Optimize Find Communities on Hypergraphs

Luigi Crisci, Domenico Liguori, Giuseppe Di Palma
Department of Computer Science
University of Salerno
Italy

***Abstract*—We optimize a sequential version of a Find Communities algorithm, written in Julia, by porting it to C++ and introducing parallelism to gain performance from multicore processors. Because of importance of finding communities in graphs, and due to the size of real world network, exploit maximum performance from current systems is required. We optimize our solutions in various steps and show a notable performance improvement.**

I. Introduction

A wide range of complex systems can be represented as network. For example, the Word Wide Web is a network of webpages interconnected by hyperlinks; social networks are represented by people as nodes and their relationships by edges. An important component of this networks are the communities: essentially, a community is a group of vertices which are densely connected internally. Finding those groups inside a network is important since allow to define a large scale graph of the network, because nodes in a community tend to act similar.[4]

In the recent years, an increasing interest has been directed towards Hypergraphs, a graph generalization, because of their capability to naturally represent high-order relationship and so to exploit the structure of several real world network, such as social network and co-authorship network [1].

Antelmi proposed an almost-linear algorithm to find communities in a Hypergraph based on a Label Propagation approach [2], which is a generalization of the graph algorithm proposed by Raghavan [3]. However,when working with real world network, the problem size can be in the order of million of nodes and edges, requiring well designed and optimized algorithm.

Although, achieve good performance when working with Hypergraphs is a major challenge: because of the sparse access made by Hypergraphs algorithms, it is hard to define a proper data layout to optimize cache utilization. Furthermore, parallelism require smart data splitting because of data dependencies between tasks, which can introduce syncronization and break performance.

In this paper, we optimize the Find Communities algorithm on Hypergraphs proposed by Antelmi [2]. Starting from a sequential implementation in Julia, we redefine the data organization and introduce fine grained parallelism using *OpenMP*, then we compare our work with the base implementation.

Related work. The Julia implementation is part of the SimpleHypergraphs.jl library [13].

II. Background

Hypergraphs. An hypergraph is a generalization of a graph in wich a single edge, called *hyperedge* can connect any number of vertices. They are very useful in modelling such things as satisfiability problems, databases, machine learning and Steiner tree problems. Their nature allow the modeler to fully represent a multi-relational (many-to-many) network. Even tho their powerful expressiveness is very appealing the complexity of the data structure is a big issue[5] [6].

Find communities. A community is a group of verticies that are densely connected. Various network have shown to have a strong tendendcy to divide into groups of different size, like Social Network [14], and the web [15]. consequently, how to efficiently find high-quality communities from big graphs is an important research topic in the era of big data. Many approach have been made to provide efficient solution focusing on different types of graphs and formulating communities in different manners. Between the many algorithm proposed, Antelmi et al. use the label propagation approach originally developed by Raghavan et al. [2].

Label propagation. Label propagation is a simple iterative semi-supervised machine learning algorithm that can be used to propagate labels through a graph with high density unlabelled areas. Each node propagate his label to his closest neighbour until convergence, where every node has a label [7]. Raghavan et al. proposed a Find Communities algorithm based on Label Propagation. It can be summarized as follow: at each iteration step, each node's label is updated by choosing the most frequent label in its neighbors (propagation rule); ties are broken with a random choice. The algorithm terminates either if it does not modify any label in two consecutive iterations, or it hits the predefined number of iterations (termination criteria). [7]

Find communities on hypergraphs. In this hypergraph version of the find communities algorithm the hypergraph is walked through a bfs to check if it is connected, then the propagation step is splitted in two phases: hyperedge labeling and vertex labeling. During the hyperedge labeling phase, the labels of the hyperedges are updated according to the most frequent label among the vertices contained in that hyperedge. Similarly, during the vertex labeling phase, the label of each vertex is updated by choosing the most frequent label among the hyperedges it belongs to. [2].

Current implementation. The current implementation of the algorithm that is optimized in this work is written in Julia

and works as follow:

Algorithm 1: Find communities ($Hypergraph$)

Result: List of edges and vertex labels

check if $Hypergraph$ is connected;
$vlabels[\,] \leftarrow map\{Int\,Int\}$;
$helabels[\,] \leftarrow map\{Int, Int\}$;
$edges[\,] \leftarrow Hypergraph.edges$;
$vertices[\,] \leftarrow Hypergraph.vertex$;
$stop \leftarrow false$;
$iter \leftarrow 0$;
while $stop \neq true$ *and* $iter <$ *MaxIterations* **do**
    $stop \leftarrow true$;
    shuffle($edges[\,]$);
    for e *in* $edges[\,]$ **do**
        if e *has vertices* **then**
            $label \leftarrow$ most frequent label among the vertices of e;
            $helabels\,[\mathrm{e}] \leftarrow label$;
        end
    end
    shuffle($vertices[\,]$);
    for v *in* $vertices[\,]$ **do**
        $label \leftarrow$ most frequent label among the hyperedges of v;
        if $label \neq vlabels[v]$ **then**
            $stop \leftarrow false$;
            $vlabels\,[\mathrm{v}] \leftarrow label$;
        end
    end
    $iter+ = 1$;
end

Julia. Julia is a dynamic programming language for technical computing. It was written in C++ and Scheme. For the compiling part it use a JIT(Just In Time) approach based on the LLVM framework. It was designed to achieve performance and give to the developer a good programming expressiveness [8].

OpenMP. OpenMp is a multi-platform API for parallel programming. It is supported by many programming languages such as C++, Fortran, etc. The parallelism is achieved by explicitly specify the actions to be taken by the compiler and runtime in order to execute the program in parallel. It doesn't require checking for data dependencies, data conflicts, race conditions, or deadlocks [9].

III. Your Proposed Method

In this section we show the various steps involved in implementing the optimizations. We first did a porting from Julia to C++, this is specified as sequential. We have divided each of these steps into the problem (found) and the solution (implemented). All these steps are shown below:

A. Sequential

Problem. The Julia language is interpreted, which lead to a notable degree in performance. We decided to port it to C++ in order to get better performance.

Solution. The important step is the porting of the data structures. We have transformed Julia *dictionary* data structure into C++ *unordered map*. After this, all functions were imported and converted into C++. Another important thing was the porting of the shuffle function. This function randomly permutes a vector, with the possibility of providing a random number generator. Julia uses the internal MersenneTwister [10] for random number generation. In our implementation we used a custom MersenneTwister implementation [11], not present in the C++ libraries.

B. Optimization 1

Problem. This is the first real optimization. We used flags from the C++ language at compile time.

Solution. We used the automatic optimization available with LLVM. We tested the optimizations that Clang++-11 makes available (quote), via the -O flags. At the beginning, most optimizations are completely disabled at -O0 or if an -O level is not set on the command line, even if individual optimization flags are specified. Turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program. In our case, with -O1, and -O2 we obtained slightly better times than with the first implementation. With -O3, however, we obtained the best performance.

C. Optimization 2

Problem. From a quick function profiling, we found that the BFS used to check if the hypergraph is connected was taking up to 90% of the execution time. This was due to two major problems:

- The current implementation was *recursive*, which is not great when working with a high number of vertices.
- The data structures used to represent the hypergraph used too much memory: we used two arrays of maps, each one of them associated to the vertices and to the edges of the hypergraph,

Solution. We proposed a new structure for the hypergraph which was relying on two *bitset*, a C++ STD library container that represent a stream of bit [12], instead of the two maps. With this approach, we drastically lowered the memory encumbrance.

To optimize the BFS, we chosed to perform a *lossy* translation from hypergraph to graph, keeping only structural information about edges, only for this step:

$$Given\ H = (V, E),\ G = (V_1, E_1), where\ V_1 = E\ \ \&$$
$$E_1 = \{(e1, e2)|\ e1 \in V_1\ \ \&\ \ e2 \in V_1\ \ \&\ \ e1 \cap e2 \neq \emptyset\}$$

This optimization was lead by a simple observation: because Hypergraphs maps high-order relationship, and is unlikely that in real world dataset a node is linked to every other node in

the network, the number of edges should be much smaller than the number of vertices. This lead to a notable reduction in the problem size. We then implemented a simple iterative version of graph BFS.

Algorithm 2: Bfs($Hypergraph$)

```
   Result: count
1  graph[ ][ ] ← hypergraphToGraph(Hypergraph);
2  count ← 0;
3  current ← −1;
4  queue[ ];
5  while queue[ ] isn't empty do
6  |  current ← queue[ ].pop();
7  |  for i ← 0 to i ≤ graph.size do
8  |  |  connected ← graph[current][i];
9  |  |  c ←!checked[i];
10 |  |  if graph[current][i] && !checked[i] then
11 |  |  |  checked[i] ← true;
12 |  |  |  frontier.push(i);
13 |  |  |  count++;
14 |  |  end
15 |  |  i++;
16 |  end
17 end
```

Fig. 1. BFS implementation within OPT2

D. Optimization 3

Problem. Our code was single threaded.

Solution. We introduced thread-level parallelism with OpenMP. This was made by parallelizing the two major for at line 11 and 18. The result is shown at the algorithm 3.

While each thread in every for loop does not have any dependency that need synchronization between them, the second loop has a *RAW (Read after write)* dependence with the first one on the *heLabels* object, which requires a barrier between the two loops, implicitly carried out by the *#pragma omp single* directive.

E. Optimization 4

Problem. To compute the vertex and edge labels, we were using two maps organized as $Vertex/Edge \implies Label$, implemented with the C++ object *std::map<int,int>*. Because we could not manipulate how a *map* object is saved in memory, when splitting the iteration space the updates at line 14 and 22 could lead to *false sharing* cases, if the location updated by the threads are too close to each other.

Solution. We switched from *map* to an int vector, aligned to the cache line size. To be sure that each thread will access elements in distinct cache lines, the iteration space was expanded from *n* to the smallest multiple of the cache line size bigger that *n*: with this approach, each thread access to evenly splitted memory location in the main memory in different cache lines, except for the last thread which iterates on $n - \frac{next\_multiple\_64(n)}{num\_threads} * (num\_threads - 1)$, which is a tolerable unbalance for a high number of verticies or edges. Because at each iteration the vertices accessed by each thread are random, to keep the accessed memory location contiguous we duplicated the *vLabels* and *heLabels* vectors in two vectors called *ordered_vLabels* and *ordered_heLabels*: with this structure, *vLabels* and *heLabels* were shuffled togheter with the vertex and edge indices, so that each thread updates contiguous memory location, and then *ordered_vLabels* and *ordered_heLabels* are updated to the values in their corresponding vectors.

Algorithm 3: Find communities details with *OpenMp*

```
1  #pragma omp parallel{
2  #pragma omp single{ shuffle(edges )
3  #pragma omp parallel for nowait
4  for e in edges do
5  |  if e has vertices then
6  |  |  label ← most frequent label among the vertices
   |  |    of e;
7  |  |  helabels [e] ← label;
8  |  end
9  end
10 #pragma omp single{ shuffle(vertices) };
11 #pragma omp parallel for nowait
12 for v in vertices do
13 |  label ← most frequent label among the hyperedges
   |    of v;
14 |  if label ≠ vlabels[v] then
15 |  |  stop ← false;
16 |  |  vlabels ← label;
17 |  end
18 end
```

Fig. 2. Find communities details with *OpenMp* (C++ syntax)

IV. Experimental Results

This section shows how the tests were conducted and discusses the results. We compare our version of Find Communities algorithm with the Julia base implementation, included in the SimpleHypergraphs.jl package [13]. To ensure a fair comparison among the competitors, we performed all experimental with the same input.

Experimental setup. This paragraph presents our experimental setup. The benchmarks were conducted on a system with the following requirements:

- **Processor**: Intel® Core™ i7-8565U @ 1.80Ghz;
- **Cores**: 4;
- **Cache**:
  - **L1Data Cache**: 32 KBytes X 4, 8-way set associative, 64-byte line size;
  - **L1Instruction Cache**: 32 KBytes X 4, 8-way set associative, 64-byte line size;
  - **L2-Cache**: 256 KBytes X 4, 4-way set associative,
  - **L3-Cache**: 8 MBytes, 16-way set associative, 64-byte line size;64-byte line size;

- **Ram**: 12 GB;
- **OS**: Ubuntu 20.04.2 LTS on Windows 10 x86_64.
- **Language**: C++17;
- **Compiler**: Clang 11.1.0.

Synthetic data-set. We defined 9 synthetic data-sets, combining different size and *density*. By density, we mean how many connection every vertex in the hypergraph have: we implemented this parameter as a probability p, which is used to determine if two vertex are connected. Those inputs were generated in order to simulate different cases (sparse network, dense network etc.). The specifications of each data-set created are shown below:

- **S**mall: 5000 vertex, 300 edges;
- **M**edium: 10000 vertex, 600 edges;
- **L**arge: 20000 vertex, 1200 edges.

And different density probability:

- **S**parse: 0.3;
- **N**ormal: 0.5;
- **D**ense: 0.7.

Running experiments. All benchmarks were performed on the previously specified system. Each run, for each file and therefore for each optimization, is launched 5 times, taking the arithmetic median. From these five values we calculate the variance, and if it exceeds 5% the benchmark were repeated. In table I all the results obtained from the benchamarks are listed.

Results and discussion. This section reports and discusses the results of our experimental evaluation in table I. We aimed at answering the questions as follows:

Q1: Compared with Julia code, how efficient are our optimizations?
Q2: What are the effects of density variation?
Q3: How scalable are Opt3 and Opt4?
Q4: How to explain the comparison between opt3 and opt4?

A1. In table I the results for each size and optimization are showed, while in table II a more detailed analysis of the performance for the **L**arge**M**edium dataset is presented.

From Julia to C++, we gain a speedup of 1.41, which is simply due to the different performance between an interpreted language and a compiled one.

Looking at table II, is clear than the most important optimization was the iterative BFS on graph, with the data structure reworking and the hypergraph translation to graph: only by doing this, we gained a notable absolute speedup of 102,68, and a relative one of 15,42.

The third and fourth optimization lead to poor improvement in terms of time: with 4 threads, only a speedup of 2 were reached. We discuss these results in the next sections.

A2. Reading the benchmarks, it can be seen that by keeping the size constant and varying the density (S, N, D) the times increase. This occurs because with higher hypergraph densities we have more elements to visit, and then the algorithm takes longer to propagate the label.

It is interesting that, while with the C++ versions the time slowly increase, in the Julia implementation we had an almost linear time increase: we suppose that it is due to the recursive BFS implementation, which suffers from the high number of connections that introduce a high number of function calls.

TABLE I
ALL TIMES (SECONDS) OF ALL BENCHMARKS FOR EACH DATA-SET

Data-set	Julia	Sequential	Opt1	Opt2	$Opt3_3$	$Opt4_4$
SS	12.89	11.89	2.60	0.53	0.40	0.42
SN	25.62	16.71	3.37	0.65	0.39	0.44
SD	85.16	22.22	4.61	0.77	0.52	0.54
MS	114.44	73.96	13.77	1.71	1.10	1.08
MN	291.98	109.07	22.61	2.591	1.44	1.50
MD	517.64	147.03	31.11	3.10	1.70	1.70
LS	810.41	476.95	99.08	7.18	3.90	4.10
LN	1091.54	774.39	163.94	10.63	5.55	5.80
LD	1389.23	1060.36	228.50	13.07	6.96	7.30

TABLE II
VARIATION OF RELATIVE AND ABSOLUTE SPEED-UP IN RELATION TO TIME IN DIFFERENT IMPLEMENTATIONS
(DATASET 20000 VERTEX, 1200 EDGES, DENSITY: 0.5)
RELATIVE SPEEDUP, ABSOLUTE SPEEDUP

Ver.	Implementation	Run Time	R Speedup	A Speedup
1	Julia	1091,54s	1	1
2	C++ Naive	774,40s	1,41	1,41
3	+ optimization flags	163,95s	4,72	6,66
4	+ bitset, bfs on graph	10,63s	15,42	102,68
5	+ OpenMP (4 threads)	5,56s	1,91	196,32
6	+ padding alignment (4 threads)	5,89s	0.94	185,32

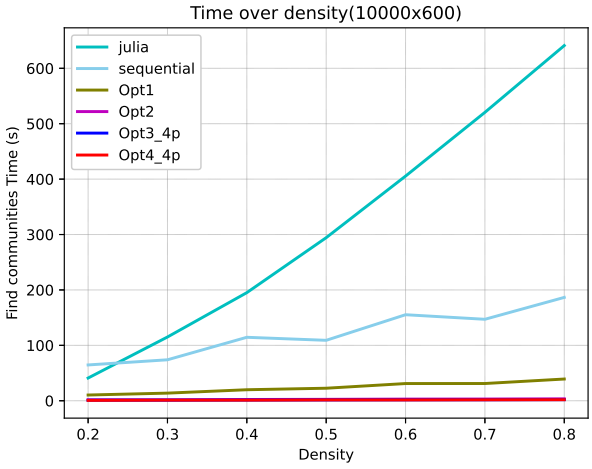

Fig. 3. Time over density, all optimizations

A3. Looking at the figure 5, the third optimization have a decent scalability from 1 to 2 cores of 1.4, which is slightly below half-way of the optimal result, however it slightly decrease when adding more threads resulting in almost no gain from 3 to 4 threads. The fourth optimization, instead,

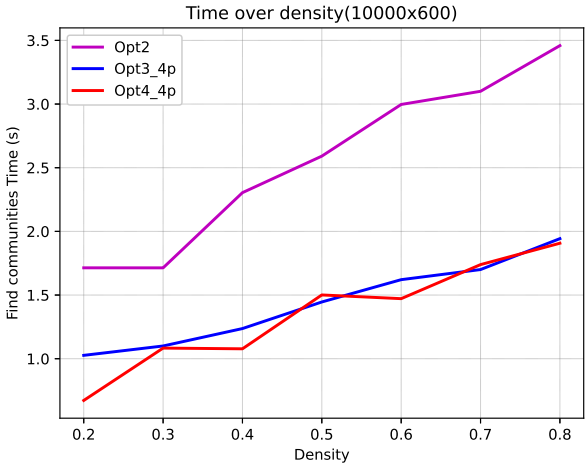

Fig. 4. Time over density, Opt2, Opt3, Opt4

showed to not be scalable at all: fro 2 to 4 threads, we had no performance improvement.

A4. These results taken us by surprise: keeping threads on different memory location should have lead to at least a minimum speedup. However, the cause of this failure are not obscure: trying to avoid false sharing, opt4 introduced new data transfers to be handled, because the label vectors were duplicated, and because it was necessary to reorganize the label vectors at the end of each label computation step, we had to remove the *nowait* clause on the two major for loop, introducing some synchronization overhead. It's clear that the trade-off between false sharing\data movement was not convenient.

Looking at opt4 we introduce a way higher overhead in the computation, which result in no gains from 1 to 2 cores with some degree of gains from 3 to 4, anyway it doesn't perform any better than opt3.

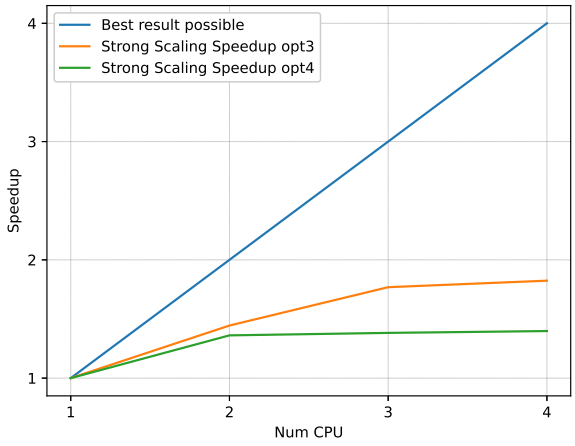

Fig. 5. Speed-up on size MN - Strong scaling Opt3 and Opt4, 1 to 4 processors

V. Conclusions

We optimized the Julia version of the Find communities algorithm on Hypergraph porting it to C++, reworking the data structures used and introducing parallelism. From the base version, we gained a huge absolute speed-up of 196,32 in the best cases. The poor speed-up obtained with the introduction of parallelism confirmed that obtains good multithread performance on hypergraphs is challenging, and that more detailed studies about parallel algorithm on this structure are required. However, we showed that only by reworking how the data are memorized and defining *ad-hoc* algorithm solution it is possible to achieve strong improvement in performance and therefore we believe that this should be the main goal of algorithm designer when working with Hypergraphs.

References

[1] Bretto, *A.Hypergraph Theory: An Introduction*. Springer Publishing Company, Incorporated, 2013.

[2] Antelmi, Alessia, et al. "Analyzing, exploring, and visualizing complex networks via hypergraphs using SimpleHypergraphs. jl." arXiv preprint arXiv:2002.04654 (2020).

[3] Raghavan, U. N., Albert, R., and Kumara, S. *Near linear time algorithm to detect community structures in large-scale networks*.Physical review. E, Statistical, nonlinear, and soft matter physics 76(2007)

[4] M.E.J.Neman (2006). *Finding community structure in networks using the eigenvectors of matrices*. Phys. Rev. E. 74 (3): 1–19. arXiv:physics/0605087

[5] Peng, Y., Shi, J., Fantinato, M., and Chen, J. A study on the author collaboration network in big data. Information Systems Frontiers 19, 6 (jun 2017), 1329–1342

[6] Vargas, D. L., Bridgeman, A. M., Schmidt, D. R., Kohl, P. B., Wilcox, B. R., and Carr, L. D. Correlation between student collaboration network centrality and academic performance. Physical Review Physics Education Research 14, 2 (oct 2018).

[7] Learning from Labeled and Unlabeled Data with Label Propagation, Xiaojin Zhu and Zoubin Ghahramani, School of Computer Science Carnegie Mellon University Pittsburg, PA 15213, June 2002
A Survey of Community Search Over Big Graphs Yixiang Fang · Xin Huang · Lu Qin · Ying Zhang · Wenjie Zhang · Reynold Cheng · Xuemin Lin

[8] Julia: A Fast Dynamic Language for Technical Computing Jeff Bezanson, Stefan Karpinski, Viral B. Shah, Alan Edelman September 25, 2012

[9] The OpenMP API covers only user-directed parallelization, wherein the programmer explicitly specifies the actions to be taken by the compiler and runtime system in order to execute the program in parallel. https://www.openmp.org/spec-html/5.1/openmp.html

[10] Matsumoto, Makoto, and Takuji Nishimura. "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator." ACM Transactions on Modeling and Computer Simulation (TOMACS) 8.1 (1998): 3-30.

[11] *The original C implementation of Mersenne Twister is not thread-safe. Here I provide a modification which makes the code thread-safe.* http://users.umiacs.umd.edu/ yangcj/mtrnd.html

[12] The class template bitset represents a fixed-size sequence of N bits. Bitsets can be manipulated by standard logic operators and converted to and from strings and integers. https://en.cppreference.com/w/cpp/utility/bitset

[13] A simple hypergraphs package for the Julia programming language. https://github.com/pszufe/SimpleHypergraphs.jl

[14] M. Girvan and M. E. J. Newman, Community structure in social and biological networks. Proc. Natl. Acad. Sci. USA 99, 7821–7826 (2002)

[15] G. W. Flake, S. R. Lawrence, C. L. Giles, and F. M. Coetzee, Self-organization and identification of Web communities. IEEE Computer 35, 66–71 (2002)