PARALLEL GRAPH COLORING

Alain Denzler, Philippe Mösch, Spyridon Margomenos

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

Abstract: Graph coloring is an important and difficult to solve efficiently problem. We reimplemented Boman's Graph Coloring (add reference to paper) for a distributed memory setup, with focus on compact data structures and predictable memory accesses, and we are proposing a new color assignment algorithm, evaluated on synthetic and real-world graphs. Our experiments show good scalability, independent of the graph partitioning, and show a substantial performance gain in comparison to the Boost parallel graph library **??**.

1. INTRODUCTION

Motivation. Graph coloring is an interesting problem that arises in many different applications, for example an important problem is task scheduling where each job is allocated to a specific time slot. We can draw a graph whose vertices correspond to tasks and whose edges denote a conflict (i.e. the two connected tasks cannot be schedules at the same time) and finally the colors denote time slots. Solving the graph coloring problem in that case allows us to find an assignment satisfying the problem. It is always desirable to obtain the minimul amount of colors that satisfy the given problem. However since this problem is NP complete, there exists no known algorithm to solve the graph coloring problem in polynomial time, instead we need to approximate the optimal solution and focus on parallelizing the algorithm that only in order to achieve better performance. The algorithm that we chose requires communication between processes which can become a bottleneck, solving this issue was one of the main difficulty in achieving better performances. Other important issues where how to partition the graph in order to distribute it among the working processes where we opted for a random distribution. One important task was to turn synchronous communication into asynchronous communication, this in order to prevent communication from becomming a bottleneck, furthermore we wanted to achieve speedup by reducing the frequency of communiation between processes. Furthermore, despite causing more conflicts, asynchronous has an obvious latency advantage. Another goal was to decide on the fastest color allocation algorithm to use, where we opted for a modified staggered-fist-fit scheme as described in the next section. To store the adjacency matrix we picked the Compressed Sparse Row (CSR) format, without the val array (since it is redundant) since this suits well the communication between processes. For the implementation, MPI and C++ were used.

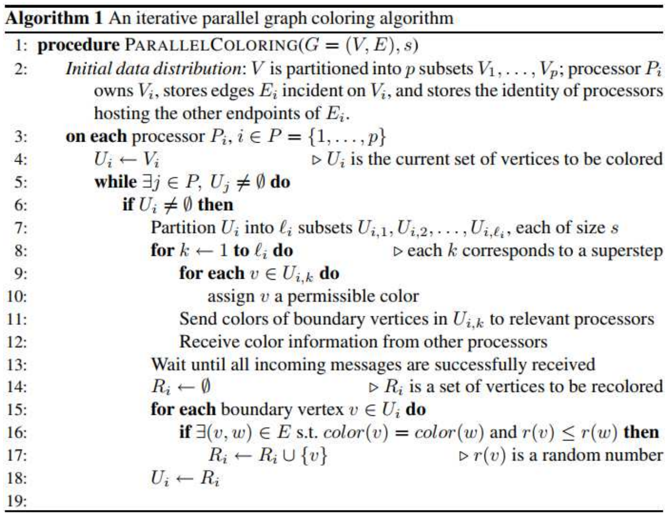
Related work. We took our inspiration for this project through the paper [1], where a well-partitioned graph was used, a superstep size of 100 as well as the First-fit coloring assigment and where the experiments where carried out with a cluster with dual 900 MHz Intel Itanium 2 CPUs and 4GB of memory. They themselves took inspiration in papers such as [2] where processors are assigned vertices, coloring is done with first fit and simultaneous coloring is causing conflict. Our implementation differs in the choice of more diverse coloring heuristics

2. BACKGROUND

In this section we introduce graph coloring in general and our algorithm of choice, Boman's graph coloring algorithm.

Graph coloring. The idea behind graph coloring is simple, given the graph G = (V,E), we assign a color to each vertex so that no two neighbors share the same color, we assume that each color can be denoted by an integer. This problem is NP-complete and thus needs to be approximated. In section **??** we describe the different coloring algorithm used in our work, they all aim at minimizing the total number of color used, we used the first-fit greedy algorithm which can be rather slow (complexity of O(n)), we also implemented the staggered first-fit allocation scheme with and without some modifications.

Boman graph coloring. Boman's graph coloring is an iterative distributed memory algorithm, it works by splitting the vertices among the processes. We distringuish interior vertices that connect only vertices belonging to the same process and Boundary vertices connecting vertices belonging to other processes. An important tunable parame-



Algorithm 1 An iterative parallel graph coloring algorithm

```
1: procedure ParallelColoring($G = (V, E), s$)
2:     Initial data distribution: $V$ is partitioned into $p$ subsets $V_1, \dots, V_p$; processor $P_i$
           owns $V_i$, stores edges $E_i$ incident on $V_i$, and stores the identity of processors
           hosting the other endpoints of $E_i$.
3:     on each processor $P_i$, $i \in P = \{1, \dots, p\}$
4:         $U_i \leftarrow V_i$                          ▷ $U_i$ is the current set of vertices to be colored
5:         while $\exists j \in P$, $U_j \neq \emptyset$ do
6:             if $U_i \neq \emptyset$ then
7:                 Partition $U_i$ into $\ell_i$ subsets $U_{i,1}, U_{i,2}, \dots, U_{i,\ell_i}$, each of size $s$
8:                 for $k \leftarrow 1$ to $\ell_i$ do          ▷ each $k$ corresponds to a superstep
9:                     for each $v \in U_{i,k}$ do
10:                        assign $v$ a permissible color
11:                    Send colors of boundary vertices in $U_{i,k}$ to relevant processors
12:                    Receive color information from other processors
13:                Wait until all incoming messages are successfully received
14:                $R_i \leftarrow \emptyset$                  ▷ $R_i$ is a set of vertices to be recolored
15:                for each boundary vertex $v \in U_i$ do
16:                    if $\exists (v, w) \in E$ s.t. $color(v) = color(w)$ and $r(v) \leq r(w)$ then
17:                        $R_i \leftarrow R_i \cup \{v\}$          ▷ $r(v)$ is a random number
18:                $U_i \leftarrow R_i$
19:
```

Fig. 1: `pseudocode of the algorithm`

ter throughout the work is the superstep size, we denote by superstep a share of the vertices of any process. The coloring phase of the algorithm is divided in superstep, in order not to send information concerning newly colored boundary vertices one by one but instead, packing those vertices through supersteps. Different superstep size affect performance (large superstep induce less communication overhead but more conflicts whereas smaller conflict lead to more frequent communication), we chose the size of 100 vertices to suit our needs the best. The algorithm can be divided into two distrinct phases, the first phase which we call ”tentative coloring phase”, this is the phase where each process color (or recolor) its vertices according to the information it has collected through previous phases, which vertices has to be recolored is determined in the next phase called ”conflict resolution phase”. Here, we detect conflicting boundary vertices, mark only one of them per conflict to be recolored. Communication concerning boundary vertices is made during the tentative coloring phase. The algorithm completes when no more conflicts are detected in the conflict resolution phase. Conflicts typically arise when the superstep of one process overlap with the superstep of another process. See above where we provide a pseudocode of the algorithm

3. IMPLEMENTATION DETAILS

This section introduces the technical details of our implementation of Boman graph coloring. Due to the flexible and general formulation of the algorithm, there are several strategies for implementing the algorithm, and their performance impact depends on both the system and their respective implementation. Specifically, this includes the choice of color assignment algorithm, the size of the superstep and the communication strategy. We implemented several versions of these choices to evaluate and reason about the performance impact of each choice.

We implemented the algorithm for a distributed memory setup using MPI and C/C++. We assume the graph is already distributed and every process has complete knowledge about the topology, in particular each process knows how many vertices and edges there are in total, as well as which process owns a particular vertex. Additionally, each process knows the set of boundary vertices, meaning the vertices which share an edge with another processor. We do not assume any special partitioning of the graph in contrast to the implementation in the original paper [**?**].

Each vertex in the graph has both a global and a local id, which are implemented using unsigned 32-bit integers. We assume that all the vertices of process i have smaller global ids than the vertices of process $i + 1$, and greater global ids than the vertices of process $i - 1$. If that is not the case, we compute a new vertex numbering that satisfies these conditions. This ordering is important because it allows computing the global id using only knowledge of the process that owns it and the local id. Additionally, the memory layout and addressing is simplified because the data can be laid out in memory in a contiguous manner.

Local for each process, the graph is stored using two C-style arrays and one integer variable. The integer variable holds the global offset, which is used to compute the global id of the vertices that are owned by this process. The two arrays are used to store the graph in compressed sparse row (CSR) format. Each process owns a continuous slice of rows in the adjacency matrix, and thus uses one of those arrays to store the offset at which the next row starts, and the other to store the columns that are nonzero. Because the adjacency matrix is binary, we do not need to store the values for the nonzero matrix entries, which are implicitly one. All the primitive data types are unsigned 32-bit integers to maximize the size of the graph we can store, while still keeping the memory footprint under control.

Colors are stored locally on each process in an array of unsigned 16-bit integers. Each process stores the color information for all the vertices. While this introduces a lot of redundancy and needs memory, it allows the processes to compute the next superstep independently and also facilitates the exchange of color informations during the communication phases. We minimize the memory overhead by using unsigned 16-bit integers, whose numerical range was sufficient for all our test graphs. The color array is indexed using the global id of the vertices.

As explained in Section **??**, color information is exchanged after each process completed a superstep. There are several options to implement the communication, and we explain the guiding principles that are common to all the variants we implemented here and refer to Section **??** for further details on the specific variants.

After each superstep, each process has to communicate the colors it changed to all the other processors. Because we store all the color information on each process, we use the MPI Broadcast function to distribute the updated colors to all the other processes. Even though only the processes that have an edge to the sending process actually need the color information, and even in this case do not need all the updated colors of this superstep, we decided to use broadcasts for scalability reasons: When increasing the number of processes the number of inter-process edges grows, until eventually most of the processes share edges. Additionally, broadcasting contigous chunks of memory is very efficient because the message does not need to contain any metadata and position information, especially in the first round of the algorithm. Knowing only the sender, the receiver can independently compute the position information and copy the message into the right place in the color array. There are no race conditions because each process only writes to the part of the array it owns and only performs read operations on the rest of the color array.

The communication using broadcasts is especially efficient during the first round of the algorithm because in this round every vertex is colored and all of the elements of the color array are set. Thus the messages only need to include the actual colors and the starting index to copy the colors into can be computed locally on each process knowing the sender of the colors. Starting from the second round however, the algorithm does not stream through memory as nicely as in the first round and only select colors are updated. To minimize the message size and the amount of communication needed, we treat the first round separately from the succesive rounds. Starting from round 3, two messages are broadcast from each process: one including the indices to update, and another including the new color values. While this doubles the amount of communication, we only send the values that change. It also introduces more complexity locally at each process because instead of using a memcpy call, each process needs to parse the messages and update the colors. Nevertheless, because the number of conflicts after the first round is a small fraction of the number of vertices, this communication scheme performs better.

After performing one round of vertex coloring, each process locally computes the set of conflicting vertices for the vertices it owns. This requires no communication, because at the end of each round each process owns the complete color information and thus only performs a lookup on the local color array. For each conflict, there are two vertices involved and only the vertex with the lower global id is recolored. This vertex is added to a local std::vector that holds the vertices to be recolored in the next round. To guarantee termination and correct communication, the maximum number of conflicts has to be sent to all the processes because all the processes have to know how many supersteps to perform.

The main guiding principles for our implementation were to make things as simple and predictable as possible. Especially during the first round where all the vertices are colored, the implementation can greatly benefit from caching and prefetching effects because most of the memory accesses are streaming through memory in a continuous fashion. For later rounds, the adjacent vertices are stored closely together in memory where the implementation again benefits from spatial locality. All the data is stored using the smallest data type permissible to reduce the memory footprint. Compared to a primitive implementation using linked lists or pointers, our implementation is tuned to leverage the power of the hardware, especially allowing the application to use caches and memory level parallelism efficiently.

Communication style. One major design choice when implementing a message-passing based algorithm is whether to use blocking or non-blocking communication. In the blocking case, all the processes wait for all the communication calls to complete before they restart another computation phase. This introduces a synchronization barrier and usually involves some processes waiting idle until all the processes arrived at the barrier. In the non-blocking case, the communication call immediately returns and the process is allowed to continue the computation. This allows overlapping the communication latencies with the computation.

In our specific case, we implemented both styles and evaluate them in Section 4. The main trade-off involved is how coloring vertices based on outdated color information affects the number of conflicts at the end of each round. In the blocking communication case, a vertex can only conflict with vertices colored at the same superstep, while using non-blocking communication it can also conflict with vertices colored in the next superstep. This is the case because during the next superstep the updated colors are still in-flight and not ready to be looked up yet. The smaller time spent waiting for communication to complete may however compensate for having to recolor more vertices due to conflicts.

Color assignment algorithms. Another important choice is how to assign colors to vertices. In this section, we introduce the 3 variants of color assignment we implemented, the first two of which were proposed in the inital paper **??** and the third being a new proposition extending the second variant.

The first variant implemented is the simple First Fit (FF) algorithm: For each vertex, the colors of the neighbouring vertices are looked up and the smallest legal color is assigned to the vertex. While First Fit is simple and efficient to implement, it has however one disadvantage in parallel applications: the color distribution is highly skewed. Small colors are assigned much more often, and thus the likelyhood for conflicts is involving those small colors is much higher.

To reduce the likelihood of conflicts in parallel applications, Staggered First Fit was introduced. Instead of all processes starting their color assignmets at the smallest, each process gets a different starting color and assigns colors starting from that value. Using an initial guess of the number of colors needed, the process assigns colors starting from 0 if there is no legal color in the interval between his starting color and the initial guess. This balances the color distribution because each process starts from its own base color and thus reduces conflicts. The assignment is however computationally more complex.

As a third variant, we propose "conflict-avoiding staggered first fit". The basic mechanism is identical to Staggered First Fit. Once the process overflows its assigned color range however, the conflict-avoiding variant does not proceed to assign the starting color of the next process. Instead, it starts assigning the last color of the range of the previous process, and then proceeds towards the smallest color. This allows for an even better distributed color assignment, because it avoids the most common colors of other processes and starts assigning rare colors.

Superstep size.

The last major design choice is the choice of superstep size. Smaller supersteps introduce more communication and synchronization points, but reduce conflicts because the local color information is better. Larger supersteps on the other hand allow for less idle times waiting for communication to complete. We chose to expose superstep size to the user as a runtime parameter for fine-tuning the algorithm to the specific graph to be colored.

4. EXPERIMENTAL RESULTS

In this section we present our experimental setup and evaluate the results taken from multiple runs of a large set of random and real-life graphs.

Experimental setup. Two different experimental setups were used. The first was a local server used for running randomly generated graphs of different sizes. The second was Euler Cluster **??** used to run real world graphs. Detailed information about both setups, is shown in Table **??**.

For the first setup we generated random graphs using the PaRMAT graph generator **??**. We generated graphs with a size of 1, 10 and 15 million nodes and vertexes with a 1:3, 1:7 and 1:15 ratio to the nodes of each graph. For each category 10 graphs were generated. For the second setup we ran each of the following real world graphs:

- Californian Road Network with 1.9 million nodes and 2.8 million vertexes
- Google Web Graph with 0.9 million nodes and 8.6 million vertexes
- Livejournal Social Network with 4 million nodes and 69 million vertexes
- Orkut Social Network with 3 million nodes and 234 million vertexes

As mentioned in Section 3, we run all graphs through the following different approaches:

- non blocking-staggered
- non blocking-avoiding
- non blocking-first fit
- blocking-staggered
- blocking-avoiding
- blocking-first fit

Finally, all graphs were run through a serial implementation, to compare the speed and the validity of the results, and, through the parallel graph coloring algorithm implemented in the widely used Boost library **??**.

Results.

In the experiments with the randomly generated graphs, there were no important observations to be made. As seen in Figure **??**, even though there is a speed up in regard to the sequential run time, it is not important and does not scale well. We believe that these results are mostly due to the fact that the graphs were not dense enough to produce any useful outcome, as mentioned in details below.

Starting with the real world graphs, Figure **??** presents the number of colors that each implementation used. We observe that, in all cases, the first fit approach is much more precise in comparison to all other approaches, and close, if not using the same number of colors, with the Boost's implementation. As discussed in Section 3, this was something expected as with the first fit approach generally less colors are used, moving the weight to the higher conflict numbers. We do observe less conflicts using the advanced assignments, but it mostly does not reduce the number of rounds needed, which is what makes those approaches expensive; there needs to be another conflict resolution phase. Also, there is much more computational complexity involved in selecting the colors, and this being "hot-path" code executed for every vertex, makes it much more expensive than the simple first fit assignment. In addition to this, there is an increase in the total number of colors used as the number of processors increases, as with more processors, more conflicts arise, forcing a usage of higher number of colors to solve the higher number of conflicts.

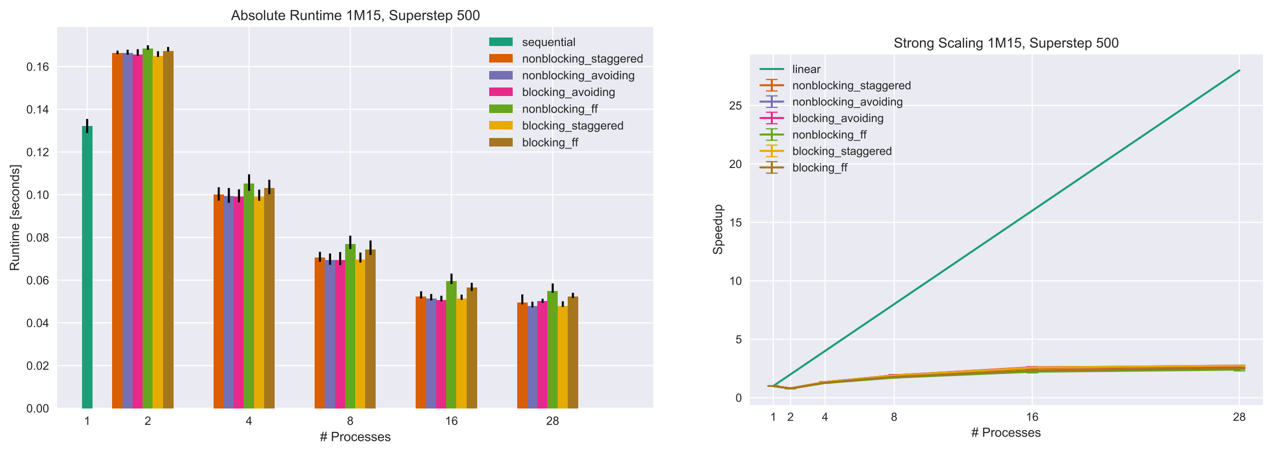


Fig. 2: Sample results for the random generated graphs

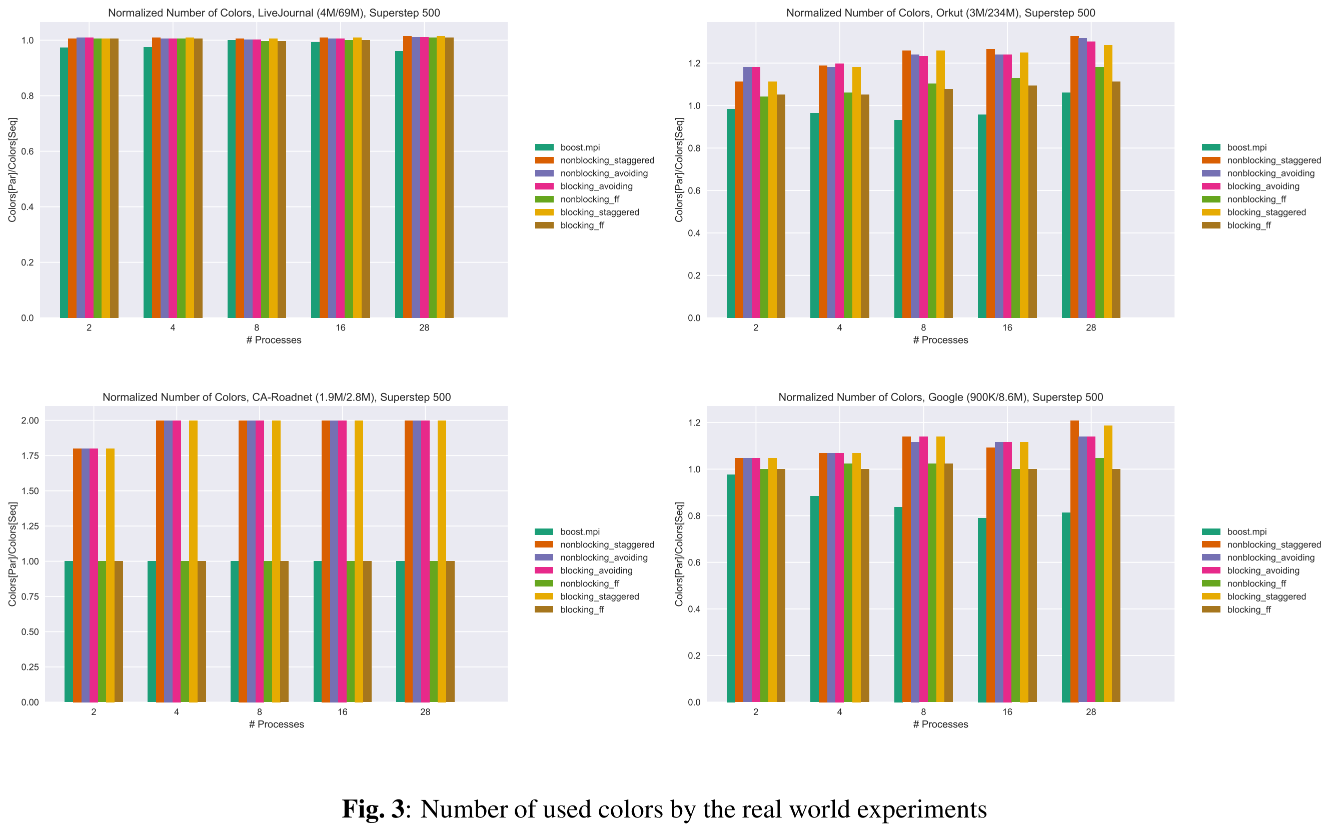


Fig. 3: Number of used colors by the real world experiments

Table 1: Experimental setup

Option	Randomly Generated Graphs	Real World Graphs
CPU	2x Intel Xeon E5-2697 v3	3x Intel Xeon E5-2680 v3
#Cores	28 total cores (2x 14)	28 total cores (2x 12 + 4)
RAM	256GB DDR4 ECC 2133 MHz	150GB DDR4 2133 MHz
Operating System	Ubuntu Server 16.04.3 LTS	Centos 7.4.1708
Kernel	4.4.0-87	3.10.0
Compiler	gcc-5.4.0	gcc-4.9.2
MPI	OpenMPI 1.10.2	OpenMPI 1.6.5
Boost Library	1.58.0	1.59.0

In Figure **??** the run time of each implementation is shown. The first observation we make is that our implementation is reasonably faster than the serial implementation and extremely faster in comparison to the Boost's implementation. Moreover, we observe that all non blocking approaches produce better results than the blocking approaches, and taking into account the results of the Californian Road Network graph, which is a sparse graph (1.9M nodes / 2.8M vertexes), we can see how the communication between cores affect the results. It is obvious that when the processors had to exchange more messages, in regards to the size of the graph, it affected heavily the result, even leading to run times more than their sequential counterpart. Furthermore, there seems to be no impact in the performance due to the use of the simpler first fit approach while we can observe a negative impact for the non blocking approaches, as the number of processors increase. Blocking communication scales better than nonblocking communication. Initially the differences are negligible, but with more processors it starts becoming important. Non blocking does introduce more conflicts because more vertices are colored with old information. With few processes it's possible to perform as good as blocking because even though there are many more conflicts they can be hidden in the communication latency and save some performance there, but introducing more processors also introduces more inter-partition edges and thus even more conflicts. At some point, it becomes too much to be handled. On the other side, when increasing the number of processors, the working set per processor decreases, and in total there are fewer communication phases, making the blocking communication approach more favorable.

Figure **??** show the scaling for the two denser real world graphs. The experiments show that the denser the graph, the better the scaling. There are even cases when the experiments returned superlinear results. As for denser graphs, each processor has more computations to do in each step, meaning the time between communications is longer because they have to look up more colors from other vertices. In sparse graphs, the run time is completely dominated by communication latency. Nevertheless, our implementation is fast when looking at absolute run time. Scaling, though, stops at some point due to the communication/synchronization overhead becoming too large, compared to the working set. Also, having more processors means that there are more edges that cross partition boundaries, which in turn gives more conflicts.

In terms of the superstep size, generally the results show that it does not have a significant impact to the end result. In the original paper the superstep size was 100, while in our experiments 500 seems to be the best, while increasing it does not influence run times or scaling. It does, though, introduce more conflicts, but this is balanced by the algorithm being able to proceed faster because there are fewer synchronization points and thus less time waiting for communication to complete.

5. CONCLUSIONS

Here you need to summarize what you did and why this is important. *Do not take the abstract* and put it in the past tense. Remember, now the reader has (hopefully) read the report, so it is a very different situation from the abstract. Try to highlight important results and say the things you really want to get across such as high-level statements (e.g., we believe that .... is the right approach to .... Even though we only considered x, the .... technique should be applicable ....) You can also formulate next steps if you want. Be brief. After the conclusions there are only the references.

6. REFERENCES

[1] Erik G. Boman, *A Scalable Parallel Graph Coloring Algorithm for Distributed Memory Computers*, 2005

[2] Assefaw Hadish Gebremedhin and Fredrik Manne, *Scalable parallel graph coloring algo- rithms*, 2000

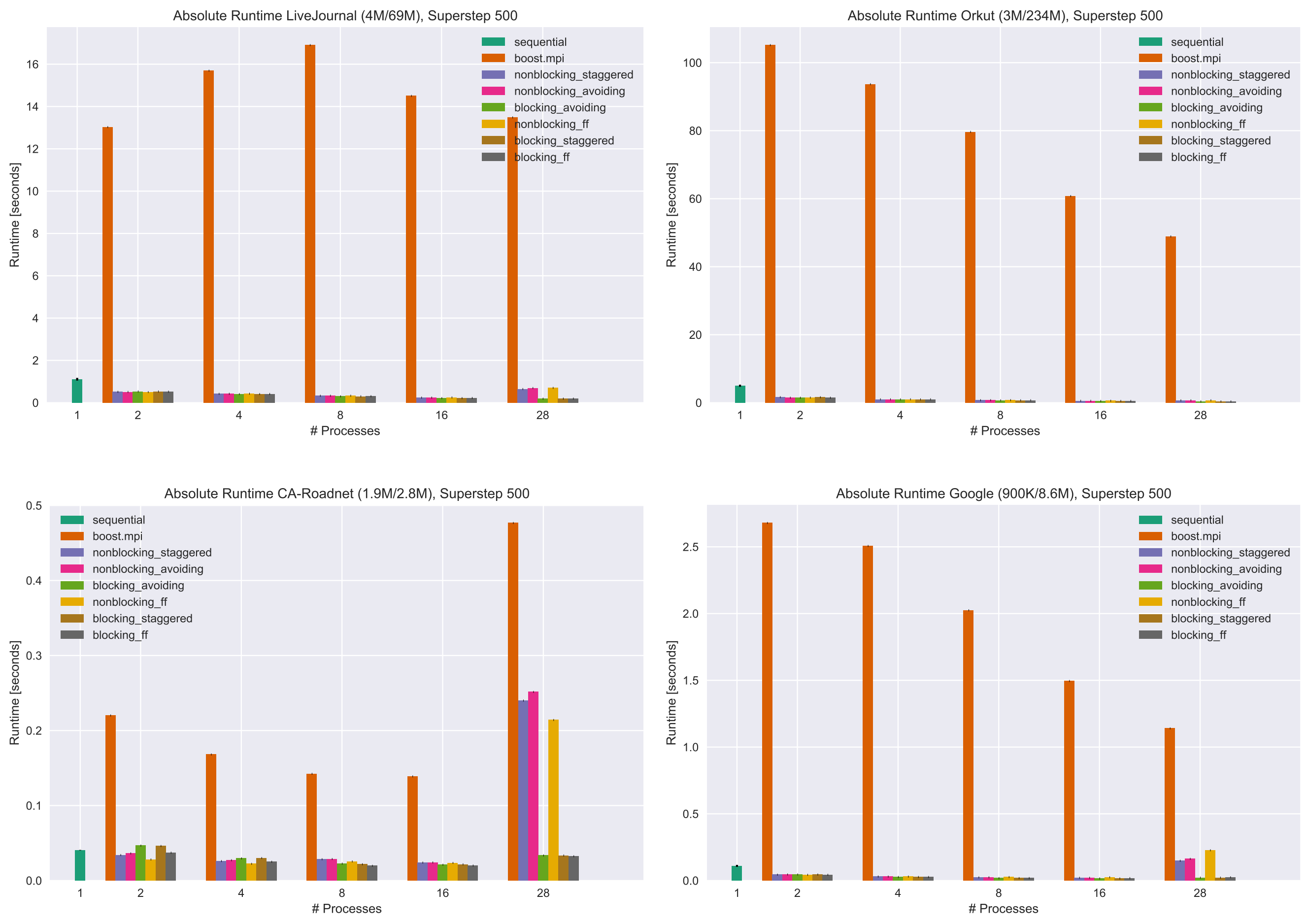


Fig. 4: Absolute run time of the real world experiment

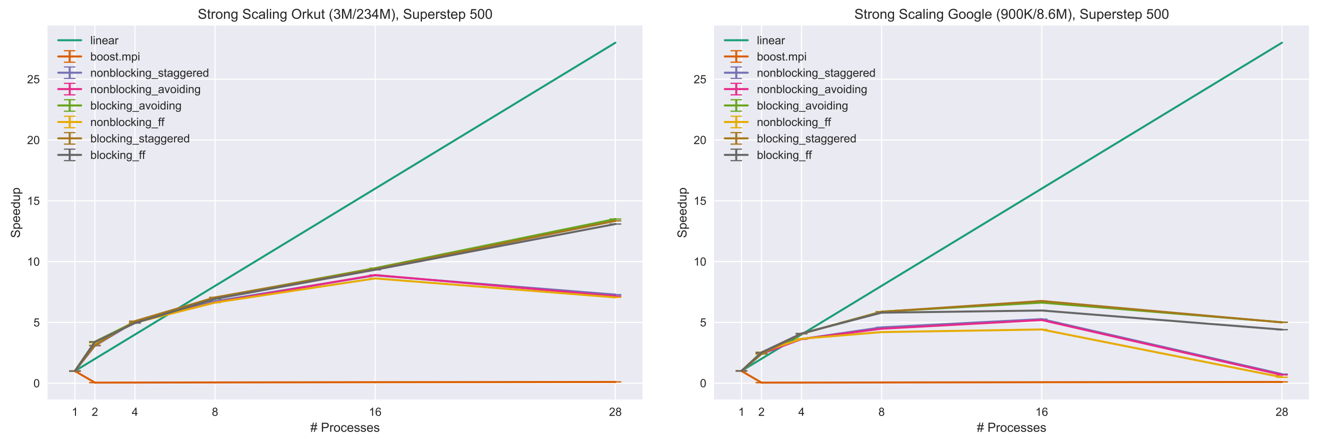


Fig. 5: Scaling for the real world experiments