Volume 3 ▪ Issue 1 ▪ October 2012

INTERNATIONAL JOURNAL OF EXPERIMENTAL ALGORITHMS (IJEA)

ISSN : 2180-1282

Publication Frequency: 6 Issues / Year

CSC PUBLISHERS
http://www.cscjournals.org

INTERNATIONAL JOURNAL OF EXPERIMENTAL ALGORITHMS (IJEA)

VOLUME 3, ISSUE 1, 2012

EDITED BY
DR. NABEEL TAHIR

ISSN (Online): 2180-1282

International Journal of Experimental Algorithms (IJEA) is published both in traditional paper form and in Internet. This journal is published at the website http://www.cscjournals.org, maintained by Computer Science Journals (CSC Journals), Malaysia.

IJEA Journal is a part of CSC Publishers
Computer Science Journals
http://www.cscjournals.org

INTERNATIONAL JOURNAL OF EXPERIMENTAL ALGORITHMS (IJEA)

Book: Volume 3, Issue 1, October 2012

Publishing Date: 25-10-2012

ISSN (Online): 1985-4129

IJEA Journal is a part of CSC Publishers
http://www.cscjournals.org

© IJEA Journal
Published in Malaysia

Typesetting: Camera-ready by author, data conversation by CSC Publishing Services – CSC Journals, Malaysia

CSC Publishers, 2012

TABLE OF CONTENTS

Volume 3, Issue 1, October 2012

An OpenCL Method of Parallel Sorting Algorithms for GPU Architecture

Krishnahari Thouti *kthouti@gmail.com*
Department of Computer Science Engg.
Visvesvaraya National Institute of Technology
Nagpur, 440010, Maharashtra, India

S. R. Sathe *srsathe@cse.vnit.ac.in*
Department of Computer Science Engg.
Visvesvaraya National Institute of Technology
Nagpur, 440010, Maharashtra, India

Abstract

In this paper, we present a comparative performance analysis of different parallel sorting algorithms: Bitonic sort and Parallel Radix Sort. In order to study the interaction between the algorithms and architecture, we implemented both the algorithms in OpenCL and compared its performance with Quick Sort algorithm, the fastest algorithm. In our simulation, we have used Intel Core2Duo CPU 2.67GHz and NVidia Quadro FX 3800 as graphical processing unit.

Keywords: GPU, GPGPU, Parallel Computing, Parallel Sorting Algorithms, OpenCL.

1. INTRODUCTION

The GPU (Graphics Processing Unit) [1] is a highly tuned, specialized machine, designed specifically for parallel processing at high speed. In recent years, Graphic Processing Unit (GPU) has been evolved as massive parallel processor for achieving high computing performance. The architecture of GPU is suitable not only for graphics rendering algorithms but for also general parallel algorithms in a wide variety of application domains.

Sorting is one of the fundamental problems of computer science, and parallel algorithms for sorting have been studied since the beginning of parallel computing. Batcher's $\Theta(\log^2 n)$ - depth bitonic sorting network [2] was one of the first methods proposed. Since then many different parallel sorting algorithms have been proposed [7, 9, 10]. The $\Theta(\log n)$ - depth sorting circuit was proposed in [4, 6].

Given, a diversity of parallel architectures and a number of parallel sorting algorithms, there is a question of which is the best fit for a given problem instance. An extent to which an application will benefit from these parallel systems, depend on the number of cores available and other parameters. Thus, many researchers have become interested in harnessing the power of GPUs for sorting algorithms. Recently, there has been increased interest in such research efforts [8, 11, 16]. However, more studies are needed to claim whether a certain algorithm can be recommended for a particular parallel architecture.

In this paper, we present an experimental study of two different parallel sorting algorithms: Bitonic sort and Parallel Radix sort.

This paper is organized as follows. Section - 2 provides previous work done. In Section - 3, we present GPU architecture and OpenCL Programming model. Parallel Sorting algorithms are explained in Section - 4. Test results and analysis are provided in Section - 5. Section - 6 concludes our work and makes future research plans.

2. RELATED WORK

In this section, we review previous work on parallel sorting algorithms. Study of parallel algorithms using OpenCL is still in progress and there is not much work done in this topic. However, an overview of parallel sorting algorithms is given in [5]. Here we review parallel algorithms with respect to GPU architecture.

A parallel sorting algorithm is presented in [12] for general purpose internal sorting on MIMD machines where performance of the algorithm on the Fujitsu AP1000 MIMD supercomputer is discussed. A comparative performance evaluation of parallel sorting algorithms presented in [13]. They implement parallel algorithms with respect to the architecture of the machine. An on-chip local memory version of radix sort for GPU's has been implemented [21]. As expected, OpenCL local memory is much faster than global memory. Bitonic sorting algorithm has been implemented using stream processing units and Image Stream processors in [17, 15].

An *O(n)* radix sort is implemented in [21]. As reported in [21] radix sort is roughly twice as fast as the CUDAPP[19] radix sort. Quick-sort algorithm for GPU's using CUDA has been implemented in [20] where their results suggest that given a large data set of elements, quick-sort still gives better performance as compared to radix and Bitonic sort. A portable OpenCL implementation of the radix sort algorithm is presented in [24] where authors test radix sort on several GPUs and CPUs. An analysis of parallel and sequential bitonic, odd-even and rank-sort algorithms for different CPU and GPU architectures are presented in [23] where they exploit task parallelism using OpenCL.

3. GPU ARCHITECTURE and OPENCL FRAMEWORK

NVidia GPUs comprises of array of multi-processor units called Streaming Multiprocessors (SMs), also called as Compute Units (CU) and each one consists of multiple Scalar Processor (SP) cores, also known as Processing Elements (PE). The NVidia Quadro FX 3800 has 24 SMs with 8 PEs in each SM as shown in Figure 1. There is on-chip local store called shared memory, through which the PEs communicate with SM and different SMs communicate through off-chip memory called global memory.

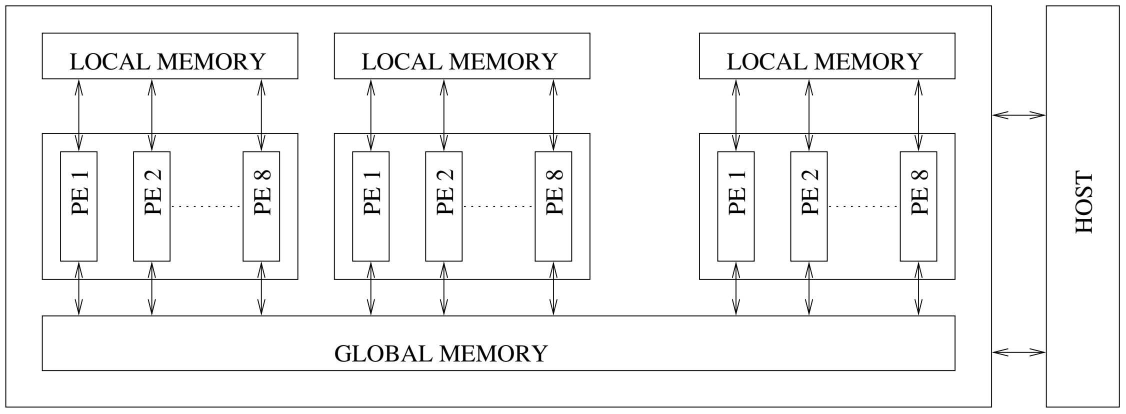


FIGURE 1: GPU Architecture

The GPU is programmable using vendor provided API's such as NVIDIA's CUDA [18], OpenCL specification by Khronos group [22]. While CUDA targets GPU specifically, OpenCL targets heterogeneous system which includes GPUs and/or CPUs. OpenCL programming model involves a host program on the host (CPU) side that launches Single Instruction Multiple Threads (SIMT) based programs called kernels consisting of groups of threads called as warps on the target device. Although management of warps is hardware dependent, programmer can organize problem domain into several work-items, consisting of one or more work-groups. This is explained as ND-Range in GPU architecture. For more information on managing and optimizing ND-Range refer to OpenCL Specifications [22]. In summary, we say, following steps are needed to initialize an OpenCL Application.

- Setting Up OpenCL Environment – Declare OpenCL context, choose device type and create the context and a command queue.
- Declare Buffers & Move Data across CPU & GPU – Declare buffers on the device and enqueue input data to the device.
- Runtime Kernel Compilation – Compile the program from the kernel array, build the program, and define the kernel.
- Run the Program – Set kernel arguments and the work-group size and then enqueue kernel onto the command queue to execute on the device.
- Get Results to Host – After the program has run, read back result array from device buffer to host memory.

See [25, 26, 27, 22] for more details on this topic.

4. PARALLEL SORTING ALGORITHMS

In this section we give brief descriptions of two parallel sorting algorithms selected for implementation.

4.1 Bitonic Sort

Batcher's Bitonic sort [2] is a parallel sorting algorithm which merges two bitonic sequences. Bitonic sorting was originally defined in terms of sorting networks. Sorting networks are comparison networks that always sort their inputs. A sorting network [14, 3] is a special kind of sorting algorithm, where the sequence of comparisons is data independent. This makes sorting networks suitable for implementation in hardware or in parallel processor arrays.

A bitonic sequence is a sequence of values $a = \{a_0, a_1 \ldots, a_{p-1}\}$ with the property that either (1) there exist an index k, where $0<k<p-1$ such that $a_0 \leq a_1 \leq \ldots \leq a_k \geq \ldots \geq a_{p-1}$ or $a_0 \geq a_1 \geq \ldots \geq a_k \leq \ldots \leq a_{p-1}$ or (2) there exist a cyclic shift of indices so that (1) is satisfied. For example, (4, 8, 12, 15, 11, 6, 3, 2) is a bitonic sequence.

Let $s = \{a_1, a_2 \ldots a_p\}$ be bitonic sequence such that $a_0 \leq a_1 \leq \ldots \leq a_{p/2-1}$ and $a_{p/2} \leq a_{p/2+1} \leq \ldots \leq a_{p-1.}$ The bitonic sequence s can be sorted with bitonic split operation which halves the sequence into two bitonic sequences s_1 and s_2 such that all values of s_1 are smaller than or equal to all the values of s_2. That is, bitonic split operation performs:

$$S_1 = \{min\ (a_0, a_{p/2}), \ldots, min\ (a_{p/2-1}, a_{p-1})\}$$
$$S_2 = \{max\ (a_0, a_{p/2}), \ldots, max\ (a_{p/2-1}, a_{p-1})\}$$

For example, the bitonic sequence mentioned above s = (4, 8, 12, 15, 11, 6, 3, 2) will be divided to two bitonic sequences s_1 = (4, 6, 3, 2) and s_2 = (11, 8, 12, 15). Thus, given a bitonic sequence, we can use bitonic splits recursively to obtain short bitonic sequences until we obtain sequences of size one, at which point the input bitonic sequence is sorted. This procedure of sorting a bitonic sequence using bitonic splits is called bitonic merge (BM).

The bitonic sorting network for sorting N numbers consists of $log(N)$ bitonic sorting stages, where i^{th} stage is composed of $N/2^i$ alternating increasing and decreasing bitonic merges of size 2^i. In OpenCL implementation, we set kernel arguments for each of the stages and call the kernel sub-routine bitonic sort. Algorithm 1, 2, and 3 shows bitonic sorting algorithm on GPU device using OpenCL. The algorithm executes on every core in GPU kernel in parallel.

```
__kernel void bitonic_sort(__global *data, int dir)
{
        divide data into $in_1$ and $in_2$
        sort($in_1$, ASC)
        sort($in_2$, DES)
        swap($in_1$, $in_2$, dir)
        sort($in_1$, dir)
        sort($in_2$, dir)
    result = ($in_1$, $in_2$)
}
```

Algorithm 1: Bitonic Sort Kernel for SIMD Architecture

```
for each level i = 1, …, log(n)
{
    for each pass of level j = 1 to i +1
        run_kernel ();
}
```

Algorithm 2: Generalized Bitonic Sort

Algorithm 1 is bitonic sort kernel for SIMD architecture where input data is multiple of 8 data sequence. Algorithm 2 is generalized bitonic sort and its corresponding kernel is shown in algorithm 3.

```
__kernel sort(__global *data, int stage i, int pass_of_stage j,
int dir)
{
    /* using values of i, j, dir – get left_Id & right_Id */
    left_child = data [left_Id]
    right_child = data [right_Id]
    compare(left_child, right_child)

    /* copy left & right child values to data with respect to dir
*/
    data [left_child] = max(left_child, right_child)
    data [right_child] = min(left-child, right_child)
}
```

Algorithm 3: Generalized Bitonic Sort Kernel Using OpenCL

Initially, the host (CPU) device distributes unsorted vector in form of work_groups to GPU cores using the global_size and local_size OpenCL Parameters. Alternate work_items in work_group perform sorting in ascending and descending order. Next, merging stage is performed and result is obtained. For more information, on this parameters please refer OpenCL Specifications [22].

4.2 Parallel Radix Sort

Like the bitonic sort, the radix sort [14] uses a divide-and-conquer strategy; it splits the dataset into subsets and sorts the elements in the subsets. But instead of sorting bitonic sequences, the radix sort is a multiple pass distribution sort algorithm that distributes each item to a bucket according to least significant digit of the elements. After each pass, items are collected from the buckets, keeping the items in order, then redistributed according to the next most significant digit.

Suppose, the input elements are 34, 12, 42, 32, 44, 41, 34, 11, 32, 63.

After First Pass: {[41, 11],   [12, 42, 32, 32],   [63],   [34, 44, 34]}

After Second Pass: {[11, 12], [32, 32, 34, 34], [41, 42, 44], [63]}

When we collect them they are in order: {11, 12, 32, 32, 34, 34, 41, 42, 44, 63}

In OpenCL, the first step of each pass is to compute histogram to identify the least significant digit. Let '*p*' be the number of number of processing elements available on GPU device. Each processing element is responsible for $\lceil n/p \rceil$ input elements. In next step, each processing element counts the number of its elements and then computes the prefix sums of these counts. Next, the prefix sums of all processing elements are combined by computing the prefix sums of the processing element-wise prefix sums. Finally, each processing element places its elements in the output array. More details are given in the pseudo-code below.

```
b ← no. of bits
A← Input Data
cmp ← 1
$cnt_0$ ← contains zero's count
$cnt_1$ ← contains one's count
One, Zero ← Bucket Arrays
Mask ← Temporary Array

for ( i = 0 to $2^b$ – 1)
{
    for ( j = 0 to A.size)
    {
        if (A [j] && cmp)
            $cnt_1$ ++
            One [$cnt_1$]  ← a[j]
        else
            $cnt_0$ ++
            Mask [$cnt_0$] ← j
    }
    for( j = $cnt_0$ to A.size)
    Mask [j] ← A.size – $cnt_0$ + j

  A ← shuffle(A, one, Mask)
  cmp ← left_shift(cmp)
}
result ← A
```

Pseudo-code: Parallel Radix Sort Kernel

The code performs bitwise AND with *cmp*. If AND result is non-zero, code places the element in *One* array and increments one's counter. If the result is zero, the code set appropriate value in *Mask* array and increment zero's counter. Once every element is analyzed, the *Mask* array is further updated to identify each element in *One;s* array. The *shuffle* function re-arranges the *Mask* array data and then process continues.

The computation of histogram is shown in algorithm 4. After this step, histogram is scanned and prefix sum is calculated using the algorithm 5. After this step, re-ordering of histogram takes place and finally result is obtained by transposing the re-ordered histogram. Other implementation details are not mentioned here; only the method is presented in this paper. For more information refer [27].

5. EXPERIMENTAL RESULTS

In this section, we discus machine specifications on which experiments were carried out and present our experimental results. In all cases, the elements to be sorted were randomly generated 10 bit integers. All experiments were repeated 30 times and the results were reported are averaged over 30 runs.

```
Let n = no. of elements
w_i = no. of work_items
w_g = no. of work_groups

/* w_i & w_g can be computed using clDeviceInfo()
   : see [22] */
for ( i = w_i to w_i + w_g)
{
    Extract the group of bits of pass i,and
    Store the result in hist []
}
```

Algorithm 4: Compute Histogram

```
for each processing element, PE_i
{
    sum[i] = list [ (n/p) * i]
    for ( j = 1 to n/p)
    sum[i] = sum[i] + list[(n/p) * i + j ]

  result = ∑(sum)
}
```

Algorithm 5: Parallel Prefix Sum

5.1 Machine Descriptions

The GPU device used for testing simulation is NVidia Quadro FX 3800 which has 192 processing cores and 1 GB device global memory. For comparison purpose, we have implemented and tested the results of quick-sort algorithm on 2.66GHz Intel Core2DUO CPU E7300 with 1GB RAM. The cache specifications are 32KB data cache, 32KBinstruction cache and 3MB shared L2 cache.

5.2 Comparison of the Algorithms

Figure 2 shows the comparison of above mentioned algorithms for different size of input sequence. For comparison purpose, we have taken the sequential version of Quick sort and have compared with OpenCL version of Parallel Bitonic Sort and Parallel Radix Sort. As expected, in all cases, radix sort is fastest, followed by Bitonic sort, and then quick sort. GPU is a large computation unit and thus we measured the GPU runtime called as GPU PROFILE time only, excluding the time for GPU memory allocation, data and memory transfer between CPU and GPU. However, if we take into account, all the parameters concerning GPU application, as explained in Section – 3, we find that quick sort is still the fastest.

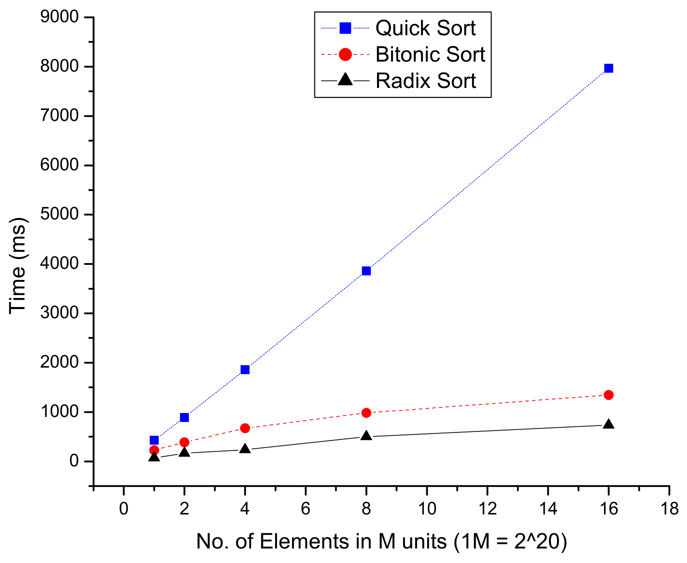


FIGURE 2: Comparison of Sorting Algorithms

6. CONCLUSION AND FUTURE SCOPE

We have presented an analysis of parallel bitonic and radix sort algorithms for GPUs using OpenCL and their comparison with the serial implementation of quicksort on CPU Dual-core machine. We have shown their GPU performance and compared with CPU implementation of quick sort. Our finding reports that radix sort is still the fastest, followed by Bitonic sort, and then quick sort. In future work, along with these sorting algorithms, we are planning to investigate some other parallel sorting algorithms including quick sort and use different GPU architecture from different vendors for our analysis.

REFERENCES

[1] General Purpose Computations Using Graphics Hardware, http://www.gpgpu.org/

[2] K. E. Batcher. “Sorting networks and their applications”. in AFIPS Spring Joint Computer Conference, Arlington, VA, Apr. 1968, pages 307–314.

[3] D.E. Knuth. The Art of Computer Programming. Vol. 3: Sorting and Searching (second edition). Menlo Park: Addison-Wesley, 1981.

[4] M. Ajtai, J. Komlos, Szemeredi. “Sorting in parallel steps”. Combinatorica 3. 983, pp. 1 -19.

[5] S. G. Akl. “Parallel Sorting Algorithms”, Academic Press, 1985.

[6] J. H. Reif, L. G. Valiant. “A Logarithmic Time Sort for Linear Size Networks”. Journals of the ACM, 34(1): 60 – 76, 1987.

[7] G.E. Blelloch,” Vector Models for Data-Parallel Computing”. The MIT Press, 1990.

[8] G.E. Blelloch, C.E. Leiserson, B.M. Maggs, C.G. Plaxton, S.J. Smith, M. Zagha. “A Comparison of Sorting Algorithms for the Connection Machine CM-2”. in Annual ACM Symp. Paral. Algo: Arc. 1991, Pages 3 -16.

[9] F. T. Leighton, “Introduction to Parallel Algorithms and Architectures: Arrays, Trees and Hypercubes”. Morgan Kaufmann, 1992.

[10] J.H. Reif. ”Synthesis of Parallel Algorithms”. Morgan Kaufmann, San Mateo, CA, 1993.

[11] H. Li, K.C. Sevcik. “Parallel Sorting by Over-partitioning”. in Annual ACM Symp. Paral. Algor.Arch. 1994, pages 46 – 56.

[12] A. Tridgell, R. P. Brent. “A general-purpose parallel sorting algorithm” in International J. of High Speed Computing 7 (1995), pp. 285-301.

[13] N. Amato, R. Iyer, S. Sundaresan, Y. Wu. “A Comparison of Parallel Sorting Algorithms on Different Architectures” Texas A & M University, College Station, TX, 1998.

[14] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. Introduction to Algorithms. 2nd edition, The MIT Press. 2001.

[15] T. J. Purcell, C. Donner, M. Cammarano, H. Jensen, P. Hanrahan “Photon mapping on programmable graphics hardware”, in Annual ACM SIGGRAPH / Eurographics conference on Graphics Hardware, 2003, pp. 41 – 50.

[16] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, T. J. Purcell. “A Survey of General-Purpose Computation on Graphics Hardware.” in Eurographics 2005, State of the Art Reports, August 2005, pp. 21-51.

[17] A. Greb, G. Zachmann. “GPU-AbiSort: Optimal Parallel Sorting on Stream Architectures” in IPDPS'06 Proceedings of the 20th international conference on Parallel and distributed processing. 2006.

[18] NVidia CUDA GPGPU Framework. http://www.nvidia.com/

[19] S. Sengupta, M. Harris, Y. Zhang, J. D. Owens. “Scan primitives for GPU computing,” in Graphics Hardware 2007, Aug. 2007, pp. 97–106.

[20] D. Cedermann, P. Tsigas. “A practical quicksort algorithm for graphic processors”, Tech. Rep, Chalmers University of Technology and Goteberg University, 2008.

[21] N. Satish, M. Harris, M. Garland. “Designing efficient sorting algorithms for manycore GPUs”. In Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing. May 23-29, 2009, pp.1-10.

[22] OpenCL Specification, http://www.khronos.org/opencl/

[23] F. Gul, O. Usman Khan, B. Montrucchio, P. Giaccone. “Analysis of Fast Parallel Sorting Algorithms for GPU Architectures”. in Proceeding FIT '11 Proceedings of the 2011 Frontiers of Information Technology Pages 173-178.

[24] P. Helluy. “A portable implementation of the radix sort algorithm in OpenCL”. http://code.google.com/p/ocl-radix-sort/ May 2011

[25] B. Gaster, L. Howes, D.R. Kaeli, P. Mistry, D. Schaa. Heterogeneous Computing with OpenCL. Morgan Kaufmann. 2011.

[26] AMD Accelerated Parallel Processing OpenCL Programming Guide, Advanced Micro Devices, Inc. 2012. http://developer.amd.com/appsdk

[27] M. Scarpino. OpenCL in Action. Manning Publications, 2011.

A Fast Near Optimal Vertex Cover Algorithm (NOVCA)

Sanjaya Gajurel *sxg125@case.edu*
Advanced Research Computing
Case Western Reserve University
Cleveland, OH, US

Roger Bielefeld *rab5@case.edu*
Advanced Research Computing
Case Western Reserve University
Cleveland, OH, US

Abstract

This paper describes an extremely fast polynomial time algorithm, the Near Optimal Vertex Cover Algorithm (NOVCA) that produces an optimal or near optimal vertex cover for any known undirected graph G (V, E). NOVCA is based on the idea of (i) including the vertex having maximum degree in the vertex cover and (ii) rendering the degree of a vertex to zero by including all its adjacent vertices. The two versions of algorithm, NOVCA-I and NOVCA-II, have been developed. The results identifying bounds on the size of the minimum vertex cover as well as polynomial complexity of algorithm are given with experimental verification. Future research efforts will be directed at tuning the algorithm and providing proof for better approximation ratio with NOVCA compared to any other available vertex cover algorithms.

Keywords: Vertex Cover Problem, Combinatorial Problem, NP-Complete Problem, Approximation Algorithm.

1. INTRODUCTION

The Vertex Cover (VC) of a graph G(V,E) with vertex set V and edge set E is a subset of vertices C of V ($C \subseteq V$) such that every edge of G has at least one endpoint in C. In 1972 Richard Karp [1] showed that identification of minimal VC in a graph is an NP-complete problem.

Various algorithmic approaches have been used to tackle NP complete problems. The Vertex Cover problem, one of the NP complete problems, has been actively studied because of its important research and application implications. Polynomial-time approximation and heuristic algorithms for VC have been developed but none of them guarantee optimality. By using the definition of approximation ratio, VC has an approximation ratio of $\rho(n)$ for any input of size n. The solution C produced by approximation algorithm is within the factor of $\rho(n)$ of the solution C* of an optimal algorithm i.e. $C^*/C \le \rho(n)$. Also, the approximation algorithm has approximation ratio of $2 - \varepsilon$, where $0 < \varepsilon < 1$. A 2-approximation [2] algorithm has been trivially obtained and similar approximation algorithms have been developed [3], [4] with an approximation of $(2 - (\ln (\ln n)/2\ln n))$, where n is the number of vertices. Halperin [5] achieved an approximation factor of $(2 - (1 - o(1))(2\ln (\ln \Delta)/ \ln \Delta))$ with maximum degree at most Δ. Karakostas [6] attained an approximation factor of $(2 - \theta(1/(\log n)^{1/2})))$, the best approximation yet, by using the semidefinite programming relaxation of VC. Evolutionary algorithms (EA) that are randomized search heuristics have also been used for solving combinatorial optimization problems including VC [7], [8].

Vertex Cover problems have been solved in O (1.2738k + kn) time [9] by using a bounded search technique where a function of a parameter restricts the search space. Abu-Khazm et al. have identified crown structure to reduce the size of both n and k [10]. It has been known that when relevant parameters are fixed, NP-complete problems can be solved in polynomial time. In both [10] and [11], n is the input size and k is the positive integer parameter. Though not guaranteed to find a minimum vertex cover, an approximation of 3/2 for almost every single graph was obtained in [11]. According to Dinur and Safra [12], it is NP-Hard to get $\varepsilon < 1.3606$.

The paper is organized as follows: the NOVCA algorithm is described in Section 2; Section 3 provides experimental results; Section 4 is the conclusion.

2. NEAR OPTIMAL VERTEX COVER ALGORITHMS (NOVCA)

NOVCA is motivated by the fact that a vertex cover candidates are those that are adjacent to minimum degree vertex so that its degree will be forcibly rendered to zero without choosing it. This fact has been reinforced during tie when the vertex with neighbors having maximum degrees is preferred over other minimum vertices. Without any optimization effort, the complexity of NOVCA is $O(E\ (V + \log^2 V))$; with V = n, the complexity becomes $O(n^2\ (n + \log^2 n))$ which is polynomial. The pseudo-code of NOVCA is presented in Fig. 1. Network Bench Node Degree algorithm [13] has been applied to determine the degree of each node. Then, the sum of the degree of adjacent nodes for each node is calculated. Both these values are included as data structures in a node - deg[v]/adj_deg_sum[v] as showed in Fig. 2. Initially, vertex cover set VC is empty.

NOVCA-I [14] constructs the vertex cover by repeatedly adding, at each step, all vertices adjacent to the vertex of minimal degree; in the case of a tie, it selects the one having the maximum sum of degrees of its neighbors. NOVCA-II, on the other hand, builds vertex cover by including vertices in descending order of degree; in the case of a tie, it chooses the vertex having the minimum sum of degrees of its neighbors. The vertices are chosen in increasing order of their degrees i.e. the adjacent vertices of minimum degree vertex are included in VC first. The magic function GetMinVertex () breaks a tie in selecting the best candidate vertex in a vertex cover. The implementation forcibly renders the degree of low degree vertices to zero without choosing them.

```
Declarations:
        V is the set of vertices of G
        E is the set of edges of G
        deg[V] is an integer array indexed by V for a set
            of vertices V
        sum_adj_deg[V] is an integer array indexed by V for
                a set of vertices V
        VC is the set of vertices comprising a vertex cover
        $Q_{sum\_adj\_deg}$ is the set of vertices having min deg[V]
             (local variable in GetMinVertex())

Functions:
        Degree(v) is the degree of the vertex v ϵ V
        Adj(v) gives the set of vertices that are adjacent
            to v ϵ V
        GetMinVertex() identifies the next adjacent
                vertices to include in the cover

        Heap_MIN(deg) returns the value of min. deg[V]
        HEAP_MAX($Q_{sum\_adj\_deg}$) returns the vertex having max
                  $Q_{sum\_adj\_deg}$


  for each v ϵ V {
    deg[v] = Degree(v)
  }
```

```
 for each v є V {

  sum_adj_deg[v] =$\Sigma_{v' \varepsilon\ Adj(v)}$deg[v']
}

E' = E
VC = ф

while (E'≠ ф){
  $v_c$ = GetMinVertex(deg, sum_adj_deg)
  VC = VC + { Adj($v_c$) }
  for each v є Adj(Adj($v_c$)){  //for NOVCA-I
  **//for each v є Adj($v_c$){  //for NOVCA-II**
   E' = E – { (adj($v_c$), v) }
          deg[v] = deg[v] – 1
  }
  V = V – { Adj($v_c$) } //for NOVCA-I
 **//V = V – { $v_c$ } //for NOVCA-II**
         for each v є V{
       If (Adj(v) == ф) continue
                    sum_adj_deg[v] = $\Sigma_{v' \varepsilon\ Adj(v)}$deg[v']
  }
 } //end while

 /// Magic Function GetMinVertex() Declarations ///

 Vertex GetMinVertex(deg, sum_adj_deg){
   $Q_{sum\_adj\_deg}$ =  ф
   vmin_deg = HEAP_MIN(deg)   //for NOVCA-I
   **//vmax_deg = HEAP_MAX(deg) //for NOVCA-II**
   for each v є V{
     If (deg[v] == vmin_deg)   //for NOVCA-I
     **//If (deg[v] == vmax_deg) //for NOVCA-II**
       $Q_{sum\_adj\_deg}$ = $Q_{sum\_adj\_deg}$ + {v}
   }
   return Heap_MAX($Q_{sum\_adj\_deg}$)   //for NOVCA-I
   **//return Heap_MIN($Q_{sum\_adj\_deg}$) //for NOVCA-II**
 }
```

FIGURE 1: Pseudo-code for NOVCA; E[G]: set of edges of graph G; VC: Vertex Cover Set; Q: Priority Queue; note that the commented bold statements are for NOVCA-II.

3. EXPERIMENTAL WORK AND RESULTS

Experiments to corroborate the theoretical results have been conducted on the CWRU High Performance Computing Resource using compute nodes with 3.0 GHz Intel Xeon processors running Red Hat Enterprise Linux 4 and using the gcc 3.4.6 compiler. Tests are performed in both serial and parallel environments. Results for all example graphs as described above always return optimal (minimum) vertex cover. We have selected Complete Graph as a test graph to determine time complexity of NOVCA for two reasons:

- optimal vertex cover is known; n – 1; where n is the number of vertices
- requires exhaustive search; there is an edge from each vertex to all other vertices

The shell script in Fig. 2 "graph_gen.sh" generates a complete graph of size n entered as input. This graph is then fed to executable "vc (serial) or vc_openmp (parallel)" (C++ program compiled with g++ compiler) to get vertex cover for that particular graph. The outputs are showed in Fig. 3.

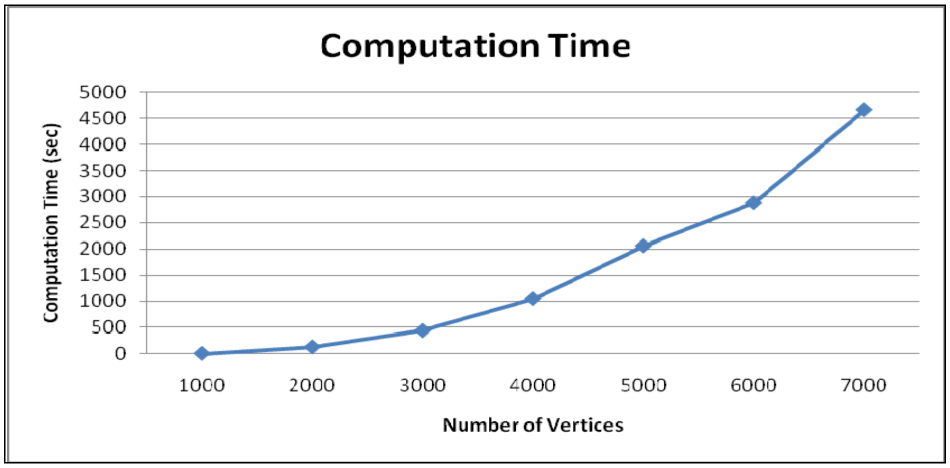
```
#PBS -l walltime=36:00:00
#PBS -l nodes=1:ppn=4:quad
#PBS -N graph1000
#PBS -j oe
cd $PBS_O_WORKDIR
/usr/local/bin/pbsdcp -s vc graph_gen.sh $TMPDIR
cd $TMPDIR
sh graph_gen.sh 1000
cp gen_graph graph1000
time ./vc graph1000 #vc_openmp for parallel
/usr/local/bin/pbsdcp -g '*' $PBS_O_WORKDIR
cd $PBS_O_WORKDIR
```

FIGURE 2: The graph_gen.sh takes 1000 (number of vertices) as an input that creates a netlist in a file, graph1000, input to the executable vc; execuatable vc will be vc_openmp and ppn = 4 respectively for parallel implementation.

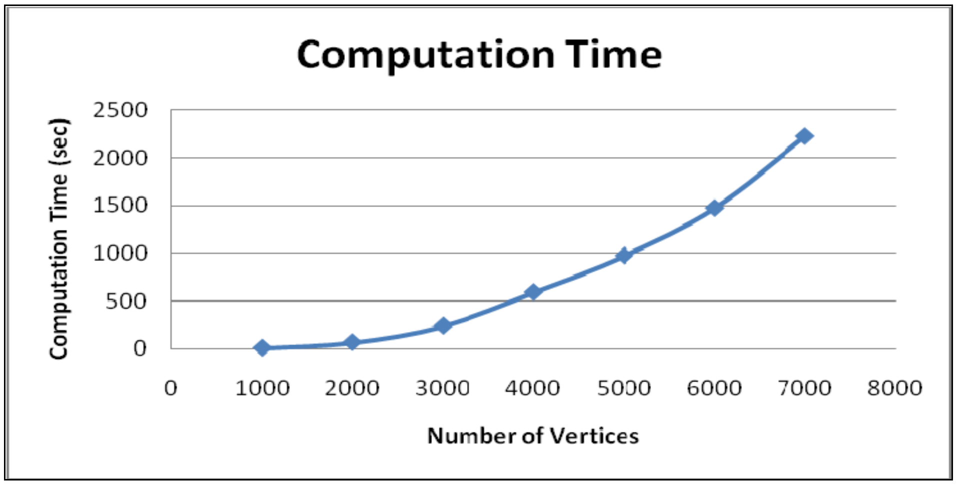
```
The cover consists of the following vertices:
  0     1     2     3     4     5     6     7
  8     9    10    11    12    13    14    15
…
…
994   995   996   997   998
There are 999 vertices in the cover.
real    0m7.161s
user    0m7.156s
sys     0m0.004s
```

FIGURE 3: Output showing the vertices in a vertex cover, number of vertices, and execution time

We have recorded the computation time for different sizes of the graphs for both serial and parallel implementation to elucidate the polynomial complexity of NOVCA algorithm as depicted in Fig. 4(a)(b). We used MATLAB's polyfit(x,y,n) command to verify polynomiality as shown in Fig. 5 and Fig 6(a)(b).



(a)

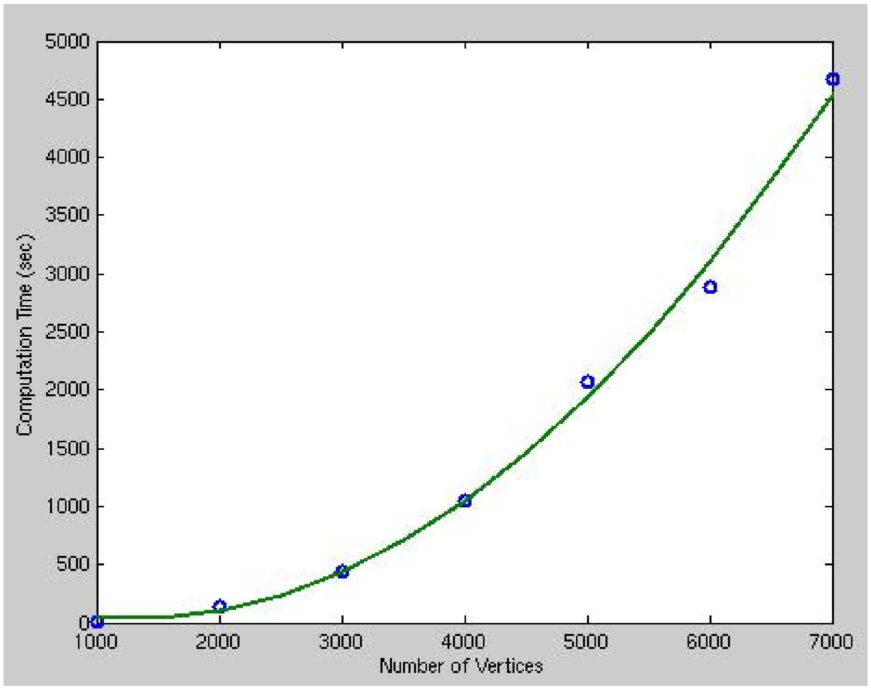


(b)

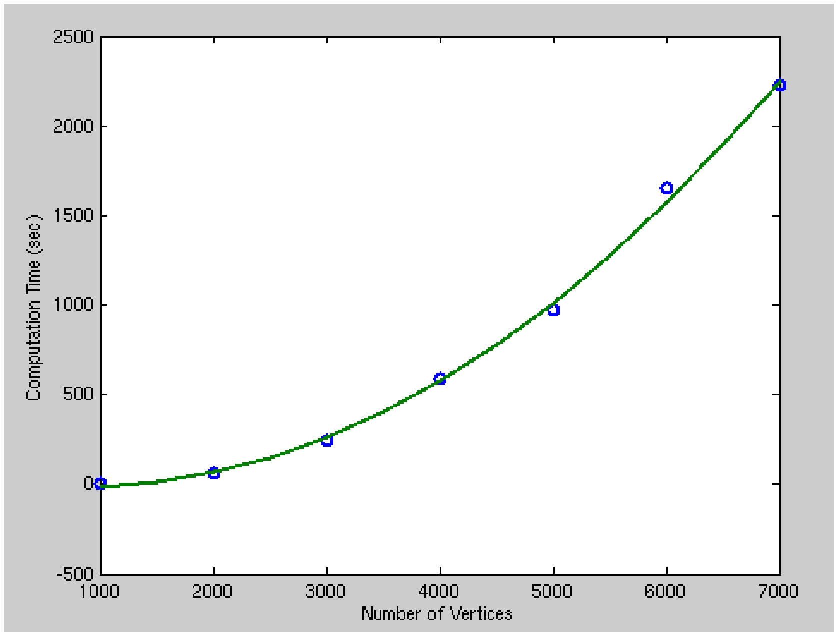
FIGURE 4: Computational Time of NOVCA for different sizes of complete graphs for (a) Serial and (b) Parallel

x = [1000,2000,3000,4000,5000,6000,7000];
y=[7.124,129.21,437.274,1046.93,2061.037,2882.444,4666.
976]; % from serial implementation
y=[7.083,65.08,238.669,589.784,971.582,1649.391,2223.02
0]; % from parallel implementation
p = polyfit(x,y,2)
p = 0.0001   -0.3592 258.4364
x2 = 1000:500:7000;
y2 = polyval(p,x2);
plot(x,y,'o',x2,y2)

FIGURE 5: MATLAB commands used for output data (computation time) from simulation for both serial and parallel implementation



(a)



(b)

FIGURE 6: MATLAB plot using polyfit with n=2; (a) Serial and (b) Parallel

NOVCA has approximation ratio smaller than 1.3606 for all available bench mark (Table 1, Table 2[15]; not showed all of the instances) graphs. For some instances like c-fat, Johnson, and random graphs NOVCA provides optimal cover. Noticeably, the execution time of NOVCA for any instance is remarkable. NOVCA has been found to perform very well compared to other available algorithms. For the instances where it provides near optimal solutions, it outperforms other algorithms in terms of execution time. We have compared NOVCA with COVER [16]. COVER is a stochastic local search algorithm for k-vertex cover. It constructs the initial candidate solution C greedily. When the several vertices satisfy the criterion for inclusion in C, COVER selects one of them randomly with uniform probabilities. The COVER algorithm terminates when either the vertex cover is found or max number of steps (MAX_ITERATIONS), has been reached. NOVCA, on the other hand doesn't have any randomness element and terminates when there are no more vertices in V. So, it has only one run unlike average execution time calculated using random seeds in different runs in COVER.

Though COVER is found to obtain better vertex cover in most of the instances of the benchmarks, NOVCA is very simple and it outperforms COVER in execution time. In case of the graph instance, MANN_a81, where both NOVCA and COVER return the same value 2225, NOVCA is 20 times faster. Though NOVCA-I outperforms NOVCA-II in terms of approximation ratio in almost all instances except keller, p-hat, and sanr, NOVCA-II has better execution time than NOVCA-I. For the challenge instances of frb100-40 [15], NOVCA-I is off by just 17 vertices (NOVCA returns 3917 vertices whereas the optimal vertex cover is 3900), but the execution time is just remarkable; only 2013.667 sec. The challenge is stated as "Based on theoretical analysis and experimental results of smaller instances, I conjecture that in the next 20 years or more (from 2005), these two benchmarks cannot be solved on a PC (or alike) in a reasonable time (e.g. 1 day) [15]." The graphs for number of vertices returned and the execution times, as showed in Fig. 7 and Fig. 8 respectively, portray that NOVCA, though comparable to COVER in terms of number of vertices returned, is significantly faster than COVER. We have also carried out comparisons of NOVCA against two other heuristic Minimum Vertex Cover (MVC) Algorithms, PLS [17] and EWCC [18], with similar results (not explicitly tabulated here).

Instances	|V|	|C*|	NOVCA-I |C|	NOVCA-I |C|/|C*|	NOVCA-I Time (sec)	COVER $\|C\|_{avg}$	COVER $Time_{avg}$ (sec)
frb59-26-1	1534	1475	1485	1.007	80.258	1477	18611.3
frb59-26-2	1534	1475	1484	1.006	79.297	1478	18589.5
frb100-40	4000	3900	3917	1.004	2013.667	-	-
broc200_1	200	179	181	1.011	0.115	179	768.2
broc800_4	800	774	782	1.010	10.832	775	4051.2
C2000.9	2000	1922	1932	1.005	207.060	1922	21489.7
c-fat200-5	200	142	142	1	0.092	142	1549.1
c-fat500-10	500	374	374	1	2.117	374	4401.2
gen200_p0.9_44	200	156	163	1.045	0.092	156	1543.6
hamming10-2	1024	512	512	1	10.297	512	2412.2
hamming10-4	1024	984	988	1.004	21.505	986	3457.6
johnson16-2-4	120	112	112	1	0.076	112	297.9
johnson32-2-4	496	480	480	1	2.273	480	2351.9
keller4	171	160	164	1.025	0.007	160	985.7
keller5	776	749	761	1.016	9.125	749	2364.9
MANN_a27	378	252	253	1.004	0.493	252	756.3
MANN_a81	3321	2221	2225	1.002	773.963	2225	15672.1
p_hat500-1	500	491	492	1.002	2.683	491	1810.2
p_hat1500-3	1500	1406	1414	1.006	74.991	1406	1298.9
san200_0.7_1	200	170	183	1.077	0.117	170	713.7
san1000	1000	985	991	1.006	22.901	989	4972.8
sanr200_0.7	200	183	185	1.011	0.857	183	788.2
sanr400_0.7	400	379	382	1.008	1.030	380	2112.5
graph50-10	50	35	35	1	0.006	35	124.5
graph100-10	100	70	70	1	0.034	70	205.3
graph200-05	200	150	150	1	0.114	150	854.1
graph250-05	250	200	200	1	0.300	200	988.5
graph500-05	500	290	290	1	1.604	290	22555.2

TABLE 1: Performance Comparison between NOVCA-I and COVER on DIMACS and BHOSLIB benchmarks |V|: number of vertices; |C*|: optimal cover; NOVCA |C|: cover returned by NOVCA; COVER $|C|_{avg:}$ Cover returned by COVER; NOVCA Time (sec): Execution time for NOVCA; COVER $Time_{avg}$: Average execution time for COVER; no data available for the instance frb100-40 in COVER

Instances	|V|	|C*|	NOVCA-II |C|	NOVCA-II |C|/|C*|	NOVCA-II Time (sec)	COVER $\|C\|_{avg}$	COVER $Time_{avg}$ (sec)
frb59-26-1	1534	1475	1494	1.014	34.770	1477	18611.3
frb59-26-2	1534	1475	1496	1.014	35.686	1478	18589.5
frb100-40	4000	3900	3944	1.011	885.860	-	-
broc200_1	200	179	182	1.017	1.316	179	768.2
broc800_4	800	774	786	1.016	6.162	775	4051.2
C2000.9	2000	1922	1942	1.010	88.604	1922	21489.7
c-fat200-5	200	142	142	1	1.238	142	1549.1
c-fat500-10	500	374	374	1	1.514	374	4401.2
gen200_p0.9_44	200	156	170	1.090	1.514	156	1543.6
hamming10-2	1024	512	512	1	5.584	512	2412.2
hamming10-4	1024	984	992	1.008	10.350	986	3457.6
johnson16-2-4	120	112	112	1	1.248	112	297.9
johnson32-2-4	496	480	480	1	2.245	480	2351.9
keller4	171	160	162	1.013	1.500	160	985.7
keller5	776	749	761	1.016	5.115	749	2364.9
MANN_a27	378	252	261	1.036	1.641	252	756.3
MANN_a81	3321	2221	2241	1.009	297.236	2225	15672.1
p_hat500-1	500	491	492	1.002	2.595	491	1810.2
p_hat1500-3	1500	1406	1412	1.004	34.535	1406	1298.9
san200_0.7_1	200	170	185	1.088	1.535	170	713.7
san1000	1000	985	992	1.007	11.657	989	4972.8
sanr200_0.7	200	183	184	1.005	1.351	183	788.2
sanr400_0.7	400	379	384	1.013	1.947	380	2112.5
graph50-10	50	35	35	1	1.667	35	124.5
graph100-10	100	70	70	1	1.552	70	205.3
graph200-05	200	150	150	1	1.523	150	854.1
graph250-05	250	200	200	1	1.653	200	988.5
graph500-05	500	290	290	1	2.366	290	22555.2

TABLE 2: Performance Comparison between NOVCA-II and COVER on DIMACS and BHOSLIB benchmarks |V|: number of vertices; |C*|: optimal cover; NOVCA |C|: cover returned by NOVCA; COVER $|C|_{avg:}$ Cover returned by COVER; NOVCA Time (sec): Execution time for NOVCA; COVER $Time_{avg}$: Average execution time for COVER; no data available for the instance frb100-40 in COVER

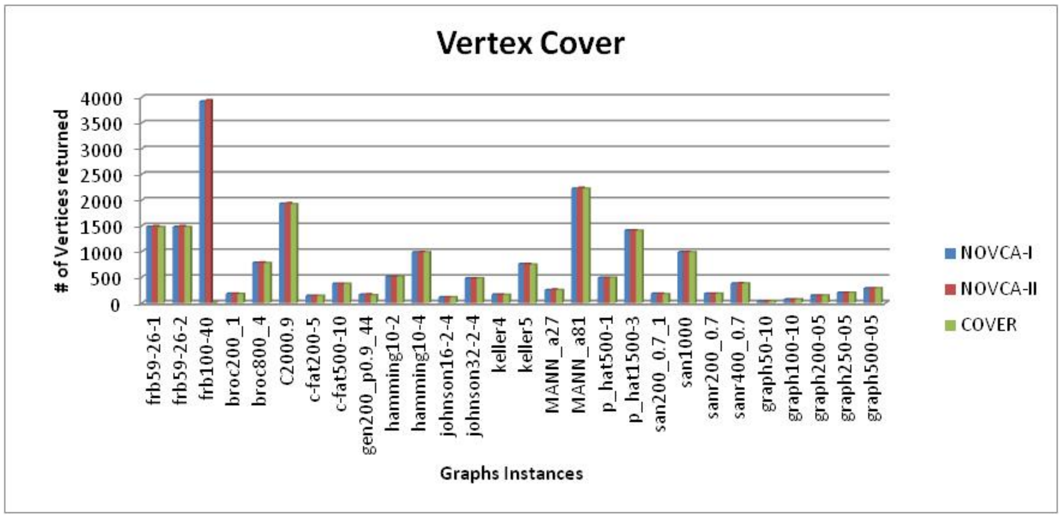


FIGURE 7: Number of Vertices returned by NOVCA-I, NOVCA-II, and COVER; no results from COVER for the instance frb100-40

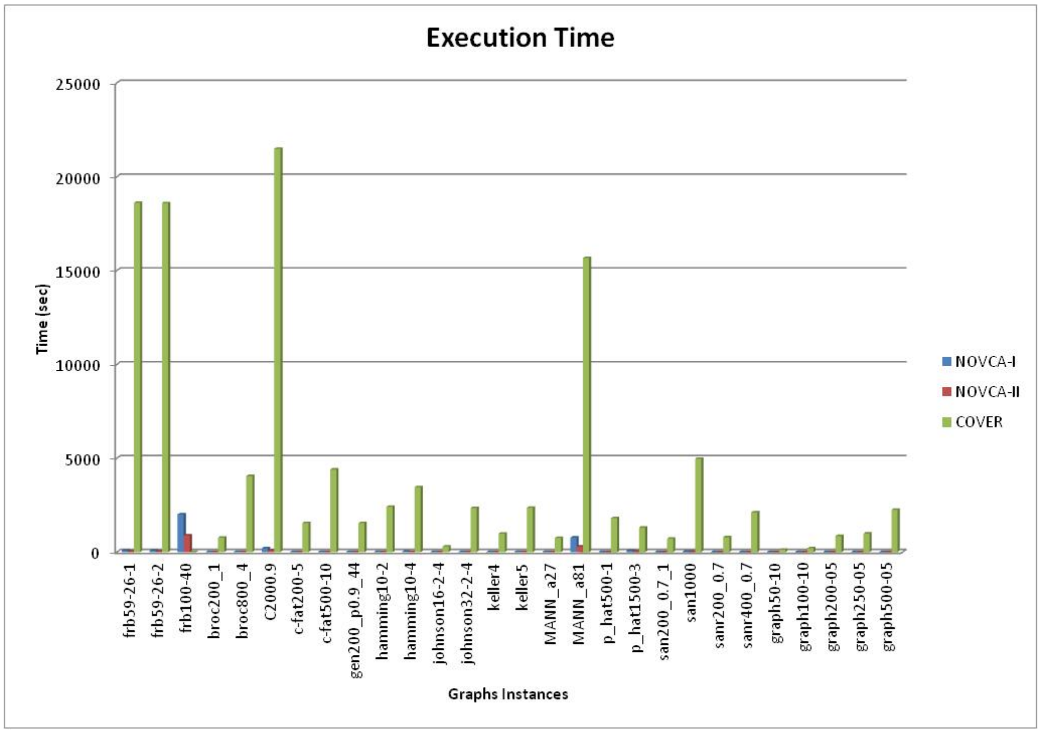


FIGURE 8: Execution time for NOVCA-I, NOVCA-II, and COVER; no results from COVER for the instance frb100-40

4. CONCLUSION AND FUTURE WORK

NOVCA algorithm provides optimal or near optimal vertex cover for known benchmark graphs. The experimental results depict that the algorithm is extremely fast compared to other available state-of-the-art MVC algorithms including COVER, PLS, and EWCC.

Future research will be focused in two areas: deriving a mathematical statement regarding the closeness of the approximation ratio to 1, and investigating approaches to parallelizing the NOVCA algorithm.

5. ACKNOWLEDGEMENT

I would like to thank Geeta Dahal and Pujan Joshi for suggesting counter examples to early versions of the algorithm.

REFERENCES

[1] R. Karp. “Reducibility among combinatorial problems”. In R. E. Miller and J. W. Thatcher (eds.). Complexity of Computer Computations, Plenum Press, NY, pp. 85-103, 1972.

[2] T. Cormen, C. Leiserson, R. Rivest. *Introduction to Algorithms*. The MIT Press, pp. 1022-1024, 2001.

[3] R. Bar-Yehuda and S. Even. “A local-ratio theorem for approximating the weighted vertex cover problem”. North-Holland Mathematics Studies, vol. 109, pp. 27-45, 1985.

[4] B. Monien and E. Speckenmeyer. “Ramsey numbers and an approximation algorithm for the vertex cover problem*”. Acta Informatica*, vol. 22, pp. 115-123, 1985.

[5] E. Halperin. “Improved approximation algorithms for the vertex cover problem in graphs and hypergraphs”. *SIAM J. on Computing*, vol. 31, pp. 1608-1623, 2002. Also in Proc. of 11th SODA, pp. 329-337, 2000.

[6] G. Karakostas. “A better approximation ratio for the vertex cover problem”. ICALP, pp. 1043-1050, 2005.

[7] G. Rudolph. “Finite Markov chain results in evolutionary computation”. *A tour d’horizon, Fundamenta Informaticae*, vol. 35, pp. 67-89, 1998.

[8] P. Oliveto, J. He, X. Yao. “Evolutionary algorithms and the Vertex Cover problem”. CEC, pp. 1430-1438, 2007.

[9] J. Chen, I. Kanj and G. Xia. “Simplicity Is Beauty: Improved Upper Bounds for Vertex Cover”. Technical report TR05-008, School of CTI, DePaul University, 2005.

[10] F. Abu-Khazm, M. Fellows, M. Langston, and W. Suters. “Crown Structures for Vertex Cover Kernelization”. Theory Comput. Systems, vol. 41, pp. 411-430, 2007.

[11] E. Asgeirsson and C. Stein. “Vertex Cover Approximation on Random Graphs”. WEA 2007, *LNCS* 4525, pp. 285–296, 2007.

[12] I. Dinur and S. Safra. “The importance of being biased”. STOC’02, pp. 33-42, 2002.

[13] NWB Team. Network Workbench Tool. Indiana University, North Eastern University, and University of Michigan, http://nwb.slis.indiana.edu/, 2006.

[14] S. Gajurel, R. Bielefeld. “A Simple NOVCA: Near Optimal Vertex Cover Algorithm”. Procedia Computer Science, vol. 9, pp 747-753, 2012.

[15] K. Xu. “Vertex Cover Benchmark Instances (DIMACS and BHOSLIB)”. http://www.cs.hbg.psu.edu/benchmarks/vertex_cover.html, 2012.

[16] S. Richter, M. Helmert, and C. Gretton. “A Stochastic Local Search Approach to Vertex Cover”. In Proceedings of the 30th German Conference of Artificial Intelligence (KI), pp 412-426, 2007.

[17] S. Cai, K. Su and A. Sattar. “Local Search with Edge Weighting and Configuration Checking Heuristics for Minimum Vertex Cover”. *Artif. Intell.*, vol. 175 pp. 1672-1696, 2011.

[18] W. Pullan. “Phased Local Search for the Maximum Clique Problem”. *J. Comb. Optim.*, vol. 12, pp. 303-323, 2006.

INSTRUCTIONS TO CONTRIBUTORS

Experimental Algorithmics studies algorithms and data structures by joining experimental studies with the more traditional theoretical analyses. With this regard, the aim of The International Journal of Experimental Algorithms (IJEA) is (1) to stimulate research in algorithms based upon implementation and experimentation; in particular, to encourage testing, evaluation and reuse of complex theoretical algorithms and data structures; and (2) to distribute programs and testbeds throughout the research community and to provide a repository of useful programs and packages to both researchers and practitioners. IJEA is a high-quality, refereed, archival journal devoted to the study of algorithms and data structures through a combination of experimentation and classical analysis and design techniques. IJEA contributions are also in the area of test generation and result assessment as applied to algorithms.

To build its International reputation, we are disseminating the publication information through Google Books, Google Scholar, Directory of Open Access Journals (DOAJ), Open J Gate, ScientificCommons, Docstoc and many more. Our International Editors are working on establishing ISI listing and a good impact factor for IJEA.

The initial efforts helped to shape the editorial policy and to sharpen the focus of the journal. Started with Volume 3, 2012, IJEA appears in more focused issues. Besides normal publications, IJEA intend to organized special issues on more focused topics. Each special issue will have a designated editor (editors) – either member of the editorial board or another recognized specialist in the respective field.

We are open to contributions, proposals for any topic as well as for editors and reviewers. We understand that it is through the effort of volunteers that CSC Journals continues to grow and flourish.

IJEA LIST OF TOPICS
The realm of International Journal of Experimental Algorithms (IJEA) extends, but not limited, to the following:

- Algorithm Engineering
- Algorithmic Code
- Algorithmic Engineering
- Algorithmic Network Analysis
- Analysis of Algorithms
- Approximation Techniques
- Cache Oblivious algorithm
- Combinatorial Optimization
- Combinatorial Structures and Graphs
- Computational Biology
- Computational Geometry
- Computational Learning Theory
- Computational Optimization
- Data Structures
- Distributed and Parallel Algorithms
- Dynamic Graph Algorithms
- Heuristics
- Mathematical Programming For Algorithms
- Metaheuristic Methodologies
- Network Design
- Parallel Processing
- Randomized Techniques in Algorithms
- Routing and Scheduling
- Searching and Sorting
- Topological Accuracy
- Visualization Code
- VLSI Design
- Graphics

- Experimental Techniques and Statistics
- Graph Manipulation

CALL FOR PAPERS

Volume: 3 - **Issue: 2**

i. Paper Submission: October 31, 2012 **ii. Author Notification:** November 30, 2012

iii. Issue Publication: December 2012

CONTACT INFORMATION

Computer Science Journals Sdn BhD
B-5-8 Plaza Mont Kiara, Mont Kiara
50480, Kuala Lumpur, MALAYSIA

Phone: 006 03 6204 5627

Fax: 006 03 6204 5628

Email: cscpress@cscjournals.org

COMPUTER SCIENCE JOURNALS (CSC JOURNALS)

B-5-8 PLAZA MONT KIARA, MONT KIARA

50480, KUALA LUMPUR, MALAYSIA

PHONE: 00603 6204 5627

FAX: 00603 6204 5628

URL: http://www.cscjournals.org