GREEDY ALGORITHMS FOR DISTANCE-2 GRAPH COLORING AND BIPARTITE GRAPH PARTIAL COLORING

by
MUSTAFA KEMAL TAŞ

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfillment of the requirements for the degree of
Master of Science

Sabancı University

July, 2019

GREEDY ALGORITHMS FOR DISTANCE-2 GRAPH COLORING AND BIPARTITE GRAPH PARTIAL COLORING

APPROVED BY:

Asst. Prof. Kamer Kaya .....
(Thesis Supervisor)

Assoc. Prof. Hüsnü Yenigün .....

Assoc. Prof. Hasan Sözer .....

Date of Approval: 12.07.2019

to my family

İKİ MESAFELİ ÇİZGE BOYAMA VE İKİ PARÇALI ÇİZGE BOYAMA İÇİN AÇGÖZLÜ ALGORİTMALAR

Mustafa Kemal Taş

Bilgisayar Mühendisliği, Yüksek Lisans Tezi, 2019

Tez Danışmanı: Dr. Ögr. Üyesi Kamer Kaya

Anahtar Kelimeler: 2 uzaklıklı çizge boyama, 2 parçalı çizge boyama, paralel algoritmalar

Özet

Koşut bir uygulamanın görev etkileşim çizgesi komşu görevlerin farklı renklerle boyandığında birbirleri ile aynı renkteki görevler aynı anda pahalı bir senkronizasyon veri yapısı kullanılmadan aynı anda çalıştırılabilmektedir. Bu tür bir çalıştırmada bir renkteki görevler bitirilmeden, başka bir renkteki görev koşut halde işlenemeyeceğinden boyama esnasında kullanılan renk sayısı koşut uygulamanın çalıştırılması esnasında karşılaşılacak senkronizasyon adım sayısını belirtmektedir. Literatürde çizge boyama problemi "bir çizgeyi mümkün olan en az sayıda renk kullanarak komşu noktalara farklı renkler vermek" olarak tanımlanmıştır ve bir optimizasyon problemi olarak görüldüğünde NP-Hard sınıfındadır.

Çizge boyama probleminin farklı çeşitleri de paralel hesaplama, özellikle paralel bilimsel hesaplama alanında önemlidir. Problemin yukarıda bahsedilen

basit halinde 1-uzaklık kullanılırken, k-uzaklık tanımı da özellikle k = 2 için pratikte kullanılmaktadır. Bu tezde de bu problem üzerine yoğunlaşılmıştır. Problemin genel hali "bir çizgeyi mümkün olan en az sayıda renk kullanarak birbirinden k ve daha az uzaklıktaki nokta ikililerine farklı renkler vermek" olarak tanımlanabilir. Literatürde bu problem için az renk kullanan buluşsal yöntemler önerilmiştir ve bu yöntemler k = 1 için oldukça hızlıdır. Fakat k = 2 için özellikle büyük çizgelerde bu buluşsal yöntemler dakikalar mertebesinde zaman alabilmektedirler. çizge boyamanın bir uygulamanın çalışması için sadece bir ön işlem olduğu düşünüldüğünde bu işlemin getirdiği ekstra zamanın mümkün olduğu kadar az olması işlemin uygulanabilirliği için önemlidir. Bu tezde 2-uzaklık çizge boyama ve bu problemin farklı bir türü olan iki parçalı çizge boyama problemleri için iyimser ve açgözlü buluşsal yöntemler önerilmiştir. Bu yöntemler çok çekirdekli işlemcilerde ve Grafik İşleme Ünitelerinde koşut olarak gerçeklenmiş, ve ölçeklenebilirlikleri analiz edilmiştir. Yapılan deneylerde önerilen yöntemlerin ölçeklenebilir ve 16 çekirdek kullanıldığında literatürdeki yöntemlerden ortalama 25 kat hızlı oldukları görülmüş, özellikle sosyal ağ karakteri taşıyan çizgeler için büyük performans artışı sağladığı saptanmıştır.

Yine bu tez çerçevesinde aynı renge sahip nokta kümelerinin eleman sayılarının birbirine yakın olması üzerine de çalışılmıştır. Bu tür dengeli dağılımlı bir boyama, uygulamanın çok çekirdekli işlemciler ve özellikle GİÜ'ler üzerinde çalışması esnasında her senkronizasyon adımında bütün çekirdekleri doyuracak kadar iş yükü olmasını sağlayacağından yüksek performans için önemli olabilmektedir. Bu tezde neredeyse hiç ekstra külfet getirmeden bunu sağlayabilecek iki yöntem önerilmiştir. Yapılan deneylerde bu yöntemlerin başarılı olduğu sonucuna varılmıştır.

GREEDY ALGORITHMS FOR DISTANCE-2 GRAPH COLORING AND BIPARTITE GRAPH PARTIAL COLORING

Mustafa Kemal Taş

Computer Science and Engineering, Master's Thesis, 2019

Thesis Supervisors: Asst. Prof. Kamer Kaya

Keywords: distance-2 graph coloring, bipartite graph partial coloring, balanced coloring, parallel algorithms

Abstract

In parallel computing, a valid graph coloring yields a lock-free processing of the colored tasks, data points, etc., without expensive synchronization mechanisms. However, the coloring stage is not free and the overhead can be significant. In particular, for distance-2 graph coloring (D2GC) and bipartite graph partial coloring (BGPC) problems, which have various use-cases within the scientific computing and numerical optimization domains, the coloring overhead can be in the order of minutes with a single thread for many real-life graphs, having millions and billions of vertices and edges.

In this thesis, we propose a novel greedy algorithm for the distance-2 graph coloring problem on shared-memory architectures. We then extend the algorithm to bipartite graph partial coloring problem, which is structurally

very similar to D2GC. The proposed algorithms yield a better parallel coloring performance compared to the existing shared-memory parallel coloring algorithms, by employing greedier and more optimistic techniques. In particular, when compared to the state-of-the-art, the proposed algorithms obtain 25× speedup with 16 cores, without decreasing the coloring quality. Moreover, we extend the existing distance-2 graph coloring algorithm to manycore architectures. Due to architectural limitations, the multicore algorithm can not easily be extended to manycore. Thus several optimizations and modifications are proposed to overcome such obstacles. In addition to multi and manycore implementations, we also offer novel optimizations for both D2GC and BGPC on social network graphs. Exploiting the structural properties of social graphs, we propose faster heuristics to increase the performance without decreasing the coloring quality. Finally, we propose two costless balancing heuristics that can be applied to both BGPC and D2GC, which would yield a better color-based parallelization performance with a better load-balancing, especially on manycore architectures.

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

A coloring on a graph $G = (V, E)$ with vertex set V and edge set E, explicitly partitions the vertices in V into a number of disjoint subsets such that two vertices $u, v \in V$ that are in the same color set are independent from each other, i.e., $(u, v) \notin E$. Graphs have been frequently used to model data, e.g., matrices and tensors, as well as computations. In these models, two neighbor vertices, i.e., an edge, usually imply a potential race-condition in a parallel execution. On the other hand, given a valid coloring on V, each color set, formed by independent vertices, can be simultaneously processed in a lock-free manner and without a synchronization overhead. The total number of colors is equal to the number of synchronization points, hence minimizing the number of colors yields a better parallel performance. Unfortunately, *the distance-1 graph coloring problem* (D1GC), i.e., coloring a graph with the minimum number of colors such that all adjacent vertices have different colors, is NP-Complete and hard to approximate [Matula, 1968, Zuckerman, 2007].

The traditional adjacency-based neighborhood is not sufficient for numerous applications such as efficient computation of Hessians and Jacobians or channel assignment problem. Instead, the problem can be modeled as a *bipartite graph partial-coloring* (BGPC) problem. In BGPC, given a bipartite graph $G = (V_A \cup V_B, E)$, one wants to color the vertices in V_A with a minimum number of colors, such that all vertex pairs that are adjacent to

at least one common V_B vertex have different colors. A similar problem is *distance-2 graph coloring* (D2GC), where a graph is colored in a way that the color of each vertex is different than the colors of the vertices in its distance-2 neighborhood, which is defined as the set of vertices that can be reached in at most two hops. For more details on the applications of BGPC and D2PC as well as the parallel algorithms to solve these problems on shared-memory and distributed-memory architectures, we refer the reader to [Gebremedhin et al., 2013, Coleman and More, 1983, Bozdağ et al., 2005, Bozdağ et al., 2010, Gebremedhin et al., 2005, Gebremedhin et al., 2002].

From the parallel computing perspective, another desirable property of a good coloring is the *balance* on the color set cardinalities [Lu et al., 2015, Meyer, 1973, Hajnal and Szemeredi, 1970, Robert K. Gjertsen et al., 1996]. A balanced coloring can improve the convergence speed and the value of the final objective function for some iterative algorithms. However, a tight balance is not required if shared-memory parallelism is the only concern; if all the color set cardinalities are above a certain threshold, that depends on the number of processors/cores available and the task heterogeneities, the parallel performance will not be disrupted by the remaining imbalance since there will be enough work to feed all the available cores/processors.

Good colorings, that use less number of colors, are not free and their generation adds an overhead for parallelization. Furthermore, the impact of this overhead increases if the coloring is performed sequentially and the actual job is executed on a large number of cores. This is why parallelization of graph coloring algorithms have been extensively studied for all the problems above, e.g., [Gebremedhin et al., 2013, Bozdağ et al., 2010, Çatalyürek et al., 2012, Deveci et al., 2016]. The results in the literature show that the execution time of a sequential D1GC algorithm is less than a second for many real-life graphs. However, for D2GC and BGPC, the overhead can be in the order of minutes.

In our previous work we proposed an algorithm for both D2GC and BGPC [Taş et al., 2017]. The proposed algorithm outperformed the state-of-the-art algorithms by applying a greedier heuristic on CPU threads. In this thesis we extend that algorithm to work on GPU threads as well as CPU

threads to increase the efficiency even further. Unfortunately, the algorithm can not be directly and efficiently adapted to GPU threads due to the architectural limitations such as the shared memory size. Moreover, the nature of the algorithm does not allow too much parallelism since it decreases the solution quality at intermediate steps leading to a worse overall execution time. Thus, a new approach is needed for utilizing GPU threads. In this thesis we also propose optimizations for coloring social network graphs, which exploit the structural properties of such graphs.

In this thesis, we first propose greedier algorithms for both D2GC and BGPC problems. The proposed algorithms outperform the state-of-the-art algorithms by applying a greedier heuristic on multicore architectures. Compared to an existing parallel coloring tool, the proposed algorithm runs 25× faster on average when executed with 16 threads. All of the algorithms are tested with same parameters, meaning that the speedup comes solely from the proposed heuristics.

Second, we adapt the existing parallel D2GC algorithm to manycore architectures. Due to architectural limitations, a straightforward adaptation is not possible. Moreover, the nature of the algorithm does not allow further parallelism, as it increases the race conditions at intermediate steps leading to a worse overall execution time. Thus, we propose several optimization tricks to overcome these obstacles. Compared to the multicore counterpart, the proposed implementation runs 4× faster on average.

Third, we focus on a special type of graphs: Social Network Graphs. As social networks get more and more popular, they provide the scientific computing community with many useful datasets. Generally, such graphs have many low-degree nodes and a few high-degree, i.e., central, nodes. As we will discuss further in later sections, the quality of the coloring is strongly dependent on the maximum degree of the graph. This allows us to employ an even greedier algorithm, which relaxes the coloring criteria for low-degree nodes for better performance. This relaxation does not affect the overall coloring quality, since the overall quality is decided by the maximum degree of the graph. With the proposed heuristic, the coloring can be done 60× faster on average without decreasing the coloring quality on social network

graphs.

Last, to obtain a balanced coloring, we propose two online balancing heuristics. The first heuristic aims not to increase the total number of colors, whereas the second heuristic aggressively improves the balance by using more colors. Both heuristics are integrated on top of the proposed D2GC algorithm. The standard deviation of the color cardinalities decrease 1.44× for the first heuristic and 4.00× for the second one. Moreover, applying these heuristics is almost free, i.e., there is no computational overhead.

To summarize, the contribution of this paper is four-fold: **1)** We propose greedy algorithms for D2GC and BGPC in multicore setting. **2)** We extend the parallel D2GC algorithm to work on manycore architectures and discuss implementation challenges. **3)** We propose several optimizations for social network graphs that can be applied to both D2GC and BGPC. **4)** We integrate two costless balancing heuristics to obtain a more balanced coloring.

For the multicore experiments, we compare our results to the algorithms proposed in `ColPack`, an open source graph coloring library that provides D2GC and BPGC implementations. The selection is based on the rationale that it is the only publicly available distance coloring library to the best of our knowledge, and almost all the literature use algorithms which are less optimistic than the ones proposed in this work. In order to have a fair comparison, we have implemented the algorithms proposed in `ColPack` from the scratch so that both algorithms share the same codebase. We even fine-tuned the performance of existing less-optimistic variants for fairness. For the manycore experiments, we compare our results to the above-mentioned implementation, again to share the same codebase.

The rest of the paper is organized as follows: Chapter 2 introduces the notation and background on parallel coloring and describes the state-of-the-art. A literature survey and related work are presented in Chapter 3. The proposed algorithms as well as the optimization techniques are described in detail in Chapter 4. Chapter 5 introduces the datasets used in experiments. In Chapter 6 the experimental setup is described and the results are presented. Finally, Chapter 7 concludes the thesis.

Chapter 2

BACKGROUND AND NOTATION

2.1 Speculative Coloring

Most of the recent coloring algorithms use a speculative, iterative approach which first colors the vertices optimistically in parallel hoping that a valid coloring will be generated, e.g., [Gebremedhin et al., 2013, Çatalyürek et al., 2012, Deveci et al., 2016, Sarıyüce et al., 2012]. The validity of the coloring is then verified in a conflict removal step; if a conflict, i.e., *a pair of neighbor vertices with the same color*, is detected, one of the vertices is tagged to be colored in the next iteration. Let $G = (V, E)$ be a graph and let $V_{color} \subseteq V$ be the vertices that need to be colored. Let $\texttt{nbor}(v) \subset V_{color}$ define set of v's neighbor vertices that need to be colored. Throughout the text, non-negative integers will be used as colors and **-1** is the color of an uncolored vertex. A pseudocode of the greedy optimistic graph coloring approach is given in Algorithms 1, 2 and 3.

Algorithm 1 GreedyGraphColoring

Input: $G = (V, E)$, $V_{color} \subseteq V$: vertices to be colored, `nbor`(.): the neighborhood function for the vertices in V_{color}.
Output: c[.]: a valid coloring array for V_{color}

1: $W \leftarrow V_{color}$
2: $c[v] \leftarrow -\mathbf{1}$, $\forall v \in V_{color}$
3: **while** W is not empty **do**
4: $\quad c \leftarrow$ ColorWorkQueue(G, W, c)
5: $\quad W \leftarrow$RemoveConflicts(G, W, c)

Algorithm 2 ColorWorkQueue

Input: $G = (V, E)$, W: vertices to color, `nbor`(.): the neighborhood function, c[.]: an incomplete coloring with no conflicts.
Output: c[.]: an optimistic coloring.

1: **for** each $w \in W$ **in parallel do**
2: $\quad F \leftarrow \emptyset$ ▷ **thread private** forbidden color set for w
3: $\quad$ **for** each $u \in$ `nbor`(w) **do**
4: $\quad\quad$ **if** $c[u] \neq -\mathbf{1}$ **then**
5: $\quad\quad\quad F \leftarrow F \cup \{c[u]\}$
6: $\quad col \leftarrow 0$ ▷ first-fit coloring policy
7: $\quad$ **while** $col \in F$ **do**
8: $\quad\quad col \leftarrow col + 1$
9: $\quad c[w] \leftarrow col$

As the algorithms show, at each iteration, a set of vertices in W are optimistically colored. A conflict removal phase is then performed to check if they are conflicting with the other vertices in V_{color}. When conflicts are detected, the *conflicting vertices* are added to the next iteration's vertex queue and the procedure is repeated. This greedy and optimistic approach can be used for almost all the coloring variants and the definitions of V_{color} and `nbor`(.) change with respect to the problem. For D2GC, $V_{color} = V$ and `nbor`(u) is the set of vertices in V whose shortest-path distances to u are less than or equal to two. For the BGPC problem, on a bipartite graph $G = (V, E)$ where $V = V_A \cup V_B$ has two parts, $V_{color} = V_A$ and for each $u \in V_A$, `nbor`(u) is defined as $\{v \in V_A \setminus \{u\} : \exists w \in V_B$ s.t. $(u, w) \in E$ and $(v, w) \in E\}$.

The BGPC problem can be considered as a hypergraph coloring problem [Bozdağ et al., 2010] where the elements of V_A correspond to the *pins*

Algorithm 3 RemoveConflicts

Input: $G = (V, E)$: the graph to color, W: vertices to color, `nbor`(.): the neighborhood function, c[.]: an optimistic coloring.
Output: W_{next}: the work queue for next iteration, c[.]: a (probably incomplete) coloring with no conflicts.

```
1: W_next ← ∅                                   ▷ a shared queue for the next iter.
2: for each w ∈ W in parallel do
3:     for each u ∈ nbor(w) do
4:         if c[u] = c[w] and w > u then
5:             W_next ← W_next ∪ {w} : atomic
6:             break
```

to be colored, and the ones in V_B correspond to the *nets* in the hypergraph which define the neighborhood. Based on this analogy, for clarity, while describing our algorithms we will use the terms *vertex* and *net* to denote a V_A and V_B vertex, respectively, in the bipartite graph. Similarly, for a vertex $u \in V_A$ ($v \in V_B$), `nets`(u) (`vtxs`(v)) will denote the set of V_B (V_A) vertices adjacent to u (v).

Lastly, for the D2GC problem, the value $1+\max_{v \in V}(|\texttt{nbor}(v)|)$ is a trivial lower bound on the number of colors required for a valid coloring since all vertices in a distance-2 neighbourhood need to be colored with distinct colors. The counterpart of this bound in BGPC variant is $\max_{v \in V_B}(|\texttt{vtxs}(v)|)$.

2.2 Compute Unified Device Architecture

One of the most commonly used manycore architectures used today in scientific computing are graphical processing units (GPUs). Compute Unified Device Architecture (`CUDA`) is a parallel computing platform developed by `NVIDIA` for general computing on GPUs. For manycore implementations, we have used `CUDA` to leverage the high performance computing potential of thousands of GPU cores. Here we present the terminology on `CUDA`. For the rest of the paper we will use the term *device* to refer GPU and *host* to refer CPU.

- **Kernel**: A *kernel* is an application or a program that runs on the device. Typically, kernels are defined as functions and executed on

device threads.

- **Thread**: A *thread* is the smallest computation unit with the finest granularity on which the kernels are executed. Each thread has its own registers and private memory.
- **Block**: A *block* is a group of threads. The main advantage of having threads grouped into blocks is that they can share a common memory to perform related tasks together.
- **Grid**: A *grid* is the topmost container which contains a group of blocks in it. It can be used as a three dimensional arrangement of blocks.
- **Warp**: Each block is split into groups of threads called *warps*. All the threads in a single warp execute concurrently and are controlled by the same program counter. Hence, the threads in a single warp always perform the same instruction, possibly on different data.
- **Global memory**: The main memory of GPU devices that can be accessed by both the host and device.
- **Shared memory**: Block-private memory that have lower latency compared to the global memory. A shared memory region is shared between the threads in the same block but can not be accessed by other blocks.

In Figure 2.1, an overall GPU architecture is shown. In Figure 2.2, the memory hierarchy on GPUs is presented. Clearly, the memory units closer to the processors have lower latency.

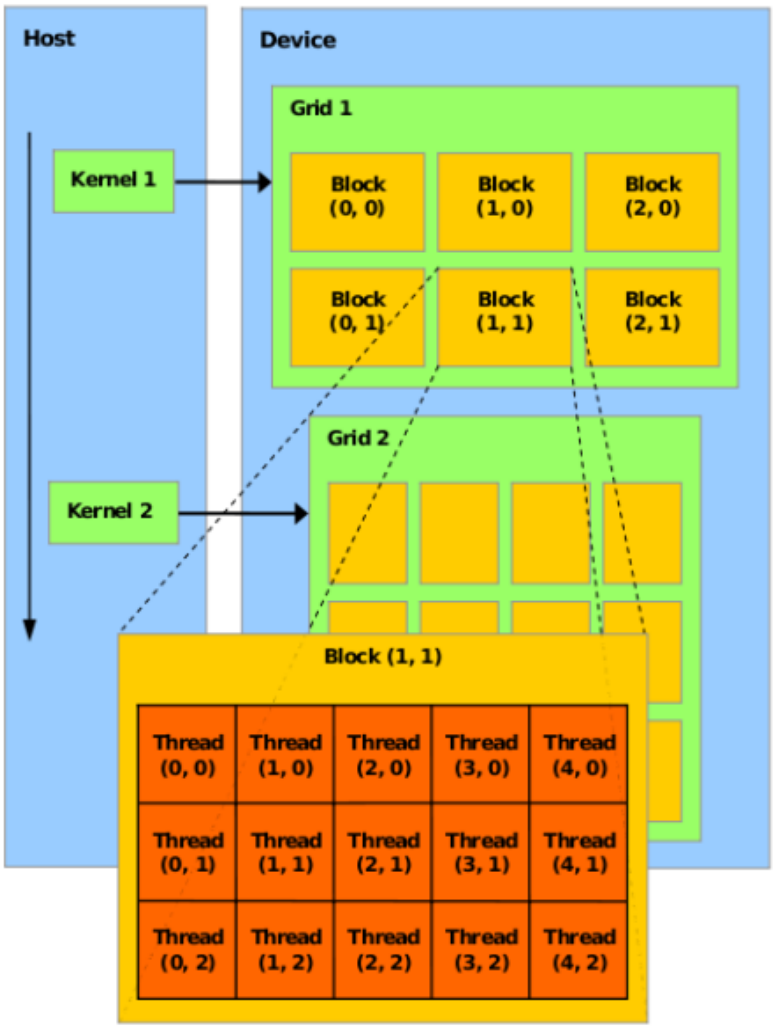


Figure 2.1: Overall GPU architecture

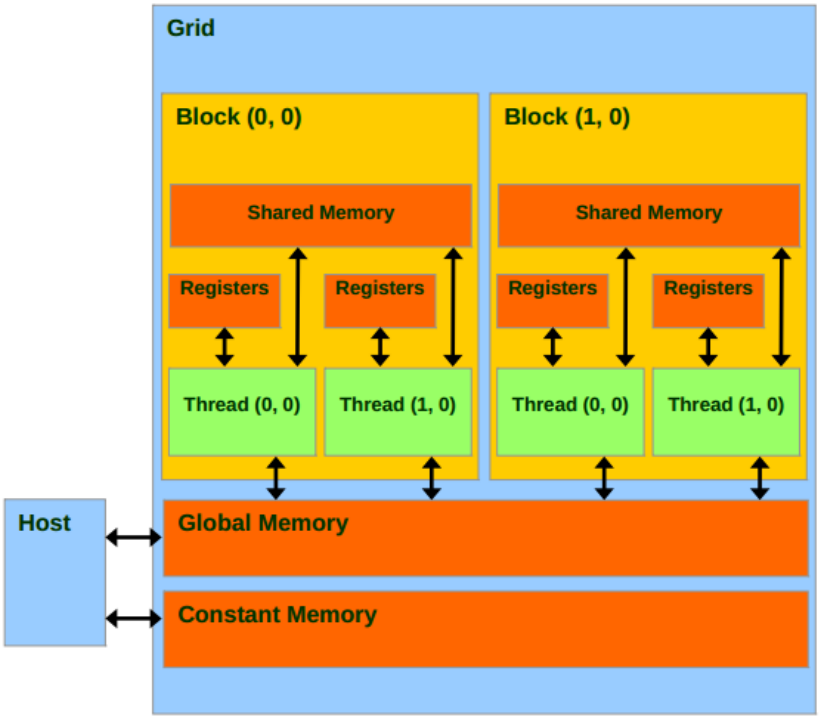


Figure 2.2: Memory hierarchy for GPUs

Chapter 3

EXISTING ALGORITHMS

Coloring has mostly been investigated for distance-1 coloring, but most of the ideas can be ported to other variants. Since graph coloring is NP-Complete [Matula, 1968] and hard to approximate [Zuckerman, 2007] in most of its variants, the vertices are greedily colored one after another, and the lowest available color for a vertex is selected. Such an algorithm produces a coloring with less than $1+\Delta$ colors for the distance-1 variant of the problem. Though to avoid the worst case, it is common to carefully choose the order in which the vertices are processed [Gebremedhin et al., 2005] using either a static [Matula and Beck, 1983, Welsh and Powell, 1967] or dynamic [Brélaz, 1979] ordering.

Earlier coloring algorithms [Allwright et al., 1994, Jones and Plassmann, 1993, Gjertsen Jr. et al., 1996] are based on generating maximum independent sets in parallel via algorithms such as [Luby, 1986]. Recent techniques optimistically color the vertices in parallel assuming that a valid coloring will be generated and then verify the validity of the coloring. In case of an invalid coloring, one of the neighbor vertices that are of the same color is tagged to be colored again in the next iteration of the algorithm. This technique was successfully applied on distributed memory machine [Boman et al., 2005, Bozdağ et al., 2008, Sarıyüce et al., 2011, Sarıyüce et al., 2014], including for BGPC and D2GC [Bozdağ et al., 2005, Bozdağ et al., 2010]. The algorithm was investigated also on shared memory, multicore and manycore architectures [Çatalyürek et al., 2012, Gebremedhin and Manne, 1999, Pat-

wary et al., 2011, Gebremedhin et al., 2002, Deveci et al., 2016] and on hybrid MPI + OpenMP systems [Sarıyüce et al., 2012]. One common point of [Bozdağ et al., 2005, Bozdağ et al., 2010] and the proposed work is that the conflict removal phase of D2GC has been performed around middle vertices which is similar to the net-based conflict removal. Nevertheless, the authors studied D2GC in the distributed setting and applied the approach for all iterations.

Parallel algorithms tend to obtain a higher color count than their sequential counterparts because a strict vertex ordering is not enforced and conflicts can cause vertices to be colored completely out of order. Culberson proposed a post optimization technique [Culberson, 1992] that iteratively recolors vertices in an order depending on the color they were given in the previous iteration. This was successfully applied on shared-memory systems [Gebremedhin and Manne, 1999] and distributed memory systems [Sarıyüce et al., 2011, Sarıyüce et al., 2014]. In this work, for BGPC and D2GC, we did not observe a significant increase on the number of colors with parallelization compared to the sequential execution. However, such post-optimization techniques can be employed to further reduce the color counts in our algorithms.

Chapter 4

PARALLEL GRAPH COLORING

The state-of-the-art algorithms for a parallel D2GC and BGPC have a quadratic complexity for both coloring and conflict resolution phases. This complexity comes from the distance-2 traversal that is carried out at each iteration to detect used colors, and in fact it is the bottleneck of the algorithm. However, based on the rationale that for every vertex v, all the vertices in $\texttt{nbor}(v)$ are distance-2 connected through v, we propose greedier algorithms that yield linear complexity by letting v *distribute* colors among its neighbors.

4.1 Parallel Algorithms for Distance Two Graph Coloring

The traditional implementation of a parallel distance-2 graph coloring is a straightforward extension of the speculative distance-1 coloring algorithm. In the coloring phase, each vertex traverses its distance-2 neighbourhood and records all the used colors as *forbidden*, then selects a suitable color accordingly. Similarly, in the conflict resolution phase, each vertex again traverses the distance-2 neighbourhood and clears itself if a conflict is encountered. These phases alternate until a valid coloring is obtained. For the rest of the thesis, we will refer to these algorithms as *vertex-based* algorithms since each thread is responsible for a single *vertex*.

The pseudocode for the vertex-based coloring and conflict removal phases

are given in Algorithm 4 and Algorithm 5. In lines 3-8 of Algorithm 4, the forbidden colors are stored in a fixed-size, thread-private array. After that, the first non-forbidden color, i.e., a color that is not in the forbidden colors array, is assigned to the corresponding vertex. For Algorithm 5, the only difference from its distance-1 counterpart is that both distance-1 and distance-2 neighbours are traversed to find a conflicting neighbour.

For both algorithms, the threads traverse only the most recent work queue. Thus, the early iterations are the most time consuming ones. As the algorithm proceeds to later iterations, the work queue gets drastically smaller thus the execution time decreases.

Algorithm 4 D2GC-ColorWorkQueue-Vertex

Input: $G = (V, E)$: a graph, $c[.]$: an incomplete coloring, W: vertices to color
Output: $c[.]$: the (most) optimistic coloring array.

```
1:  for each v ∈ W in parallel do
2:      F ← ∅ : thread private forbidden color set for v
3:      for each u ∈ nbor(v) do
4:          if c[u] ≠ −1 and c[u] ∉ F then
5:              F ← F ∪ {c[u]}
6:          for each w ∈ nbor(u) do
7:              if c[w] ≠ −1 and c[w] ∉ F then
8:                  F ← F ∪ {c[w]}
9:      col ← 0                                  ▷ first-fit coloring policy
10:     while col ∈ F do
11:         col ← col + 1
12:     c[w] ← col
```

The time complexity of both algorithms is quadratic in terms of the size of the graph. For both algorithms, each vertex traverses its distance-2 neighborhood. Namely, for each vertex v, all the vertices adjacent to v are visited. Then for each vertex u which are adjacent to v, the neighbors of u is visited. Thus the time complexity of an iteration is

$\mathcal{O}\left(\sum_{v \in V} \sum_{u \in \texttt{nbor}(v)} |\texttt{nbor}(u)|\right)$.

For the conflict removal phase, there might be early terminations (line 6 in Alg. 3). However, this worst-case bound is tight; if the optimistic coloring is valid then the whole neighborhood should be traversed.

Algorithm 5 D2GC-RemoveConflicts-Vertex

Input: $G = (V, E)$: a graph $c[.]$: an optimistic coloring, W: most recent work queue

Output: W_{next}: the work queue for next iteration, $c[.]$: an incomplete coloring.

```
 1: W_next ← ∅
 2: for each v ∈ W in parallel do
 3:     for each u ∈ nbor(v) do
 4:         if c[v] = c[u] and v > u then
 5:             W_next ← W_next ∪ {v} : atomic
 6:             break
 7:         for each w ∈ nbor(u) do
 8:             if c[v] = c[w] and v > w then
 9:                 W_next ← W_next ∪ {v} : atomic
10:                 break
```

As mentioned above, the traditional approach traverses the most recent work queue at each iteration. Numerous experiments have shown that the size of the work queue decrease drastically after the early iterations, meaning that the most time consuming part of the execution is the early iterations. Empirically, 78% of the runtime is observed to be used on the first iteration. That number goes up to 89% for the first two iterations. In this work we propose a greedier and more optimistic method to attack these early iterations. After the early iterations, both algorithms switch back to their *vertex-based* counterparts.

Instead of having all the vertices traverse their distance-2 neighbourhood, the proposed algorithm attacks the most time consuming early iterations by having all the vertices traversing their distance-1 neighbourhood and assigning *different* colors to each neighbour. The rationale behind this idea is the fact that for a vertex v, any two vertices are distance-2 neighbours if they are both in $nbor(v)$, connected by v, thus they should be assigned different colors. The same idea is also applied to the conflict removal phase; all the vertices traverse their distance-1 neighborhood and resolve conflicts amongst the vertices in that neighborhood. The proposed coloring and conflict removal algorithms are given in Algorithm 6 and Algorithm 8. For the rest of the thesis, these algorithms will be referred as *net-based* algorithms. The term *net* is taken from the hypergraph terminology since each vertex is treated

similar to a *net* in a hypergraph.

The coloring algorithm (Algorithm 6) starts with traversing the distance-1 neighborhood and assigning colors in a first-fit manner to uncolored or conflicting neighbours (line 5). If the visited neighbour already has a valid color, then its color is marked as forbidden (line 9). This is done by keeping a thread-private, fixed-size array F for each thread. The algorithm terminates when the whole distance-1 neighbourhood is traversed.

This algorithm is an order of magnitude faster than its vertex-based counterpart, namely, it is linear in terms of the size of the graph ($|V| + |E|$). However, while coloring, each thread only checks local conflicts within the distance-1 neighbourhood of the current net; this is the optimism. Since most of the vertices are members of many distance-1 neighbourhoods, most of them are assigned to conflicting colors due to race conditions. This is the most optimistic net-based coloring since threads "hope" that the assigned colors in earlier positions will not appear in the same neighbourhood. Unfortunately, our preliminary experiments have shown that this level of optimism is maleficent due to the large number of conflicts it incurs. To keep the coloring process in the right track by reducing the number of conflicts, we propose Algorithm 7, which is a modified version of Algorithm 6. In this algorithm, two main modifications are made.

First, instead of a first-fit coloring strategy, a *reverse first-fit* strategy is used. The main source of conflicts is multiple threads assigning the same color to vertices in the same neighbourhood. Since all threads use **0** as the initial color for the first-fit strategy, the same small colors are more frequently used and cause conflicts. The straightforward idea would be assigning different initial colors for each thread, however a randomized approach would not guarantee maintaining solution quality, i.e., it might increase the total number of colors since it doesn't take into account any lower or upper bounds. However the reverse first-fit strategy assigns $|\texttt{nbor}(v)|$ to each thread as the initial color, and goes backwards looking for the *largest* possible color at each iteration. The advantage of this strategy is that, it prioritizes different colors for each net instead of using the same small colors for each neighbourhood, thus decreases the possibility of having conflicts. Moreover, since $|\texttt{nbor}(v)|$ is

an obvious lower bound on the total number of colors used, we do not expect a large increase in the final number of colors. Also, for the same reason this approach is guaranteed to use non-negative colors whatsoever.

The second modification is having an additional traverse of the distance-1 neighbourhood to mark the forbidden colors, at the beginning of the algorithm. In Algorithm 6, a single pass is done over the distance-1 neighbourhood and vertices are *recolored* if they are conflicting with any of the previously colored vertices. As previously mentioned, since threads are oblivious about the colors of unvisited neighbours it is highly probable that a thread assigns a color that is already claimed by another vertex in the same neighbourhood. In such cases, the latter vertex is recolored, leading to an avalanche of conflicts. With our proposed modification, first the whole neighbourhood is traversed and forbidden colors are stored in F, a thread-private fixed-size array. While doing so, any uncolored vertex or any vertex that causes a conflict (due to the actions of other threads) is added to a local work queue W_{local}, again a thread-private array. After this traversal is done, only the vertices in W_{local} are colored using the proposed reverse first-fit strategy. The pseudocode of the modified version of the net-based coloring algorithm is given in Algorithm 7.

Algorithm 6 D2GC-ColorWorkQueue-Net-Naive

Input: $G = (V, E)$: a graph, $c[.]$: an incomplete coloring.
Output: $c[.]$: the (most) optimistic coloring array.

```
1: for each v ∈ V in parallel do
2:     F ← ∅ : thread private forbidden color set for v
3:     col ← 0                                          ▷ first-fit coloring
4:     for each u ∈ nbor(v) do
5:         if c[u] = −1 and c[u] ∈ F then
6:             while col ∈ F do
7:                 col ← col + 1
8:             c[u] ← col
9:         F ← F ∪ {c[u]}
```

To demonstrate the benefits of these two modifications, in Table 4.1, we present the number of uncolored (remaining) vertices after the first iteration of the algorithm on two randomly selected graphs. The results for different

Algorithm 7 D2GC-ColorWorkQueue-Net

Input: $G = (V, E)$: a graph, $c[.]$: an incomplete coloring.
Output: $c[.]$: the (most) optimistic coloring array.

1: **for** each $v \in V$ **in parallel do**
2: $F \leftarrow \emptyset$: **thread private** forbidden color set for v
3: $W_{local} \leftarrow \emptyset$: **thread private** vertices to be colored
4: **if** $c[v] \neq -1$ **then**
5: $F \leftarrow F \cup \{c[v]\}$
6: **else**
7: $W_{local} \leftarrow W_{local} \cup \{v\}$
8: **for** each $u \in \texttt{nbor}(v)$ **do**
9: **if** $c[u] \neq -1$ **and** $c[u] \notin F$ **then**
10: $F \leftarrow F \cup \{c[u]\}$
11: **else**
12: $W_{local} \leftarrow W_{local} \cup \{u\}$
13: $col \leftarrow |\texttt{nbor}(v)|$ ▷ reverse first-fit coloring
14: **for** each $u \in W_{local}$ **do**
15: **while** $col \in F$ **do**
16: $col \leftarrow col - 1$
17: $c[u] \leftarrow col$
18: $col \leftarrow col - 1$

Matrix-Graph	$\|V\|$	Remaining $\|W_{next}\|$ after the first iteration		
		Alg. 6	Alg. 6 + reverse	Alg. 7
bone010	986,703	863,785	806,264	610,924
coPapersDBLP	540,486	409,621	303,152	133,874

Table 4.1: The number of uncolored (remaining) vertices after the first iteration for two graphs, obtained from matrices bone010 and coPapersDBLP, when Algorithms 6 and 7 are used on 16 threads.

graphs are similar to those presented, so only two of them are presented. The performance results for all the graphs will be presented in Chapter 6

The conflict resolution phase is relatively simpler. Similar to the coloring phase, the conflict resolution phase also populates a forbidden colors array F by traversing the distance-1 neighbourhood. If a color is encountered for the first time, it is added to F. If it has already been added to F in previous iterations then the color of that vertex is cleared, thus the conflict is resolved. Also, the conflicting vertex is added to W_{next}, the work queue of the next iteration. The pseudocode of the net-based conflict resolution algorithm is presented in Algorithm 8

Algorithm 8 D2GC-RemoveConflicts-Net

Input: $G = (V, E)$: a graph to color, $c[.]$: an optimistic coloring.
Output:W_{next}: the work queue for next iteration $c[.]$: an incomplete coloring.

```
1:  W_next ← ∅
2:  for each v ∈ V in parallel do
3:      F ← ∅ : thread private forbidden color set for v
4:      if c[v] ≠ −1 then
5:          F ← F ∪ {c[v]}
6:      for each u ∈ nbor(v) do
7:          if c[u] ≠ −1 then
8:              if c[u] ∈ F then
9:                  W_next ← W_next ∪ {u}
10:             else
11:                 F ← F ∪ {c[u]}
```

Since the net-based algorithms traverse only distance-1 neighborhood instead of the full distance-2 neighborhood, the proposed algorithms are expected to be significantly faster than their traditional counterparts. In other

words, for each vertex v, only the distance-1 neighbors are visited and the complexity is $\mathcal{O}\left(\sum_{v \in V} |\texttt{nbor}(v)|\right)$ which is linear in terms of the size of the graph.

However, despite being faster than their vertex-based counterparts, net-based algorithms require a traversal over the whole graph. On the other hand, vertex-based approaches operate only on the most recent work queue. Since after the early iterations the work queue gets drastically smaller, net-based algorithms become suboptimal. Thus, they are only preferred when the work queue is sufficiently large, and we switch to vertex-based algorithms for later iterations.

4.2 Parallel Algorithms for Bipartite Graph Partial Coloring

Intuitively, BGPC problem is very similar to D2GC with only one difference: the neighborhood is defined differently. In BGPC, given a bipartite graph $G = (V_A \cup V_B, E)$, a valid coloring is obtained by assigning colors to vertices in V_A such that all vertex pairs that are adjacent to at least one vertex in V_B have different colors.

The traditional approach again employs the vertex-based algorithms. Each thread is responsible for one vertex in V_A and traverses the corresponding neighborhood. For clarity concerns, for a vertex v in V_A we will refer its neighbors in V_B as $\texttt{nets}(\mathbf{v})$ and for a vertex u in V_B we will refer its neighbors in V_A as $\texttt{vtxs}(\mathbf{u})$. The vertex-based coloring and conflict removal algorithms are given in Algorithm 9 and Algorithm 10.

Similar to the D2GC problem, the vertex-based algorithms have a quadratic complexity. First, each net $v \in V_B$ is visited $|\texttt{vtxs}(v)|$ times and for each visit, all $|\texttt{vtxs}(v)|$ will be processed. Hence the complexity of the neighborhood traversal of an iteration is $\mathcal{O}\left(\sum_{v \in V_B} |\texttt{vtxs}(v)|^2\right)$. Note that, for the conflict removal phase there can be early terminations, however the given worst-case is tight.

For BGPC, we employ the same net-based idea used in D2GC for both coloring and conflict removal phases. The net-based coloring algorithm pro-

Algorithm 9 BGPC-ColorWorkQueue-Vertex

Input: $G = (V_A \cup V_B, E)$: a bipartite graph, W: vertices to color, $c[.]$: an incomplete coloring with no conflicts.
Output: $c[.]$: an optimistic coloring.

```
1: for each v ∈ W in parallel do
2:     F ← ∅ : thread private forbidden color set for w
3:     for each v ∈ nets(w) do
4:         for each u ∈ vtxs(v) \{w} do
5:             if c[u] ≠ −1 then
6:                 F ← F ∪ {c[u]}
7:     ...                                  ▷ first-fit coloring (lines 6-9 in Alg. 2)
```

Algorithm 10 BGPC-RemoveConflicts-Vertex

Input: $G = (V_A \cup V_B, E)$, W: vertices to color, $\texttt{nbor}(.)$: the neighborhood function, $c[.]$: an optimistic coloring.
Output: W_{next}: the work queue for next iteration, $c[.]$: a (probably incomplete) coloring with no conflicts.

```
1: W_next ← ∅ : a shared queue for the next iter.
2: for each w ∈ W in parallel do
3:     for each v ∈ nets(w) do
4:         for each u ∈ vtxs(v) \{w} do
5:             ...                          ▷ detect conflicts (lines 4-6 in Alg. 3)
```

cesses the vertices in V_B, i.e., the *nets*, in parallel and colors their corresponding adjacency lists. That is achieved by again keeping a thread private *forbidden colors* array. Each color encountered during the traversal is added to the array if it has not been added before. If a vertex has no colors or the color of a vertex is already *forbidden*, then the vertex is marked to be recolored in that iteration. This way, an online conflict removal is also carried out during the coloring phase. After the whole neighborhood is traversed, the vertices marked to be recolored are colored using the *reverse first-fit* strategy mentioned in previous sections.

Similarly, the net-based conflict removal algorithm performs a net-based traversal and marks the conflicting vertices to be colored in the next iteration, again with the help of a thread-private *forbidden colors* array. The pseudocodes for net-based coloring and conflict removal phases are given in Algorithm 11 and Algorithm 12.

Algorithm 11 BGPC-ColorWorkQueue-Net

Input: $G = (V_A \cup V_B, E)$: a bipartite graph, $c[.]$: an incomplete coloring.
Output: $c[.]$: an optimistic coloring array.

```
1: for each v ∈ V_B in parallel do
2:     F ← ∅ : thread private forbidden color set for v
3:     W_local ← ∅ : thread private vertices to be colored
4:     for each u ∈ vtxs(v) do
5:         if c[u] ≠ −1 and c[u] ∉ F then
6:             F ← F ∪ {c[u]}
7:         else
8:             W_local ← W_local ∪ {u}
9:     col ← |vtxs(v)| − 1                          ▷ reverse first-fit coloring
10:    for each u ∈ W_local do
11:        while col ∈ F do
12:            col ← col − 1
13:        c[u] ← col
14:        col ← col − 1
```

The complexity of each iteration of net-based algorithms are linear in terms of the size of the graph ($|V_A \cup V_B| + |E|$). As in the net-based D2GC algorithms, since each net $v \in V_B$ traverses only $\texttt{vtxs}(v)$, the complexity is $\mathcal{O}\left(\sum_{v \in V_B} |\texttt{vtxs}(v)|\right)$.

Algorithm 12 BGPC-RemoveConflicts-Net

Input: $G = (V_A \cup V_B, E)$: a bipartite graph to color, $c[.]$: an optimistic coloring.
Output: $c[.]$: an incomplete coloring.

```
1: for each v ∈ V_B in parallel do
2:     F ← ∅ : thread private forbidden color set for v
3:     for each u ∈ vtxs(v) do
4:         if c[u] ≠ −1 then
5:             if c[u] ∈ F then
6:                 c[u] ← −1
7:             else
8:                 F ← F ∪ {c[u]}
```

4.3 Proposed Algorithms

As mentioned above, the proposed net-based algorithms for coloring and conflict resolution are an order of magnitude faster than their vertex-based counterparts. However since they require a traversal over the whole graph, they become inefficient compared to their vertex-based counterparts when the work queue gets smaller and smaller. Experimental results indicate that 89% of the execution time is spent on the first two iterations on average. Thus, attacking these two iterations results in a significant speedup. Here we propose several algorithms that are obtained by using different net-based and vertex-based algorithm combinations.

- **VV**: Vertex-based coloring with first-fit policy and vertex-based conflict removal for all iterations. This is the traditional approach and is used as a baseline for all other algorithms.
- **VN1**: Vertex-based coloring with net-based conflict removal for just the first iteration.
- **VN2**: Vertex-based coloring with net-based conflict removal for the first two iterations. Empirical results suggest that using net-based conflict removal for the first two iterations is the best configuration. Thus it is adapted for the rest of the algorithms.
- **N1N2**: Net-based coloring for the first iteration with net-based conflict removal for the first two iterations.

- **N2N2**: Net-based coloring for the first two iterations with net-based conflict removal for the first two iterations.

These algorithms are applied to both D2GC and BGPC. Thus in total there are 10 algorithms. The experimental results presented and algorithms are compared in terms of performance in Chapter 6.

4.4 Manycore Implementation for GPUs

The existing literature on D2GC and BGPC focuses on multicore implementations and are limited to vertex-based approaches mentioned in the previous section. The reason is, intuitively both problems as well as the abovemetioned algorithms are hard to adapt to manycore architectures. Here we discuss several technical obstacles that make a straightforward adaptation from the multicore implementations infeasible, then propose solutions to overcome such obstacles.

Parallelism: The parallel speculative coloring algorithm iteratively tries obtaining a valid coloring and resolves the conflicts if there are any. Intuitively, at any iteration, the time spent on coloring depends on the number of *uncolored* vertices, i.e., number of conflicts, at the previous iteration. In other words, number of conflicts obtained at intermediate steps of the execution has a direct impact on the execution time. Thus, an algorithm that generates minimal conflicts at intermediate steps terminates faster.

The conflicts occur when multiple threads assign different colors to the same vertex. Clearly, the possibility of a conflict occurring increases as the number of threads increase [Gebremedhin and Manne, 1999]. This creates a paradox: as the number of threads increases an intermediate coloring phase is executed faster, however the resulting intermediate coloring has more conflicts, thus the overall time increases. So, parallelism is a useful way to speed up coloring but *too much parallelism hurts.*

In the case of GPU implementation, a straightforward adaptation of the CPU algorithm fails as the number of threads on GPU can go up to thousands compared to tens of threads on CPU. Despite being significantly faster than

the CPU implementation for a single iteration, this method generates too many conflicts and the algorithm takes too long to converge.

We propose an approach that lowers the *vertex-level* parallelism while still utilizing the execution power of GPUs. The proposed method assigns a group of GPU threads called *warps* to each vertex, hence its corresponding neighborhood, instead of assigning a single thread to each vertex. This way, the number of vertices that are processed at a given time is decreased while the total number of GPU threads being used remains the same. Specifically, each thread in a warp traverses different parts of the neighborhood of the same vertex and populate a common forbidden colors array. This array is held in the *shared memory* of each CUDA block as I/O operations are much faster compared to the *global memory*. After the forbidden colors are detected, threads cumulatively search the forbidden colors array and find a suitable color. Then the warp skips to the next vertex in the work queue until there are no more vertices left in the queue.

Memory Limitations: As described in previous sections, the first step of both coloring and conflict resolution phases is to determine which colors are already being used in the neighbourhood. This information needs to be stored in order to either select an available color or to resolve conflicts. In the CPU implementations, in order to avoid dynamic memory allocations and deallocations, a two dimensional matrix with t rows and $|V|$ columns is created where t is the number of threads and each row is a thread-private array. When a color is encountered in the neighbourhood of a vertex, the corresponding entry in that thread-private array is marked to indicate that color is forbidden. Apparently, the memory complexity of the *forbidden colors* array is $O(t|V|)$ for t threads and $|V|$ vertices. The memory requirements can be lowered by using $\max_{v \in V}(|\texttt{nbor}(v)|)$ columns instead of $|V|$ columns, but that would increase the computation complexity since it requires a smarter forbidden color marking technique, hence it is not preferred. More specifically, once the memory requirements are lowered, a more sophisticated search mechanism would be needed to find an available color for a vertex. Moreover, $O(t|V|)$ is an acceptable memory complexity for modern architectures, even for graphs with billions of vertices. However, the same idea can not be

applied to the GPUs for two reasons: **1)** GPUs have much less fast shared memory compared to CPUs. **2)** GPUs have many more threads compared to CPUs. Clearly, keeping a thread-private array for each thread or warp is not an option.

A possible solution is using the *global memory* of the GPU device to store the *forbidden colors* array. Today, the global memory a GPU has 2-20 GBs of global memory. While this approach allows using much more space, based on our preliminary experiments, the latency of reading and writing on global memory is too much compared to the CPU latencies. In fact, this approach works significantly slower than the CPU implementations and also generates more conflicts due to the reasons discussed in the previous section.

To overcome the memory limitations, we propose using minimal warp-private arrays that represent only a small portion of the color space. The proposed implementation only considers the colors in a given interval and ignores the others. Empirical results have shown that for most of the vertices in many graphs, the selected limit (which is fine-tuned as 3072) is sufficient to cover the neighborhood. The benefit of this approach is that, the arrays can be small enough to fit into the *shared-memory* of the CUDA blocks which is much faster in terms of I/O latency compared to the global memory. Also, since only *membership queries* will be executed on this array, we allow race conditions. Thus, there is no synchronization overhead.

In Algorithm 13, the *warp level* GPU implementation of the *vertex-based* algorithm is given. For this pseudocode, the keyword $next(.)$ is used to denote fetching the next member from a set. Note that, all the memory accesses are *coalesced* to combine multiple memory accesses into a single operation.

The algorithm starts with an empty, warp-private *forbidden colors* array of size k which is stored in the *shared memory* (line 2). Then, each *warp* fetches a vertex from the work queue. For the fetched vertex, the threads in a single warp traverse the neighborhood; threads visit the distance-1 neighbors in a *coalesced* manner and each thread is responsible for the distance-1 neighborhood of the corresponding neighbor which incurs a burden for high performance. The coalesced memory access pattern is given in Figure 4.1. For each iteration, the array needs to be cleared for reuse. In order to get

Algorithm 13 D2GC-ColorWorkQueue-Warp

Input: $G = (V, E)$: a graph, $c[.]$: an incomplete coloring, W: vertices to color, k: mask size

Output: $c[.]$: the (most) optimistic coloring array.

```
1:  while W ≠ ∅ do
2:      F ← ∅ : warp private, shared set of size k
3:      v ← next(Q)                          ▷ Fetch the next vertex from work queue
4:      for each thread t ∈ warp do in parallel
5:          u ← next(nbor(v))
6:          if c[u] ≠ −1 and c[u] < k then
7:              F ← F ∪ {c[u]}
8:          for each w ∈ nbor(u) do
9:              if c[w] ≠ −1 and c[w] < k then
10:                 F ← F ∪ {c[w]}
11:     ...                                  ▷ cumulative first-fit coloring
```

rid of the clearing overhead the forbidden colors are marked with the corresponding vertex id.

The advantage of employing coalesced memory access is, multiple memory accesses can be combined into a single transaction. Since consecutive threads access consecutive memory locations, every successive 128 bytes can be accessed by a warp in a single transaction. In Figure 4.1, in the first transaction first 32 neighbors are loaded from the memory (yellow). After all the threads in a warp finish their execution, next 32 neighbors are loaded (blue).

As mentioned above, to keep track of the forbidden colors, a small array is used which can fit into the shared memory of GPU blocks. Thus, not all colors can be stored in the forbidden colors array. Instead, only the colors smaller than k, the size of the array, are stored (lines 6 and 9). Despite causing additional conflicts, the performance gained from utilizing the shared memory compensates the time lost for additional conflicts.

Finally, a *cumulative first-fit coloring* is applied. Each thread in a warp starts from a different color index and searches the color space until a valid color is found. Namely, a thread t_i starts the search from the index $\left(i \times \frac{k}{32}\right)$. When a valid color is found, all threads terminate with the help of a shared flag. Again, there is no synchronization overhead as race conditions are

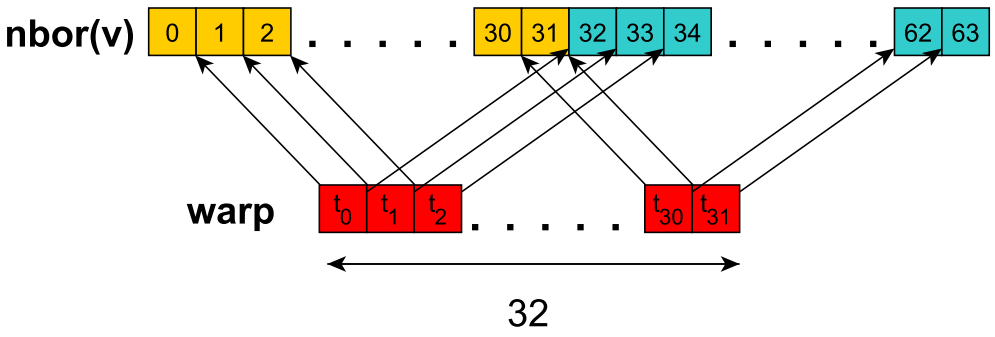


Figure 4.1: Coalesced memory access for a single warp

allowed at this phase.

Unfortunately, a net-based implementation on manycore architectures could not be easily implemented due to high memory requirements and race conditions. Since there is only vertex-based implementations for manycore architectures, two approaches have been adapted.

- **VertexGPU**: Vertex-based coloring on GPU for the first iteration, followed by vertex-based coloring on CPU for the rest of the execution and net-based conflict resolution on CPU for the first two iterations followed by vertex-based conflict resolution on CPU for the rest of the execution.

- **HybridGPU**: Net-based coloring on CPU for the first iteration, followed by vertex-based coloring on GPU for one iteration and vertex-based coloring on CPU for the rest of the execution. Net-based conflict resolution on CPU for the first iteration and vertex-based conflict resolution for the rest of the execution.

Compared to the CPU implementations, VertexGPU and HybridGPU are the manycore counterparts of VN2 and N1N2 described in the previous section.

4.5 Optimizations for Social Networks

For D2GC and BGPC problems, the maximum degree is a trivial lower bound for a valid coloring. In other words, a valid coloring must use at least $\max(|\texttt{nbor}(v)|)$ colors. This requirement proposes an opportunity for social network graphs.

From the structural point of view, social network graphs have a few central vertices with high degrees and many vertices with low degrees [Scott, 1988]. For such graphs, the number of colors to use is determined by a few vertices whereas the low-degree vertices have no impact to the solution quality. Inspired by this observation, the requirements for low-degree vertices can be relaxed to decrease the conflicts observed at intermediate steps, thus the overall execution time. In other words, low-degree vertices can assign more colors to their distance-1 neighborhood without disturbing the final solution quality.

For both D2GC and BGPC, the proposed net-based coloring method employs a *reverse first-fit* coloring strategy in which each vertex $v \in V$ starts the coloring process with $|\texttt{nbor}(v)|$. However as mentioned, this initial number can be increased *as long as it does not exceed the maximum degree*. We propose a heuristic, that takes advantage of this observation to decrease the conflicts at intermediate steps and increase the overall performance. Note that this heuristic is built on top of the net-based coloring described in previous sections.

The proposed heuristic attacks the first, net-based iteration by performing multiple coloring calls before the conflict removal phase. At each call, the initial color for the *reverse first-fit* coloring strategy is increased by a factor e. For example, for an initial color c, when $e = 50\%$ the second iteration starts with an initial color $c' = 1.5 \times c$ and when $e = 100\%$ the second iteration starts with an initial color $c' = 2 \times c$. In the cases where this initial color exceeds the maximum degree, it is set back to the maximum degree so the overall color count is not increased. In Figure 4.2, the impact of this heuristic is demonstrated on a social network graph, `coPapersDBLP`.

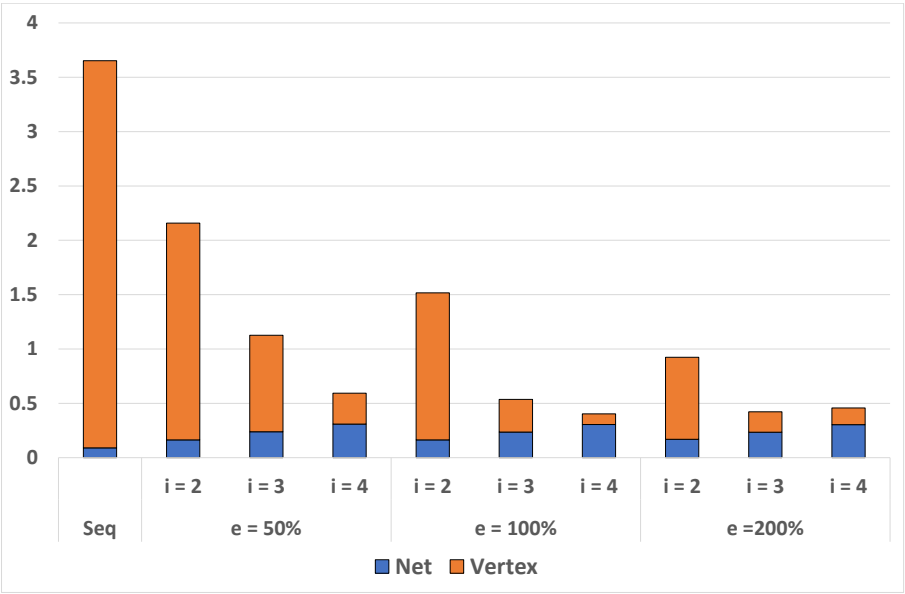


Figure 4.2: Execution timess (in seconds) of the net-based (blue) and vertex-based (orange) phases for a single thread, where i consecutive net-based calls are executed with e increase factor on coPapersDBLP graph.

4.6 Balanced Coloring

As mentioned before, graph coloring has been frequently used to parallelize a large task with many sub-tasks. In our preliminary experiments, the (reverse) first-fit policy generated a few large color sets (of small colors) and thousands of color sets with less than 2 elements for a real-life optimization problem. This result is in concordant with a comprehensive recent study focusing solely on balancing, parallel balancing heuristics, and their practical impacts on parallel computing [Lu et al., 2015]. In fact, on a single multicore CPU socket, the performance reduction (in FLOPS) may not hurt too much since most of the vertices, with small colors, can still be processed in parallel. However, the impact of the imbalance increases with the number of processors/cores. Furthermore, in most of the iterative algorithms, processing only a few vertices and updating the current solution can be harmful from the optimization perspective since this restricts the dimensions of the moves in the search space performed to reach a better solution.

In this work, we experimented on cost-free and unsupervised balancing heuristics within the BGPC and D2GC algorithms proposed above. The straightforward choice would be keeping color set cardinalities dynamically throughout the execution; but this is expensive especially for large number of cores. Instead, we propose two heuristics: the first heuristic tries to keep the number of colors the same as much as possible and the second one aggressively applies balancing hence increases the number of colors (only around 10% on average). The heuristics are given in Algorithms 14 and 15 for the vertex-based approach. The net-based variants are also similar.

In the first balancing heuristic B1, each thread keeps track of the maximum color it uses (col_{max} at line 1). The threads employ the first-fit policy for the odd-numbered vertices (or nets) and otherwise, they employ the reverse first-fit policy starting from col_{max}. Unlike the original BGPC and D2GC algorithms, starting from col_{max}, instead of $|\texttt{nbor}(w)|-1$, necessitates a safety check (line 8). If this is the case, the heuristic initiates a first-fit starting from $col_{max}+1$. By performing alternating policies w.r.t. the vertex (or net) id, B1 hopes to distribute the colors evenly in the interval $[0, col_{max}]$. If there

Algorithm 14 ColorWorkQueue-B1

Input: $G = (V, E)$, W: vertices to color, `nbor`(.): the neighborhood, c[.]: an incomplete coloring with no conflicts.

Output: c[.]: an optimistic coloring.

```
1: col_max ← 0 : thread private
2: for each w ∈ W in parallel do
3:     ...                                          ▷ lines 2-6 of Alg. 2
4:     if w mod 2 = 0 then
5:         col ← col_max
6:         while col ∈ F do
7:             col ← col − 1
8:         if col = −1 then
9:             col ← col_max + 1
10:            while col ∈ F do
11:                col ← col + 1
12:    else
13:        col ← 0
14:        while col ∈ F do
15:            col ← col + 1
16:    c[w] ← col
17:    col_max = max(col_max, col)
```

is no color between this interval, it extends the size of the interval.

The second heuristic B2, given in Algorithm 15, keeps a variable col_{next} in addition to col_{max} to start from for the color search. The idea is the same: the heuristic wants to distribute the colors in between $[0, col_{max}]$ but increments the color to start by one for each vertex/net. To aggressively favor large color numbers and focus the later colors in the interval more, the minimum color to start is set to $col_{max}/3 + 1$ (the last line of Alg. 15). However, filling these color sets with more vertices increases the probability of them being in a forbidden-color array. Thus, more colors are expected to appear during the course of execution due to the conflicting nature of balancing and using less number of colors.

Algorithm 15 ColorWorkQueue-B2

Input: $G = (V, E)$, W: vertices to color, `nbor`(.): the neighborhood, c[.]: an incomplete coloring with no conflicts.
Output: c[.]: an optimistic coloring.

1: $col_{max} \leftarrow 0$: **thread private**
2: $col_{next} \leftarrow 0$: **thread private**
3: **for** each $w \in W$ **in parallel do**
4: ... ▷ lines 2-6 of Alg. 2
5: $col \leftarrow col_{next}$
6: **while** $col \in F$ **do**
7: $col \leftarrow col + 1$
8: **if** $col > col_{max}$ **then**
9: $col \leftarrow 0$
10: **while** $col \in F$ **do**
11: $col \leftarrow col + 1$
12: $c[w] \leftarrow col$
13: $col_{max} = max(col_{max}, col)$
14: $col_{next} = min(col + 1, col_{max}/3 + 1)$

Chapter 5

DATASETS

The algorithms that have been presented in this thesis have different strengths and weaknesses depending on the structures of the input graphs. To fairly present the quality of the algorithms and minimize the bias that can be caused by input graph selection we have selected three sets of input graphs. These sets vary in terms of size, average degree and maximum degree.

5.1 Graphs from Literature

In Table 5.1, the properties of the graphs taken from the coloring literature are presented. The graphs are generated from their corresponding UFL matrices. Duplicate edges and self loops are removed to guarantee a valid coloring. The table presents the basic properties such as number of rows and columns as well as the number of non-zeros. Also, the maximum and the average degree are shown as well as the variance of degree distribution, since they are closely related to the coloring quality and the execution time. Also for each graph, the source is notated by a reference. The graphs are selected with the purpose of increasing variety and demonstrating the flexibility of the algorithms.

5.2 Social Network Graphs

In addition to the graphs taken from the coloring literature, we carried out numerous experiments on social network graphs as well. By definition, so-

Matrix	Properties #rows	#cols	#edges	Column deg. max.	Avg.
af_shell [Patwary et al., 2011]	1,508,065	1,508,065	27,090,195	34	33.92
audikw_1	943,695	943,695	39,297,771	344	81.28
Bump_2911	2,911,419	2,911,419	124,818,480	194	42.87
cage15	5,154,859	5,154,859	99,199,551	46	18.24
channel [Lu et al., 2015]	4,802,000	4,802,000	42,681,372	18	17.77
europe_osm	50,912,018	50,912,018	54,054,660	13	2.12
hollywood-2009	1,139,905	1,139,905	57,515,616	11,467	98.91
indochina-2004	7,414,866	7,414,866	194,109,311	256,425	40.72
kron_g500	1,048,576	1,048,576	44,620,272	131,503	85.1
nlpkkt120	3,542,400	3,542,400	50,194,096	27	26.33
nlpkkt240	27,993,600	27,993,600	401,232,976	27	26.66
Queen_4147	41,471,10	41,47,110	166,823,197	80	78.45

Table 5.1: Properties of the graphs/matrices used in the experiments, taken from the literature

cial network graphs have several large degree vertices and many small degree vertices. Since execution time and the total number of colors are directly affected by the maximum degree, social network graphs are intuitively *harder* to color. Also, the structure of such graphs yield an imbalance over the workloads of threads, making it even harder to have an efficient parallelization. In Table 5.2, the social network graphs used in the experiments are presented, again with basic structural properties.

Matrix	Properties #rows	#cols	#edges	Column deg. max.	Avg.
coPapersDBLP [Lu et al., 2015]	540,486	540,486	15,245,729	3,299	66.23
soc-LiveJournal1	4,847,571	48,47,571	68,993,773	20,333	17.67
soc-pokec-relationships	1,632,803	1,632,803	30,622,564	14,854	27.31
soc-Slashdot0902	82,168	82,168	948,464	2,552	12.27
wikipedia-20051105	1,634,989	1,634,989	19,753,078	75,757	22.67
wiki-topcats	1,791,489	1,791,489	28,511,807	238,342	28.40

Table 5.2: Properties of the graphs used in the social network experiments.

5.3 Random Graphs

In order to fully represent a wide spectrum of graphs, we also have experimented on *synthetically generated* graphs. The graphs are generated using R-MAT graph generator [Chakrabarti and Faloutsos, 2006]. The R-MAT generator generates a graph by recursively dividing the adjacency matrix into four quadrants and placing $|E|$ edges to corresponding quadrants with given probabilities : (a, b, c, d).

We have generated RMATs, using the parameters used in the literature [Çatalyürek et al., 2012]: $(0.25, 0.25, 0.25, 0.25)$, $(0.45, 0.15, 0.15, 0.25)$, $(0.55, 0.15, 0.15, 0.15)$. We adopt the naming conventions suggested by Çatalyürek et al. and name these graphs rmat-er, rmat-g and rmat-b, respectively. To test the scalability, two sets of graphs have been generated with 2^{24} and 2^{25} vertices respectively, both having 2^{27} edges.

In addition to RMATs, three Random Geometric Graphs(RGG) are also used for the experiments on random graphs. RGGs are undirected graphs that are generated by randomly placing N vertices in a geometric space and connecting two nodes with edges if their distance is within a given range. In this work we use graphs with $N \in \{2^{22}, 2^{23}, 2^{24}\}$. In Table 5.3 generated random graphs are presented with their structural properties.

	Properties			Column deg.	
Graph	#rows	#cols	#edges	max.	Avg.
rmat-b	16,777,216	16,777,216	134,217,654	47,060	16.00
rmat-er	16,777,216	16,777,216	134,217,654	42	16.00
rmat-g	16,777,216	16,777,216	134,217,654	1244	16.00
rmat-b2	33,554,432	33,554,432	134,217,654	70,419	16.00
rmat-er2	33,554,432	33,554,432	134,217,654	42	16.00
rmat-g2	33,554,432	33,554,432	134,217,654	1504	16.00
rgg_n_2_22	4,194,304	4,194,304	30,359,198	36	14.74
rgg_n_2_23	8,388,608	8,388,608	63,501,393	40	15.13
rgg_n_2_24	16,777,216	16,777,216	132,557,200	40	15.80

Table 5.3: RMATs and RGGs used for Random Graph experiments.

Chapter 6

RESULTS

All the experiments in the paper are performed on a single machine running on 64 bit CentOS 6.5 equipped with 64GB RAM and a quad-socket Intel Xeon E7-4870 v2 clocked at 2.30 GHz where each socket has 15 cores (60 in total). For the multicore implementations, we used OpenMP and all the codes are compiled with `gcc 4.9.2` with the `-O3` optimization flag enabled. For the GPU implementations, we used `CUDA 7.5` on a Tesla K40C machine with 15 multiprocessors and 192 `CUDA` cores per processor (2880 in total) clocked at 0.75 GHz.

The experiments are presented in four sections: **(1)** Multicore Experiments, **(2)** Manycore Experiments, **(3)** Social Network Graph Experiments, **(4)** Balanced Coloring Experiments.

For all experiments, the sequential VV algorithm is used as the baseline. For the evaluation of the algorithms, we used two parameters: the execution time and the total number of colors, i.e., the quality of the coloring. Also, for the balancing algorithms the variance of the cardinality of color sets is also considered as an indicator of the *balance* of the coloring.

6.1 Multicore Experiments

The execution times of D2GC algorithms for the graphs taken from the coloring literature are given in Figures 6.1- 6.4. The algorithms are shown above the charts. The coloring (red) and the conflict resolution (blue) phases are

color coded for a more clear visualization. Corresponding speedup values over the sequential VV algorithm are presented in Figures 6.5-6.7. The experimental results show that for the aforementioned graphs, a maximum of 401× speedup can be obtained and on average 25× speedup is obtained with the *best* algorithm on 16 threads with around 1% increase on the number of colors on average. The geometric mean of the speedups over a sequential VV algorithm is shown in Table 6.1.

	Speedup over seq. VV Number of Threads				**Color Change(%)** Number of Threads			
Algorithm	$t = 2$	$t = 4$	$t = 8$	$t = 16$	$t = 2$	$t = 4$	$t = 8$	$t = 16$
VV	1.86	3.38	5.79	10.36	**7.25**	12.07	14.47	14.02
VN1	3.13	5.47	9.39	16.39	**6.39**	11.57	16.43	13.75
VN2	3.13	5.33	8.97	15.51	**6.98**	12.40	14.40	14.25
N1N2	**4.34**	**8.27**	**15.27**	**24.74**	8.16	**10.14**	**10.72**	**10.62**
N2N2	3.03	4.97	8.90	14.14	7.94	**9.85**	**10.37**	**11.03**

Table 6.1: Average speedup of the multicore algorithms on graphs taken from the coloring literature for number of threads $t \in \{2, 4, 8, 16\}$ calculated by taking the geometric mean and the average change in the total number of colors for graphs taken from coloring literature. The maximum speedup for each column and the minimum color change for each column is shown in bold. The color changes that are within 1% margin of the minimum are also shown in bold.

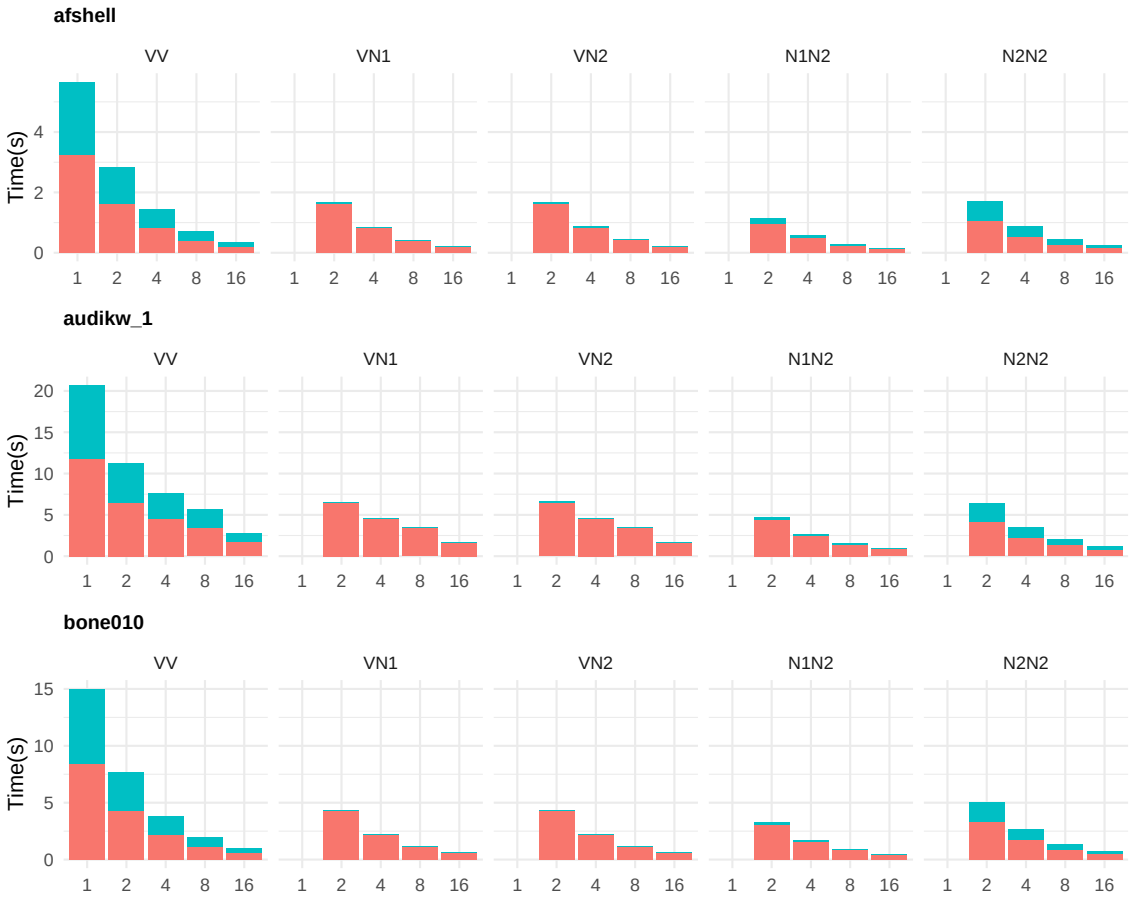


Figure 6.1: The execution times for the multicore algorithms on the graphs taken from coloring literature. The y-axis denotes the time in seconds and the x-axis denotes the number of threads. The algorithms are denoted above the bars. Red color shows the execution time of the coloring phase and the blue color shows the execution time of the conflict resolution phase.

Figure 6.2: The execution times for the multicore algorithms on the graphs taken from coloring literature. The y-axis denotes the time in seconds and the x-axis denotes the number of threads. The algorithms are denoted above the bars. Red color shows the execution time of the coloring phase and the blue color shows the execution time of the conflict resolution phase.

Figure 6.3: The execution times for the multicore algorithms on the graphs taken from coloring literature. The y-axis denotes the time in seconds and the x-axis denotes the number of threads. The algorithms are denoted above the bars. Red color shows the execution time of the coloring phase and the blue color shows the execution time of the conflict resolution phase.

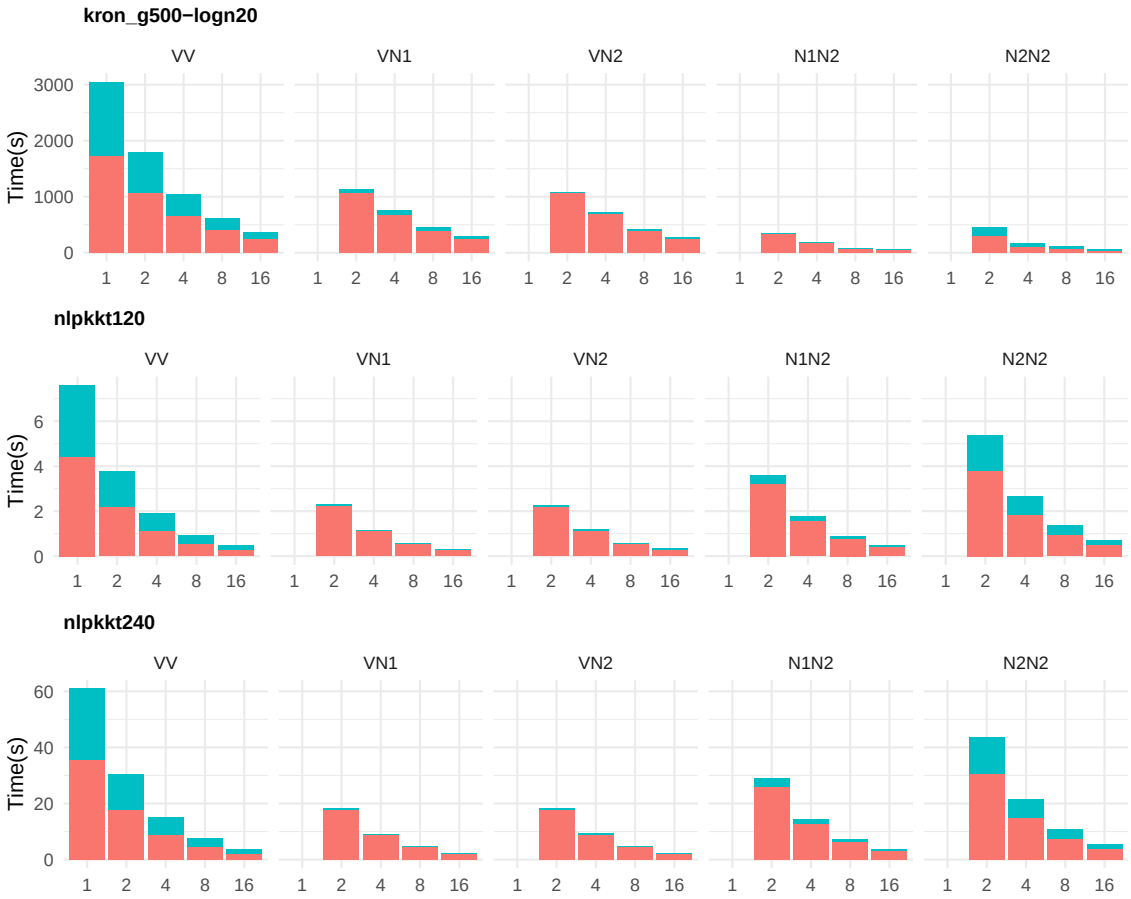


Figure 6.4: The execution times for the multicore algorithms on the graphs taken from coloring literature. The y-axis denotes the time in seconds and the x-axis denotes the number of threads. The algorithms are denoted above the bars. Red color shows the execution time of the coloring phase and the blue color shows the execution time of the conflict resolution phase.

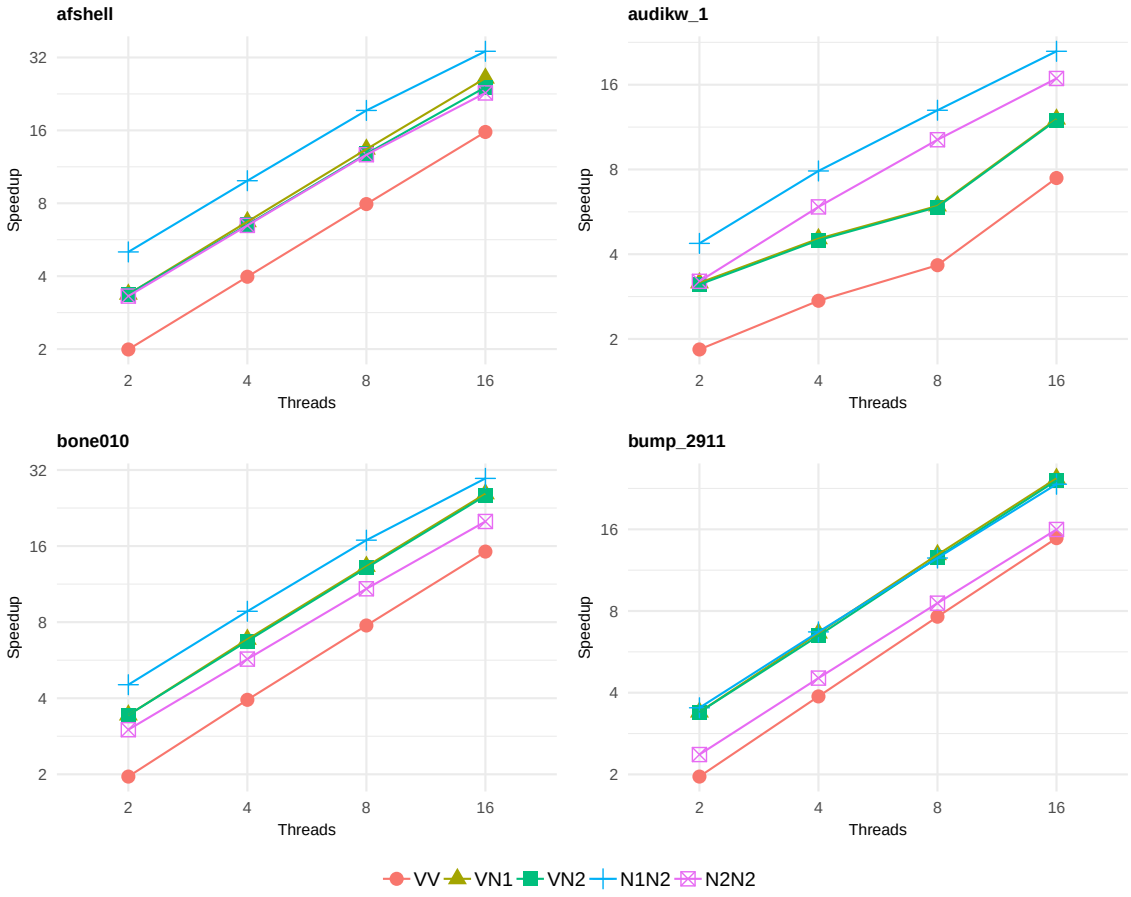


Figure 6.5: The speedup values for the multicore algorithms over the sequential VV algorithm on the matrices taken from coloring literature. The y-axis denotes the speedup values and the x-axis denotes the number of threads.

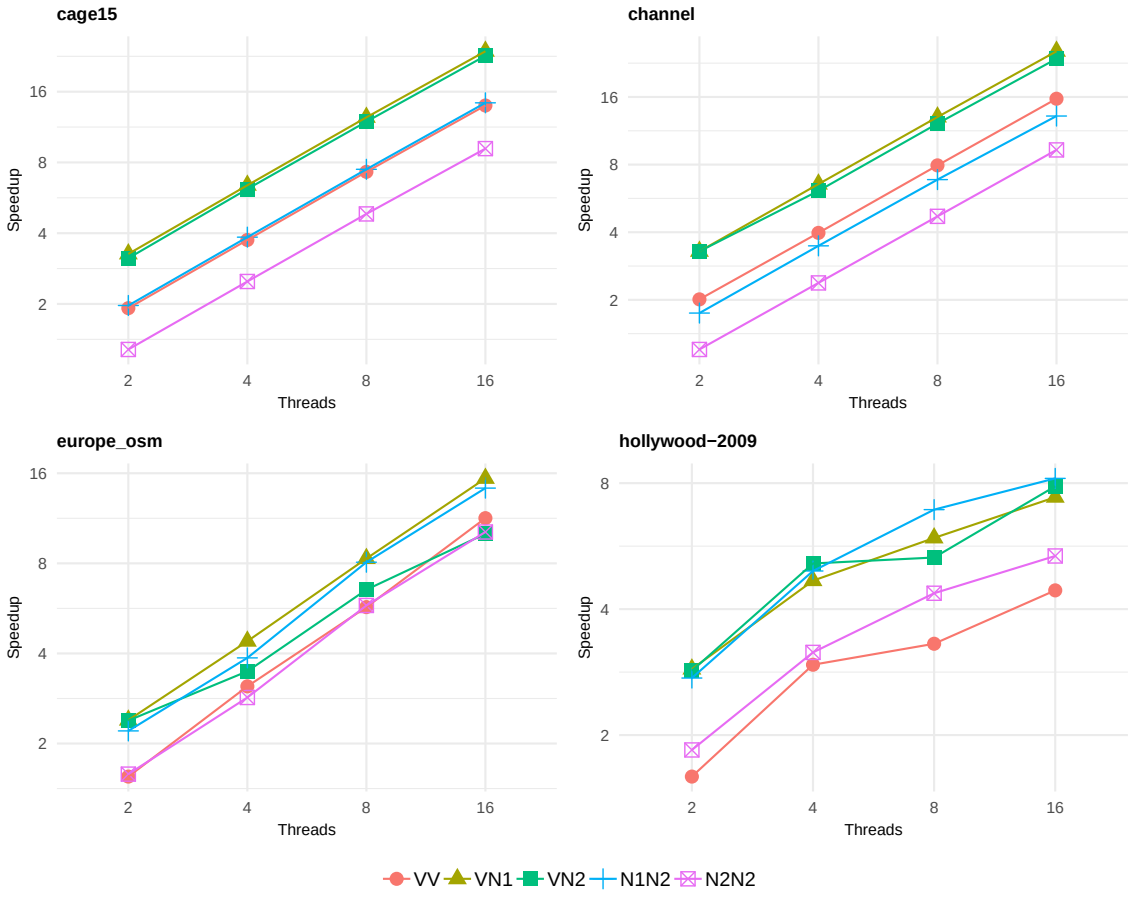


Figure 6.6: The speedup values for the multicore algorithms over the sequential VV algorithm on the matrices taken from coloring literature. The y-axis denotes the speedup values and the x-axis denotes the number of threads.

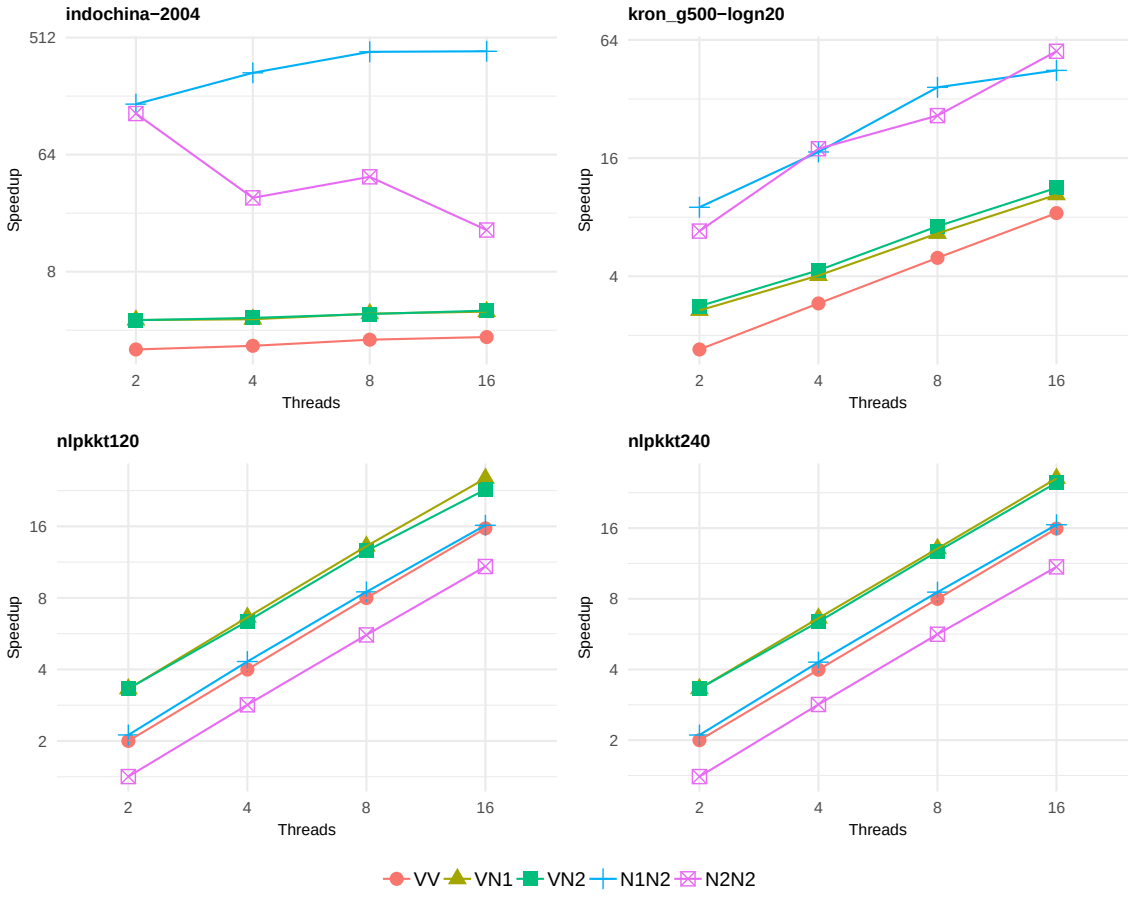


Figure 6.7: The speedup values for the multicore algorithms over the sequential VV algorithm on the matrices taken from coloring literature. The y-axis denotes the speedup values and the x-axis denotes the number of threads.

In Figures 6.8-6.10, the execution times for all the algorithms on random graphs are given. The x-axis is the number of threads and y-axis is the execution time in seconds. The coloring and conflict detection phases are color coded. The corresponding speedup values are presented in Figures 6.11-6.13. The geometric mean of the speedup of each algorithm is presented in Table 6.2

	Speedup over seq. VV Number of Threads				**Color Change(%)** Number of Threads			
Algorithm	$t = 2$	$t = 4$	$t = 8$	$t = 16$	$t = 2$	$t = 4$	$t = 8$	$t = 16$
VV	1.54	2.67	4.24	6.88	**0.02**	**0.91**	**0.66**	**1.83**
VN1	2.45	3.78	6.82	11.09	**0.24**	**0.75**	**0.49**	**1.83**
VN2	2.45	3.85	6.69	10.77	**0.39**	**0.75**	**0.64**	**1.83**
N1N2	**3.92**	**6.13**	**11.32**	**19.24**	3.20	4.11	2.88	3.31
N2N2	2.65	4.29	7.77	13.02	3.58	2.63	2.33	3.03

Table 6.2: Average speedup of the multicore algorithms on random graphs for number of threads $t \in \{2, 4, 8, 16\}$ calculated by taking the geometric mean and the average change in the total number of colors for random graphs. The maximum speedup for each column and the minimum color change for each column is shown in bold. The color changes that are within 1% margin of the minimum are also shown in bold.

Figure 6.8: The execution times for the multicore algorithms on random graphs. The y-axis denotes the time in seconds and the x-axis denotes the number of threads. The algorithms are denoted above the bars. Red color shows the execution time of the coloring phase and the blue color shows the execution time of the conflict resolution phase.

Figure 6.9: The execution times for the multicore algorithms on random graphs. The y-axis denotes the time in seconds and the x-axis denotes the number of threads. The algorithms are denoted above the bars. Red color shows the execution time of the coloring phase and the blue color shows the execution time of the conflict resolution phase.

Figure 6.10: The execution times for the multicore algorithms on random graphs. the y-axis denotes the time in seconds and the x-axis denotes the number of threads. The algorithms are denoted above the bars. Red color shows the execution time of the coloring phase and the blue color shows the execution time of the conflict resolution phase.

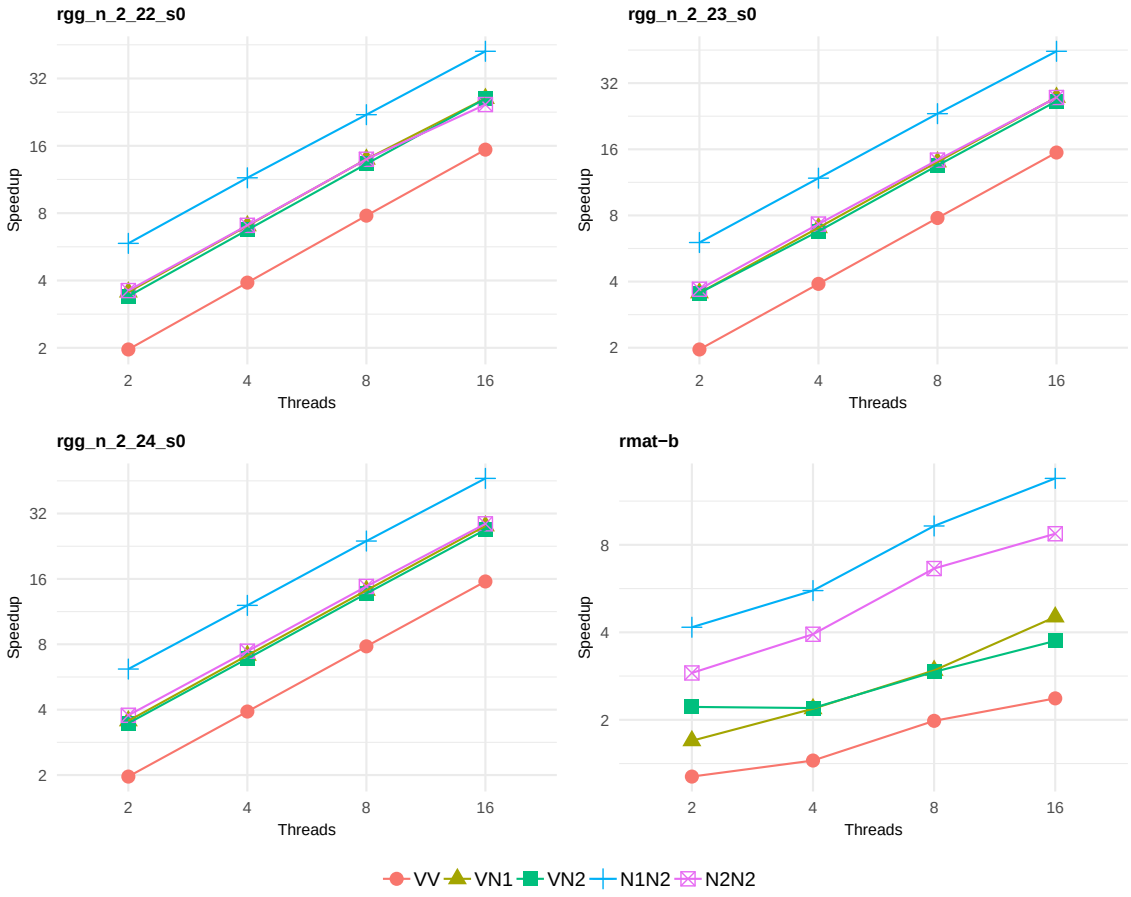


Figure 6.11: The speedup values for the multicore algorithms over the sequential VV algorithm on random graphs. The y-axis denotes the speedup values and the x-axis denotes the number of threads.

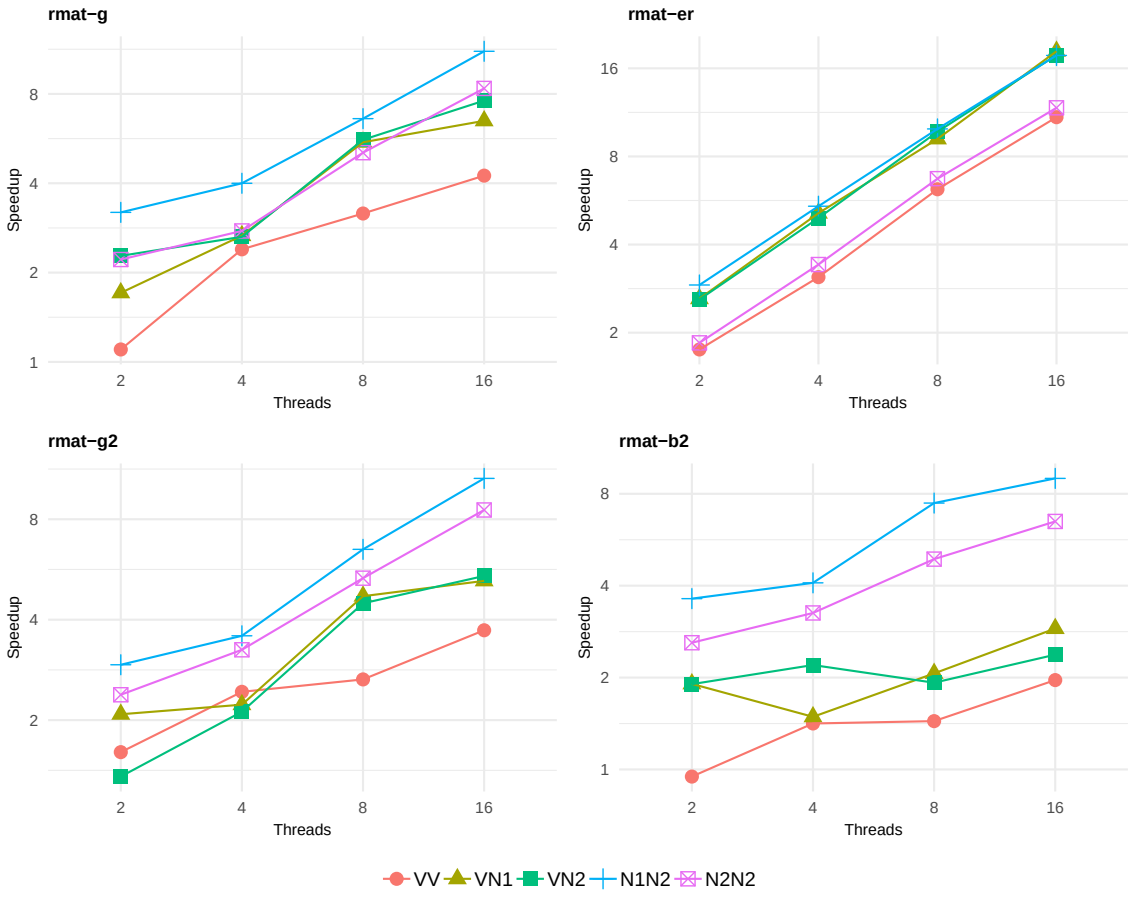


Figure 6.12: The speedup values for the multicore algorithms over the sequential VV algorithm on random graphs. The y-axis denotes the speedup values and the x-axis denotes the number of threads.

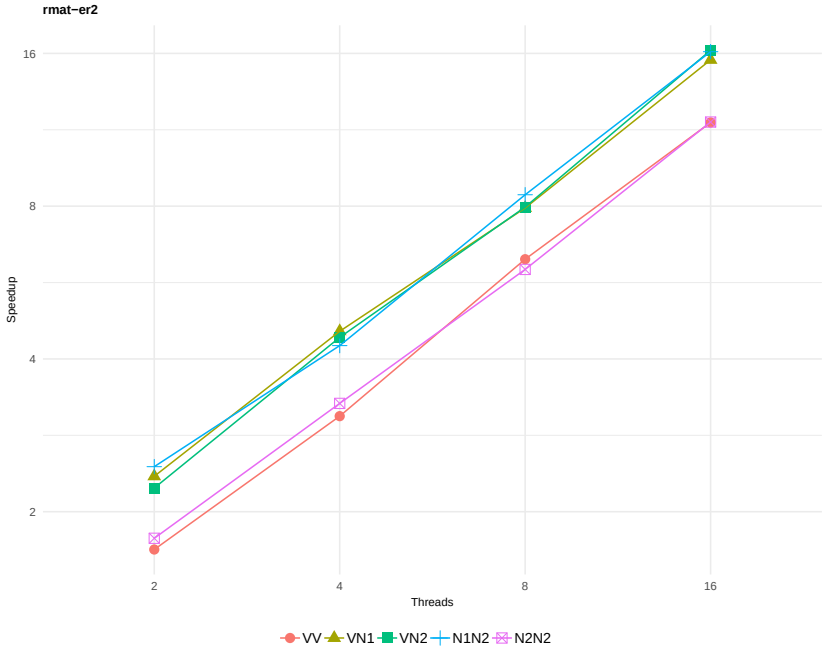


Figure 6.13: The speedup values for the multicore algorithms over the sequential VV algorithm on random graphs. The y-axis denotes the speedup values and the x-axis denotes the number of threads.

Last, the execution times of all the algorithms on social network graphs are presented in Figures 6.14-6.15 and the corresponding speedup values are shown in Figures 6.16-6.17. Table 6.3 summarizes the average speedup and increase in number of colors for social network graphs.

	Speedup over seq. VV Number of Threads				**Color Change(%)** Number of Threads			
Algorithm	$t = 2$	$t = 4$	$t = 8$	$t = 16$	$t = 2$	$t = 4$	$t = 8$	$t = 16$
VV	1.26	1.65	2.26	3.48	**0.03**	**0.02**	**0.01**	**0.02**
VN1	2.01	2.62	3.61	5.37	**-0.07**	**-0.04**	**-0.01**	**0.16**
VN2	2.06	2.57	3.63	5.52	**-0.06**	**-0.06**	**-0.02**	**0.16**
N1N2	**6.64**	**10.31**	**18.42**	**25.25**	1.09	**0.76**	**0.96**	**0.84**
N2N2	4.46	6.23	9.91	18.17	**0.49**	**0.71**	**0.45**	**0.37**

Table 6.3: Average speedup of the multicore algorithms for number of threads $t \in \{2, 4, 8, 16\}$ on social network graphs, calculated by taking the geometric mean and the average change in the total number of colors for social network graphs. The maximum speedup for each column and the minimum color change for each column is shown in bold. The color changes that are within 1% margin of the minimum are also shown in bold.

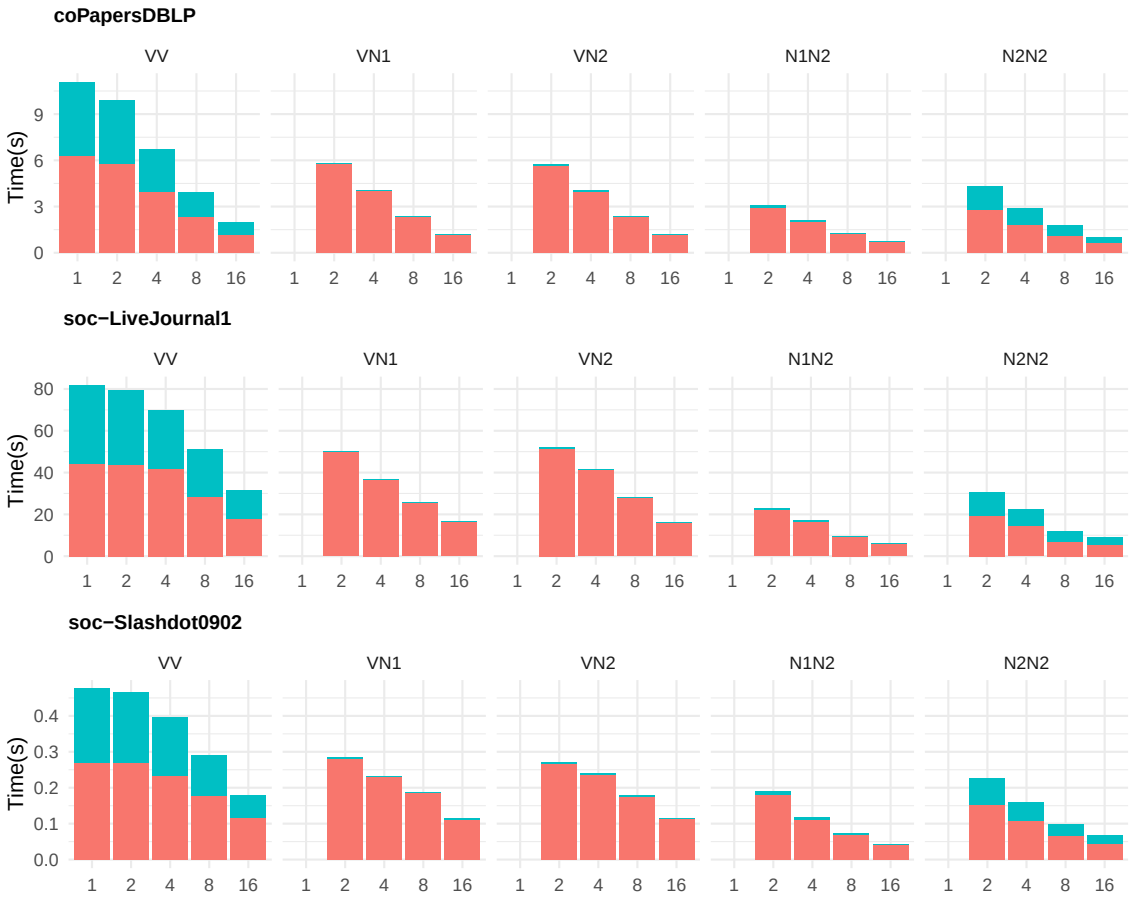


Figure 6.14: The execution times for the multicore algorithms on social network graphs. The y-axis denotes the time in seconds and the x-axis denotes the number of threads. The algorithms are denoted above the bars. Red color shows the execution time of the coloring phase and the blue color shows the execution time of the conflict resolution phase.

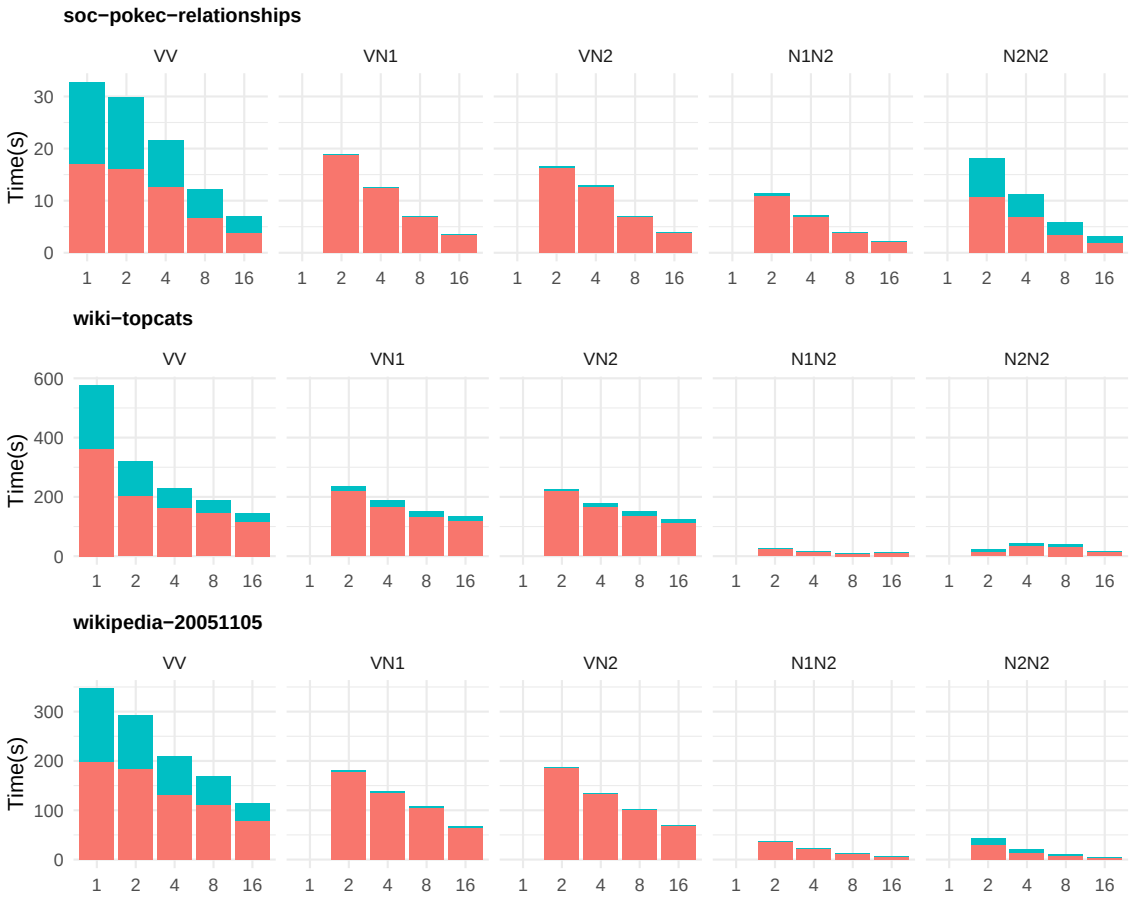


Figure 6.15: The execution times for the multicore algorithms on social network graphs. The y-axis denotes the time in seconds and the x-axis denotes the number of threads. The algorithms are denoted above the bars. Red color shows the execution time of the coloring phase and the blue color shows the execution time of the conflict resolution phase.

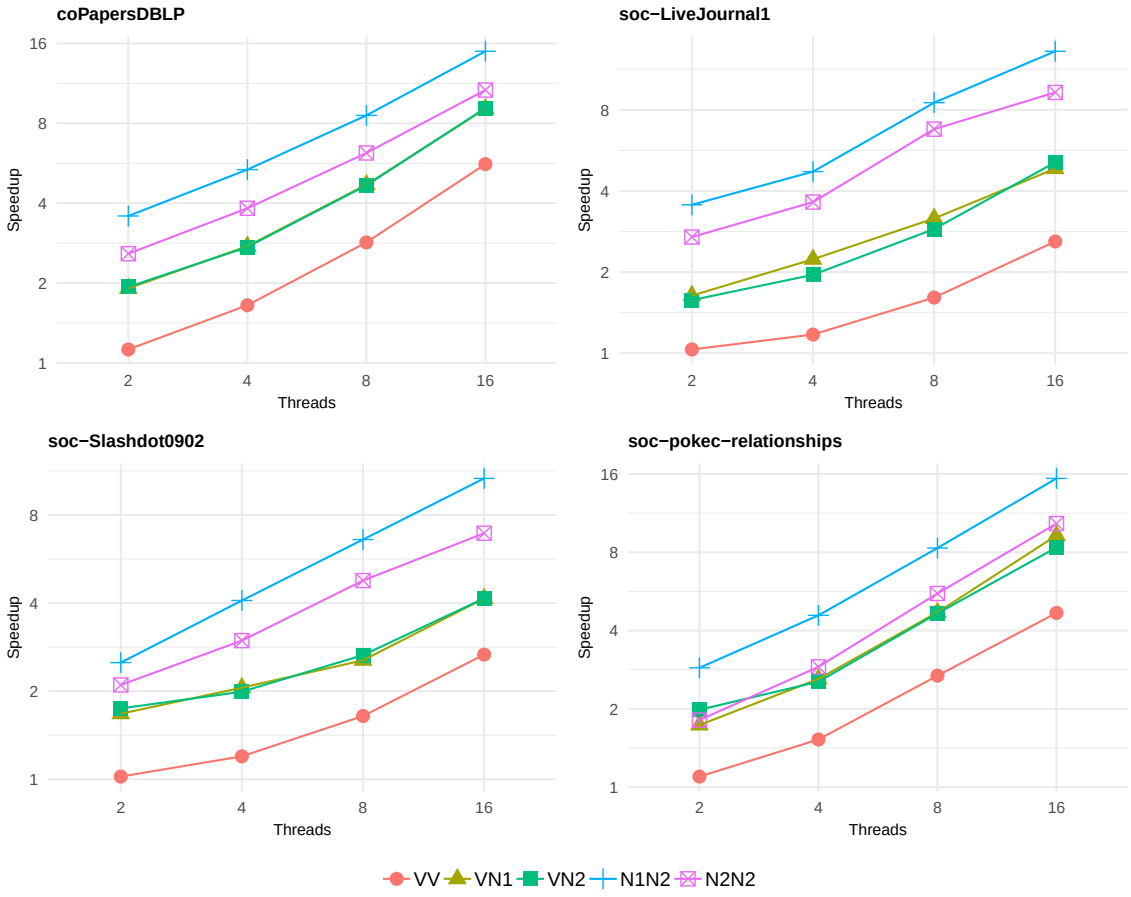


Figure 6.16: The speedup values for the multicore algorithms over the sequential VV algorithm on social network graphs. The y-axis denotes the speedup values and the x-axis denotes the number of threads.

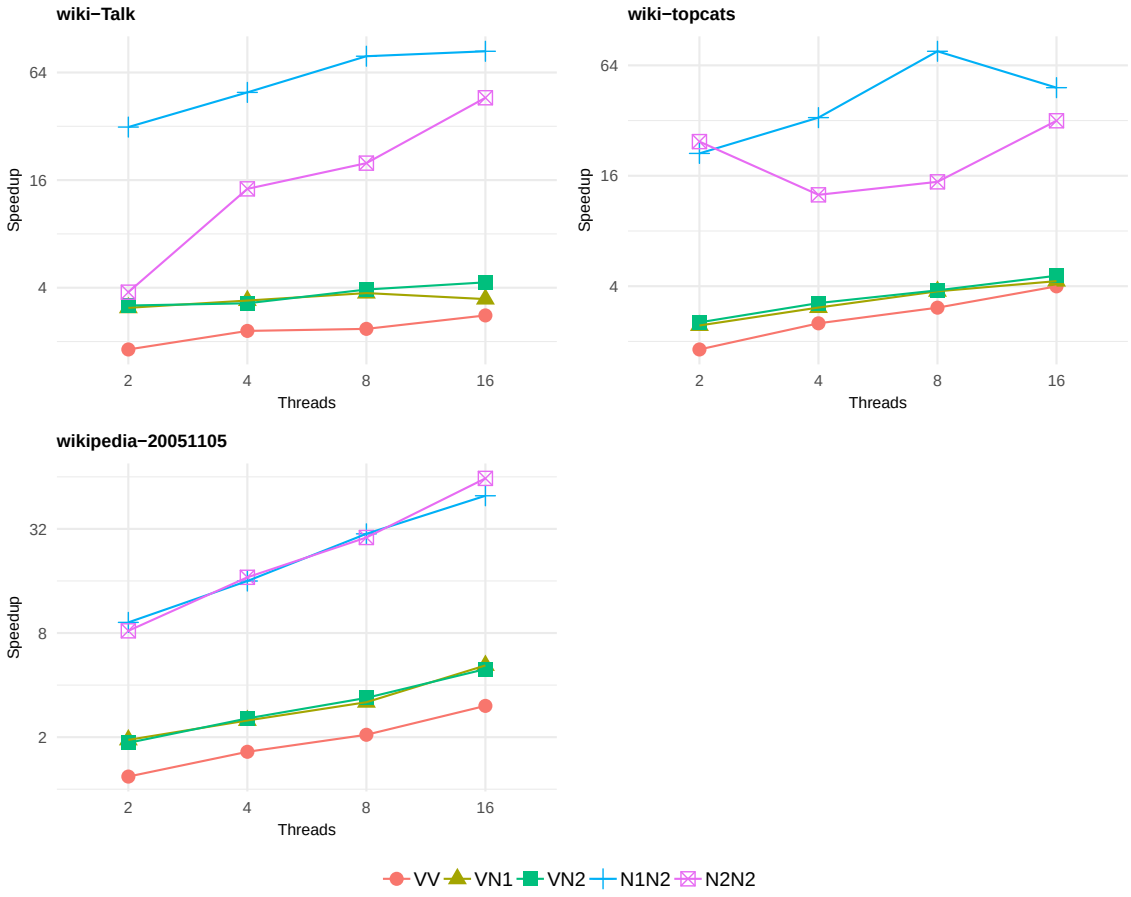


Figure 6.17: The speedup values for all the algorithms over the sequential VV algorithm. The y-axis denotes the speedup values and the x-axis denotes the number of threads.

6.2 Manycore Experiments

As explained before, we propose two algorithms for manycore architectures: **VertexGPU** and **HybridGPU**. These algorithms are implemented on top of $VN2$ and $N1N2$ respectively, with the first vertex-based coloring carried out on GPU thus the baseline is selected as $VN2$ and $N1N2$, respectively. Note that for all multicore baselines $t = 16$ is used. The execution times of manycore experiments are given in Figures 6.18-6.22 for 16 threads. The coloring (red) and the conflict resolution (blue) phases are color coded.

The experimental results show that manycore algorithms are not suitable for all the graphs. Since manycore algorithms require high degrees and large graphs to utilize the large number of threads, the graphs taken from the coloring literature can not fully utilize the computation power of GPUs. In such cases, the GPU execution takes longer than its CPU counterpart. However, for graphs showing social network characteristics, i.e., smallworld structure and power-law degree distribution, proposed algorithms obtain a maximu of 4.00× and 5.84× speedup over their multicore counterparts. On average, manycore algorithms are 1.50× and 1.32× faster than their multicore counterparts without any change in the number of total colors. Despite being significantly faster than their multicore counterparts per iteration, manycore algorithms can still generate more conflicts because of higher parallelism, which increases the total execution time.

Figure 6.18: The execution times for manycore algorithms on random graphs, side-by-side with their multicore counterparts. Y-axis denotes the time in seconds and x-axis denotes the algorithms executed with $t = 16$ threads. Red color shows the execution time of the coloring phase and the blue color shows the execution time of the conflict resolution phase.

Figure 6.19: The execution times for manycore algorithms on random graphs, side-by-side with their multicore counterparts. The y-axis denotes the time in seconds and the x-axis denotes the algorithms executed with $t = 16$ threads. Red color shows the execution time of the coloring phase and the blue color shows the execution time of the conflict resolution phase.

Figure 6.20: The execution times for manycore algorithms on random graphs, side-by-side with their multicore counterparts. The y-axis denotes the time in seconds and the x-axis denotes the algorithms executed with $t = 16$ threads. Red color shows the execution time of the coloring phase and the blue color shows the execution time of the conflict resolution phase.

Figure 6.21: The execution times for manycore algorithms on social network graphs, side-by-side with their multicore counterparts. The y-axis denotes the time in seconds and the x-axis denotes the algorithms executed with $t = 16$. Red color shows the execution time of the coloring phase and the blue color shows the execution time of the conflict resolution phase.

Figure 6.22: The execution times for manycore algorithms on social network graphs, side-by-side with their multicore counterparts. The y-axis denotes the time in seconds and the x-axis denotes the algorithms executed with $t = 16$. Red color shows the execution time of the coloring phase and the blue color shows the execution time of the conflict resolution phase.

6.3 Social Network Experiments

The social network optimization experiments are only carried out on the social network dataset. The experiments are carried out with $i \in \{2, 3, 4\}$ consecutive net-based coloring calls and $e \in \{50, 100, 200\}$ increase factor with $t = 16$ threads. The baseline is selected as the $N1N2$ algorithm since the proposed optimizations are implemented on top of that algorithm. The speedup over the sequential VV algorithm is also presented to demonstrate the overall impact. The execution times of social network experiments are given in Figures 6.23-6.24 for 16 threads. In each chart the first bar shows the baseline execution. The experimental results show that for social network graphs, a maximum of 174× speedup can be obtained and on average 60× speedup is obtained with the *best* algorithm on 16 threads over the sequential VV algorithm with 1.56% increase on the number of colors. The geometric mean of the speedups over a sequential VV algorithm is shown in Table 6.4 and the average change in the total number of colors is shown in Table 6.5.

	$i = 2$			$i = 3$			$i = 4$		
τ	$e = 50$	$e = 100$	$e = 200$	$e = 50$	$e = 100$	$e = 200$	$e = 50$	$e = 100$	$e = 200$
2	7.62	8.57	11.74	10.57	15.44	20.03	17.69	19.79	22.53
4	13.02	12.55	15.21	14.05	23.60	22.66	18.02	22.99	30.66
8	15.94	19.03	21.39	22.95	28.82	35.35	27.77	33.56	35.33
16	25.07	38.17	35.32	36.38	44.30	**59.99**	40.79	49.81	45.91

Table 6.4: Average speedup of social network experiments over sequential VV algorithm where $i \in \{2, 3, 4\}$ consecutive net-based coloring calls are executed with $e \in \{50, 100, 200\}$ increase factor on $\tau \in \{2, 4, 8, 16\}$ threads.

	$i = 2$			$i = 3$			$i = 4$		
τ	$e = 50$	$e = 100$	$e = 200$	$e = 50$	$e = 100$	$e = 200$	$e = 50$	$e = 100$	$e = 200$
2	11.19	10.92	7.94	8.40	6.06	5.10	4.79	5.24	4.64
4	6.14	7.97	7.05	8.53	3.84	7.11	8.74	7.97	3.64
8	8.24	6.76	8.55	6.07	7.01	5.18	6.56	8.94	9.54
16	4.92	2.50	5.17	4.15	5.08	**1.56**	6.49	10.42	10.70

Table 6.5: Average increase in the number of colors (%) for social network experiments over sequential VV algorithm where $i \in \{2, 3, 4\}$ consecutive net-based coloring calls are executed with $e \in \{50, 100, 200\}$ increase factor on $\tau \in \{2, 4, 8, 16\}$ threads.

	$i = 2$			$i = 3$			$i = 4$		
	$e = 50$	$e = 100$	$e = 200$	$e = 50$	$e = 100$	$e = 200$	$e = 50$	$e = 100$	$e = 200$
Speedup	1.26	1.90	1.97	1.95	2.71	**3.91**	2.58	3.65	3.40
Col. Change	0.08	-0.42	0.09	-0.09	-0.48	**-0.29**	-0.06	-0.12	-0.03

Table 6.6: Average speedup and color change (%) for social network experiments over N1N2 algorithm where $i \in \{2, 3, 4\}$ consecutive net-based coloring calls are executed with $e \in \{50, 100, 200\}$ increase factor. Both algorithms are executed on 16 threads.

Figure 6.23: The execution times for social network experiments with $i \in \{2, 3, 4\}$ consecutive net-based calls are executed with $e \in \{50, 100, 200\}$ increase factor and 16 threads. The numbers above the bars show i, the number of consecutive net-based calls, and the numbers below the chart show e, the increase factor.

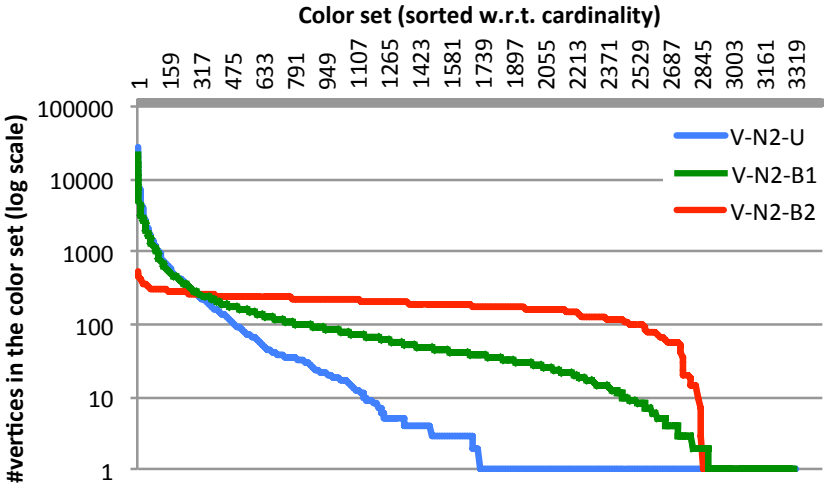
Figure 6.24: The execution times for social network experiments with $i \in \{2, 3, 4\}$ consecutive net-based calls are executed with $e \in \{50, 100, 200\}$ increase factor and 16 threads. The numbers above the bars show i, the number of consecutive net-based calls, and the numbers below the chart show e, the increase factor.

6.4 Experiments on Balancing

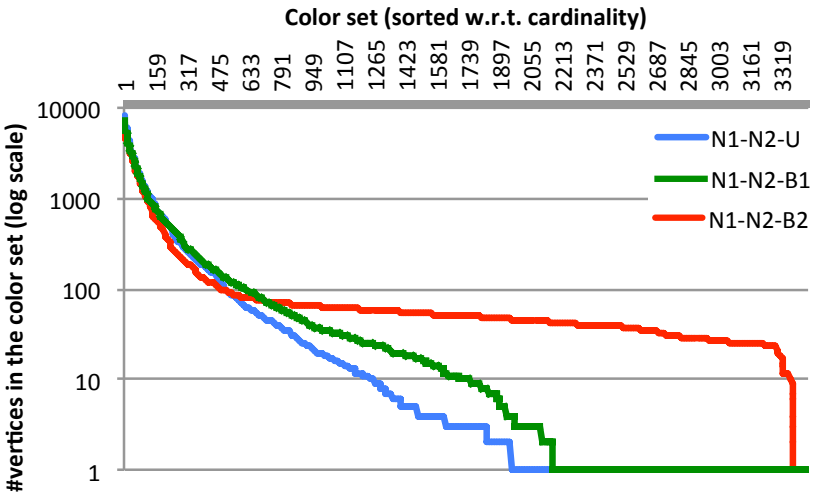
The balanced coloring experiments are carried out with $B1$ and $B2$ balancing heuristics on top of $VN2$ and $N1N2$ algorithms, respectively. Thus, the baseline for these experiments are selected as $VN2$ and $N1N2$. The *balance* is measured as the standard deviation of the cardinalities of color sets. The impact of balancing heuristics B1 and B2 are presented in Table 6.7 for D2GC experiments. The heuristics are applied to VN2 and N1N2 and the results are compared with their original implementation. Experimental results show that, applying these heuristics is for free, i.e., there is no computational overhead as expected. For B1, the standard deviation of the color cardinalities decreases to 0.69× and 0.84× of the initial result when applied to VN2 and N1N2, respectively, on the expense of 4% color increase. For B2, which aggressively tries to reduce the number of colors, the standard deviation decreases 0.25× and 0.62× with around 9% and 13% increase on the number of colors for V-N2 and N1-N2, respectively. To better visualize the impact of these balancing heuristics, Figure 6.25 shows the distribution of color set cardinalities for the original and balanced executions of VN2 and N1N2 on coPapersDBLP.

	Normalized w.r.t. X-N2			
Algorithm	Coloring time	#Color sets	Average card.	Std. Dev.
V-N2-U	1.00	1.00	1.00	1.00
V-N2-B1	0.95	1.04	0.96	0.69
V-N2-B2	0.95	1.13	0.89	0.25
N1-N2-U	1.00	1.00	1.00	1.00
N1-N2-B1	0.99	1.04	0.96	0.84
N1-N2-B2	0.99	1.09	0.91	0.62

Table 6.7: Impact of balancing heuristics, B1 and B2, on the color set cardinalities and the number of color sets for parallel D2GC algorithms VN2 and N1N2 on 16 threads. Results are normalized with the original unbalanced algorithms denoted with -U.



(a) Balancing coPapersDBLP for VN2



(b) Balancing coPapersDBLP for N1N2

Figure 6.25: Impact of balancing heuristics, B1 and B2, on the color set cardinalities and the number of color sets for D2GC algorithms parallel VN2 (left) and N1N2 (right) on 16-threads for coPapersDBLP.

Chapter 7

CONCLUSION

In this thesis, we propose novel, greedier and more optimistic parallel algorithms for parallel BGPC and D2GC on multicore architectures. Proposed algorithms outperform the state-of-the-art algorithms without decreasing the solution quality, i.e., increasing the number of colors too much. Namely, a maximum of 401× and an average of 25× speedup can be obtained with the proposed algorithms with just 1% increase on the total number of colors. To demonstrate that the efficiency of the proposed algorithms is not dependent on the structure of the graphs, we have gathered three datasets with different structural properties.

We then extended our work to manycore architectures and presented number of challenges that make manycore implementations hard as well as the solutions to overcome these challenges. Proposed manycore algorithms yield a maximum of 5.84 and an average of 1.50 speedup over the *best* multicore algorithms. Experimental results indicate that, manycore algorithms are suitable for large graphs containing high-degree nodes.

We also propose several optimization heuristics for social network graphs to further increase the performance. Proposed heuristics take advantage of the structural properties of social network graphs to obtain a maximum of 174× and an average of $60x$ speedup for social network graphs with just 1.56% increase on the total number of colors.

Finally, we propose two costless balancing heuristics that can be applied to both BGPC and D2GC, as well as other coloring variants, to balance

the color set cardinalities and improve the impact of the coloring on the real application to be parallelized. The results show that the proposed techniques are useful in practice and improves the performance and the *goodness* of the coloring.

The proposed techniques are suitable for Intel Xeon Phi architecture which will be considered in future works. which can be a comfort while parallelizing the coloring algorithms on manycore architectures. We also believe that a better net-based coloring and a better cost-free, self-balancing heuristic are worth investigating since experimental results indicate that their impact will be significant. Lastly, the optimistic techniques for BGPC and D2GC can be extended to the distance-k graph coloring problem and further performance improvements can be investigated.

Bibliography

[Allwright et al., 1994] Allwright, J., Bordawekar, R., Coddington, P. D., Dincer, K., and Martin, C. (1994). A comparison of parallel graph coloring algorithms. Technical Report SCCS-666, Northeast Parallel Architectures Center at Syracuse University (NPAC).

[Boman et al., 2005] Boman, E., Bozdağ, D., Çatalyürek, Ü., Gebremedhin, A., and Manne, F. (2005). A scalable parallel graph coloring algorithm for distributed memory computers. In *Proc. of 11th Int'l. Euro-Par Conf. on Parallel Processing*, pages 241–251.

[Bozdağ et al., 2005] Bozdağ, D., Çatalyürek, Ü., Gebremedhin, A., Manne, F., Boman, E., and Özgüner, F. (2005). A parallel distance-2 graph coloring algorithm for distributed memory computers. In *Proc. of 1st Int'l. Conf. on High Performance Computing and Communications*, pages 796–806. Springer.

[Bozdağ et al., 2010] Bozdağ, D., Çatalyürek, Ü., Gebremedhin, A., Manne, F., Boman, E., and Özgüner, F. (2010). Distributed-memory parallel algorithms for distance-2 coloring and related problems in derivative computation. *SIAM Journal of Scientific Computing*, 32(4):2418–2446.

[Bozdağ et al., 2008] Bozdağ, D., Gebremedhin, A., Manne, F., Boman, E., and Çatalyürek, Ü. (2008). A framework for scalable greedy coloring on distributed memory parallel computers. *Journal of Parallel and Distributed Computing*, 68(4):515–535.

[Brélaz, 1979] Brélaz, D. (1979). New methods to color the vertices of a graph. *Commun. ACM*, 22:251–256.

[Çatalyürek et al., 2012] Çatalyürek, Ü. V., Feo, J., Gebremedhin, A. H., Halappanavar, M., and Pothen, A. (2012). Graph coloring algorithms for

multi-core and massively multithreaded architectures. *Parallel Computing*, 38(10-11):576–594.

[Çatalyürek et al., 2012] Çatalyürek, Ü. V., Feo, J., Gebremedhin, A. H., Halappanavar, M., and Pothen, A. (2012). Graph coloring algorithms for multi-core and massively multithreaded architectures. *Parallel Computing*, 38(10):576 – 594.

[Chakrabarti and Faloutsos, 2006] Chakrabarti, D. and Faloutsos, C. (2006). Graph mining: Laws, generators, and algorithms. *ACM Comput. Surv.*, 38(1).

[Coleman and More, 1983] Coleman, T. F. and More, J. J. (1983). Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM Journal on Numerical Analysis*, 1(20):187–209.

[Culberson, 1992] Culberson, J. C. (1992). Iterated greedy graph coloring and the difficulty landscape. Technical Report TR 92-07, University of Alberta.

[Deveci et al., 2016] Deveci, M., Boman, E. G., Devine, K. D., and Rajamanickam, S. (2016). Parallel graph coloring for manycore architectures. In *2016 IEEE Parallel and Distributed Processing Symposium (IPDPS)*, pages 892–901.

[Gebremedhin and Manne, 1999] Gebremedhin, A. H. and Manne, F. (1999). Parallel graph coloring algorithms using OpenMP (extended abstract). In *In First European Workshop on OpenMP*, pages 10–18.

[Gebremedhin et al., 2002] Gebremedhin, A. H., Manne, F., and Pothen, A. (2002). Parallel distance-k coloring algorithms for numerical optimization. In *Euro-Par 2002 Parallel Processing - 8th International Conference*, pages 912–921.

[Gebremedhin et al., 2005] Gebremedhin, A. H., Manne, F., and Pothen, A. (2005). What color is your jacobian? Graph coloring for computing derivatives. *SIAM Review*, 47(4):629–705.

[Gebremedhin et al., 2013] Gebremedhin, A. H., Nguyen, D., Patwary, M. M. A., and Pothen, A. (2013). ColPack: Software for graph coloring and related problems in scientific computing. *ACM Trans. Math. Softw.*, 40(1):1:1–1:31.

[Gjertsen Jr. et al., 1996] Gjertsen Jr., R. K., Jones, M. T., and Plassmann, P. (1996). Parallel heuristics for improved, balanced graph colorings. *Journal on Parallel and Dist. Computing*, 37:171–186.

[Hajnal and Szemeredi, 1970] Hajnal, A. and Szemeredi, E. (1970). Proof of a conjecture of p. erdos. *London: North-Holland*, pages 601–623.

[Jones and Plassmann, 1993] Jones, M. and Plassmann, P. (1993). A parallel graph coloring heuristic. *SIAM Journal on Scientific Computing*, 14(3):654–669.

[Lu et al., 2015] Lu, H., Halappanavar, M., Chavarr-a-Miranda, D., Gebremedhin, A., and Kalyanaraman, A. (2015). Balanced coloring for parallel computing applications. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE*, pages 7–16.

[Luby, 1986] Luby, M. (1986). A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15(4):1036–1053.

[Matula, 1968] Matula, D. W. (1968). A min-max theorem for graphs with application to graph coloring. *SIAM Review*, 10:481–482.

[Matula and Beck, 1983] Matula, D. W. and Beck, L. L. (1983). Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, 30:417–427.

[Meyer, 1973] Meyer, W. (1973). Equitable coloring. *Amer. Math. Monthly*, 80:920–922.

[Patwary et al., 2011] Patwary, M., Gebremedhin, A., and Pothen, A. (2011). New multithreaded ordering and coloring algorithms for multicore architectures. In *Euro-Par 2011 Parallel Processing - 17th International Conference*, pages 250–262.

[Robert K. Gjertsen et al., 1996] Robert K. Gjertsen, J., Jones, M. T., and Plassmann, P. E. (1996). Parallel heuristics for improved, balanced graph colorings. *Journal of Parallel and Distributed Computing*, 37(2):171–186.

[Sarıyüce et al., 2011] Sarıyüce, A. E., Saule, E., and Catalyurek, U. V. (2011). Improving graph coloring on distributed memory parallel computers. In *18th Annual Int. Conf. on High Performance Comp.*

[Sarıyüce et al., 2012] Sarıyüce, A. E., Saule, E., and Catalyurek, U. V. (2012). Scalable hybrid implementation of graph coloring using MPI and

OpenMP. In *IPDPSW, Workshop on Parallel Computing and Optimization (PCO)*.

[Sarıyüce et al., 2014] Sarıyüce, A. E., Saule, E., and Çatalyürek, Ü. V. (2014). On distributed graph coloring with iterative recoloring. Technical Report arXiv:1407.6745, ArXiv.

[Scott, 1988] Scott, J. (1988). Social network analysis. *Sociology*, 22(1):109–127.

[Taş et al., 2017] Taş, M. K., Kaya, K., and Saule, E. (2017). Greed is good: Parallel algorithms for bipartite-graph partial coloring on multicore architectures. In *Parallel Processing (ICPP), 2017 46th International Conference on*, pages 503–512. IEEE.

[Welsh and Powell, 1967] Welsh, D. J. A. and Powell, M. B. (1967). An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Comp. Journal*, 10:85–86.

[Zuckerman, 2007] Zuckerman, D. (2007). Linear degree extractors and the inapproximability of max clique and chromatic number. *Theory of Computing*, 3:103–128.