Journal of Advanced Computing Research Vol. 1, Issue 1 (2016) 39-41

Parallel Quicksort Algorithm Application

Fatin Ameera bt Alissak [#1], Mohamed Faidz Mohamed Said [#2]

[#] *Universiti Teknologi MARA*
70300 Seremban, Negeri Sembilan, MALAYSIA
[1] ameera.alfa@gmail.com
[2] faidzms@ieee.org

***Abstract*—Experienced algorithm designers depend vigorously on an arrangement of building blocks and on the tools expected to assemble the blocks into a calculation. This presentation outlines the general engineering of the parallel vector models; introduces two imperative classes of primitive instructions, the scan and segmented instructions; shows how the model and the instructions can be used to assemble part of a parallel quicksort calculation. The engineering of a V-RAM states that the machine is a random access machine with the expansion of a vector memory, a parallel vector processor, and a vector input/output port. Every area of the vector memory can contain a vector of various length. The parallel vector processor executes operations on entire vectors.**

***Keywords*: Parallel, Quicksort, Algorithm, PRAM**

I. Introduction

Experienced algorithm designers depend vigorously on an arrangement of building blocks and on the tools expected to assemble the blocks into a calculation. The understanding of these basic blocks and tools is thus basic to the understanding of algorithms. Many of the blocks and tools required for parallel algorithms reach out from successive algorithms, for example, dynamic-programming and divide-and-conquer. The researcher presents one of the easiest and most helpful building blocks for parallel algorithms the all-prefix sums operation. The researcher also characterizes the operation, demonstrates how to implement a P-RAM and shows numerous utilizations of the operation [2].

The researcher describes GPU-Quicksort, an efficient Quicksort calculation appropriate for exceptionally parallel multi-center graphics processors. Quicksort has already been viewed as an inefficient sorting answer for graphics processors, yet we demonstrate that in CUDA, NVIDIA's customizing stage for universally useful calculations on graphical processors, GPU-Quicksort performs superior to the speediest known sorting usage for graphics processors, for example, radix and bitonic sort. Quicksort can in this manner be seen as a reasonable elective for sorting vast amounts of information on graphics processors [3].

In most parallel random-access machine (P-RAM) models, memory references are expected to take unit time. In principle, certain scan operations, otherwise called prefix calculations, can execute in no additional time than these parallel memory references. Cederman and Tsigas [4] plot a broad investigation of the effect of incorporating into the P-RAM models, such scan operations as unit-time primitives. The study infers that the primitives make strides the asymptotic running time of many algorithms by 0 factor [5], significantly improve the portrayal of many algorithms, and are significantly less demanding to actualize than memory references. The algorithm designer should feel free to use these operations because they were as cheap as a memory reference.

II. Background

Blelloch and Maggs [5] recommend an adjustment in the basic models used for examining parallel calculations. In particular, it proposes that we move far from utilizing theoretical performance models based on machines to utilizing models based on languages. As said in the article, some reference works as of now casually break down parallel calculations as far as work and profundity before mapping them onto a PRAM.

Blelloch and Maggs [5] purpose that the additional progression be taken of formalizing a model based on work and profundity. With this formal model, the PRAM can be removed of the circle, straightforwardly mapping the model onto more reasonable machines. We moreover contend that language-based models appear to be the most sensible approach to characterize a programming model based on work and profundity.

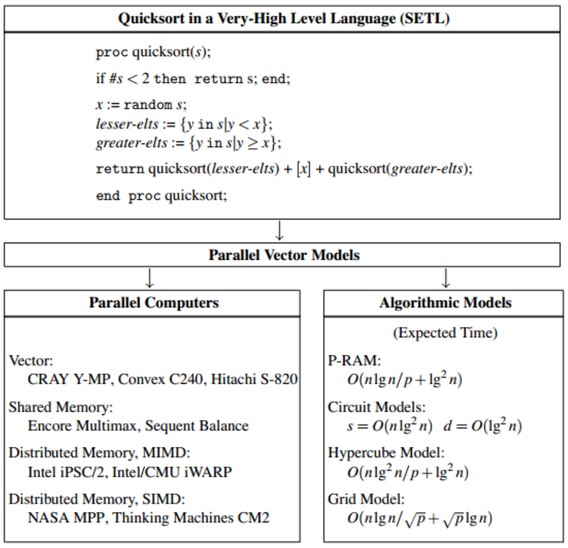
Cederman and Tsigas [6] depict a powerful parallel algorithmic execution of Quicksort, GPU-Quicksort, proposed to misuse the astoundingly parallel nature of configuration processors (GPUs) and their confined store memory. Quicksort has long been viewed as one of the speediest sorting calculations practically speaking for single processor frameworks and is additionally a standout amongst the most contemplated sorting calculations, however as of recently it has not been viewed as an efficient sorting answer for GPUs.

Cederman and Tsigas [6] demonstrate that GPU-Quicksort presents a reasonable sorting elective and that it can beat other GPU-based sorting calculations, for example, GPUSort and radix sort, considered by numerous to be two of the best GPU-sorting calculations. GPU-Quicksort is intended to exploit the high transfer speed of GPUs by minimizing the measure of accounting and between string synchronization required. It accomplishes this by utilizing a two-pass outline to keep the between string synchronization low, and mixing read operations and compelling strings so memory gets to are kept.

III. Methodology

This presentation plots the general building of the parallel vector models; presents two basic classes of primitive guidelines, the sweep and portioned directions; indicates how the model and the directions can be utilized to gather part of a parallel quicksort estimation [7].

A strange state, normally parallel, depiction of quicksort in the lingo SETL. The parameters are an arrangement of data keys. The structure s gives back the span of s, subjective s gives back a discretionarily chosen part of s, and annexes two successions. Ideally, we might want to make an interpretation of this depiction into capable code for a wide variety of designs, furthermore to choose the nature of the figuring on different hypothetical models. This book recommends that the parallel vector models are an OK premise on which to combine these objectives [7].


Quicksort in a Very-High Level Language (SETL)

```
proc quicksort(s);
if #s < 2 then return s; end;
x := random s;
lesser-elts := {y in s|y < x};
greater-elts := {y in s|y ≥ x};
return quicksort(lesser-elts) + [x] + quicksort(greater-elts);
end proc quicksort;
```

Parallel Vector Models

Parallel Computers

Vector:
CRAY Y-MP, Convex C240, Hitachi S-820
Shared Memory:
Encore Multimax, Sequent Balance
Distributed Memory, MIMD:
Intel iPSC/2, Intel/CMU iWARP
Distributed Memory, SIMD:
NASA MPP, Thinking Machines CM2

Algorithmic Models

(Expected Time)
P-RAM:
$O(n \lg n/p + \lg^2 n)$
Circuit Models:
$s = O(n \lg^2 n)$ $d = O(\lg^2 n)$
Hypercube Model:
$O(n \lg^2 n/p + \lg^2 n)$
Grid Model:
$O(n \lg n/\sqrt{p} + \sqrt{p} \lg n)$

Figure 1. [7]

Blelloch [7] shows the designing of a V-RAM and expresses the machine is an irregular access machine [1] with the extension of a vector memory, a parallel vector processor, and a vector information/yield port. Each zone of the vector memory can contain a vector of different length. The parallel vector processor executes operations on whole vector.

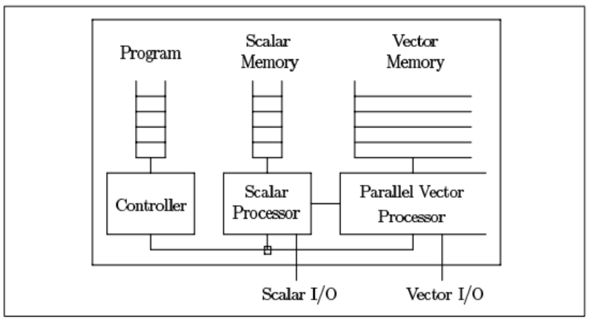


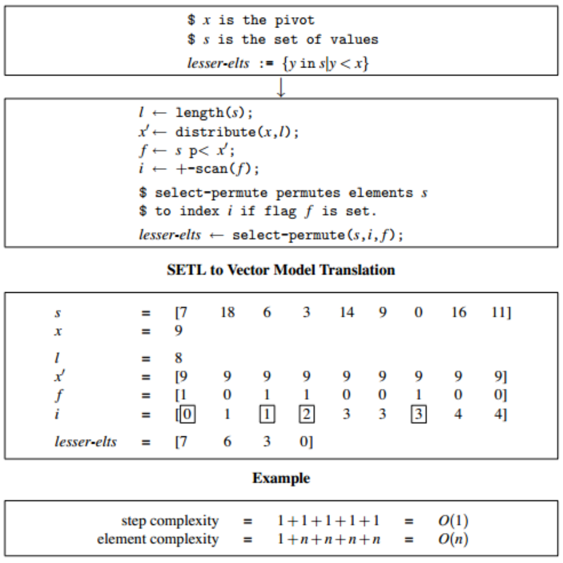
Figure 2. [7]

Essentially as with a discretionary access machine [1] model or the Turing machine model [8], the parallel vector models are accommodatingly described similarly as a machine outline, the vector RAM (V-RAM). The V-RAM is a standard serial RAM with the expansion of a vector memory and a vector processor as in Figure 2. Each memory region in the vector memory can contain a self-assertively long vector of nuclear qualities; the vector length is associated with the vector not the memory region. Each heading of the vector processor deals with a settled number of vectors from the vector memory and maybe scalars from the scalar memory. A vector guideline may, for instance, total the components of a vector, revamp the request of the components of a vector, or union the components of two sorted vectors as in Figure 3.

```
Mv[0]                           =  [3   17   7   13   5   1   11   2]
Ms[0] ← sum Mv[0]               =  59

Mv[0]  (data vector)            =  [c   o   g   n   i   t   i   o   n]
Mv[1]  (index vector)           =  [2   8   4   1   0   7   6   3   2]
Mv[2] ← permute Mv[0],Mv[1]     =  [i   n   c   o   g   n   i   t   o]

Mv[0]                           =  [4   7   11]
Mv[1]                           =  [2   9   12   21]
Mv[2] ← merge Mv[0],Mv[1]       =  [2   4   7   9   11   12   21]

Mv[0]                           =  [5   1   3   4   3   9   2   6]
Mv[1]                           =  [2   5   3   8   1   3   6   2]
Mv[2] ← p+ Mv[0],Mv[1]          =  [7   6   6   12  4   12  8   8]
```

Figure 3. [7]

The venture change anticipated that would change over the SETL code, which chooses components of s not exactly the turn x, into guidelines for a parallel vector machine ($ is the comment character in SETL). Likewise, an instance of the execution of the vector directions, besides, versatile nature of the vector model code.


```
$ x is the pivot
$ s is the set of values
lesser-elts := {y in s|y < x}
```

```
l ← length(s);
x′ ← distribute(x,l);
f ← s p< x′;
i ← +-scan(f);
$ select-permute permutes elements s
$ to index i if flag f is set.
lesser-elts ← select-permute(s,i,f);
```

SETL to Vector Model Translation

s	=	[7	18	6	3	14	9	0	16	11]
x	=	9								
l	=	8								
x'	=	[9	9	9	9	9	9	9	9	9]
f	=	[1	0	1	1	0	0	1	0	0]
i	=	[$\boxed{0}$	1	$\boxed{1}$	$\boxed{2}$	3	3	$\boxed{3}$	4	4]
lesser-elts	=	[7	6	3	0]					

Example

step complexity	=	$1+1+1+1+1$	=	$O(1)$
element complexity	=	$1+n+n+n+n$	=	$O(n)$

Complexity

Figure 4. [7]

The quicksort calculation just utilizing parallelism inside each square yields a stage diverse quality corresponding to the quantity of pieces O(n). Basically, utilizing parallelism from running the pieces in parallel yields a phase versatile quality in any occasion relative to the greatest square O(n). By using both types of parallelism the progression intricacy is proportional to the profundity of the tree expected O[9].

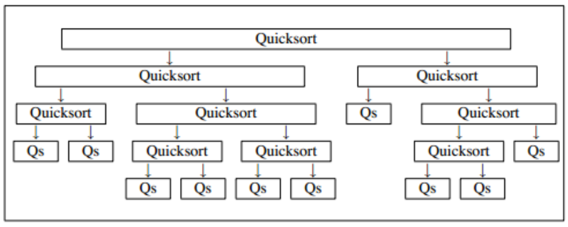


Figure 5. [7]

Sintorn and Assarsson [9] presents a computation for fast sorting of far reaching records utilizing cutting edge GPUs. The technique performs rapid by effectively using the parallelism of the GPU all through the whole count. At first, GPU-based bucketsort or quicksort parts the once-over into enough sublists then to be sorted in parallel utilizing mix sort. The computation is of unpredictability n log n, and for game plans of 8M components and utilizing a solitary Geforce 8800GTS-512, it is 2.5 times as brisk as the bitonic sort figurings, with standard many-sided quality of n(log n)^2, which for long was thought to be the speediest for GPU sorting. It is 6 times speedier than single CPU quicksort, and 10% faster than the late GPU-based radix sort. At last, the computation is further parallelized to utilize two outlines cards, bringing about yet another 1.8 times speedup.

Sorting is a general issue in software engineering. Mergesort [10] is a surely understood sorting calculation of complexity O(n log n), and it can without much of a stretch be implemented on a GPU that backings scattered writing. The GPU-sorting calculations are very bandwidth-restricted, which is represented for instance by the actuality that bitonic sorting of 8-bit values [11] are nearly four times speedier than for 32-bit values [12]. To upgrade the rate of memory comprehends, we along these lines outline a vector-based mergesort, utilizing CUDA and log n render goes, to manage four 32-bit floats at the same time, bringing about an almost 4 times speed change appeared differently in relation to solidification sorting on single floats. The Vector-Mergesort of two four float vectors is refined by utilizing a uniquely created parallel take a gander at and-swap estimation, on the 8 info floats to each string running the CUDA center.

Be that as it may, the calculation turns out to be extremely wasteful for the last m passes, where m = log 2p and p is the quantity of processors on the GPU. The reason is that parallelism is lost when the quantity of remaining records is less than double the quantity of processors. We take care of this issue by at first using a parallel GPU-based bucketsort [13] or quicksort [14], isolating the data list into ≥ 2p cans, trailed by consolidation sorting the substance of every pail, in parallel.

IV. Conclusion

GPU-based sorting calculation had been introduced. It is a creamer that at first uses one go of bucket sort to part the data list into sublists which then are sorted in parallel using a vectorized variant of parallel consolidation sort. The estimation is then exhibit generally performs to some degree speedier than the radix sort, using a single configuration card, and is on a very basic level snappier than other prior GPU-based sorting counts. It is a solicitation of degree snappier than single CPU quicksort. The calculation is unimportant to assist parallelize utilizing double representation cards, bringing about moreover 1.8 times speedup. While GPUSort records genuine timings for the nearest upper force of two size. STL Sort is a CPU-based speedy sort usage.

References

[1] C. C. Elgot and A. Robinson, "Random-access stored-program machines, an approach to programming languages," in *Selected Papers*, ed: Springer, 1982, pp. 17-51.

[2] G. E. Blelloch, "Prefix sums and their applications," 1990.

[3] G. E. Blelloch, "Scans as primitive parallel operations," *Computers, IEEE Transactions on,* vol. 38, pp. 1526-1538, 1989.

[4] D. Cederman and P. Tsigas, "A practical quicksort algorithm for graphics processors," in *Algorithms-ESA 2008*, ed: Springer, 2008, pp. 246-258.

[5] G. E. Blelloch and B. M. Maggs, "Parallel algorithms," in *Algorithms and theory of computation handbook*, 2010, pp. 25-25.

[6] D. Cederman and P. Tsigas, "Gpu-quicksort: A practical quicksort algorithm for graphics processors," *Journal of Experimental Algorithmics (JEA),* vol. 14, p. 4, 2009.

[7] G. E. Blelloch, *Vector models for data-parallel computing* vol. 75: MIT press Cambridge, 1990.

[8] A. Church, "Turing AM. On computable numbers, with an application to the Entscheidungs problcm. Proceedings of the London Mathematical Society, 2 s. vol. 42 (1936–1937), pp. 230–265," *The Journal of Symbolic Logic,* vol. 2, pp. 42-43, 1937.

[9] E. Sintorn and U. Assarsson, "Fast parallel GPU-sorting using a hybrid algorithm," *Journal of Parallel and Distributed Computing,* vol. 68, pp. 1381-1388, 2008.

[10] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30--May 2, 1968, spring joint computer conference*, 1968, pp. 307-314.

[11] N. K. Govindaraju, N. Raghuvanshi, M. Henson, and D. Manocha, "A cache-efficient sorting algorithm for database and data mining computations using graphics processors," *University of North Carolina, Tech. Rep,* 2005.

[12] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "GPUTeraSort: high performance graphics co-processor sorting for large database management," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, 2006, pp. 325-336.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to algorithms second edition," *The Knuth-Morris-Pratt Algorithm", year,* 2001.

[14] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," in *Graphics hardware*, 2007, pp. 97-106.