Find Communities

Project development for the 2015 course in
Advanced Algorithms and Parallel Programming

Author:

Andrea Bignoli

Supervisors:

Fabrizio Ferrandi
Marco Lattuada

Department of
Computer Science And Engineering

Introduction to the problem of Community Detection

What are communities?

Intuitively a community is often defined as a set of nodes that are more connected than it would occur in a graph with the same number of edges but distributed randomly. In other words, communities are highly interconnected set of nodes in a graph.

Modularity

Finding communities in a network is a graph partitioning problem.

To measure the quality of a given partitioning the concept of modularity has been defined as:

$$Q = \frac{1}{2m} \sum_{i,j} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(ci, cj)$$

Where:

A_{ij} represents the weight of the link between node i and node j

$m = \frac{1}{2} \sum_{i,j} A_{ij}$

$k_i = \sum_j A_{ij}$

$$\delta(c_i, cj) = \begin{cases} 0 & ci \neq cj \\ 1 & ci = cj \end{cases}$$

Modularity

From its definition we can notice that the modularity is a scalar value between -1 and 1.

$$Q = \frac{1}{2m} \sum_{i,j} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(ci, cj)$$

We can also point out that:

- In the sum i can be equal to j (i.e. self loops count in the measure)
- A_{ij} can be equal to zero
- The computation of this measure, considering the above formula can get to be quadratic with respect to the number of nodes if performed without any mathematical optimization

Modularity – Efficient Computation

Because of the previous considerations we may derive from the original formula:

$$Q = \sum_k Q_k$$

Where Q_k denotes the modularity contribute due to community k.

Modularity – Efficient Computation

$$\frac{1}{2m}\sum_{i,j\in C_k}\left[A_{ij}-\frac{k_i k_j}{2m}\right]=\frac{1}{2m}\left[\sum_{i,j\in C_k}A_{ij}-\sum_{i,j\in C_k}\frac{k_i k_j}{2m}\right]$$

$$=\frac{1}{2m}\left[\sum_{i,j\in C_k}A_{ij}-\frac{\sum_{i\in C_k}k_i\sum_{j\in C_k}k_j}{2m}\right]=\frac{1}{2m}\left[\sum in_k-\frac{\sum tot_k\sum tot_k}{2m}\right]$$

$$=\frac{1}{2m}\left[\sum in_k-\frac{(\sum tot_k)^2}{2m}\right]$$

Where:

$$\sum in_k=\sum_{i,j\in C_k}A_{ij}$$

$$\sum tot_k=\sum_{i\in C_k}k_i$$

Given the needed parameters, this allow the computation of modularity in O(k) steps, where k is the number of communities in the given partitioning.

This is far more efficient than the original, and it's the measure that the reference implementation, and mine, actually use.

Modularity Maximization

Having defined modularity as in the previous slides, the problem of finding communities in a graph is reduced to the maximization of the modularity measure.

The problem is that exact modularity maximization has been proven to be NP-complete in the strong sense[2].

For these reason, in modularity maximization the goal is not to find the optimum partition, but good ones as fast as possible. Several heuristics have been developed to accomplish this task. One of them is the Louvain method.

The Louvain Method

Louvain Method

The Louvain method was first described in 2008 by Blondel et. al.[1]

The Louvain method is an heuristic for modularity maximization able to produce significantly better community partitions in large graphs (up to millions of nodes) and faster than any of the other methods defined at the time of its definition.

Louvain Method – High level description

The algorithm is composed by multiple phases.

During the first phase the algorithm assigns each node to a separate community (called “singlet”).

Then for each node, the algorithm considers the modularity gain that could be obtained by moving the node from its community to any of the communities to which its adjacent nodes belong. If a positive gain is found, the algorithm select the adjacent community producing the maximum one and moves the considered node to that community. This analysis is performed sequentially over all nodes.

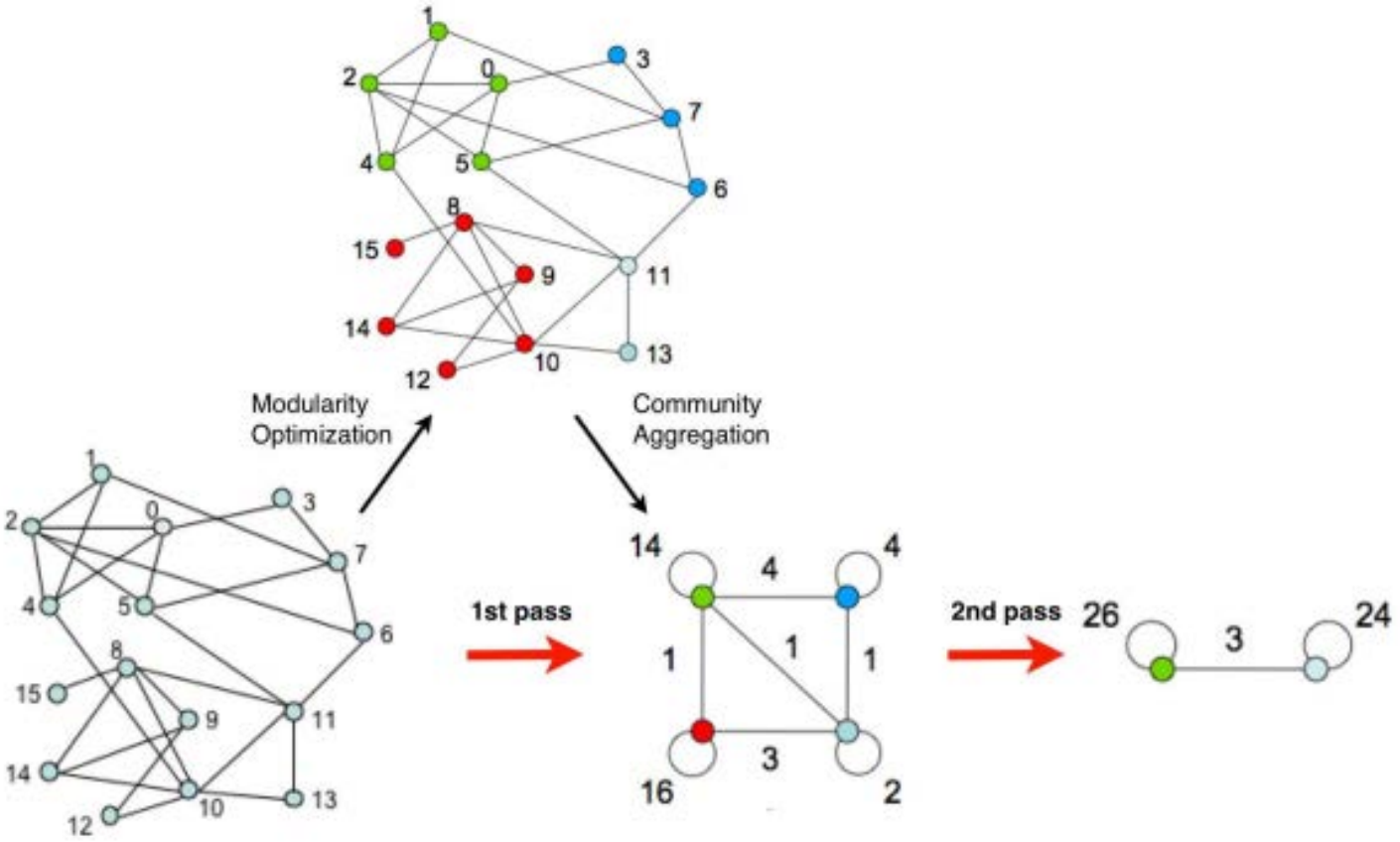
Then, the algorithm computes the modularity gain produced by the previous iterations. If the gain is greater than a given threshold (≥ 0), the iterations over all nodes are repeated.

Louvain Method – High level description

In the opposite case, communities are collapsed into super-nodes. Links internal to the communities become self loops. Links from nodes belonging to a community i to nodes belonging to a community j are unified in a single edge from community i to community j, whose weight is the sum of the original edges.

This concludes a phase. Then, a new phase starts, receiving as input the new graph. This process continues until the modularity gain produced by a phase is above a threshold (≥ 0).

Louvain Method – Graphical Representation



Taken from "Fast unfolding of communities in large networks" [1]

Louvain Method – Computing Modularity Gain

In practice, the gain produced by the transfer of a node to another community is calculated by:

- Removing the node from its community
- Computing the gain that would be achieved by moving the node into the target community

The gain produced by the second step is computed using the formula:

$$\Delta Q = \left[\frac{\sum_{in}+2k_{i,in}}{2m} - \left(\frac{\sum_{tot}+k_i}{2m}\right)^2\right] - \left[\frac{\sum_{in}}{2m} - \left(\frac{\sum_{tot}}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2\right]$$

Where:

$$k_{i,in} = \sum_{i,j \in C} A_{ij}$$

The node removal modularity loss is computed in the same way.

Louvain Method – Computing Modularity Gain

On transfer of node i to a community, the following community parameters are consequently updated as such:

$$\sum_{in} = \sum_{in} + 2k_{i,in} + self(i)$$

$$\sum_{tot} = \sum_{tot} + k_i$$

Louvain Method – Considerations

The Louvain method proved to be extremely effective in modularity computation. Some of the key reasons of its success are:

- Each node transfer happen using the latest information available for the computed communities
- The number of communities is reduced sharply during the iterations, therefore reducing the computation time for the next node transfers
- Each node can be transfered an unlimited number of times so that the algorithm, even if operating in a greedy way, can produce high values of modularity

Louvain Method – Implementation Notes

As observed, computations need the following parameters to operate:

$$\sum_{tot} \quad \sum_{in} \quad k_i \quad k_{i,in}$$

The first three parameters can actually be stored in O(n) space, and updated efficiently on node transfer. This observation doesn't hold for the fourth one, but I'll expand on this in a few slides.

This means that at the start of the phase there should be an initialization process in which all those three values are computed and stored.

On node transfer, the first two will be updated both for the source and the destination community using the formulas shown.

Preliminary Considerations In Parallelizing The Louvain Method

Parallelizing The Louvain method

The first consideration to be done, is that we are working on an heuristic.

This does not only mean that we aren’t going to obtain the optimum modularity.

This means that every method we develop should take into account the fact that we don’t know in advance the properties of the input graph. This means that in practice, the effectiveness of the methods varies depending on several properties of the input, and not only its size.

This should be a key design principle.

Parallelizing The Louvain method

A really important consideration to be done, that is also relevant for the sequential case, is that memory usage should be carefully considered.

The reason is that, since our goal is to produce algorithms able to analyze graphs even bigger than the ones initially considered by the authors of the Louvain method, the orders of magnitude are at least 10^6 for the number of nodes, and 10^9 for the number of edges.

Not only memory usage should be limited, but it also should be performed in a smart way. This means that data locality can play a relevant role in the analysis.

The above considerations are even more relevant, considering the importance of correct shared memory usage among different threads.

Parallelizing The Louvain method

As illustrated, the Louvain method is sequential in its definition. It is hard to efficiently parallelize the method and achieve significant speedups.

One possible idea would be parallelizing the best neighbor computation inside the sequential iterations over all nodes. But this approach wouldn't produce great results. In fact, when a node is considered, the following steps are performed:

1. Computation of neighbor communities and relative $k_{i,in}$s.
2. Usage of the previous computation and of community parameters to select the best among available transfers.

In the following analysis I will denote as:

- n: the number of nodes
- m: the number of edges
- k: the number of communities

Parallelizing The Louvain method

The first step can be parallelized, but additional memory is needed to reduce the $k_{i,in}$s. Moreover the reduction causes a parallelization cost that probably makes this operation be not even worth to parallelize.

It would be possible to store the $k_{i,in}$s, and effectively remove the first step, but this would require us:

- To pay O(n+m) memory. Using a matrix O(n*k) is unfeasible, considering the size of the input graphs and the fact that k is equal to n at the start of the phase, making the required memory $O(n^2)$.
- To update such values on each node transfer, which is rather difficult to perform in an efficient way.

Parallelizing The Louvain method

The second one is easier to parallelize, yet we pay a parallelization cost due to the reduction of max.

And even more important, the degree of parallelism we could obtain would depend on the degree of the node, limiting it by a great amount, to the point where I can’t see significant gains in this parallelization approach in most cases.

Obviously anyway, it would depend on the graph structure.

Parallelizing The Louvain method

So, having discarded that type of parallelization, lets analyze the original algorithm to find the most promising parallelization strategy.

We can’t obviously parallelize the subsequent iterations over all nodes during each phase, as each iteration needs the results of the computation of the previous one.

So, the most profitable strategy is probably iterate over the nodes in parallel. But this introduces several issues. The first and most important one is the fact that during the analysis of a node the data available would not be updated to the last correct value. In fact, one of the nodes in its own community might have left. Or, on the other hand, one of its neighbors may have been transfered to a community.

Can these situations have an impact over the correctness of the computation?

Yes, definitely.In fact, node transfers in these conditions may even produce negative gains. A full demonstration of this, not reported here for brevity, can be found in [3].

Moreover there are also other potential issues, like the “swap” scenario.

The easiest example to represent it is the situation in which two connected nodes i and j both decides to move to other node’s community. If i and j are considered in parallel this simply produces a swap. This hypothetic scenario is easier to imagine if both communities are singlets, like at the beginning of each phase.

It is important to note, anyway, that even if swap conditions are negative for efficiency of the algorithm, they can’t ever lead it to not terminate since a positive gain is always needed to repeat the iterations.

Parallelizing The Louvain method

A simple minimum label heuristic could actually be used to solve the problem in case of singlets:

In case a node i, belonging to a singlet community, can produce a modularity gain by moving into the community of node j, which is also a singlet, it actually can be transfered only if the label associated to node i is smaller than the one of j.

A label could be, for example, the index of the node itself.

As shown in [3] this approach can be actually expanded to solve also other local maxima scenarios (that would, in any case be solved in the subsequent phases of the algorithm).

Preliminary Notes About My Implementation

Preliminary Notes About My Implementation

My implementation is not a solution to the problem of finding communities.

My implementation is a framework in which solutions to the problem of finding communities can be integrated and tested with different execution options given as input.

In fact, my implementation includes already four different algorithms to solve the given problem. One of them being an optimized version of the original sequential one[4], while the other three are different parallel algorithms that implement the Louvain method in different ways.

Even if the obtained results are not on par with the state of the art of Grappolo[5], they still can provide valuable contributions to the study of the subject. In particular, chunk-based analysis could provide improvements even to the state of the art. This hypothesis is very recent and still to be confirmed though.

Preliminary Notes About My Implementation

My implementation is publicly available at:

https://github.com/andbig/find-communities-OpenMP

Preliminary Notes About My Implementation

The best way to show the vast number of available algorithms and execution options is the readme included with the source code.

--- Usage:

./find-communities-OpenMP input-file [options]

```
-h              Shows help
                  * Ignores the rest of the input while doing so.
-f number       Use file format identified by number.
                  * Default is 0
                  * Complete list of available file format options
                    can be found below
-t              Number of threads to use during parallel execution.
                  * Must be greater than zero
                  * Default is 1
-a number       Execute the version of the algorithm identified by number.
                  * Complete list of available algorithm versions can
                    be found below
                  * Default is 1
```

Preliminary Notes About My Implementation

```
-p number      Stop phase analysis when phase improvement is smaller
               than number.
                 * Number must be between 0.0 and 1.0
-i number      Stop phase internal iterations when iteration
               improvement is smaller than number.
                 * Number must be between 0.0 and 1.0
-e number      Enable execution option identified by number.
                 * Complete list of available execution options
                   can be found below
-c file        Save communities obtained in each phase in file.
-g file        Save graphs obtained in each phase in file.
-b number      Turn benchmarking on.
                 * Perform the given number of benchmark runs
                 * Disable file outputs and screen logging
                 * Disable verbose logging
```

Preliminary Notes About My Implementation

Available algorithm versions:

```
-a 0          Sequential
                 * Original Louvain method implementation optimized
-a 1          Parallel (Sort & Select)
                 * Iterations over all nodes are done in parallel,
                   potential transfers are sorted and selected by
                   computed modularity increase
-a 2          Parallel (Naive partitioning on first phase)
                 * Partition the input graph during first phase
                   and performed and optimized Louvain version
                   over all partitions. Next phases are performed
                   using sequential implementation
-a 3          Parallel (Sort & Select Chunk)
                 * Iterations are divided in chunks in which iterations
                   over all nodes are done in parallel, potential transfers
                   are sorted and selected by computed modularity increase
```

Preliminary Notes About My Implementation

Available execution options:

```
-e 0          Use a number of partitions equal to the smallest
               power of two greater or equal to the number of threads
               during parallel sorting.
                 * By default a number of partitions equal to the biggest
                   power of two smaller or equal to the number of threads
                   is used
                 * Applies only to parallel algorithms with sorting
-e 1 number   Use the specified chunk size during the iterations of
              Parallel (Sort & Select Chunk).
                 * Default is 2000
-e 2          Applies the vertex following heuristic before the first phase.
                 * At the moment it is meant just for testing purposes,
                   since it is not implemented efficiently
```

Preliminary Notes About My Implementation

Available file format options:

```
-f 0          Edge list, not weighted.
                 * i.e. each line is of the form:
                   source-node destination-node
-f 1          Edge list, weighted.
                 * i.e. each line is of the form:
                   source-node destination-node edge-weight
-f 2          Metis format.
```

Indexes must be non negative, and weights should be greater than zero.

Output is undefined if any of the above conditions is not met.

For benchmarking purposes, I included also the bash script collect-performance-figures.sh

Usage:

./collect-performance-figures.sh input-graph-file file-format output-benchmarks-file benchmark-runs scale-up-to-threads

Working Environment

Input Graphs

In this initial analysis I decided to focus on a limited number of input graphs.

This was necessary since, due to the lack of available time, this was the only possible method to gather valuable experience for the development of new and better heuristics.

I want to explicitely point out that, to perform a complete analysis on the subject, more input graphs should be studied. Yet, the graphs analyzed here provided on themselves plenty of information that allowed me to design new and better heuristics, identifying the drawbacks of the initial versions.

Input Graphs

The chosen graphs, available at [6], are:

Input graph	***Vertices***	***Edges***	***Max degree***
cond-mat-2003	31,163	120,029	
cond-mat-2005	40,421	175,691	
CNR-2000	325,557	2,738,970	18,236
coPapersDBLP	540,486	15,245,729	3,299

Hardware

Intel® Core™ i5-3570K Processor
(6M Cache, up to 3.80 GHz)

# of Cores	4
# of Threads	4
Processor Base Frequency	3.4 GHz
Max Turbo Frequency	3.8 GHz
TDP	77 W

Intel® Xeon® Processor E5-2620 v3
(15M Cache, 2.40 GHz)

# of Cores	6
# of Threads	12
Processor Base Frequency	2.4 GHz
Max Turbo Frequency	3.2 GHz
TDP	85 W

Composed by 2 packages, for a total of available 24 cores.

The First Attempt In
Parallelizing The Louvain Method
Sort & Select

Sort & Select

The impact of all the issues described in the previous slides over the parallelization of the algorithm was definitely not clear at the start of my analysis.

This is clearly expectable since, as I mentioned at the start of this section, each consideration made must take into account the fact that the structure of the input graph is completely unknown.

Ignoring the negative gains situation is feasible, at least partially. But saying whether this will have a negative impact over the results is not. We can just make forecasts, based from the analysis of the problem and the experience and knowledge that can be extracted from previous work performed by others.

Sort & Select

Since at the beginning of the project I had access only to the results of my analysis, but I didn’t have any sort of experience with the problem, I could not just ignore the negative gain scenario.

For this reason, I decided to avoid it completely. At least for the first implementation of the algorithm.

Even though the obtained results were not bad, it later turned out the most efficient way to deal with negative scenarios was actually another one.

Yet this first parallelization attempt already produced decent results on its own and offers valuable contributions to future analysis.

Sort & Select

The key observation is that whenever a node is transferred from a community to another, only the source and destination parameters are modified using the formulas:

$$\sum_{in} = \sum_{in} + 2k_{i,in} + self(i)$$

$$\sum_{tot} = \sum_{tot} + k_i$$

Any other community is not influenced by the exchange. Any node transfer computation not having any community in common with the performed exchange, will happen just as if the exchange never happened, even in the sequential one.

I define as ***independent transfers*** a pair of transfers for which:

$$\{C_{src_1}, C_{dest_1}\} \cap \{C_{src_2}, C_{dest_2}\} = \varnothing$$

As observed in the previous slides any computation of one of the transfers is completely ininfluenced by the execution of the other one.

This means that as transfers happening in parallel are independent, there is the garantee that no negative scenario can happen.

In fact, performing multiple iterations, the algorithm can iteratively increment the modularity by performing multiple parallel independent transfers.

Sort & Select

The structure of the algorithm stays the same, being still composed by several repeated phases.

At the start of each phase each node is assigned to a singlet, and the initialization of all the other parameters is similar to the original one, performed in parallel whenever possible.

Then iterations over all nodes are performed in parallel. Instead of applying the best transfer found though, the algorithm stores all potential transfers.

Then the stored transfers are sorted by decreasing modularity gain in parallel.

After that step the transfers are sequentially scanned and selected whenever the node and the source and destination community have not already been selected for a transfer. This produces a set of independent transfers.

Then transfers are performed in parallel. Since they are independent by construction, no negative gain can arise.

After that the modularity gained is computed. If above a given threshold the iteration is repeated. I will denote as ***global iteration*** the set of previously described steps.

When the gain is below the threshold the algorithm produces the new community graph and proceeds to the next phase.

The algorithm stops, just as the sequential, when a phase doesn’t produce sufficient gain.

Sort & Select– Neighbor Computation

The computation of neighbors is actually a relevant step, both for time efficiency and for data. This is true for multiple reasons:

- O(m) calls will be performed to the chosen data structure to update $k_{i,in}$ during each global iteration.
- O(n) calls can be performed during the analysis of each node to the procedure to extract a community and its $k_{i,in}$ from the chosen data structures to perform transfer computations.
- If using a dynamically allocated data structure, malloc locking among threads may actually be a bottleneck

Sort & Select – Neighbor Computation

In my first implementation I used, just for testing, a sorted linked list as a data structure to hold the $k_{i,in}$s of the neighbor communities. Needless to say, that wasn't really good, since the insertion operation had a complexity of O(n) and it was called O(m) times during a global iteration. But that was just to test and gain experience over the problem I was dealing with.

A possible good solution would have been a Treap, but, if implemented as a dynamically allocated tree, it probably would have still caused problems with malloc locking.

I later changed that data structure with an O(1) time complexity on insertion and query and an O(n * nthreads) space requirement. Basically it consists of a double array that allows both fast insertion and query with cleanup. I won't go in detail just for time constraints. At the current stage, this data structure has been integrated just in the Sort & Select Chunks algorithm version.

Sort & Select – Parallel Sorting

To perform the parallel sorting of the transfers I divide the input transfers array into a number of chunks equal to the biggest power of two smaller or equal to the number of threads, run quicksort in parallel over all of them and finally merge the results in a merge sort manner.

The advantage is that quicksort usually produces good results in practice, and at each step the parallelism among threads is maximum. The drawback is that this doesn’t exploit fully the number of threads available, if different from a power of two.

There is anyway an option in my implementation to change this behavior.

This is a first version and may be updated in the future for a more efficient component.

Sort & Select– Initial Objections

- The number of potential transfers is O(m). This means that at each iteration the algorithm could spend O(m*log(m)) time on sorting during each global iteration.
  - From a complexity analysis point of view, this is true. But in practice, this isn't really the case, as the number of positive gain transfers is limited to an amount generally much smaller than that. Yet, the worst case remains. In my testing I didn't observe particular problems due to this, though. In any case, there is a solution: instead of storing each potential transfer, store a smaller amount. To do that several policies may be considered. For instance, the simplest one would consinst in storing just up to a constant number of edges per node. Thus, the complexity would be reduced to O(n*log(n). If this bound is not accettable the transfers may be directly allocated in a fixed size data structure of even smaller size. To recap, even if in the current implementation that observation is correct, it didn't provoke issues until now, and it could be fixed without altering the algorithm.

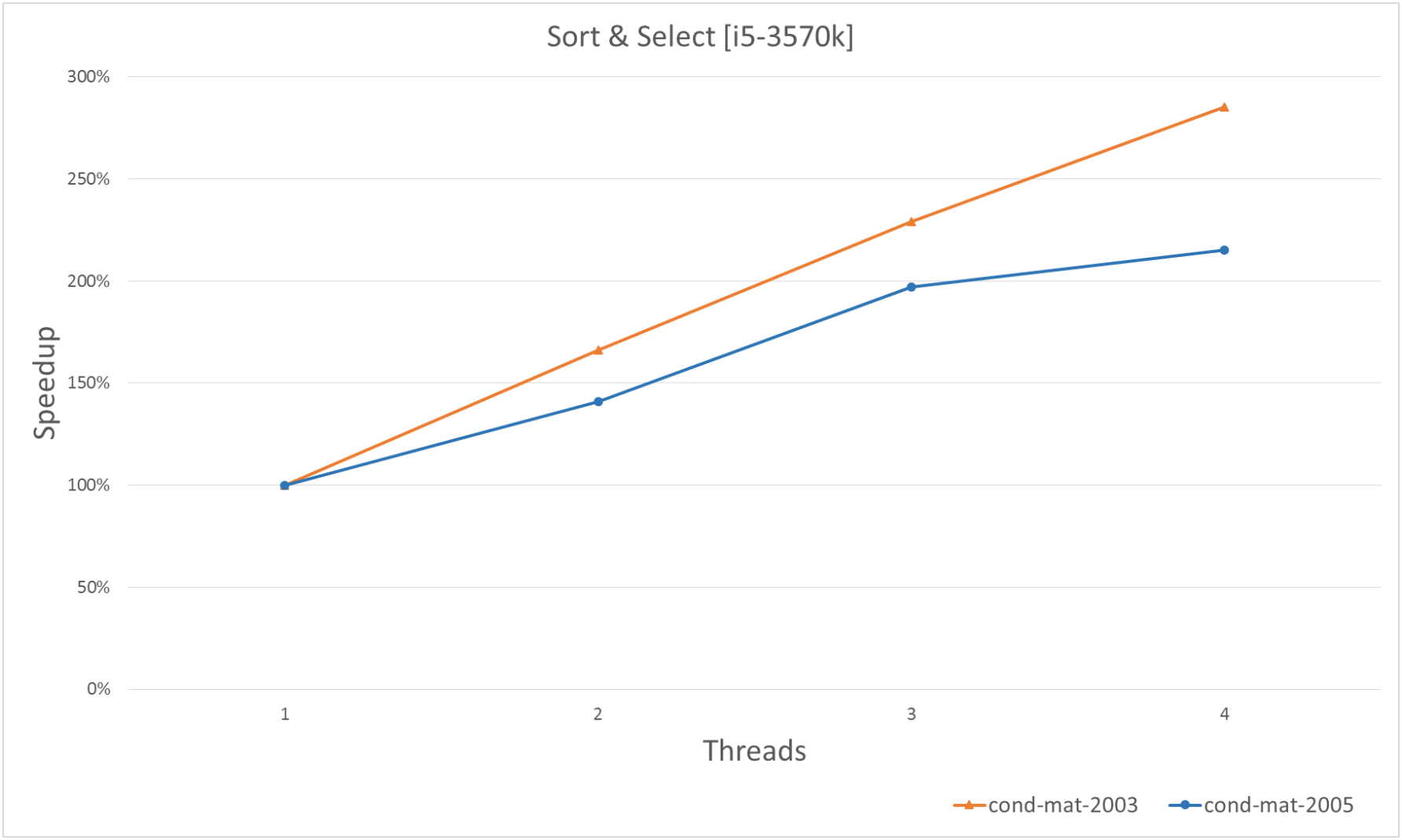
Sort & Select – Implementation

In the implementation the option to execute *Parallel (Sort & Select)* is –a 1, in the current version.

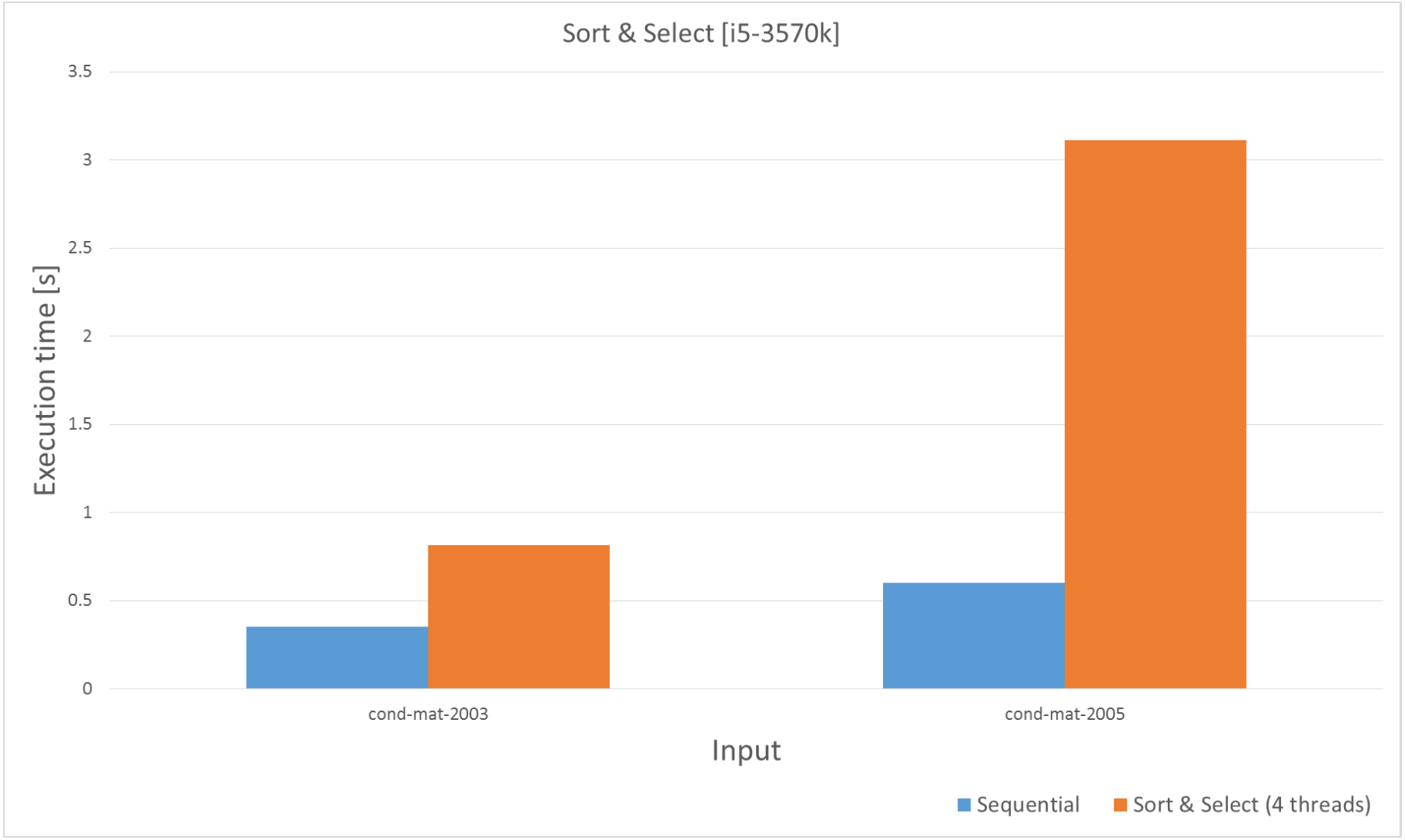
Specified number of threads is used.

In the following slides I am going to show some of the results obtained. Measurements are averaged over 10 runs of the algorithm.

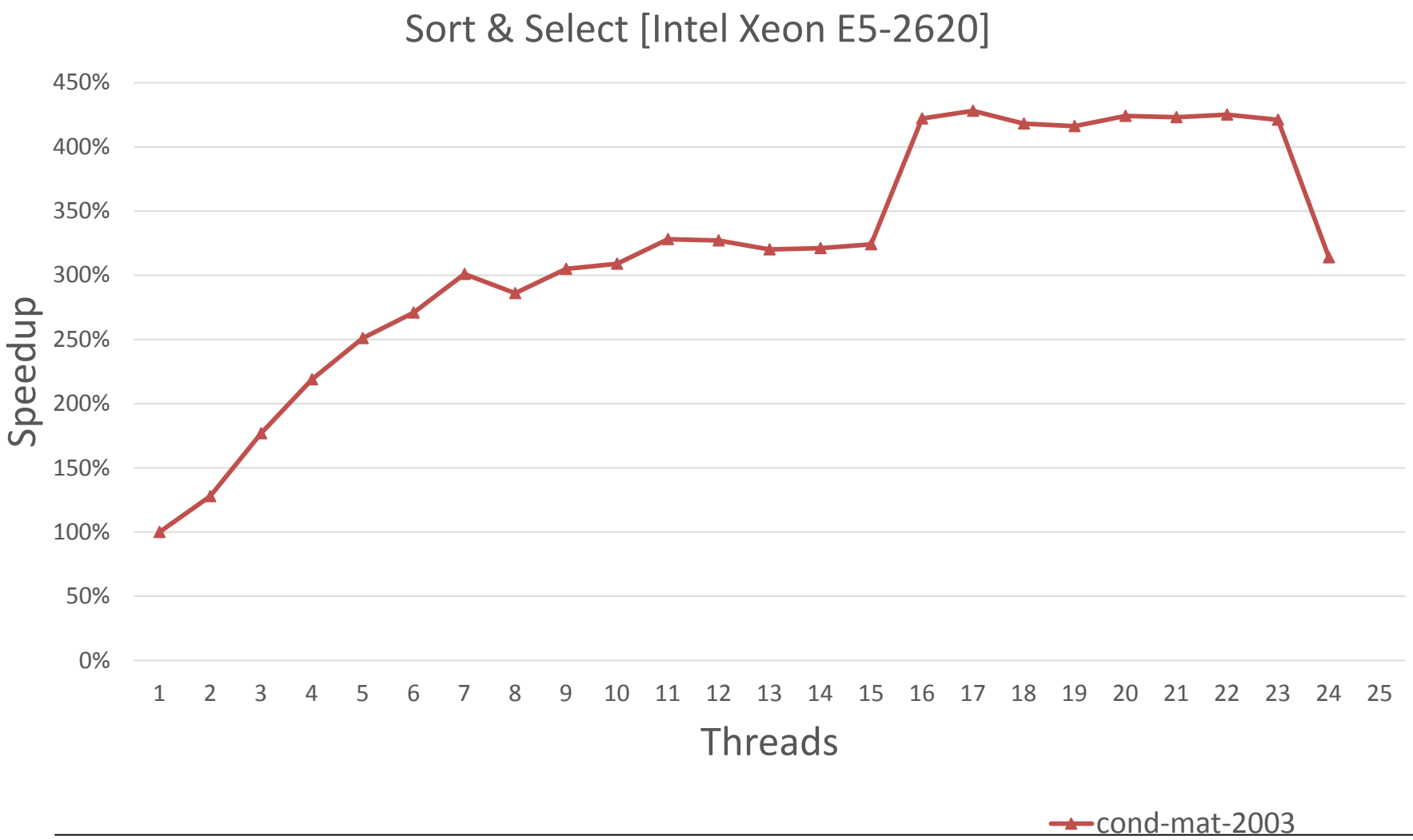
Sort & Select - Results



Sort & Select - Results



Sort & Select - Results



Sort & Select - Results

As the reader can easily notice, while the parallelism achieved by the algorithm is not that bad, the performance compared to the sequential implementation is disappointing.

This actually get even worse on other graphs considered in this analysis, like cnr-2000.

Parallel (Sort & Select) – The Real Issue

The algorithm has a drawback in its definition that I didn't consider relevant enough at the beginning but seems to be relevant in the post-implementation analysis.

At each phase of the algorithm just one node can go into a specific community (or leave it). Every other node that would go there to maximize modularity has to wait the next global iteration. Meanwhile, for that, the algorithm must pay the iterations over all nodes, the sorting and the selecting of node transfers. This situation can happen with a frequency depending on the structure of the input graph.

This is a prime example of what I wrote at the beginning of this analysis: the success of an heuristic depends on the characteristics of the input graph. In any case, complexity analysis is far from being sufficient in this case. This will become even clearer later, in the form of much more positive results.

Parallel (Sort & Select) – The Real Issue

Anyways, I already devised two solutions to this problem.

One of them is the introduction of resolution queues to handle the conflicts among transfers in an efficient way. This would also make the reduction of the number of stored potential transfers even more beneficial. The resolution queue could operate in several ways that are not investigated yet. Anyways I already performed the necessary reductions over the previous formulas for modularity computation, that I don't fully report here for brevity. My results shows that if node j wants to join the community of node i and performs its computation obtaining a potential positive gain of ΔQ_0, but then i leaves that community, the actual gain that node 2 would get by joining the community anyways can be computed easily given A_{ij} and the results of the previous computation.

The second solution will instead be the main topic of the following section.

Chunk Analysis

Chunk Analysis

The second solution to the main drawback of Sort & Select has already implemented with very good initial results. It implements a chunk partitioning strategy.

Basically global iterations are divided in chunks executed sequentially. In the first implementation of this strategy chunks are divided equally over node indexes.

e.g. In the case there are four chunks, during the analysis of the second chunk of the first global iteration the first step is still the parallel iteration over the nodes. Instead of iterating over all of them, the algorithm iterates over the second quarter of the nodes of the input graph. The transfers are then sorted, selected, performed just like before, but just for that quarter.

Chunk Analysis – Why Is It Good?

The main reason why this approach is so good is that it allows subsequent chunks to run with much newer and better information. This ultimately should always lead to a ***reduction in the number of iterations***.

It is important to note that this approach is beneficial not only to my parallel implementation, but potentially to any heuristic that performs node iterations in parallel. The state of the art, Grappolo[4], actually falls in this category. I still have to confirm whether this is actually true, but my initial analysis shows that is approach could even benefit Grappolo, when vertex coloring is not activated.

There are three observations to point out though:

- It may be incompatible with other heuristics
- There may be a loss in data locality
- Some computations may need to be repeated at the start or at the end of each chunk, depending on the algorithm used

None of the above though, seems to be particularly dangerous. Yet it should be tested.

Chunk Analysis – Why Is It Good?

It’s not only about reducing the number of iterations. It’s also about ***making them faster***.

This can easily happen because, expecially in the first iteration of the first phase, the number of non-empty communities is reduced faster. This means that the complexity in computing the cluster of neighbor communities for any given node is reduced, and also the number of considered potential transfers.

This effect is repeated for O(m) operations. This is the second reason why I see so much potential in this approach.

A Small Digression

In fact Chunk Analysis is just an example of a more general design principle, that in the articles I had the opportunity to read so far, it's not rightfully highlighted.

There are two sources for improvements given the structure of the algorithm:

- Reducing the number of iterations per phase
- Reducing the time spent on each iteration

While this is apparently a really simple point, it's also easy to loose track of it while designing new heuristics, given the huge number of different possibilities. Ultimately this comes down to one of the reasons the original Louvain method was so successful in the first place:

Using the best (last) available information

A Small Digression

This is actually worth noticing because:

- It can lead to the definition of other heuristics, obviously
- We may also use some pre-computed statistical information on the graph to perform computations in a more efficient way. This is, anyway, very complex and outside of the possibilities allowed by current, still limited practical experience on the problem

Chunk Analysis – Why Is It Good?

Back to the main topic, Chunk Analysis actually brings a benefit more to Sort & Select, that is the reason I mentioned it as a solution for its drawback.

By the way it is defined, Chunk Analysis applied to Sort & Select allows a community to be used up to c times during a global iteration in total, where c is the n. This happens because transfers between communities have to be indepent just internally to a single chunk analysis. So a community can be used just once during a single chunk analysis but up to c times during a global iteration.

It should be noted that while this is not a real complete solution to the issue, it greatly helps.

Sort & Select - Chunks

Sort & Select - Chunks

This algorithm version is the result of the combination of:

- Sort & Select
- New data structure for neighbor communities computation
- Chunk Analysis

For clarity I'll report here the same description previously given:

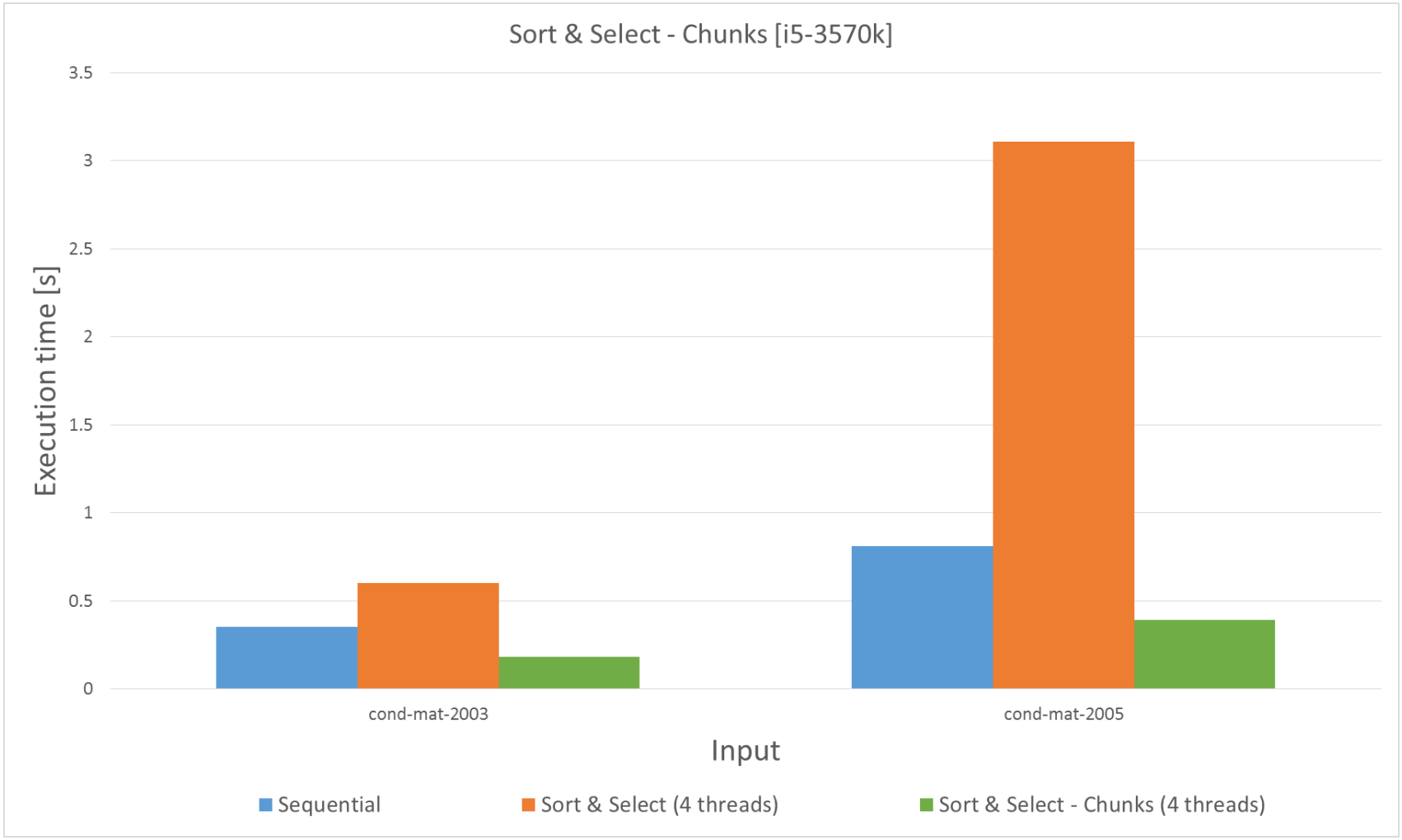
Global iterations are divided in chunks executed sequentially. In the first implementation of this strategy chunks are divided equally over node indexes.

e.g. In the case there are four chunks, during the analysis of the second chunk of the first global iteration the first step is still the parallel iteration over the nodes. Instead of iterating over all of them, the algorithm iterates over the second quarter of the nodes of the input graph. The transfers are then sorted, selected, performed just like before, but just for that quarter.

Sort & Select - Chunks - Results

The results show great improvements over the previous Sort & Select. It should also be pointed out that there are some optimizations yet to be implemented, so its final performance should be significantly better than the one shown here.

Sort & Select - Chunks - Results



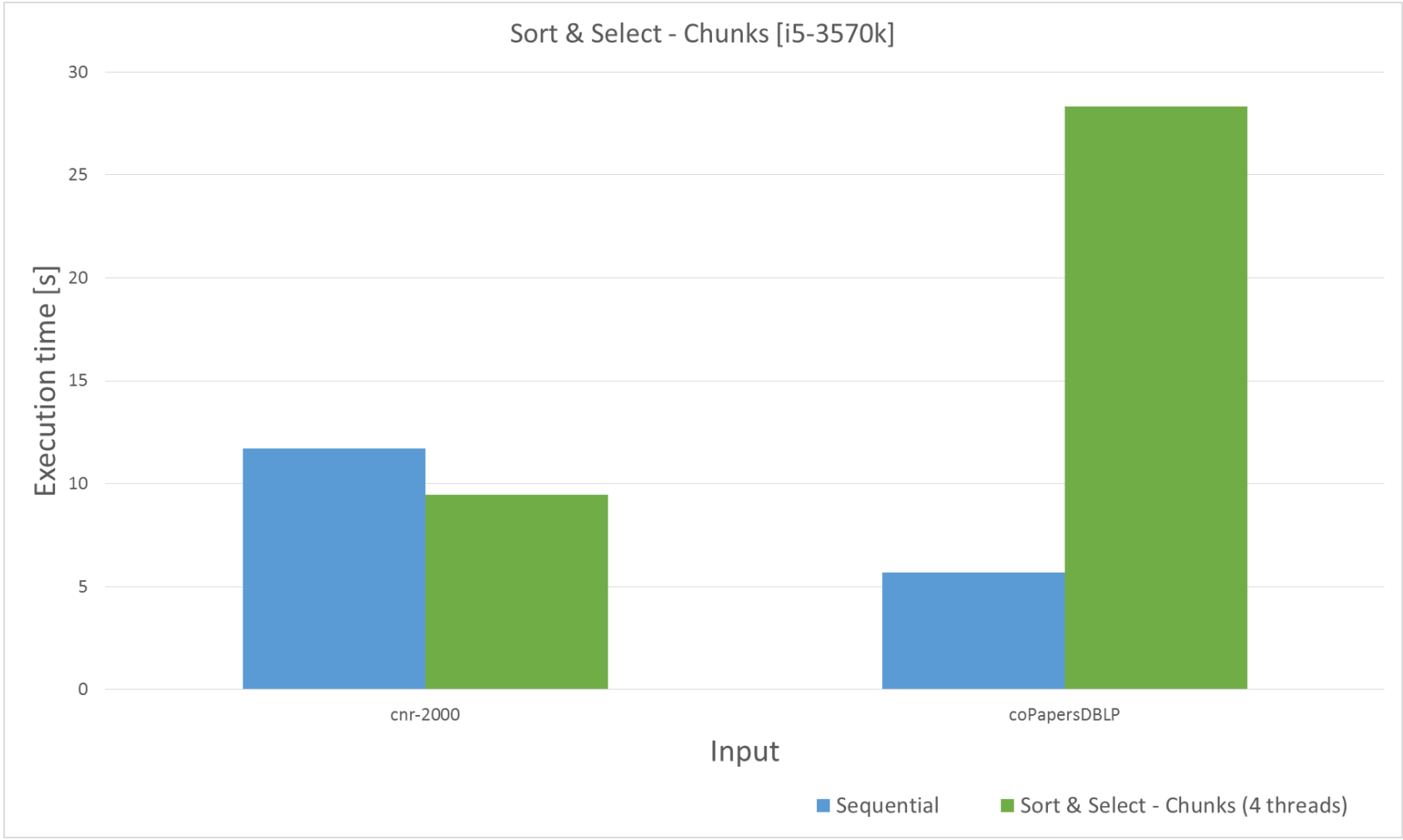
Sort & Select - Chunks - Results

As expected, the results show a substantial improvement over the first version of Sort & Select. The results can get, as the cond-mat-2003 graph shows, up to 885%.

This parallel version actually surpasses the sequential implementation in performance, achieving a speedup of 208% on the second input graph.

But as I already anticipated, chunk analysis doesn’t solve completely the major drawback of Sort & Select. As a result, on particular input graphs performances may be inferior to expectations.

Sort & Select - Chunks - Results



Sort & Select - Chunks - Results

While on CNR the speedup is still positive, on coPapers the performances of this parallel algorithm still suffer from the problem described before, proving that further measures are needed to implement it efficiently.

In any case, the performance improvement over the first version of Sort & Select is astonishing, and clearly proves the effectiveness of the chunk analysis method. In fact, the original implementation needed way more than 10 minutes even run on CNR. Sort & Select – Chunk needs less than ten seconds. This ratio growths with the sizes of the input graph in favor of the second version.

Sort & Select - Chunks - Results

I actually identified the particular issue causing low performances on Sort & Select Chunks: high average outgoing degree on nodes. This is still an hypothesis yet to be confirmed, but the results obtained so far clearly support it. In fact, the average degree is 16.8 in CNR, and 56.4 in coPapers.

The major drawback of the first version was not being able to use a community more than once per iterations. This ultimately caused a sharp increase in the number of iterations needed to get to convergence, and their duration. In practice, this behavior has been manifested in the analysis of CNR.

The same still happens on the chunked version, even if in much smaller scale. Increasing the number of chunks alleviates the problem, but doesn’t solve it. Moreover, increasing the number of chunks introduces a tradeoff with the parallelization costs associated to each iteration.

Sort & Select - Chunks - Results

As a final note, the results shows that the speedup increasing the number of threads in Sort & Select Chunks isn’t really satisfying.

This however, is due to the fact that this version is still very recent. There are lots of optimizations, already known, yet to be made on it.

As I anticipated, this should lead to improved parallel scalability, and even greater speedups over the sequential version.

Naive Partitioning

Naive Partitioning

The last of the algorithm versions I implemented as of now partitions the graph in a number of sub-graphs equal to the number of threads and perform, completely in parallel, an optimized version of the sequential Louvain method over each of them. This process is applied only on the first phase currently, since it may produce smaller values of modularity if applied repeatedly.

As of now, the partitioning is naive since it divides the input graph by indexes. In practice the set of indexes of vertices is equally split into the given number of partitions. Connections among nodes belonging to different subgraphs are deleted.

The last observation is the reason why this heuristic may produce smaller values of modularity.

Naive Partitioning - Strenghts

The positive aspect about this version is that it completely erases the complexity of dealing with the problematics of correcteness in parallelism, while providing a really high parallelism degree.

Naive Partitioning - Drawbacks

- It generally produces smaller values of modularity, since the possibility of moving single vertices is lost after the first phase. During the first phase a node might get put in the wrong community as its potential best neighbors reside in another partition
  - This is why using an heuristic to produce better min-cut approximated heuristics could improve both the modularity and the performance (time to converge) of the algorithm
- Not only the quality of communities detected by the first phase decrease, but their number increase in general. This is due to the fact that «would-be» communities are being split by the partitioning
  - Same as before
- Another drawback is that, since we are splitting the graph, we generally exploit less data locality among threads.
  - This actually depends on the implementation. At the same time the fact that data is not contiguous might reduce the need for communication among cores and cache validity issues, increasing performance.

Naive Partitioning - Results

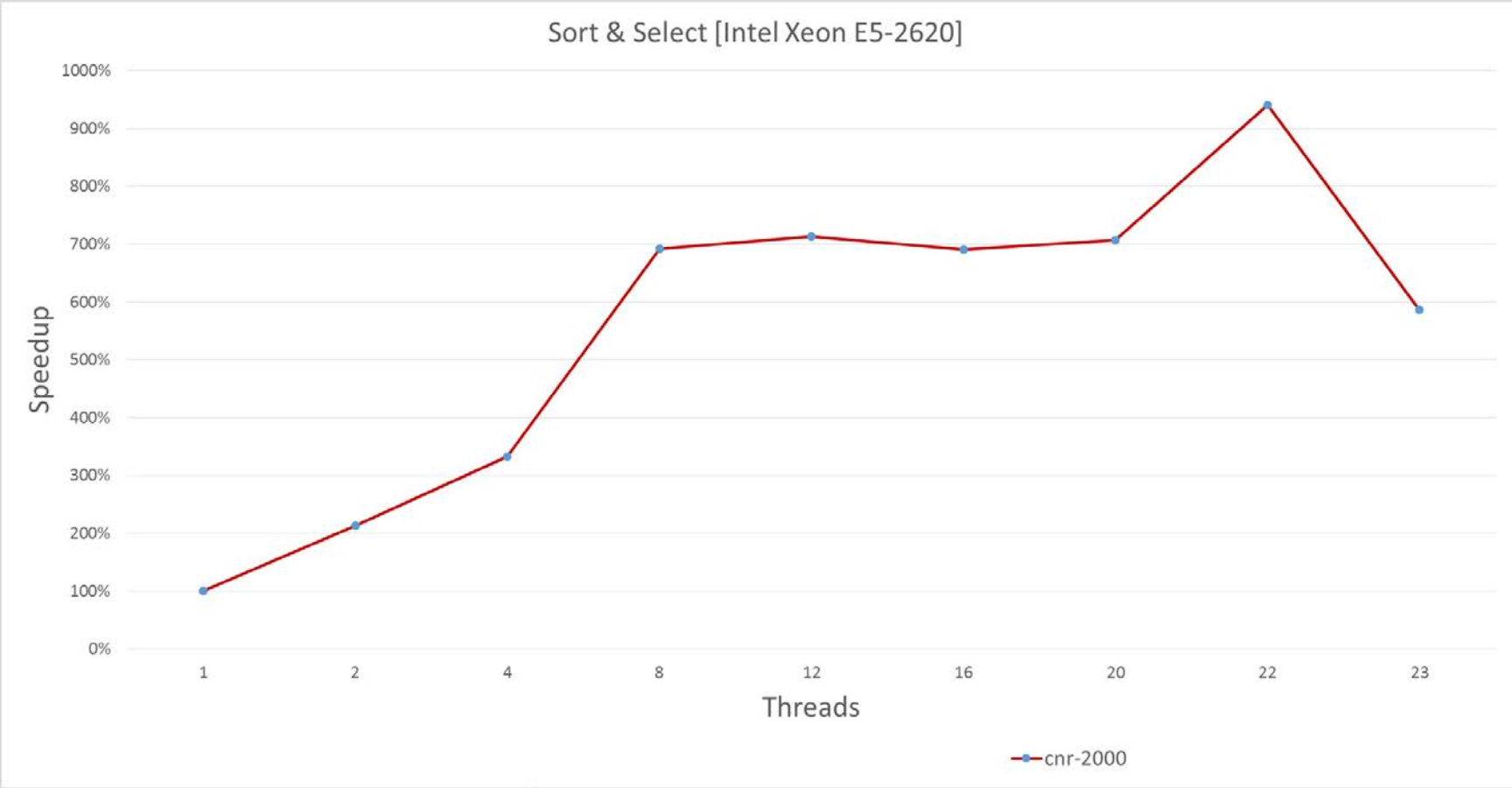
On CNR the modularity didn't decrease significantly by increasing the number of partitions:

Threads	cnr-2000
1	0.912782
2	0.912640
4	0.912626
8	0.912591
12	0.912571
16	0.912561
20	0.912581
22	0.912575
23	0.912584

The value of modularity produced by the sequential version is: 0.912782 (as expected it is equal to the value produced by the algorithm executed by one thread.

This observation, though, might not be valid for every input graph. In fact, analysis on smaller graphs produced slightly lower values of modularity.

Naive Partitioning - Results



Naive Partitioning - Results

Using 22 threads, an impressive speedup of up to 700% over the sequential version has been observed for CNR.

This however, may not hold for every input graph.

Other Heuristics In Grappolo[4]

Vertex Following

This heuristic, run before the first phase of the algorithm, places nodes with a single neighbor in the neighbor's community and stops considering them for node transfers.

This is based on the following Lemma, quoting [3]:

« Given an input graph G(V,E,w) let i and j be two different vertices such that i is a single degree vertex with only one incident edge (i,j) ∈ E. Then, in the final solution C(i) = C(j) »

This heuristic has been described in [3] and implemented in Grappolo[4]. My framework includes an execution option to run this pre-computation.

Vertex Coloring In Grappolo

Grappolo has an option to run a pre-computation on the input graph that run distance-1 coloring on the vertexes in parallel, using the parallel implementation provided by [5].

Vertices can be processed in parallel if and only if they are of the same color. This prevents swap scenarios. It doesn’t prevent though any possible negative gain scenartio. Yet, the results seems to confirm that the application of the heuristic is beneficial in some cases.

Grappolo

The algorithm performs the following steps:

- Vertex Following preprocessing (Optional)
- Coloring preprocessing (Optional)
- Phases:
  - Perform parallel iterations over all nodes using community information from the previous iteration
- Graph rebuilding:
  - Efficient passing of the generated graph data structure to the next phase

Grappolo - Results

Input	Output modularity		Run-time (in sec)		
	Parallel	Serial	Parallel (8 threads)	Serial	Speedup (8 threads)
CNR	0.912608	**0.912784**	0.8	4.3	5.37×
coPapersDBLP	**0.858088**	0.848702	3.7	7.7	2.08×
Channel	**0.933388**	0.849672	21.2	30.9	1.45×
Europe-osm	**0.994996**	N/A	63.4	N/A	N/A
MG1	**0.968723**	0.968671	28.8	126.6	4.39×
uk-2002	0.989569	**0.9897**	210.3	335.9	1.59×
MG2	0.998397	**0.998426**	457.8	1313.7	2.86×
NLPKKT240	0.934717	**0.952104**	388.4	5077.2	13.07×
Rgg_n_2_24_s0	**0.992698**	0.989637	34.2	111.1	3.24×
Soc-LiveJournal	**0.751404**	0.726785	67.05	182.7	2.72×
friendster	**0.626139**	N/A	2036.8	N/A	N/A

As reported in [3].

Grappolo – Compared to Mine on CNR

	Sequential	Sort & Select	Sort & Select Chunks	Naive Partitioning	Grappolo
cnr-2000	11.711803s		9.457714s	4.918350s	2.703322s

Tests have been performed on an i5-3570k processor, using four threads.

Grappolo Approach Compared To Mine

Grappolo Approach Compared To Mine

As I wrote at the beginning of my analysis, I decided to completely avoid negative gain scenarios. I did this to enforce the correctness of my algorithm, while also being sure not to fall into unexpected local maxima of modularity that may have blocked the analysis performed during a phase.

Grappolo on the other hand, follows a different approach. Instead of completely avoiding negative gain scenario, the designers chose to accept the possibility of encounteting some, while making sure that this can’t affect significantly computation times.

I didn’t make the same choice because I didn’t have experience on the problem,nor access to previously done research, nor time to develop my mathematical analysis as much as it is needed to come to the same conclusions that the Grappolo designers made.

Grappolo Approach Compared To Mine

Grappolo's approach is superior to mine in performance.

In fact, having proven that negative gain scenarios don't represent an issue for Grappolo, all of its operations are inherently less computation intensive.

As a matter of fact, Grappolo exploits parallelism without having to deal with the same constraints that I put on my heuristics.

For instance, removing the constraint that dictates the absence of negative gain scenarios from my Sort & Select, would result in an algorithm very similar to Grappolo.

This is not surprisingly actually, since the first considerations the Grappolo designers made in their article, are actually the same I discovered on my own during the initial phases of problem analysis. Our results start to diverge when they actually analyze the negative gains impact.

So, even if the performance results obtained are not comparable with the ones produced by Grappolo, the analysis that I performed is still valid, and may offer contributions for future development.

References

[1] Blondel V D, Guillaime J L, Lambiotte R, Lefebvre E, 2008 arXiv:0803.0476

[2] Brandes U, Delling D, Gaertler M, Goerke R, Hoefer M, Nikoloski Z and Wagner D,2006 arXiv:0608255

[3] Lu H, Halappanavar M, Kalyanaraman A, 2014 arXiv:1410.1237

[4] http://hpc.pnl.gov/people/hala/grappolo.html

[5] Catalyurek U, Feo J, Gebremedhin A H, Halappanavar M, Pothen A, Graph coloring algorithms for multi-core and massively multithreaded architectures, Parallel Computing.

[6] http://www.cc.gatech.edu/dimacs10/archive/data/