel crawler in grid environment. In *Int. Conf. on Grid and Cooperative Computing*, LNCS 3032, pages 229–232. Springer, 2004.

[9] C. A. R. Hoare. Algorithm 65 (FIND). *Comm. of the ACM*, 4(7):321–322, 1961.

[10] B. Thau Loo, S. Krishnamurthy, and O. Cooper. Distributed Web crawling over DHTs. Technical Report UCB/CSD-04-1305, EECS Department, University of California, Berkeley, Feb 2004.

[11] M. Marin. Binary Tournaments and Priority Queues: PRAM and BSP. Technical Report PRG-TR-7-97, Oxford University, 1997.

[12] M. Marin and C. Bonacic. Bulk-Synchronous On-Line Crawling on Clusters of Computers. In *16th Euromicro Intl. Conf. on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 414–421. IEEE CS, 2008.

[13] R. Paredes. *Graphs for Metric Space Searching.* PhD thesis, Universidad de Chile, 2008. Advisor: G. Navarro. Tech Report TR/DCC-2008-10. Available at `www.dcc.uchile.cl/~raparede/publ/08PhDthesis.pdf`.

[14] R. Paredes and G. Navarro. Optimal incremental sorting. In *Proc. 8th Workshop on Algorithm Engineering and Experiments and 3rd Workshop on Analytic Algorithmics and Combinatorics*, pages 171–182. SIAM Press, 2006.

[15] L.G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33:103–111, 1990.