High-Performance Priority Queues for Parallel Crawlers

Introduction

- Overall search engine architecture:
  - Crawler
  - Indexer
  - Search engine (query processing)
- Crawler consists of scheduler and robots (that make http connections to download webpages)
- Distributed nature of asynchronous crawler architecture: several clusters of computers spread over multiple processors
- Robots are asynchronous threads running on given set of processors and compete for resources
- Communication between robots is via point-to-point individual messages among processors
  - Eg. Maintain (schedule, set of r robots) pair on each processor and distribute URLs by MD5 hash/domain
- Issue:
  - Exponential growth of web needs parallel crawling to efficiently collect/process volumes of data
  - Parallel URL process management requires efficient resource allocation and utilization across multiple machines/threads

- Solution outline:
  - A new, multi-level PQueue data structure to store URLs
  - Aim: efficiently feed up the (schedule, set of r robots) pairs distributed over P processors
  - Bulk-synchronous URL computations (communication)

  - Inter-node optimization
    - Overall parallel computations are performed in blocks of R URLs in each processor
    - Each pair has a queue of URLs to download next, regularly fed by bulk-synchronous communication
    - Queue facilitates the communication between asynchronous processes
    - Top rP links are downloaded at a time

  - Intra-node optimization
    - Bulk-synchronous extraction/addition of URLs to queue
    - 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
    - Algorithms proposed for these operations
    - Assuming that T queues are maintained in parallel: T insert-many and extract-many operations can be run at a time

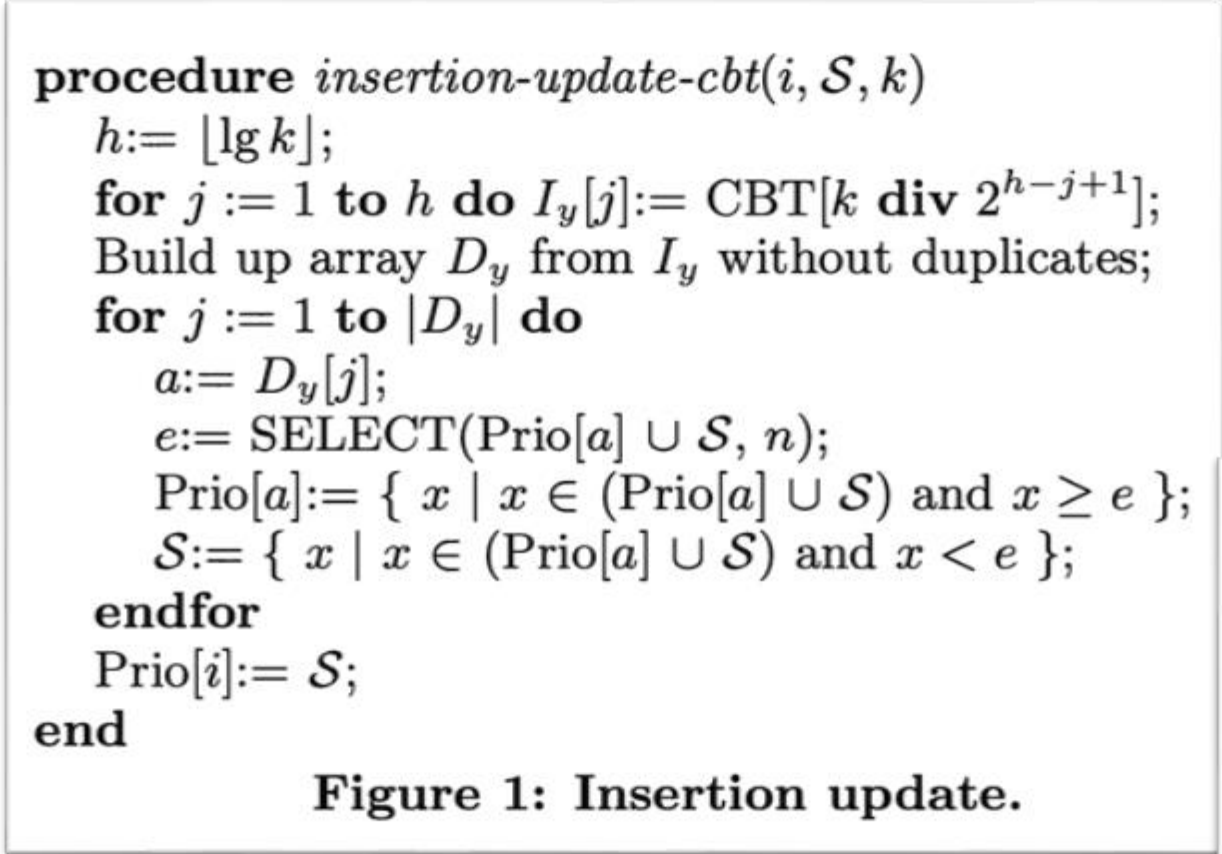
Parallel Crawling

- Overall parallel crawling process:
  - Cluster architecture setup: P processors, each with a priority queue, and a scheduler and r robots all on separate threads
- Operation:
  - Each process is treated as a bulk-synchronous parallel computer
  - Computation is performed in "supersteps"
    - Local processing/sending messages to other processors
    - Barrier synchronization of all processors
    - Messages are made available at their destination processors by underlying communication library
    - Main BSP (in crawler) runs in an infinite loop where each cycle uses functions receiveMessages(), run(), sendMessages() and bsp sync()
    - If idle robots are not found, exracted URLs are maintained in a pending jobs queue Q
    - Downloaded webpages represented by graph

Priority Queues: log worst case approach

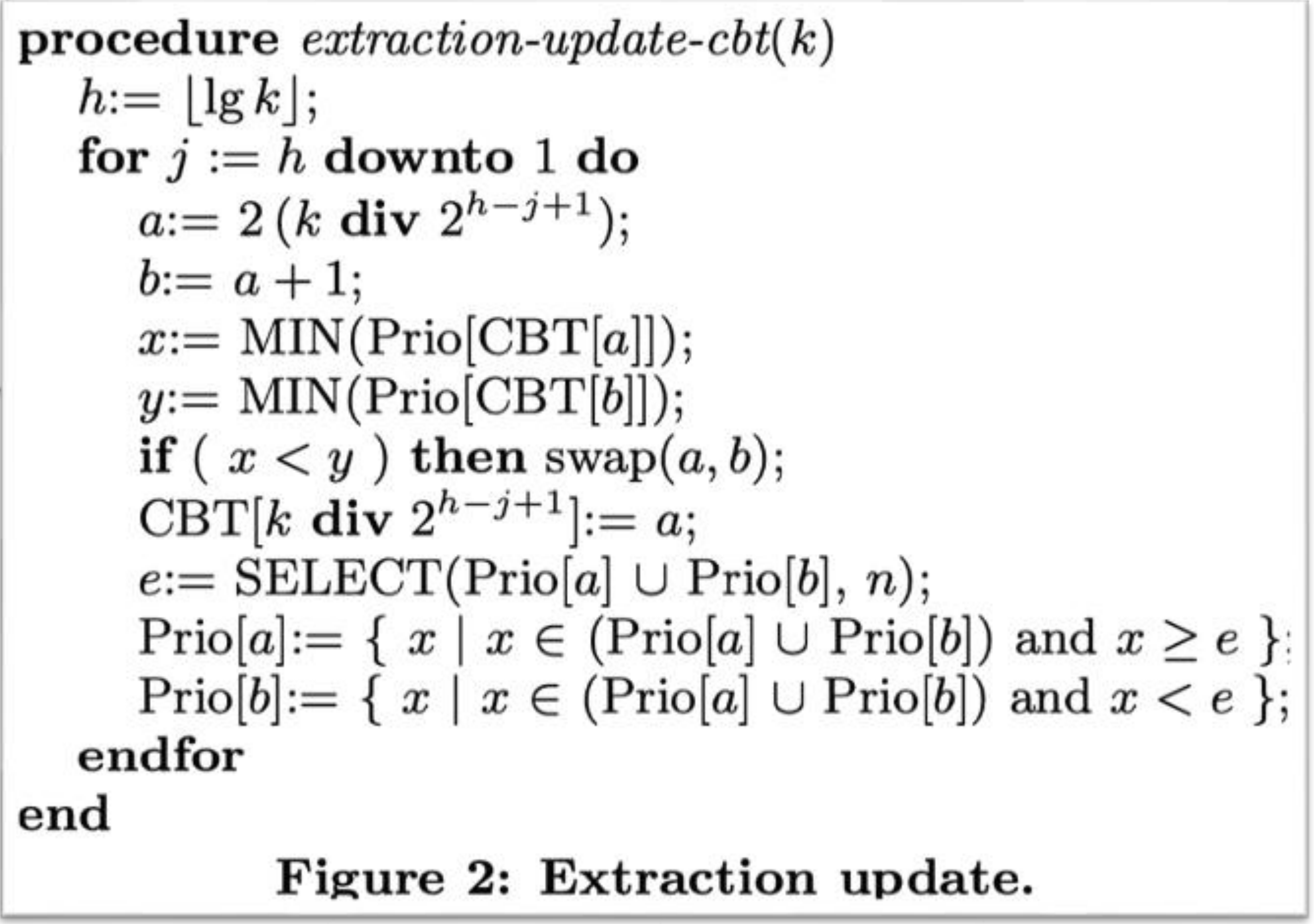
- Complete binary tree represents the PQueue (every item in the queue is a leaf node)
- Priorities are assigned by PageRank
- Internal nodes are used to maintain a continuous binary tournament to determine the item with higher priority at each step
- update_cbt() operation:
  - Leaf node k's priority is updated
  - Tournament is updated by performing matches along the unique path between k and the root of the tree

- PQueue is implemented as:
    - CBT[] of 2N nodes: maintains match results amongst nodes
    - Leaf[] of N nodes: map between items and leaves
    - Prio[] of N nodes: maintains priority values
- Highest priority (identifier i = CBT[1]) is maintained in Prio[i], and associated leaf position is Leaf[i] of the CBT
- Deletion: 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
- Insertion: appending a new rightmost leaf and updating the CBT by expanding in two leaves the first leaf of the tree
- Update-cbt worst case: O(logN)
- Near-perfect load balance while inserting and extracting URLs
- Suitable when PQueue is to be maintained in main memory



```
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.



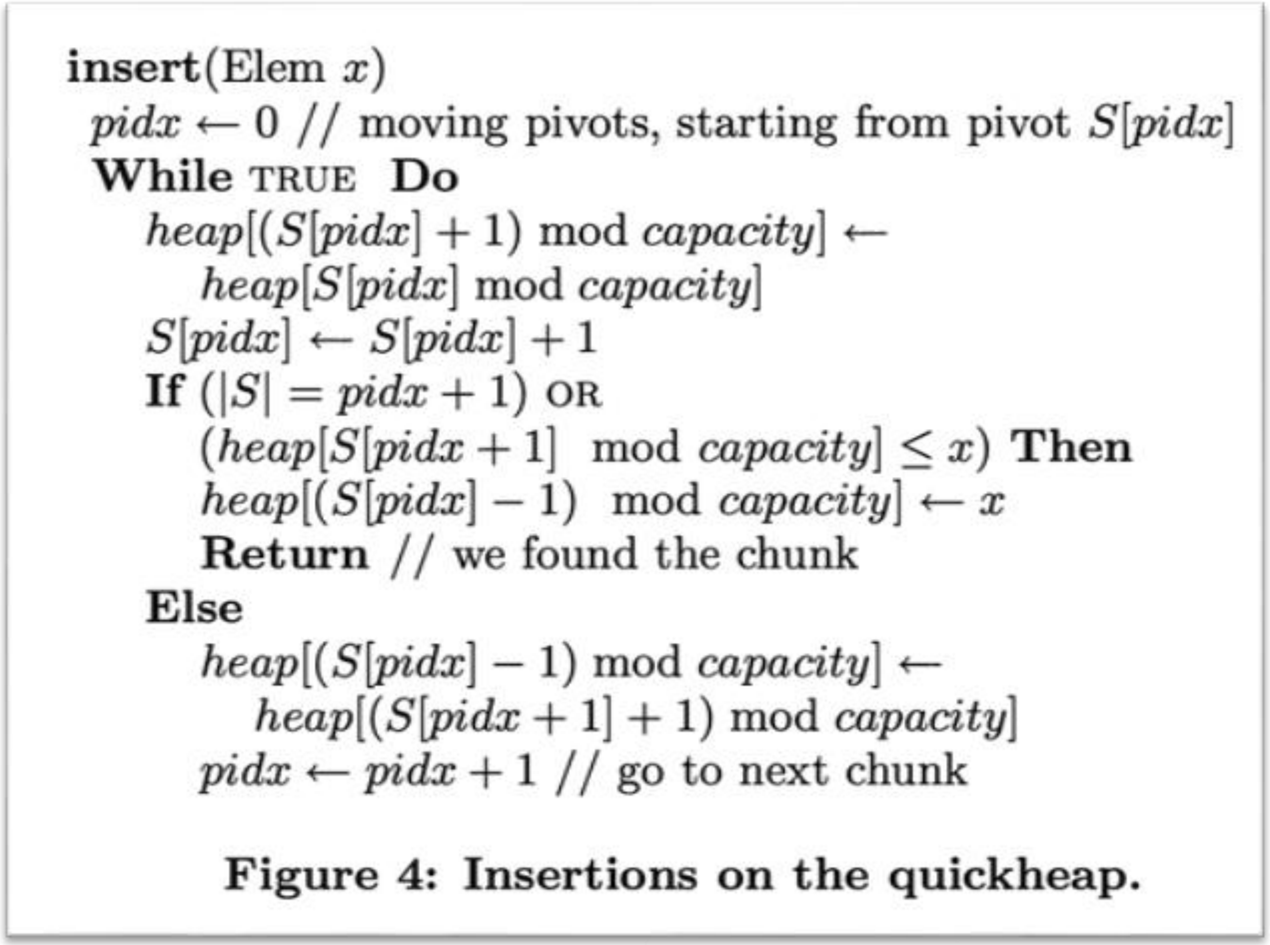
```
procedure extraction-update-cbt$(k)$
    $h := \lfloor \lg k \rfloor$;
    for $j := h$ downto $1$ do
        $a := 2\,(k$ div $2^{h-j+1})$;
        $b := a + 1$;
        $x :=$ MIN(Prio[CBT[$a$]]);
        $y :=$ MIN(Prio[CBT[$b$]]);
        if ( $x < y$ ) then swap($a, b$);
        CBT[$k$ div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

Priority Queues: amortized cost approach

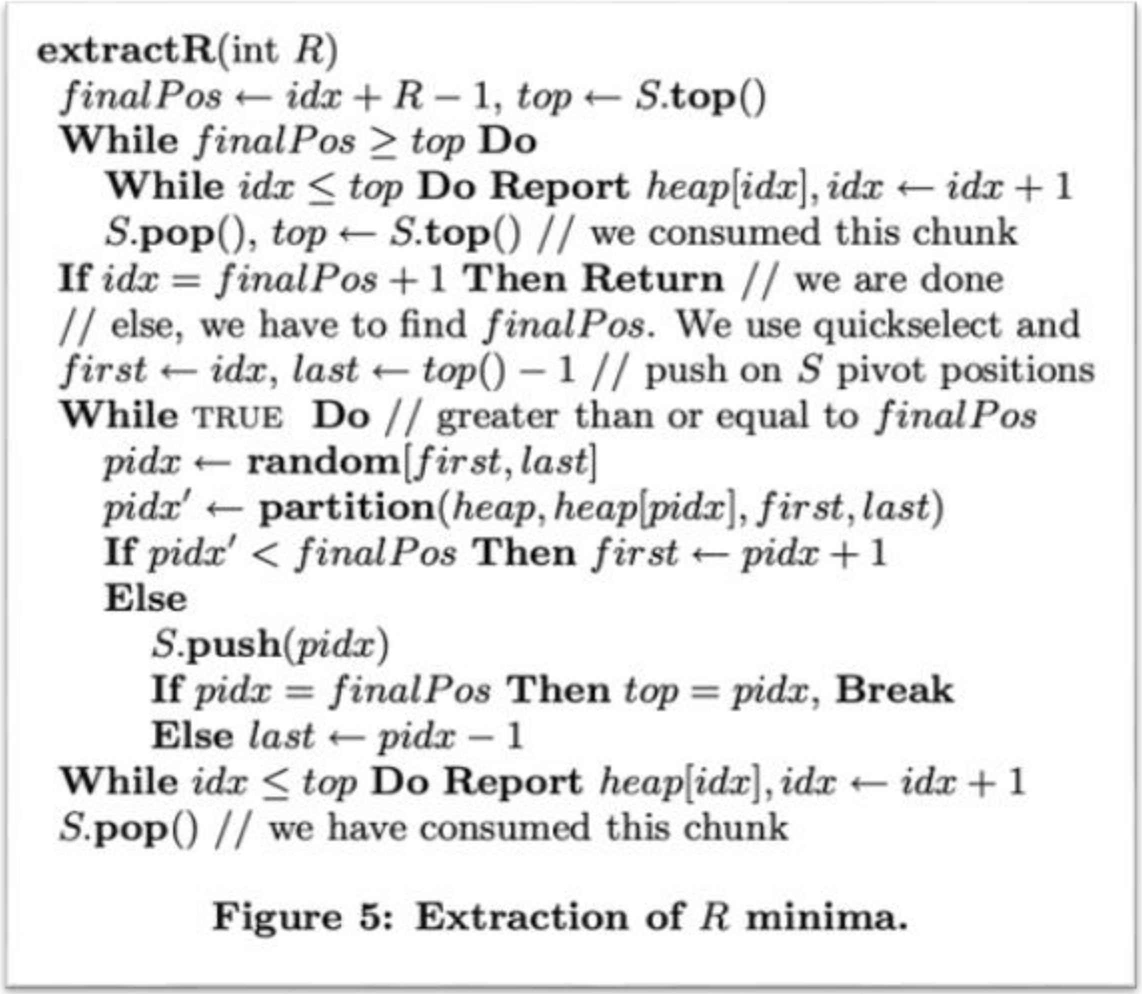
- Incremental sorting problem: Given a set A of m numbers, output the elements of A from smallest to largest, so that the process can be stopped after k elements have been output, for any k that is unknown to the algorithm.
- QuickSelect algorithm finds the smallest element of arrays A[0, m − 1], A[1,m−1], ..., A[k−1,m−1]
- This leaves the k smallest elements sorted in A[0, k − 1]:
  - O(kn) complexity avoided by reusing the work across calls to Quickselect
  - When QuickSelect is called on A[1, m-1], a sequence of pivots has already been used to partially sort A in the previous call on A[0, m − 1]
  - These pivots are stored in stack S
  - For next call: check if p (max value in S) is the index of sought minimum value:
    - Yes: pop and return A[p]
    - No: elements between A[1, p-1] are smaller than the rest (from previous partitioning), so run QuickSort on that array and push new pivots into S
  - Worst case: O(m+klogk)

- PQueue implemented over QuickHeap:
    - By QuickSelect, the array has the following structure [from right to left]: start with pivot, chunk of elements on left is smaller; reach another pivot, and so on
    - Resembles a semi-ordered heap structure
    - PQueue implemented over array processed with QuickSelect
- QuickHeap implementation:
    - Circular array heap to store all the elements
    - stack S to store the positions of pivots partitioning heap (top is the smallest pivot, bottom is the pivot corresponding to ∞)
    - integer idx to indicate the first cell of the QuickHeap
    - integer capacity to indicate the size of heap
- Construction: Elements added to tail (heap[S[0]%cap]) and extracted from head (heap[idx%cap])
- Top priority elements will be found in first chunk (heap cells between idx and S[top]-1)
- To insert a new element: compare with each pivot until the chunk it falls in is found and create a new element there
- Suitable when secondary memory efficiency is important



```
**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 $R$)
  $finalPos \leftarrow idx + R - 1$, $top \leftarrow S$.**top**()
  **While** $finalPos \geq top$ **Do**
    **While** $idx \leq top$ **Do Report** $heap[idx]$, $idx \leftarrow idx+1$
    $S$.**pop**(), $top \leftarrow S$.**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 $finalPos$
    $pidx \leftarrow$ **random**$[first, last]$
    $pidx' \leftarrow$ **partition**$(heap, heap[pidx], first, last)$
    **If** $pidx' < finalPos$ **Then** $first \leftarrow pidx+1$
    **Else**
      $S$.**push**$(pidx)$
      **If** $pidx = finalPos$ **Then** $top = pidx$, **Break**
      **Else** $last \leftarrow pidx-1$
  **While** $idx \leq top$ **Do Report** $heap[idx]$, $idx \leftarrow idx+1$
  $S$.**pop**() // we have consumed this chunk
```

Figure 5: Extraction of R minima.

Experimentation

- Advantage of working with chunks of URLs in each priority queue rather than individual URLs : comparing QuickHeap to Binary Heap implementation
- Performance metric:
  - number of key (URL priority) comparisons
- Analysis:
  - QHeap outperforms BHeap for wide range as R scales up

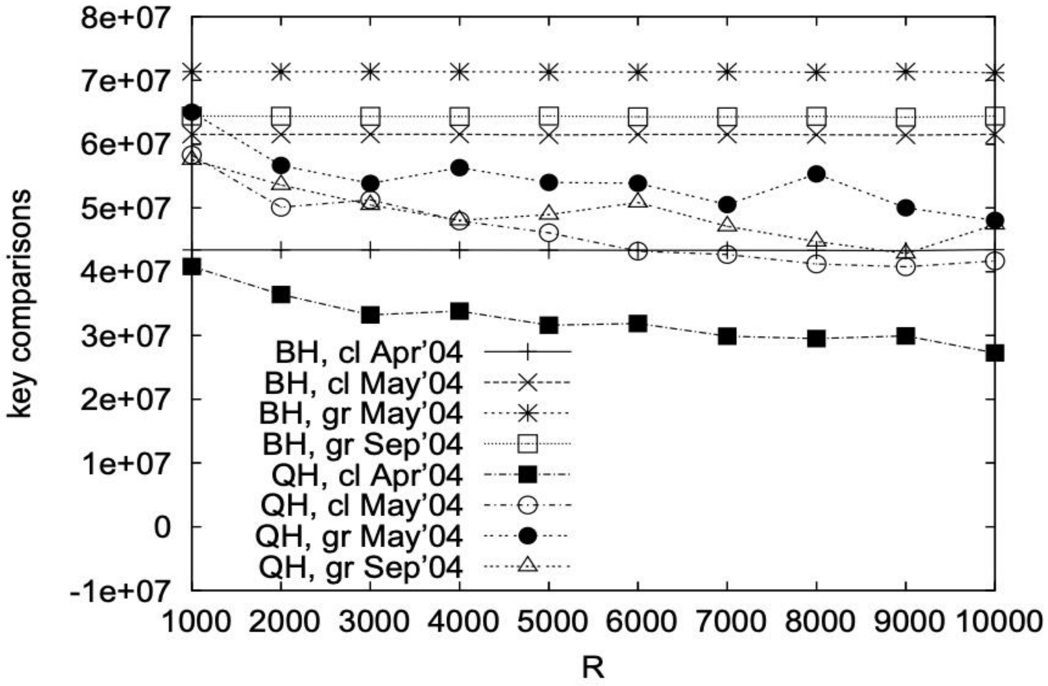


Figure 8: Number of key comparisons for different web samples.

- Number of I/O disk operations
  - BHeap has too many random disk accesses (so cannot compare)
  - QHeap performs well as R scales up
  - CBT has similar performance, but QHeap performs better than CBT for disk access operations (20% better)

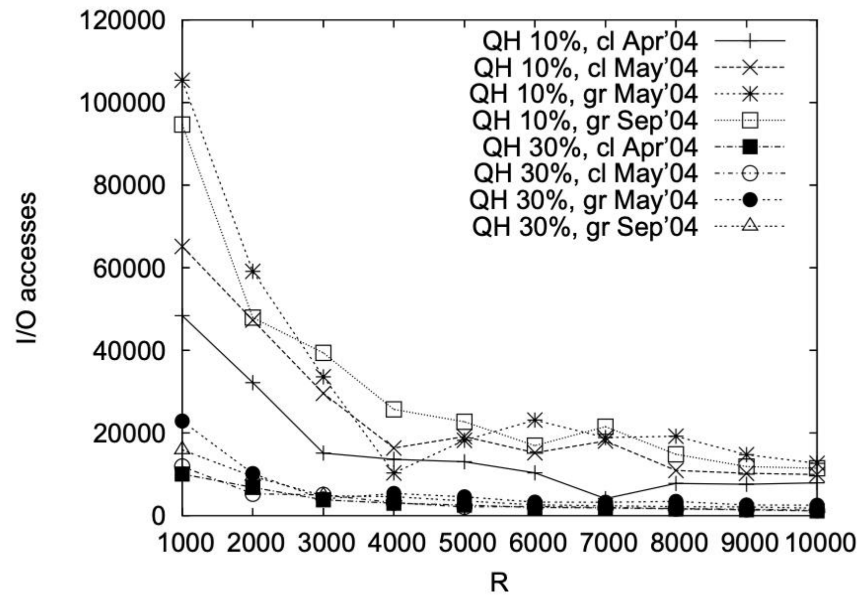


Figure 9: Number of I/O operations for different web samples, and a ram size which is 10 % and 30% of the queue size.

- Load balancing in CBT
  - Experimentation using T OpenMP threads in Intel's Quad-Xeon multi-core processor with 8 CPUs
  - Repeatedly executed an extract-top(R/T) operation immediately followed by a corresponding insert-many(R/T) operation on CBT queue

  - Speed up = running-time(T = 1)/running-time(T), namely the time with 1 thread to the time obtained with T threads
  - Near-optimal speed up observed
  - CBT queue is able to achieve very good load balance, namely on average all computations executed in each CBT by each thread are fairly similar

  - Efficiency = 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
  - Table 1 results show values very close to 1 for both operations
  - Similar performance not possible for QuickHeap: high imbalance in the extract-top(R/T) operation due to the its amortized cost strategy

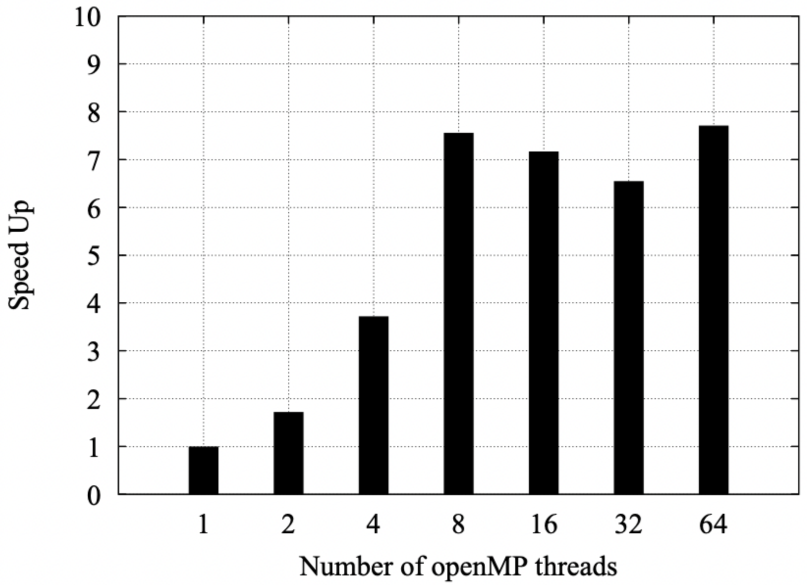
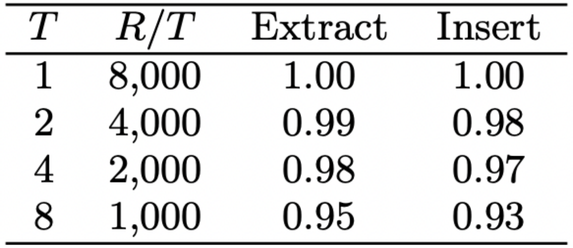


Figure 10: Speedups for T= 1, 2, 4, 8, 16, 32 and 64 light threads.



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.

Reference

- M. Marin, R. Paredes, and C. Bonacic. "High-performance priority queues for parallel crawlers." In Proceedings of the 10th ACM workshop on Web information and data management, pp. 47-54. 2008.