A PARALLEL SOLUTION TO MAXIMAL INDEPENDENT SET PROBLEM USING CUDA

Multi-core Computing

Mohammad Parsa Bashari 400104812
Amirhossein Koochakian 400105199

Fall 2023

Contents

1 Maximal Independent Set (MIS) Problem Statement **3**
1.1 Theoretic Terminology . . . 3
1.2 Sequential Algorithm . . . 3
1.3 Applications . . . 3

2 Parallel Algorithm **4**

3 Implementation **4**
3.1 The Main Kernel . . . 4
3.2 Encountered Challenges and Our Solutions . . . 5
3.2.1 The Problem of Deep Copy . . . 5
3.2.2 The Problem of Locking Inside a Warp . . . 6

4 Another Parallel Algorithm (by Karp & Wigderson) **6**
4.1 Procedures . . . 7
4.1.1 HEAVYFIND() . . . 7
4.1.2 SCOREFIND() . . . 7
4.1.3 INDFIND() . . . 7
4.2 Implementation using CUDA and OpenMP . . . 7

5 Evaluation and Results **8**

1 Maximal Independent Set (MIS) Problem Statement

Given a graph $G(V, E)$ we are supposed to find a subset of vertices $I \subseteq V$ where I is a maximal independent set.

1.1 Theoretic Terminology

Let $G(V, E)$ be an undirected graph without loops or multiple edges. For any set $S \subseteq V$, let $N(S)$, the *neighborhood* of S, be defined as $\{w \in V |$ for some $u \in S, \{u, w\} \in E\}$. Then S is *independent* if $S \cap N(S) = \varnothing$; that is, no two vertices in S are adjacent. An independent set S is called a *maximal independent set* if S is not properly contained in any independent set. Equivalently, $S \subseteq V$ is a maximal independent set if $S \cap N(S) = \varnothing$ and $S \cup N(S) = V$.

1.2 Sequential Algorithm

Let $G(V, E)$ be a graph with vertex set $V = \{1, 2, ..., n\}$. Then the following sequential algorithm constructs a maximal independent set I.

Algorithm 1 Sequential Algorithm to Construct an MIS

```
1: I ← ∅
2: for i = 1 to n do
3:     if i ∉ N(I) then
4:         I ← I ∪ {i}
```

There is no apparent way to make the sequential algorithm run in $o(n)$ time through the use of $n^{O(1)}$ processors. The intuition that this algorithm is inherently sequential is supported by the following theorem by Cook [1].

Theorem 1.1. The problem of deciding whether vertex n lies in the independent set created by the sequential algorithm is complete in P with respect to logspace reducibility.

1.3 Applications

Maximal independent sets find applications in various fields such as:

- **Wireless Sensor Networks:** In sensor networks, where nodes are deployed to monitor an area, maximal independent sets can be used to schedule the activities of non-interfering sensors.
- **Graph Coloring:** In graph coloring, the vertices are assigned colors such that no two adjacent vertices share the same color. Maximal independent sets can be used to find colorings with the minimum number of colors.
- **Distributed Computing:** In distributed systems, maximal independent sets are useful for tasks like leader election, where nodes in a network need to select a leader without conflicts. Each node in the maximal independent set can be a potential leader.
- **Cryptography:** In some cryptographic protocols, the use of maximal independent sets can enhance security. For example, in key distribution schemes, nodes in a maximal independent set can be used to exchange secret keys securely.
- **VLSI Design:** In the design of Very Large Scale Integration (VLSI) circuits, maximal independent sets can be used for tasks such as scheduling operations to optimize the usage of resources and reduce delays.

- **Social Network Analysis:** In social networks, identifying maximal independent sets can help in identifying non-overlapping groups of individuals.
- **Bioinformatics:** In the analysis of biological networks, such as protein-protein interaction networks or gene regulatory networks, maximal independent sets can help identify sets of elements (proteins or genes) that do not directly interact with each other.

2 Parallel Algorithm

Let $G(V, E)$ be a graph with vertex set $V = 1, 2, ..., n$ and Flag be a state array of size $|V|$. $Flag_i$ is 0 if vertex i is undecided, 1 if it's chosen for MIS, and 2 if it's not.

Algorithm 2 Top-Level Description of the code executed by thread i

```
1: if Vertex i is not already checked then
2:     Try to acquire the lock of itself and its neighbors
3:     if Lock acquired then
4:         Set its Flag to 1 and its neighbors to 2
```

We will explain the exact code for this algorithm in the following section.

3 Implementation

3.1 The Main Kernel

Each thread manages a vertex. If the vertex has already been checked, the thread will not do any additional process. However, if its Flag is 0, the thread will try to lock the vertex and its neighbors. Flags of the neighbors will be set to 2 if the lock of the vertex and its neighbors are acquired. Otherwise, the locks will be released to be used by other threads.

```
__global__ void maximalIndependentSet(const Graph* G, int* Flags, int* V) {
    size_t v = blockIdx.x * gridDim.y + blockIdx.y;

    while (v < G->n) {
        if (Flags[v]) break;

        if (atomicCAS(&V[v], 0, 1) == 0) {
            size_t k = 0;
            for (size_t j = 0; j < G->V[v].degree; j++) {
                int ngh = G->V[v].Neighbors[j];
                if (Flags[ngh] == 2 || atomicCAS(&V[ngh], 0, 1) == 0) {
                    k++;
                } else {
                    break;
                }
            }
            if (k == G->V[v].degree) {
                // Win on self and neighbors, fill flags
                Flags[v] = 1;
                for (size_t j = 0; j < G->V[v].degree; j++) {
                    int ngh = G->V[v].Neighbors[j];
                    if (Flags[ngh] != 2) {
                        Flags[ngh] = 2;
```

```
                    }
                }
            } else {
                // Lose, reset V values up to the point where it lost
                V[v] = 0;
                for (size_t j = 0; j < k; j++) {
                    int ngh = G->V[v].Neighbors[j];
                    if (Flags[ngh] != 2) {
                        V[ngh] = 0;
                    }
                }
            }
        }
    }
}
```

Code Snippet 1: CUDA kernel for our parallel implementation of MIS problem

3.2 Encountered Challenges and Our Solutions

3.2.1 The Problem of Deep Copy

We have the following `structs` to store the graph.

```
typedef struct Vertex {
    int degree;
    int *Neighbors;
} Vertex;

typedef struct Graph {
    int n;
    int max_degree;
    Vertex *V;
} Graph;
```

Code Snippet 2: The structs for storing the graph

In this case (that we have pointers in our `structs`), when we want to move a `Graph` from host to device, a simple `cudaMemcpy()` won't work. This is because `cudaMemcpy()` copies the contents of the `Graph` struct (including the pointer value `Vertex *v`) into the device; but this pointer value is still pointing to somewhere in the host memory. Note that we have this problem in two levels because in each `Vertex` there is a `int *Neighbors` field.

To solve this issue, we should use a temporary pointer variable in the host pointing to somewhere in the device memory. Then we copy the array from the host into the device using `cudaMemcpy()`. At the end, we copy the pointer value itself from the host to the desired field of the graph in the device. As I mentioned earlier, we should do this in two levels in our case. A more complete explanation of the problem is in stack overflow [2]. In our code, the `deep_copy()` function moves a whole `Graph` from the host to the device.

```
void deep_copy(Graph *graph, Graph **graph_dev) {
    Graph *dev_graph;
    cudaMalloc(&dev_graph, sizeof(Graph));
    cudaMemcpy(dev_graph, graph, sizeof(Graph), cudaMemcpyHostToDevice);

    Vertex *v;
    cudaMalloc(&v, graph->n * sizeof(Vertex));

    cudaMemcpy(&(dev_graph->V), &v, sizeof(Vertex *), cudaMemcpyHostToDevice);
```

```
10
11      for (int i = 0; i < graph->n; i++) {
12          cudaMemcpy(&(v[i]), &(graph->V[i]), sizeof(Vertex), cudaMemcpyHostToDevice);
13          int *neighbors;
14          cudaMalloc(&neighbors, graph->max_degree * sizeof(int));
15          cudaMemcpy(neighbors, graph->V[i].Neighbors, graph->max_degree * sizeof(int),
    cudaMemcpyHostToDevice);
16          cudaMemcpy(&(v[i].Neighbors), &neighbors, sizeof(int *), cudaMemcpyHostToDevice);
17      }
18      *graph_dev = dev_graph;
19  }
```

Code Snippet 3: The function to solve the deep copy issue

3.2.2 The Problem of Locking Inside a Warp

In GPU programming, managing concurrent access to shared resources like vertices in parallel algorithms is crucial for efficient execution. When utilizing multiple threads within the same block, it's crucial that threads within a block share the same resources, including shared memory. This can lead to challenges in implementing locking mechanisms effectively to protect critical sections of the algorithm. At first, we implemented the algorithm using multiple threads in a block, but due to the fact that threads in the same block execute same instructions, we were not able to lock threads successfully.

Therefore, instead of considering multiple threads in a single block, we assigned a thread to each block. By assigning a single thread to each block, we ensured that each block operates independently, thereby avoiding the issue of multiple threads within a block trying to synchronize operations.

The following code calculates and determines grid dimension. It is used to make sure that we have enough number of blocks to assign to threads.

```
1  dim3 calculateGridDim(int n) {
2      if (n < 65535)
3          return dim3(n, 1);
4      return dim3(65535, (n + 65535 - 1) / 65535);
5  }
```

Code Snippet 4: The function to calculate the grid dimension

4 Another Parallel Algorithm (by Karp & Wigderson)

The algorithm that we use in order to implement a parallel solution to the MIS problem is the one offered by *Karp* and *Wigderson* [3]. A top-level description of their algorithm is as follows:

Algorithm 3 Top-Level Description of the Parallel Algorithm

1: $I \leftarrow \varnothing; H \leftarrow V$
2: **while** $H \neq \varnothing$ **do**
3: $\quad S \leftarrow$ an independent set in induced subgraph H
4: $\quad I \leftarrow I \cup S$
5: $\quad H \leftarrow H - (S \cup N_H(S))$

The paper divides line 3 into three steps each of which is a parallel procedure:

Algorithm 4 Parallel Algorithm for the MIS Problem

1: $I \leftarrow \varnothing; H \leftarrow V$
2: **while** $H \neq \varnothing$ **do**
3: $K \leftarrow \textsc{HeavyFind}(H)$
4: $T \leftarrow \textsc{ScoreFind}(K)$
5: $S \leftarrow \textsc{IndFind}(T)$
6: $I \leftarrow I \cup S$
7: $H \leftarrow H - (S \cup N_H(S))$

4.1 Procedures

There are three procedures in this algorithm: HeavyFind(), ScoreFind(), and IndFind(). These procedures are explained in detail in the paper [3]. Here, we briefly describe what these procedures do, deferring the intricate details to the paper.

4.1.1 HEAVYFIND()

Procedure $\textsc{HeavyFind}(H)$ produces a subgraph K with at least $|H|/\lceil \log |H| \rceil$ heavy vertices. It requires $O(\log |H|)$ executions of the body of the while loop, and each of these executions can be performed in $O(\log |H|)$ time using $|H|^2$ processors.

4.1.2 SCOREFIND()

Having found a set K with many heavy vertices, the algorithm proceeds to find a set T within K such that $Score_K(T)$ is large. The *Score* is a metric for a set of vertices defined rigorously in the paper [3].

4.1.3 INDFIND()

Given a set of vertices T, $\textsc{IndFind}(T)$ constructs an independent subset of T by "killing" one end-point of each edge occurring in T. In this procedure, one processor is assigned to each pair $\{u, w\} \subseteq T$. The processor assigned to $\{u, w\}$ kills (arbitrary) u or v if $\{u, w\} \in E(T)$[1].

4.2 Implementation using CUDA and OpenMP

There exists an implementation of this algorithm in Piotr Mikołajczyk's GitHub repository [4]. We have run this code and compared its results with ours in the next section.

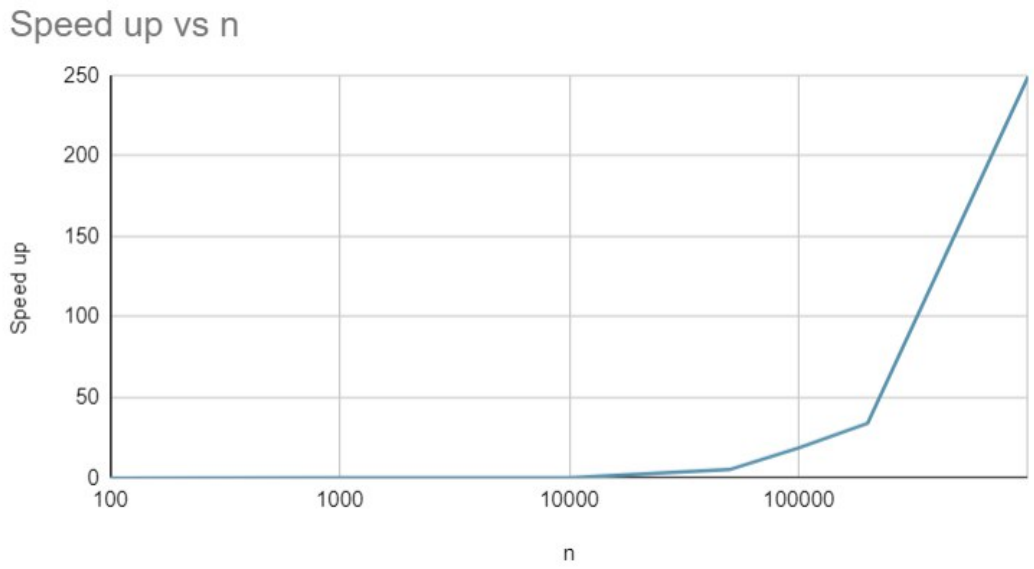
[1] where $E(T) = \{\{u, w\} \subseteq T | \{u, w\} \in E\}$

5 Evaluation and Results

In order to evaluate our implementation, we implemented the serial algorithms and compared the results. The following figure shows an example run on a graph with 1,000,000 vertices and 900,000 edges:

```
parsa … Multicore project MIS-cuda make clean
rm -rf a.out
parsa … Multicore project MIS-cuda make
nvcc helpers.cu main.cu -o a.out && ./a.out
CUDA implementation:
execution time: 0.000266 sec
Correct Maximal Independent Set!
--------------------------
Serial implementation:
execution time: 0.066267 sec
Correct Maximal Independent Set!
--------------------------
speedup: 249.12406015x
parsa … Multicore project MIS-cuda
```

As you can see, we got about 250× speedup.
In the following diagram, we report our speedup for different values of $|V|$.



As you can see, the speedup is increasing when n grows big. The speedup passes 1 when n is about 10,000. This means the parallel algorithm is more efficient when the graph has more than 10,000 vertices.

References

[1] Stephen A. Cook. The classification of problems which have fast parallel algorithms. In Marek Karpinski, editor, *Foundations of Computation Theory*, pages 78–93, Berlin, Heidelberg, 1983. Springer Berlin Heidelberg.

[2] M. Harris. Stack overflow. https://stackoverflow.com/a/9323898.

[3] Richard M. Karp and Avi Wigderson. A fast parallel algorithm for the maximal independent set problem. *J. ACM*, 32(4):762–773, oct 1985.

[4] Piotr Mikołajczyk. Cuda-maximal-independent-set. https://github.com/pmikolajczyk41/CUDA-Maximal-Independent-Set, 2018.