Phong Vo
6/24/2019

QUICKSORT ALGORITHM and APPLICATIONS

1. Introduction

This paper describes the properties and specs of Quicksort Algorithm such as its implementation, running time, utilized algorithms and applications. Quicksort is one of the best effective algorithmic functions in sorting. It was first introduced in 1959 by Tony Hoare then widely published in 1961.

Quicksort is a divide-and-conquer algorithm which divides the input array into two smaller size subarrays, then recursively sorts up those two. Quicksort has three typical steps: [1]

- Divide: the algorithm selects an element to be assigned as a pivot, divides the array into two subarrays which contain elements less and greater than pivot on the left and right subarrays, respectively.
- Conquer: recursively call for the quicksort function for those two subarrays.
- Combine: The whole array is now sorted because two subarrays are sorted. Thus, no work is needed.

Pseudo code for Quicksort: [1]

```
QUICKSORT(A, p, r)
    1  if p < r
    2      q = PARTITION(A, p, r)
    3      QUICKSORT(A, p, q – 1)
    4      QUICKSORT(A, q + 1, r)
PARTITION(A, p , r)
```

```
1   x = A[r]
2   i = p – 1
3   for j = p to r – 1
4       if A[j] ≤ x
5           i = i + 1
6           exchange A[i] with A[j]
7   exchange A[i + 1] with A[r]
8   return i + 1
```

Performance of quicksort:

- Best-case: when two subarrays have almost the same amount of elements, for example, one has $\left\lfloor \frac{n}{2} \right\rfloor$ and one has $\left\lceil \frac{n}{2} \right\rceil$ – 1. This case helps quicksort runs faster.

  The recurrence for the running time is $T(n) = 2T(\frac{n}{2}) + \Theta(n)$

  The solution is $T(n) = \Theta(n\,lg\,n)$.

- Average-case: The solution is $T(n) = \Theta(n\,lg\,n)$.

- Worst-case: when two subarrays are not balanced in partitioning where one array has no element and the other has (n – 1) elements.

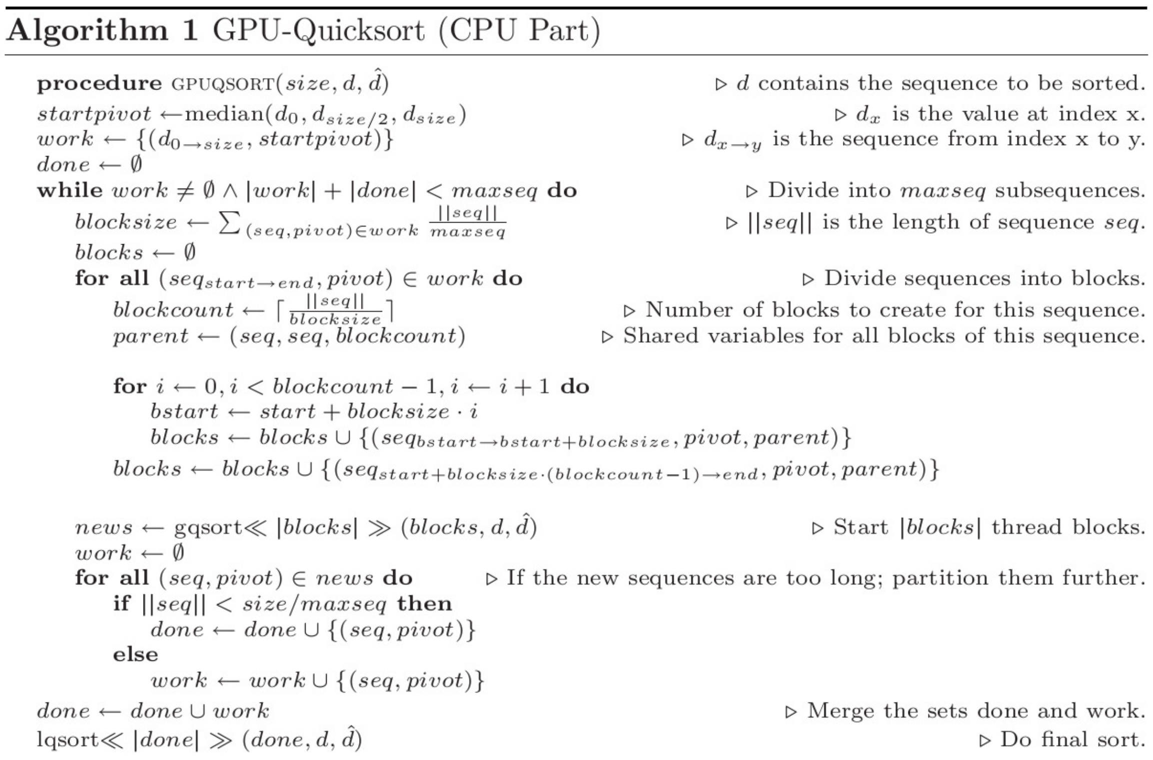
  The recurrence is $T(n) = T(n – 1) + \Theta(n)$, solution is $T(n) = \Theta(n^2)$.

  The worst-case can be prevented by using an enhanced algorithm named Randomized Quicksort. This algorithm picks a random element in the array to be assigned as the pivot. It usually can be the smallest or the largest element rather than the left-most or the right-most element of the original array [2].

2. Applications

(a) Graphic Processors Quicksort (GPU-Quicksort)

This application which is used “an efficient algorithmic parallel implementation of Quicksort, named GPU-Quicksort” [3]. The study shows the result that GPU-Quicksort can execute better than GPUSort and Radix sort thank to “taking advantage of the high bandwidth of GPUs by minimizing the amount of bookkeeping and inter-thread synchronization need” [3].



Algorithm 1 GPU-Quicksort (CPU Part)

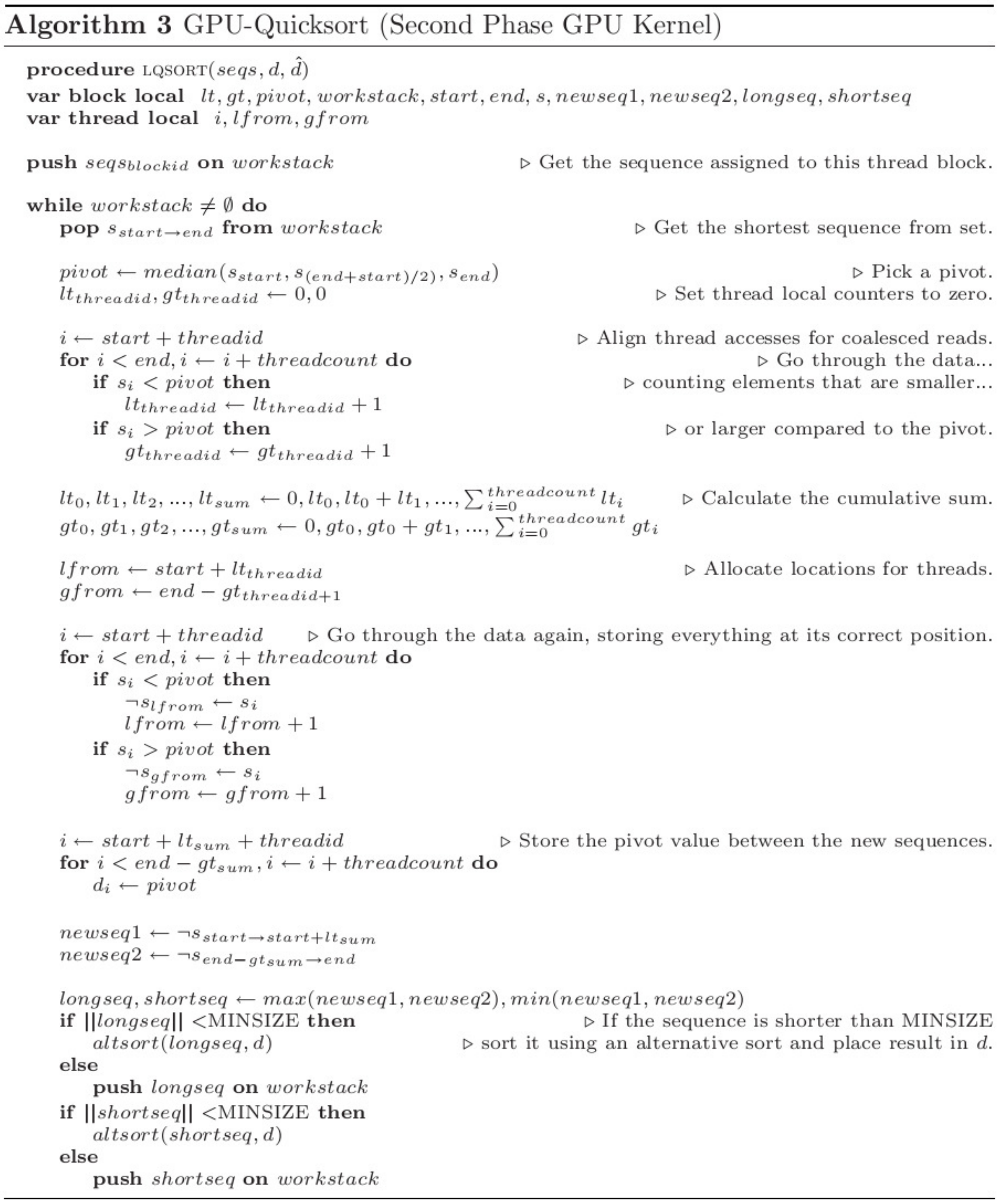
```
procedure GPUQSORT($size, d, \hat{d}$)                          ▷ $d$ contains the sequence to be sorted.
$startpivot \leftarrow$ median($d_0, d_{size/2}, d_{size}$)               ▷ $d_x$ is the value at index x.
$work \leftarrow \{(d_{0 \rightarrow size}, startpivot)\}$              ▷ $d_{x \rightarrow y}$ is the sequence from index x to y.
$done \leftarrow \emptyset$
while $work \neq \emptyset \wedge |work| + |done| < maxseq$ do         ▷ Divide into $maxseq$ subsequences.
    $blocksize \leftarrow \sum_{(seq,pivot) \in work} \frac{||seq||}{maxseq}$     ▷ $||seq||$ is the length of sequence $seq$.
    $blocks \leftarrow \emptyset$
    for all $(seq_{start \rightarrow end}, pivot) \in work$ do            ▷ Divide sequences into blocks.
        $blockcount \leftarrow \lceil \frac{||seq||}{blocksize} \rceil$       ▷ Number of blocks to create for this sequence.
        $parent \leftarrow (seq, seq, blockcount)$          ▷ Shared variables for all blocks of this sequence.

        for $i \leftarrow 0, i < blockcount - 1, i \leftarrow i + 1$ do
            $bstart \leftarrow start + blocksize \cdot i$
            $blocks \leftarrow blocks \cup \{(seq_{bstart \rightarrow bstart+blocksize}, pivot, parent)\}$
        $blocks \leftarrow blocks \cup \{(seq_{start+blocksize \cdot (blockcount-1) \rightarrow end}, pivot, parent)\}$

    $news \leftarrow$ gqsort$\ll |blocks| \gg (blocks, d, \hat{d})$          ▷ Start $|blocks|$ thread blocks.
    $work \leftarrow \emptyset$
    for all $(seq, pivot) \in news$ do      ▷ If the new sequences are too long; partition them further.
        if $||seq|| < size/maxseq$ then
            $done \leftarrow done \cup \{(seq, pivot)\}$
        else
            $work \leftarrow work \cup \{(seq, pivot)\}$
$done \leftarrow done \cup work$                                ▷ Merge the sets done and work.
lqsort$\ll |done| \gg (done, d, \hat{d})$                                     ▷ Do final sort.
```

Figure 2.1: GPU-Quicksort (CPU Part)

Algorithm 2 GPU-Quicksort (First Phase GPU Kernel)

```
function GQSORT($blocks, d, \hat{d}$)
var global $sstart, send, oldstart, oldend, blockcount$
var block local $lt, gt, pivot, start, end, s$
var thread local $i, lfrom, gfrom, lpivot, gpivot$

$(s_{start \to end}, pivot, parent) \leftarrow blocks_{blockid}$   ▷ Get the sequence block assigned to this thread block.
$lt_{threadid}, gt_{threadid} \leftarrow 0, 0$   ▷ Set thread local counters to zero.

$i \leftarrow start + threadid$   ▷ Align thread accesses for coalesced reads.
for $i < end, i \leftarrow i + threadcount$ do   ▷ Go through the data...
    if $s_i < pivot$ then   ▷ counting elements that are smaller...
        $lt_{threadid} \leftarrow lt_{threadid} + 1$
    if $s_i > pivot$ then   ▷ or larger compared to the pivot.
        $gt_{threadid} \leftarrow gt_{threadid} + 1$

$lt_0, lt_1, lt_2, ..., lt_{sum} \leftarrow 0, lt_0, lt_0 + lt_1, ..., \sum_{i=0}^{threadcount} lt_i$   ▷ Calculate the cumulative sum.
$gt_0, gt_1, gt_2, ..., gt_{sum} \leftarrow 0, gt_0, gt_0 + gt_1, ..., \sum_{i=0}^{threadcount} gt_i$

if $threadid = 0$ then   ▷ Allocate memory in the sequence this block is a part of.
    $(seq_{sstart \to send}, oseq_{oldstart \to oldend}, blockcount) \leftarrow parent$   ▷ Get shared variables.
    $lbeg \leftarrow FAA(sstart, lt_{sum})$   ▷ Atomic increment allocates memory to write to.
    $gbeg \leftarrow FAA(send, -gt_{sum}) - gt_{sum}$   ▷ Atomic is necessary since multiple blocks access this
variable.

$lfrom = lbeg + lt_{threadid}$
$gfrom = gbeg + gt_{threadid}$

$i \leftarrow start + threadid$
for $i < end, i \leftarrow i + threadcount$ do   ▷ Go through data again writing elements
    if $s_i < pivot$ then   ▷ to the their correct position.
        $\neg s_{lfrom} \leftarrow s_i$   ▷ If $s$ is a sequence in $d$, $\neg s$ denotes the corresponding
        $lfrom \leftarrow lfrom + 1$   ▷ sequence in $\hat{d}$ (and vice versa).
    if $s_i > pivot$ then
        $\neg s_{gfrom} \leftarrow s_i$
        $gfrom \leftarrow gfrom + 1$

if $threadid = 0$ then
    if $FAA(blockcount, -1) = 0$ then   ▷ Check if this is the last block in the sequence to finish.
        for $i \leftarrow sstart, i < send, i \leftarrow i + 1$ do   ▷ Fill in pivot value.
            $d_i \leftarrow pivot$
        $lpivot \leftarrow median(\neg seq_{oldstart}, \neg seq_{(oldstart+sstart)/2}, \neg seq_{sstart})$
        $gpivot \leftarrow median(\neg seq_{send}, \neg seq_{(send+oldend)/2}, \neg seq_{oldend})$
        result $\leftarrow$ result $\cup \{(\neg seq_{oldstart \to sstart}, lpivot)\}$
        result $\leftarrow$ result $\cup \{(\neg seq_{send \to oldend}, gpivot)\}$
```

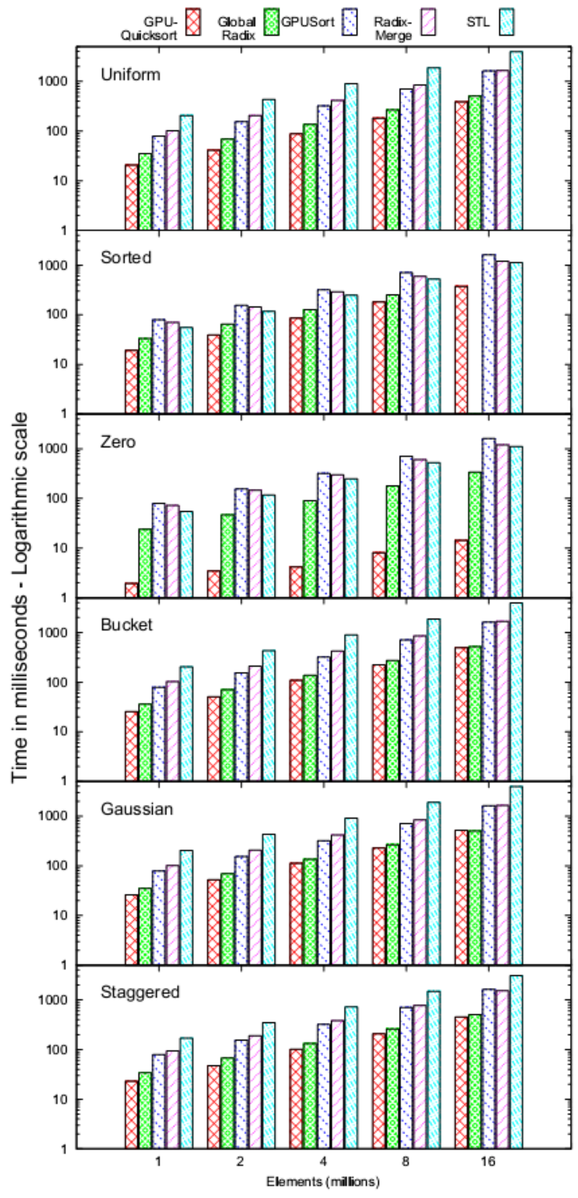
Figure 2.2: GPU-Quicksort (First Phase GPU Kernel)



Algorithm 3 GPU-Quicksort (Second Phase GPU Kernel)

```
procedure LQSORT($seqs, d, \hat{d}$)
var block local $lt, gt, pivot, workstack, start, end, s, newseq1, newseq2, longseq, shortseq$
var thread local $i, lfrom, gfrom$

push $seqs_{blockid}$ on $workstack$        ▷ Get the sequence assigned to this thread block.

while $workstack \neq \emptyset$ do
    pop $s_{start \to end}$ from $workstack$        ▷ Get the shortest sequence from set.

    $pivot \leftarrow median(s_{start}, s_{(end+start)/2}, s_{end})$        ▷ Pick a pivot.
    $lt_{threadid}, gt_{threadid} \leftarrow 0, 0$        ▷ Set thread local counters to zero.

    $i \leftarrow start + threadid$        ▷ Align thread accesses for coalesced reads.
    for $i < end, i \leftarrow i + threadcount$ do        ▷ Go through the data...
        if $s_i < pivot$ then        ▷ counting elements that are smaller...
            $lt_{threadid} \leftarrow lt_{threadid} + 1$
        if $s_i > pivot$ then        ▷ or larger compared to the pivot.
            $gt_{threadid} \leftarrow gt_{threadid} + 1$

    $lt_0, lt_1, lt_2, ..., lt_{sum} \leftarrow 0, lt_0, lt_0 + lt_1, ..., \sum_{i=0}^{threadcount} lt_i$        ▷ Calculate the cumulative sum.
    $gt_0, gt_1, gt_2, ..., gt_{sum} \leftarrow 0, gt_0, gt_0 + gt_1, ..., \sum_{i=0}^{threadcount} gt_i$

    $lfrom \leftarrow start + lt_{threadid}$        ▷ Allocate locations for threads.
    $gfrom \leftarrow end - gt_{threadid+1}$

    $i \leftarrow start + threadid$        ▷ Go through the data again, storing everything at its correct position.
    for $i < end, i \leftarrow i + threadcount$ do
        if $s_i < pivot$ then
            $\neg s_{lfrom} \leftarrow s_i$
            $lfrom \leftarrow lfrom + 1$
        if $s_i > pivot$ then
            $\neg s_{gfrom} \leftarrow s_i$
            $gfrom \leftarrow gfrom + 1$

    $i \leftarrow start + lt_{sum} + threadid$        ▷ Store the pivot value between the new sequences.
    for $i < end - gt_{sum}, i \leftarrow i + threadcount$ do
        $d_i \leftarrow pivot$

    $newseq1 \leftarrow \neg s_{start \to start + lt_{sum}}$
    $newseq2 \leftarrow \neg s_{end - gt_{sum} \to end}$

    $longseq, shortseq \leftarrow max(newseq1, newseq2), min(newseq1, newseq2)$
    if $||longseq||$ <MINSIZE then        ▷ If the sequence is shorter than MINSIZE
        $altsort(longseq, d)$        ▷ sort it using an alternative sort and place result in $d$.
    else
        push $longseq$ on $workstack$
    if $||shortseq||$ <MINSIZE then
        $altsort(shortseq, d)$
    else
        push $shortseq$ on $workstack$
```

Figure 2.3: GPU-Quicksort (Second Phase GPU Kernel)



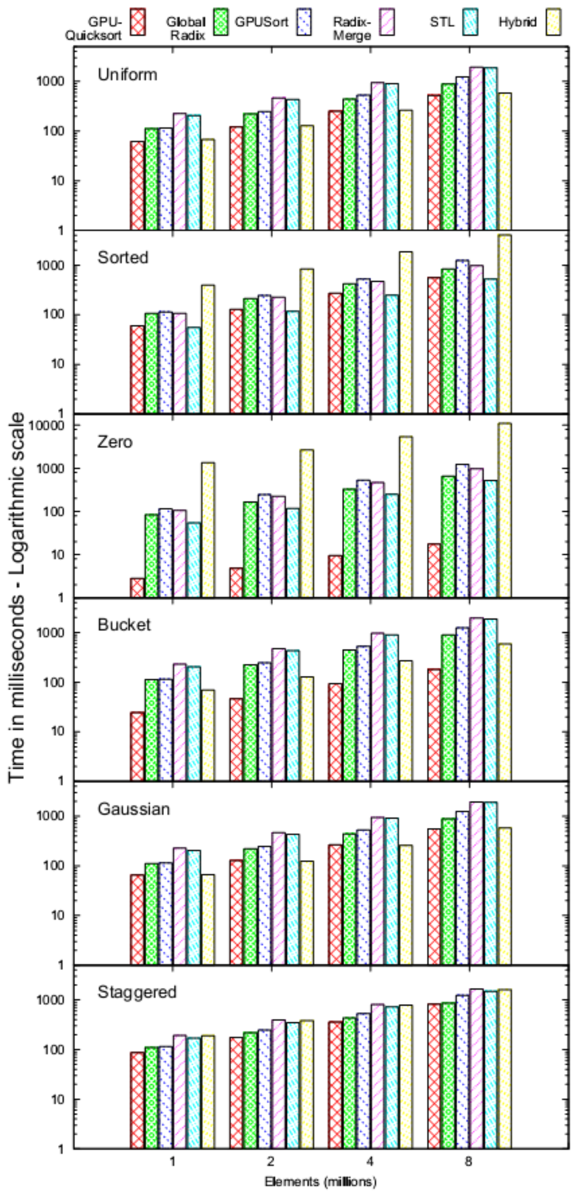


Figure 2.4: Results on the 8800GTX Figure 2.5: Results on the 8600GTS

Following the results showing in Figures 2.4 and 2.5, we can see that GPU-Quicksort performs much better than the conventional Quicksort algorithm does because it chooses the pivot among all elements which will be assigned as sorted elements. In addition, there are no extra steps required if the array is completely sorted [3].

(b) Enhanced Robust Index Model for Load Scheduling of a Home Energy Network

The goal of Quicksort in this research is "robust optimization method that considers the load shifting strategy" [4]. The Quicksort algorithm is used to "improve the robust level of appliances" [4] by making constraints of load optimization problems.

Algorithm 1 Quick Sort Algorithm for the Order of Loads

```
1: **Set** the desired objective function
2: **Initialize** the importance and deadline of appliances
3: **Quick Sort** (int $S[]$, int $i$ = first, $j$ = last){
        If ($i < j$){
            Int split = part($S$, $i$, $j$)
            Quicksort($S$, $i$, split-1)
            Quicksort($S$, split + 1, $j$)
        }
    }
4:**Return** results
```

Figure 2.6: Quicksort Algorithm of the Order of Loads

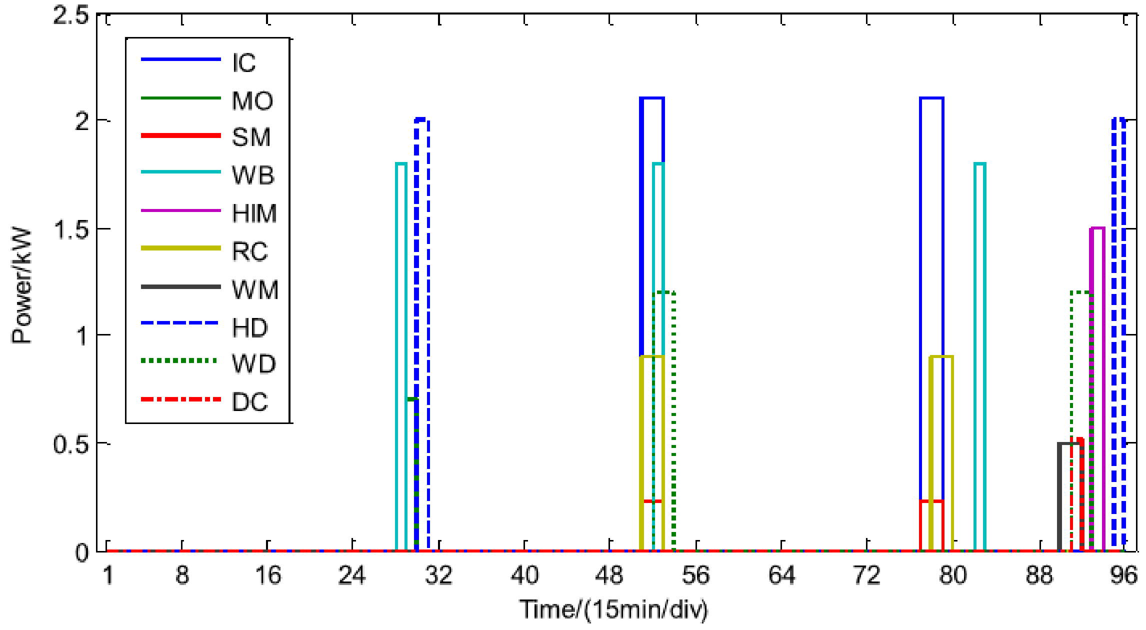


Figure 2.7: Load schedule with Orders of Loads Algorithm applied [4]

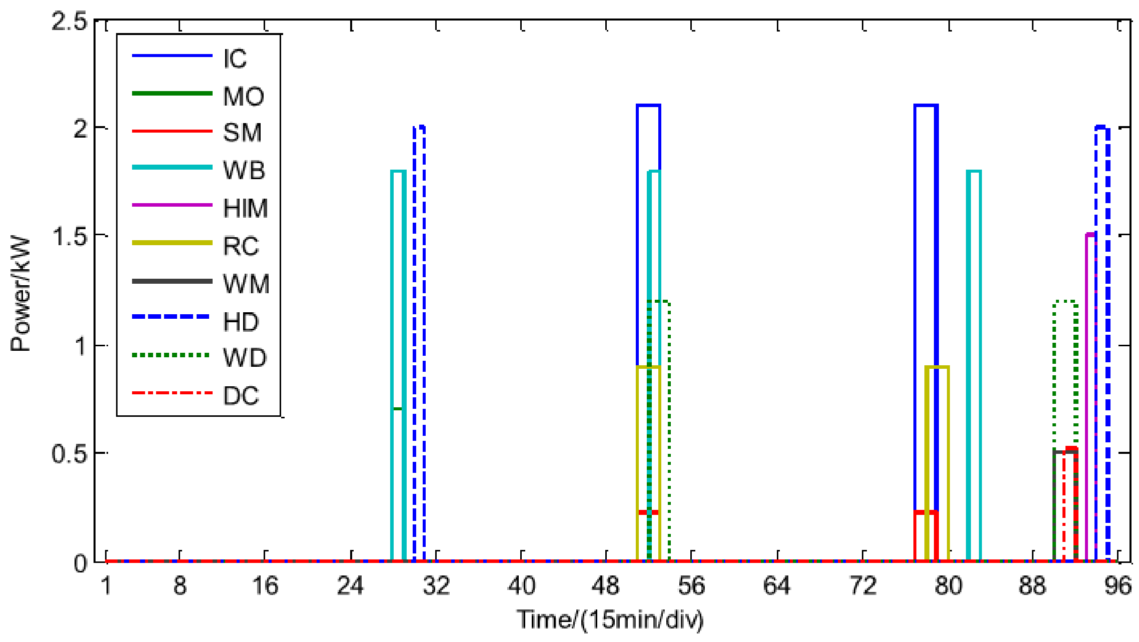


Figure 2.8: Load schedule with no Orders of Loads Algorithm applied [4]

The results of the research show that the energy consumptions of particular appliances with no Orders of Loads Algorithm applied are higher than the one by applying one, which thanks to the optimization of the Quicksort algorithm is utilized.

3. Conclusions

This piece of research paper is to gather and present the *pros* of Quicksort Algorithm when it is widely used in computer science and electrical technology. Quicksort is considered one of the best algorithms in hi-performance not only it does not require extra memory space, but it reduces the computing time for dedicated programs. Specially, GPU-Quicksort with assistance of Quicksort helps reducing the consuming of the system resources but keeps the computers' performance in appropriate levels.

4. References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein. “Introduction to Algorithms,” 3rd ed. Cambridge, pp.170-180, 2009.

[2] M. Ragab and U. Roesler. “The Quicksort process,” Stochastic Processes and their Applications, vol. 124, no. 2, pp. 1036–1054, Feb. 2014.

[3] D. Cederman and P. Tsigas, “GPU-Quicksort,” Journal of Experimental Algorithmics, vol. 14, p. 1.4, Dec. 2009.

[4] Ran, L. and Leng, S (2019). Enhanced Robust Index Model for Load Scheduling of a Home Energy Local Network with a Load Shifting Strategy. IEEE Access, vol. 7, pp. 19943-19953.