Parallel graph coloring on general purpose operating systems: a review of performances

Project report

Politecnico di Torino

Antonio Vespa 284101
Salvatore Pappalardo 281621
Andrea Speziale 292559

Contents

1 Introduction

In graph theory, the *graph coloring problem* consists in labeling the vertices with a certain color respecting the constraint that adjacent vertices cannot have the same color. Said in a more technical way:

> An undirected graph G is a set of vertices V and a set of edges E. The edges are of the form (i; j) where i; j ∈ V. A coloring of a graph G is a mapping c : V → {1, 2, ..., s} such that c(i) ≠ c(j) for all edges (i; j) ∈ E. c(i) is referred to as the color of vertex i. Vertices i and j are said to be neighbors if (i; j) ∈ E. The number of vertices is denoted by V.

The aim of our project is describing how it is possible to implement different well-known parallel coloring algorithms, and then compare their efficiency in terms of time, memory used and number of colors. During the progress of our work we have always had as a point of reference the results of Greedy algorithm, which, although trivial, has its benefits on small graphs.

The implementations are based on the C ++ programming language, the **Boost Graph** library for the graph representation, construction and visit and obviously the well-known standard library with its synchronization primitives.

NOTE: Later in the document we shall use the following notation:

- LDF: Larger Degree First
- SDL: Smallest Degree Last
- JP: Jones-Plassmann
- CSR: Compressed Sparse Row
- ADJL: Adjacent List
- ADJM: Adjacent Matrix

2 Design choices

2.1 The Boost Graph Library in brief

Graphs are mathematical abstractions that are useful for solving many types of problems in computer science. Consequently, these abstractions must also be represented in computer programs. The Boost Graph Library [1] is a *generic* interface that allows handling graph concepts, structures and algorithms, but hides the details of the implementation. Note that are three ways in which this library is generic:

1. Algorithm/Data-Structure Interoperability: each algorithm is written in a neutral way wrt data-structure
2. Extension through Function Objects: containers are extensible
3. Element Type Parameterization: containers are parameterized on the element type

Looking at the objectives it is clear how this library uses a lot generic programming to define a required typology of graph through a hierarchy of concepts, and then it provides many generic algorithms on these graphs: just think those needed to cross the nodes.

2.2 Graph Representation

In our project we have chosen to use three different representations of a graph to analyze the impact [2] of data structures on the performance of algorithms:

- **Adjacency List** [3]
- **Adjacency Matrix** [4]
- **Compressed Sparse Row Matrix** [5]

The BGL provides many configuration options to customize the graph's instance in memory. The main ones are:

- **Directed**: stands for the direction of the edges, i.e the graph could be directed, undirected or bidirectional;
- **VertexProperties**: for specifying internal node property storage;
- **EdgeProperties**: for specifying internal edges property storage;
- **VertexList**: The selector for the container used to represent the vertex-list of the graph.
- **EdgeList**: The selector for the container used to represent the edge-list for the graph.

Note that not all the different graph types provide the same parameters, for example the CSR representation cannot define the *VertexList* and *EdgeList* parameters because of its structure and in addiction it can be only directed and bidirectional.

Consistently with the types of *representation* explained above, we implemented them as:

```
typedef boost::adjacency_list<boost::listS,boost::vecS,boost::directedS,vertexDescriptor>
↪ graphAdjL;
typedef boost::adjacency_matrix<boost::undirectedS,vertexDescriptor> graphAdjM;
typedef boost::compressed_sparse_row_graph<boost::bidirectionalS, vertexDescriptor>
↪ graphCSR;
```

and as *vertex properties*:

```
struct vertexDescriptor { int random,num_it,weight; bool toBeDeleted; int16_t color; };
```

Then it's possible to interact with the constructed graph using the functions made available by BGL. Some are mentioned here for the following purposes:

- **Get info about the graph**: the fuctions **boost::num_vertices** and **boost::num_edges** return respectively the total number of vertices and edge of the specified graph.
- **Get info about a node**: the fuction **boost::out_edges** returns the number of outgoing edges, so implictly the number of neighbors vertices of a specified node.
- **Iterate**: the iteration can be on *vertices* or *edges*. To get all the edges, you can use **boost::edges()** which returns two iterators that refer to the begin and the end of them. In a similar way **boost::vertices()** can be called to obtain all vertices.

```
std::pair<graph::edge_iterator,
   graph::edge_iterator> es = boost::edges(g);
```

A possible variant, in our opinion even more practical because iterators are not used explicitly, is using the available macros called **BGL_FORALL_VERTICES** and **BGL_FORALL_EDGES**. Here is the function we used to iterate on vertices:

```
void GraphCSR::forEachVertex(node* current_vertex, std::function<void()> f){
   BGL_FORALL_VERTICES(curr, graph, graphCSR){
           *current_vertex = curr;
           f();
       }
}
```

We must also mention another macro which is **BGL_FORALL_ADJ** that allowed us to iterate the adjacent vertices of a specific vertex.

- **Modify**: the fuctions **boost::add_vertex** and **boost::add_edge** allows respectively to add a vertex in the graph and to connect two nodes (In graph theory, lines vertices are called edges).

2.3 CRTP structure

The curiously recurring template pattern (CRTP) is an idiom in C++ in which a class X derives from a class template instantiation using X itself as a template argument.

```
// The Curiously Recurring Template Pattern (CRTP)
template <class T>
class Graph
{
    // methods within Graph can use template to access members of Derived
};
class GraphCSR : public Graph<GraphCSR>
{
    // ...
};
```

The graph class template will take advantage of the fact that member function bodies (definitions) are not instantiated until long after their declarations, and will use members of the derived class within its own member functions, via the use of a cast; e.g.:

```
/*** Base class for CRTP ***/
    template <typename T>
    class Graph {
    private:
        friend T;
        ...
    public:
        ...
        /*** algoritmi colorazione ***/
        void sequential();
        void jonesPlassmann();
        void largestDegree_v3(); //jp structure
        void largestDegree_v2(); //without STL
        void largestDegree_v1(); //STL implementation
        void smallestDegree();
        /*** da specializzare in ogni rappresentazione interna ***/
        void forEachVertex(node* current_vertex, std::function<void()> f){
            return static_cast<T&>(*this).forEachVertex(current_vertex,f);
        };
        void forEachNeighbor(node v, node* neighbor, std::function<void()> f){
            return static_cast<T&>(*this).forEachNeighbor(v,neighbor,f);
        };
        int getDegree(node v){
            return static_cast<T&>(*this).getDegree(v);
        };
    };

    /*** 1) CSR Internal representation ***/
    class GraphCSR : public Graph<GraphCSR>{
    private:
        graphCSR graph;
        ...
    public:
        GraphCSR(vector<std::pair<node, node>>&,int,int);
        /*** specializzazioni ***/
        void forEachVertex(node* current_vertex, std::function<void()> f);
        void forEachNeighbor(node v, node* neighbor, std::function<void()> f);
        int getDegree(node v);
    };
```

Note that a member function like Graph<GraphCSR>::forEachVertex(), though declared before the existence of the struct GraphCSR is known by the compiler (i.e., before GraphCSR is declared), is not actually instantiated by the compiler until it is actually called by some later code which occurs after the declaration of GraphCSR (not shown here), so that at the time the function "forEachVertex" is instantiated, the declaration of GraphCSR::forEachVertex() is known.

This technique achieves a similar effect to the use of virtual functions, **without the costs** (and some flexibility) [6] of dynamic polymorphism (a possible alternative for a generic programming, explored in an initial phase of our project but then discarded for the aforementioned reasons).

The use of the CRTP pattern allowed us to use 3 different structures for the representation of the graph, which although different in the end shared the same basic functions. If we had had to declare three different classes, we would had to duplicate the code for three! Instead, using CRTP we were able to write the almost totality of functions once (just think to those implementing coloring algorithms), excluding only the ones more *dependent* on the structure itself (about iteration on edges/nodes).

```
//Example of how to generalize coloring algorithms through the
// use of other member function for structure dependent actions
template<typename T>
void asa::Graph<T>::sequential() {
    ...
    forEachVertex(&current_vertex, [this, &color, &current_vertex]() { //!!!!!!
        //ciclo su ogni vertice e cerco il colore da usare
        color = searchColor(current_vertex);
        static_cast<T &>(*this).graph[current_vertex].color = color; //coloro il vertice
        ↪  corrente
    });
};
```

3 Input

3.1 Input graph implementation

Our project is able to read two different types of input graph format defined by a dedicated file extension, with the aim of extracting the edges (to be passed later to the graph constructor). More precisely, for this purpose we created the **ReadInput** class, which, starting from the extension, is able to extract the pairs of connected nodes.

```
    //reading a graph in the "fin_name" file example
    ReadInput read(fin_name);
    V = read.getV();
    E = read.getE();
    edges = read.getEdges();
```

We support two graph formats:

- **.graph** or **.txt**: it represents an undirected graph and contains as first row two numbers that are the number of vertices **V** and edges **E**. After this first row there will be **V** further rows containing all the neighbors of each i-th node, where **i** is the row number. In this representation we have to consider nodes starting from 1.
- **.gra**: it is a bit different and more compact because it seems like an oriented graph but we consider it as bidirectional, so we will manually put edges of both directions in the edges vector. In this representation we have to consider nodes starting from 0.
  In particular, the first row contains the number of edges **V** and after it there will be **V** other rows. Each row is in the format *0: 84 92 95 23 72 2 75 88 87 90 67 #* where the first number is the actual node followed by each neighbour and finally the row is ended with an hash symbol.

Once the reading is finished, we can notice that our vector **edges**, for each pair of nodes contains two edges to represent both directions. This is useful for the CSR *bidirectional* representation and it is not a problem for the adj list and adj matrix because the duplicated ones are not considered and consequently ignored by the (Boost) constructor.

Note that the two different input formats start from an id_node 0 and 1 respectively. This difference is managed during the insertion of each edge in **edges** vector: in the case of starting id_node = 1, we insert an adge [#readInputLine, neighbour - 1].

```
for(int i=0; i<V; i++){
    getline(f, line);
     ...
    edges.emplace_back(std::pair<int, int>(i,neighbour-1));
}
```

This step is fundamental for the construction of the graph in an appropriate way, as the *Boost Graph data structure* expects id_nodes starting from 0, and instead vertices starting from 1 would result in the instantiation of a node 0 with 0 neighbors.

3.2 Input performances concerns

As the reading speed is important, we preferred to read the file all at once putting its content in a previously resized **buffer** (whose dimension is computed thanks to the std::filesystem::file_size() function). This method requires more memory but reading from that is much more efficient then reading the file line by line from the file system (1 IO vs #nlines IO actions performed). Then, thanks to the *stringstream* function of the *iostream* library, we read the buffer content line by line.

Sidenote. We tried to implement a multi-thread version following the producer-consumer problem [8] with a producer and a consumer, but it resulted to be much less performing than the previous single-thread job. It uses a shared queue where the producer will push strings that represents edges as pairs of nodes divided by a space, while the consumer will pop them to push in the **edges** vector (it's needed a **mutex** for managing push and pop operations performed to the queue and a **CV** for waking up threads). Probably this approach is not advantageous as, in addiction to *std::thread* creation, the reading and loading operations in the data structures are much faster than the IO operations: a producer does not make sense if the consumers are much faster than him.

4 Algorithms code implementation

Following the literature[9] we have implemented the following algorithms.

4.1 Jonnes-Plassmann

$U := V$
while ($|U| > 0$) **do**
  for all vertices $v \in U$ **do in parallel**
    $I := \{v$ such that $w(v) > w(u) \forall$ neighbors $u \in U\}$
    for all vertices $v' \in I$ **do in parallel**
      $S :=$ {colors of all neighbors of v'}
      $c(v') :=$ minimum color not in S
    end do
  end do
  $U := U - I$
end do

Figure 1: Jones Plassmann pseudocode

Our implementation firstly defines a random number for each vertex, then are created several threads: to each of them is assigned the same number of vertices in order to have a good balance during processing (*graph partitioning*). In more detail, each thread will execute the same function called *threadFn*, with a specific *id* as parameter that indicate the progressive thread number.

4.1.1 Select vertices to color

In this function, after the vertices range to colour is computed and assigned to the given thread (*min* and *max* stands for the extremes), a for loop is executed in order to find the first set of vertices:

```
for (int i = min; i < max; i++) {
    node neighbor;
    bool major = true;
    if (static_cast<T &>(*this).graph[i].color != -1) major = false;
    else
        forEachNeighbor(i, &neighbor, [this, &neighbor, i, &major, &isMinor]() {
            // Non necessito lock: i thread agiscono su porzioni di memoria disgiunte
            if (static_cast<T &>(*this).graph[neighbor].color == -1 && isMinor(i,
            ↪  neighbor))
                major = false;
        });
    if (major) {
        //if major remains true, this node has to be coloured
        toColor_set[tcs_length[id]++] = i; //thread local variable, tcs_length is the
        ↪  actual number of vertices to color
    }
}
```

You can notice that this for loop consist in scanning each vertex of the defined range and check if it is already coloured (color != -1) and whether it has the **highest random value** among its neighbors (*isMinor* function will compare the random values of the vertices). Then, if this condition is true, the vertex is added to the *toColor_set* for being coloured (next step).

4.1.2 Coloring vertices selected in this round

The *colouring part* consists in another for loop (in the **same thread**) that will assign the smallest available colour to each vertex in *toColor_set*, that are the greatest in their neighborhood founded in this round.

```
for (int j = 0; j < tcs_length[id]; j++) {
    //pop from the vector of nodes to color
    int i = toColor_set[j];
    //compute of the smallest available color among neighbours
    int16_t color = searchColor(i);
    //assign the color
    static_cast<T &>(*this).graph[i].color = color;
}
```

Note that **no locks** are used when assigning the color because the neighbours can't be selected in other threads.

4.1.3 Start the next round

The variable so called *round* represents in the algorithm the number of loop done by threads (in which a set of "maximal" nodes is *created and coloured*) and rules the synchronization between the main thread and all the others: no one can start a new round if the others are still managing the previous round. This main loop will continue until all vertices of graph are colored.

```
//n-th thread
...
ulk.lock();
cv.wait(ulk, [&roundMain, &round, id]() { return round < roundMain;  });
round++;
ulk.unlock();

//Main thread
...
slk.lock();
cv.wait(slk, [&done, this]() { return done == concurrentThreadsActive; }); //aspetto gli
↪  altri thread
```

```
12  roundMain++;
13  cv.notify_all();
14  slk.unlock();
```

4.1.4 Main thread

The purpose of the *main* function will be to create the threads, run them and wait for them to complete a coloring loop or *round*, in order to update the number of remaining non-colored vertices and restart them. Only when the number of remaining vertices becomes zero the *toTerminate* flag will be set true to let all threads terminate.

```
1   while (v_length > 0) {
2       //aspetto gli altri thread
3       slk.lock();
4       //aspetto fin quando tutti i thread creati vanno in wait perchè hanno finito il
        ↪  round
5       cv.wait(slk, [&done, this]() { return done == concurrentThreadsActive; });
6       // tolgo i vertici di toColor_set da verteces
7       for (int i = 0; i < concurrentThreadsActive; i++)
8           v_length -= tcs_length[i];
9       if (v_length == 0)
10          toTerminate = true;
11      roundMain++;
12      done = 0;
13      slk.unlock();
14
15      cv.notify_all();
16  }
```

In this implementation, the number of threads does not take into account the main thread. The latter has one important tasks: update the termination condition, which is to the number of vertices still to be colored. We can conceptualize the algorithm in two steps:

1. the secondary threads determine individually the vertices to be colored and color them,
2. the main thread update the termination condition.

Originally the algorithm was written with two different synchronization points: one after the determination of the set of vertices to be colored in a round, and the other one with after the colorization itself. It is easy to understand that the second synchronization point introduced more waiting times, thus slowing down the algorithm without a good reason.

4.2 Largest Degree First

"The Largest-Degree-First algorithm can be parallelized using a very similar method to the Jones Plassmann algorithm.The only difference is that instead of using random weights to create the independent sets, the weight is chosen to be the degree of the vertex in the induced subgraph. Random numbers are only used to resolve conflicts between neighboring vertices having the same degree. In this method, vertices are not colored in random order, but rather in order of decreasing degree, with those of largest degree being colored first."

We implemented three different versions of this algorithm. They all have a dummy main thread which very simply creates the threads and then waits for them to finish, there the entire job is executed. The differences resides in the interpretation of the original paper. In the following subsection, each version will be explained in details.

4.2.1 V1 - STL data structures

The first version is the "naive" one and also the oldest. This version is not much optimized since **STL data-structures** were used: it means slowness.

In this version, each thread operates on all the vertices simultaneously (no graph partitioning), processing them one by one in order to find those with maximum degree in the neighborhood.

Once found, a color is chosen and **immediately** after colored, so no multiple rounds as in Jones Plassmann algorithm.

A simple description of the code for each thread is as follows. Note that in this code snippet we omitted the implementation details (such as locks management) in favor of a simpler description.

```
deque<int> total_set; // contains all the vertices

while(total_set.size() > 0){  //there are are still nodes to be colored
    int v = total_set.front();
    total_set.pop_front(); //pop of the node from the queue, it will be considered in
    ↪  this way only by me  as a thread
    if( hasMaximumDegree(v) ){
        doColor(v);
    }else{
        total_set.push_back(v); //push again in the queue (vertex to be considered
        ↪  again, but later)
    }
}
```

So, each vertex is popped out from the main queue (synchronization through mutex is necessary because we are modifying a structure that is common to all threads). Then, this and only this thread checks whether it is the one with maximum degree between the **not-colored** neighbors: if so, that vertex is colored with the minimum color available in the neighborhood, otherwise the vertex is pushed back into the total set once again until it has actually the maximum degree the next time it will be valuated (again).

This version of the algorithm, as previously said, is quite inefficient. The first reason is to be searched in the data structure. Obviously, in order to use such an approach, it mandatory to use locks, which might slow down the threads quite a lot. Furthermore, the detailed implementation of deque class is unknown to the authors, so it is possible that push's and pops could be not so performing. Lastly, it is also true that is simpler (and faster) to operate directly on a range of integers, rather than any data structure holding integers. Thus, a second version of the algorithm was implemented.

4.2.2 V2 - Graph Partitioning

This version is conceptually the same as the first one, but it is more efficient, solving the problems described above.

To each thread is assigned an equal range of vertices (graph partitioning) and can access only those, in a very similar way as in Jones Plassmann implementation.

```
int min = id_thread * range, max = (id_thread + 1) * range;
```

For all the assigned vertices still uncoloured, the thread checks whether it is the one with maximum degree or not. If it is the case, then the vertex is colored. Otherwise the cycle continues (after the increases of the local variable *remaining* which counts the number of nodes left uncolored), until every assigned vertex is colored: *remaining = 0 WHEN i = max - 1*. Every vertex is processed at least once, in order to determine its color.

NOTE If you read the code carefully, you will notice that no locks have been used. This was achieved thanks to:

1. different data structures between threads,
2. *[empirical intuition]* how the for loop was built and, more generally, the coloring method

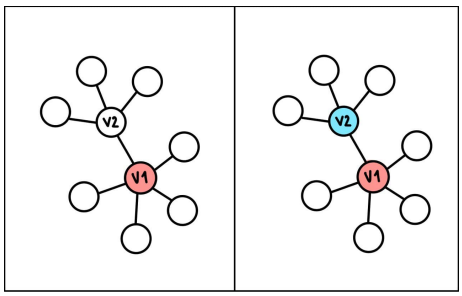

Figure 2: [d(v1):5, d(v2):4] In any case v1 will be colored before v2

No lock required Starting from the idea that locks are used to prevent race conditions, we affirm that in this case they are not there. Assuming you are coloring a node *vertex1* and at the same time considering a nearby node *vertex2*, con degree(vertex2)<degree(vertex1):

1. if during the evaluation of vertex2, vertex1 is already colored, then vertex2 can in turn be marked with a color if it is maximal among its neighbors;

2. if during the evaluation of vertex2, vertex1 is not yet colored, then vertex2 in any case cannot be colored because vertex1 is maximal, it has a greater degree than its own.

From this simple intuition, it can be seen that the order of coloring according to the degree of the nodes is preserved in any case and that simply between the two cases described, times change: in the second the vertex2 node will have to wait to be considered further on in the iteration.

4.2.3 V3 - Jones Plassmann structure

Unlike the previous ones that immediately color a "maximal" node when found (similar to how the greedy algorithms does), in this interpretation we first select the set of maximum nodes and then color it all together. The concept of *round* comes back.

Thus, the third version is a little bit more complex than the first one and it's basically the same as the Jones-Plassmann algorithm, with the exception of the *color condition*. Indeed, it was possible to generalize the jones-plassmann algorithm structure. The only difference is that this version checks which one of the vertices has greater degree, while in the jones-plassmann algorithm only the random value matters. For this reason, every consideration made about the jones-plassmann implementation is valid in this context.

4.3 Smallest Degree Last

This algorithm is an improvement of the **Largest Degree First** algorithm, it uses a different system of weights.

*"There is at first a **weighting phase** which begins by finding all vertices with degree equal to the smallest degree d presently in the graph. These are assigned the current weight and removed from the graph, thus **changing the degree** of their neighbors. The algorithm repeatedly removes vertices of degree d, assigning successively larger weights at each iteration. When there are no vertices of degree d left,the algorithm looks for vertices of degree d+1. This continues until all vertices have been assigned a weight."*

$k := 1$
$i := 1$
$U := V$
while $(|U| > 0)$ **do**
  while $\{\exists$ vertices $v \in U$ with $d^U(v) \leq k\}$ **do in parallel**
    $S = \{$all vertices v with $d^U(v) \leq k\}$
    for all vertices $v \in S$, $w(v) := i$
    $U = U - S$
    $i := i + 1$
  end do
  $k := k + 1$
end do

Figure 3: Smallest degree pseudocode for weighting phase

Once the *weighting phase* is done, can start the **coloring phase** which will assign colors. This proceeds as in the *Largest Degree First* algorithm but it will start coloring from the vertices with the highest weight (previously assigned starting from the lowest grade).

This implementation uses STL data structures, in a similar way of *Largest Degree First v.1*.

4.3.1 Weighting phase

```
/*** n-th thread ***/
while (doContinueWhile) {
    if (total_set.empty())
        //end loop se check positivo...
    else {
        /*** pop dalla coda di nodi da valutare ***/
        current_vertex = total_set.front();
        total_set.pop_front();
        //se sto iniziando nuovo round, aspetto che tutti i nodi selezionati allo scorso
        ↪  siano pesati dal main thread...
        ...
        /*** confronto con i vicini ***/
        int degreeCurrVertex = computeDegree(current_vertex);
        if (degreeCurrVertex <= numIteration)
            minor = true;
        ulk.lock();
        if (minor) {
            toColor_set.push_back(current_vertex);
        } else {
            /*** reinserisco in coda,valuto al prossimo giro ***/
            ...
        }
    }
}
/*** main thread ***/
...
while (doContinueWhile) {
    cv.wait(ulk, [this]() { return isEnded || increase_numIteration == active_threads;
    ↪  });
    while (!toColor_set.empty()) {
        /*** pop da coda + weight + mark deleted ***/
        ...
    }
    if (isEnded)
        doContinueWhile = false;
    numIteration++;
}
```

During this phase a certain number of threads will be created and they will search for those vertices with minimum degree to assign them a progressive weight. *round* concept is present there. The role of the main thread is central: secondary threads only put vertices into a *deque* data structure so that the *main* one can assign the weight and *delete* them from the graph. In this way the threads will keep looking for vertices with the same degree. Only when they are all weigthed, we can start looking for other nodes characterized by a greater degree, until no vertices are left.

NOTE: The vertices are not really removed from the Graph data structure, but they have a specific flag that will be set to mark them as *deleted*. In fact, we had to create a dedicated function **int computeDegree(node current_vertex)** for computing the degree using that flag, because the **boost::num_edges** is useless in this case (the node is still present).

4.3.2 Coloring phase

Then, the coloring phase can start. It consists in creating a previously defined number of threads that will color the vertices contained into a deque **toColor_set** with the minor available color, starting from the **higher weight**.

```
/*** n-th thread ***/
while (doContinueWhile) {
 //sincronizzazione round...
 if (toColor_set.empty() && total_set.empty()) {
     //check terminazione ...
     ...
     doContinueWhile = false;
 } else {
     /*** pop dalla coda ***/
     current_vertex = toColor_set.front();
     toColor_set.pop_front();
     /*** coloring ***/
     int16_t color = searchColor(current_vertex);
     static_cast<T &>(*this).graph[current_vertex].color = color; //coloro il vertice
     ↪  corrente
 }
}
/*** main thread ***/
//cerco per peso decrescente
for (wei = current_weigth; wei >= 0; wei--) {
    forEachVertex(&current_vertex, [this, &current_vertex, current_weigth, wei, &C]() {
        /*** pop coda ***/
        current_vertex = total_set.front();
        total_set.pop_front();
        if (static_cast<T &>(*this).graph[current_vertex].weight == wei) {
            // inserisco nei vertici da colorare se peso giusto
            toColor_set.push_back(current_vertex);
        } else
            // reinserisco se weight minore
            total_set.push_back(current_vertex);
    });
    //sincronizzazione a fine round..
}
```

5 Results

5.1 Limits

5.1.1 False Sharing

Memory is stored within the cache system in units know as cache lines. Cache lines are a power of 2 of contiguous bytes which are typically 32-256 in size. The most common cache line size is 64 bytes. False sharing is a term which applies when threads unwittingly impact the performance of each other while modifying independent variables sharing the same cache line. Write contention

on cache lines is the single most limiting factor on achieving scalability for parallel threads of execution in an SMP system. False sharing is a silent performance killer because it is far from obvious when looking at code.
To achieve linear scalability with number of threads, we must ensure no two threads write to the same variable or cache line. Two threads writing to the same variable can be tracked down at a code level. To be able to know if independent variables share the same cache line we need to know the memory layout, or we can get a good profiling tool to tell us, such as Intel VTune.
For more details, see false sharing

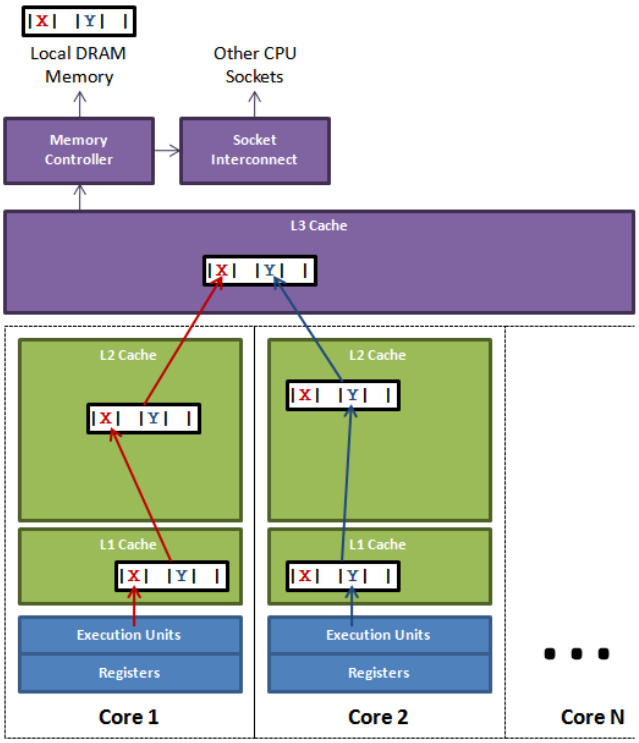

Figure 4: Cache lines usage example

5.1.2 Operative system

The tests were performed on linux, and therefore an operating system that implements a very specific scheduling strategy to support the execution in RAM of many different programs at the same time, among which our GraphColoring is the last of a long list.

Ideally it would be necessary to run our program in a dedicated, almost 'bear metal' environment, where it could have unlimited resources, but it is not possible to do so due to the innumerable dependencies of the libraries used.

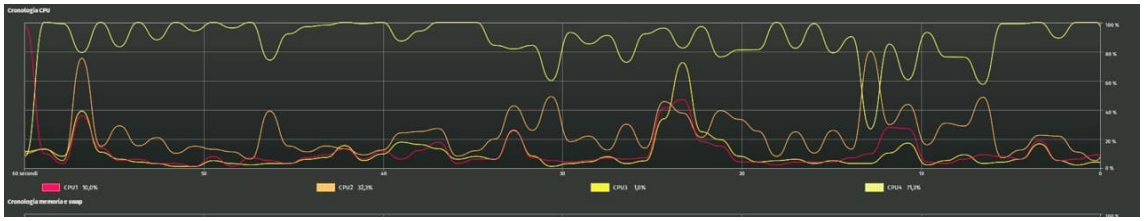

Figure 5: [System monitor] Resources usage

Looking at figure 5 you can easily imagine how the low average usage of the cpu cores is due to the alternation of execution with other programs. You should try to play with the scheduler priorities for better performance.

5.1.3 Threads

Thread creation is expensive because there is a fair bit of work involved:

1. a large block of memory has to be allocated and initialized for the thread stack
2. system calls need to be made to create / register the native thread with the host OS
3. descriptors need to be created, initialized and added to data structures
4. the thread ties down resources as long as it is alive.

The costs of all of these things are platform specific, but they are not cheap on any of them. For example reference to this page.

5.2 Benchmark

All the tests were executed on a linux platform. Specifically, we used a laptop equipped with a Intel i7-8550U (4 cores, 8 threads) and 16Gb of RAM. The operating system is Ubuntu 20.04.3 LTS.

In order to perform all the benchmark toghether we used **runlim**, a tool developed by jku whose purpose is sampling and limiting time and memory usage. The output of the tool is parsed by a bash script we called *runlim.sh*. This script is capable of running the program with a specific configuration. Optionally, it can perform multiple runs and display the average time.

Finally, we created *benchmark.sh* that is responsible for running runlim.sh with all the specified configurations.

5.2.1 Times

The times were gathered using the tool explained above. Each configuration is run 5 times. The tested configurations vary in used algorithms and number of threads. Each algorithms was tested with 2, 4, 8 and 16 threads. The physical threads of the processor are 8, but it was worth seeing the behavior of the OS with 16 threads. The graph benchmarks used come from [10]. We decided to use graphs with a number of node variable between 2^{15} and 2^{21} nodes.

	Sequential	jones_plassmann				ldf_v3				ldf_v2			
Files\Threads	1	2	4	8	16	2	4	8	16	2	4	8	16
rgg15	**0,836**	1,09	0,998	1,012	0,906	1,128	0,934	1,04	1,02	1,162	0,896	1,048	1,062
rgg16	1,336	1,736	1,332	1,702	**1,262**	1,892	1,662	1,482	1,556	1,442	1,408	1,294	1,552
rgg17	2,254	2,8	2,514	**2,192**	2,544	3,408	2,88	2,718	2,812	2,744	2,836	2,534	2,624
rgg18	**4,25**	5,804	4,914	4,662	4,778	6,95	5,628	5,218	5,292	5,384	4,576	4,768	4,584
rgg19	**8,33**	11,41	9,718	9,278	9,804	14,54	11,79	10,96	11,09	10,59	9,25	9,182	9,166

Figure 6: Times for the algorithms to complete. All times measured in seconds.

As shown in figure 6 For the smallest graphs tested, the sequential algorithm performed the best. That's likely due to the useless overhead generated to allocate resources for the new threads in the other algorithms. The parallel algorithms outperformed the sequential on middle-sized graphs.

In relationship to big-graphs (Figure 8), the sequential algorithm was faster than the parallel ones. That's likely due to the OS scheduler, which we didn't set up. The best guess is that the OS manages to distribute evenly the available resources to all the threads, thus allocating less than necessary for the program to run faster. In contrast, the OS might have "decided" to dedicate a single thread to one process for a slightly longer period of time. This explains the obtained results.

It's visible that some of the implementations are greatly inefficient. In Figure 7 the worst algorithms are shown. The implementation with Smallest Degree Last and Larger Degree First v1 are much slower than the others. That's due to the use of complex data structures. In these first versions `std::deque` was the preferred class used, which is slow for operation of `push` and `pop` in comparison to the simple basic C array combined with indeces.

> **Note:** some of the configurations were so slow that we decided not to test them. Thus an 'x' is reported in the tables.

	Sequentia	ldf_v1				sdl			
Files\Threads	1	2	4	8	16	2	4	8	16
rgg15	**0,836**	3,06	3,13	3,356	3,062	4,018	5,832	8,348	12,18
rgg16	1,336	6,18	6,538	6,222	5,8	8,814	12,75	18,46	27,11
rgg17	2,254	12,31	13,13	12,84	12,02	17,8	25,69	36,68	54,76
rgg18	**4,25**	25,88	28,11	27,74	26,01	X	X	X	X
rgg19	**8,33**	55,28	59,58	59,48	53,84	X	X	X	X

Figure 7: Times for the algorithms to complete. ldf stands for "Largest Degree First", sdl stands for "Smallest Degree Last". All times measured in seconds.

	Sequential	jones_plassmann				ldf_v2			
	1	2	4	8	16	2	4	8	16
rgg20	**16,93**	23,52	20,02	19,23	19,1	21,62	19,48	18,87	**18,76**
rgg21	**35,23**	49,26	42,41	40,98	53,75	46,27	41,34	**39,36**	45,48

Figure 8: Times for the algorithms to color graphs with 2^{20} and 2^{21} vertices. ldf stands for "Largest Degree First", sdl stands for "Smallest Degree Last". All times measured in seconds.

An interesting trend to notice is the difference in times related to the number of threads.

Generally the 8 threads configuration performed faster or equal than the 16 one, except for one single case, as show in Figure 9. This is related to the physical available threads in the processor. For `rgg21`, every algorithm show the same trend. The time improves with increasing number of threads, to worsen in the last configuration.

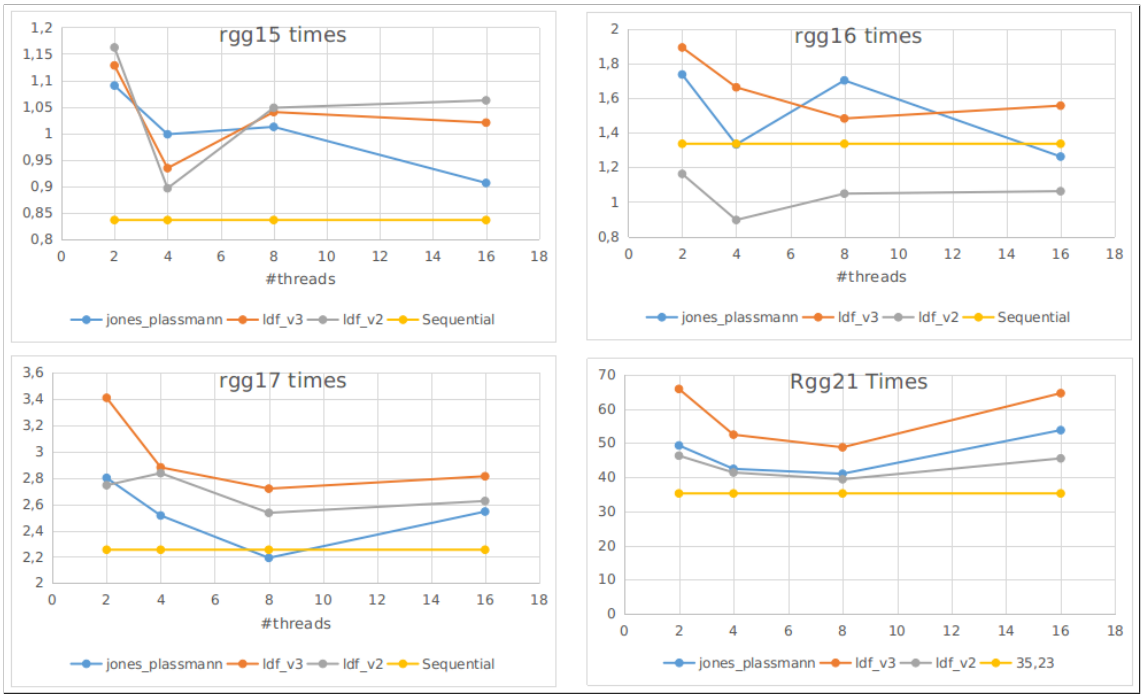

Figure 9: Comparison of times between algorithms related to the number of threads The sequential algorithms is reported for comparison only. All times measured in seconds.

5.2.2 Space

In this section we want to highlight the different required spaces for different configurations. Firstly we will illustrate the spaces for different algorithms with the same representation and finally the different space for different graph representations.

As shown in Figures 10, 11 and 12, there is no significant difference in the occupied spaces between different algorithms. This result is almost obvious, since most of the space is occupied by the graph, making any other structure negligible. We also want to recall that for these tests the same representation was used, reducing even more the difference in spaces.

	Sequential	ldf_v1				sdl			
Files\Threads	1	2	4	8	16	2	4	8	16
rgg15	22,9	**22,6**	22,9	22,9	22,8	23,1	23,1	23,3	23,3
rgg16	43,6	43,6	43,6	43,6	43,6	44	43,8	44,2	44,2
rgg17	87,4	87,3	87,4	87,4	88,1	88,2	88	88,3	88,1
rgg18	**179,4**	179,9	180	179,9	179,9	X	X	X	X
rgg19	**373,7**	374,6	375,1	374,1	375,7	X	X	X	X

Figure 10: Required space for LDF and SDF v1 algorithms. Measures are expressed in Mega-Bytes.

	Sequential	jones_plassmann				ldf_v3				ldf_v2				
Files\Threads	1	2	4	8	16	2	4	8	16	2	4	8	16	2
rgg15	22,9	22,8	22,8	22,9	22,9	22,9	22,9	22,9	22,8	22,9	22,9	22,9	22,8	**22,6**
rgg16	43,6	43,7	43,6	43,7	43,6	43,6	43,6	43,6	43,9	**43,4**	43,6	43,7	44	43,6
rgg17	87,4	87,4	**87,2**	**87,2**		87,4	87,4	87,4		87,3	87,4	**87,2**		87,3
rgg18	**179,4**	**179,4**	179,7	179,7	179,9	179,7	179,9	179,7	179,9	179,7	179,9	**179,4**	180,1	179,9
rgg19	**373,7**	373,9	374,2	373,8	374,3	374,1	374,1	374,2	373,8	373,9	373,9	374,1	374,1	374,6

Figure 11: Required space for JP, LDF v3 and LDF v2 algorithms. Measures are expressed in Mega-Bytes.

	Sequentia	jones_plassmann				ldf_v2			
	1	2	4	8	16	2	4	8	16
rgg20	780,5	781	780,9	781,1	781,4	780,5	780,6	780,9	781
rgg21	**1631,8**	1632,7	1632,6	1633	1633	1632,3	1632,3	1632,4	1632,4

Figure 12: Required space for rgg20 and rgg21 graphs. Measures are expressed in Mega-Bytes.

Regarding the representations, it is noticeable that some representations are definitely more efficient than others. Figure 13 shows how the space increase with respect to the different files tested.

It is visible that the space grows exponentially. This is better shown in figure 15. The worst representation is the adjacent matrix. It was not possible to test it with rgg18, rgg19 and rgg20, since the memory would overflow physical RAM, using the swap partition on the disk. The values with yellow background are interpolated based on the actual values for the other files.

The smallest representation is the Compressed Sparse Row. This representation has some drawback, tough. It requires compression and decompression of the data, slightly increasing the processing time. In addition to that, the construction of this type of graph is a little bit more complicated, since it is necessary to have all the edges of the file in advance, while for the other two representations it is possible to just add data in the structures after they are allocated, thus knowing only the number of vertices.

		Space (MB)		
	Files\representation	**CSR**	**ADJL**	**ADJM**
	rgg15	22,9	65,5	484,3
	rgg16	43,1	134,6	2074,3
	rgg17	87,1	281	8242,9
	rgg18	179,6	591	34583
	rgg19	373,8	1241,7	142831
	rgg20	780,4	2608	590000
		Yellow values are interpolated exponetially		

Figure 13: Required space for different graph's representations. Measures are expressed in Mega-Bytes.

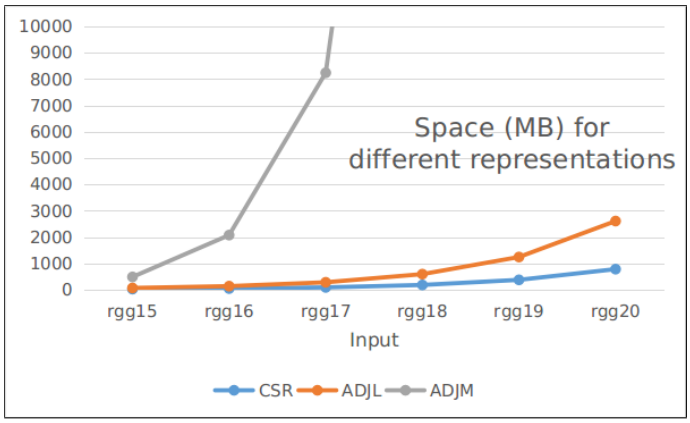

Figure 14: Required space with respect to files. Measures are expressed in Mega-Bytes.

5.2.3 Colors

Finally, in this section we represent a comparison of the number of colors used by each algorithm, highlighting the most performing ones.

Files	Sequential	Jones plassman	ldf v2	ldf v1	sdf
rgg15	12	14	13	13	**10**
rgg16	14	16	13	13	**12**
rgg17	15	16	14	14	**12**
rgg18	16	16	15	15	**14**
rgg19	18	17	17	17	**15**
rgg20	17	18	17	17	**16**

Figure 15: Number of colors used

As expected from the literature [7], the smallest degree last is the one that uses fewer colors even if in general it has lower performances (our implementation particularly).

6 Conclusions

We noticed that parallel algorithms are faster than the greedy sequential only under specific conditions. On general purpose OS's like the used one, the algorithms are not better "out of the box". It is necessary to reconfigure the OS in order to obtain the best performance.
The differences between spaces is huge. Overall, the best representation might be Adjacent List, which does not occupy much space, but does not require decompression.

References

[1] http://www.boost.org/doc/libs/1_52_0/libs/graph/doc/index.html

[2] https://www.usenix.org/system/files/login/articles/login_winter20_16_kelly.pd

[3] https://www.boost.org/doc/libs/1_77_0/libs/graph/doc/adjacency_list.html

[4] https://valelab4.ucsf.edu/svn/3rdpartypublic/boost/libs/graph/doc/adjacency_matrix.html

[5] https://www.boost.org/doc/libs/1_77_0/libs/graph/doc/compressed_sparse_row.html

[6] https://stackoverflow.com/questions/19062733/what-is-the-motivation-behind-static-polymorphism

[7] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.102.611&rep=rep1&type=pdf

[8] https://en.wikipedia.org/wiki/Producer-consumer_problem

[9] J. R. Allwright, R. Bordawekar, P. Coddington, K. Dinçer, Christine Martin, "A Comparison of Parallel Graph Coloring Algorithms", 1995

[10] Ryan A. Rossi and Nesreen K. Ahmed, The Network Data Repository with Interactive Graph Analytics and Visualization - AAAI, 2015, https://networkrepository.com