Using Community Detection Algorithms to Identify Clusters of Ranks in an MPI Application Based on the Communication Matrix

Manfred Calvo Sánchez*, Esteban Meneses†
School of Computer Science, Costa Rica Institute of Technology
Email: *calvomanfred@gmail.com, †emeneses@ic-itcr.ac.cr

***Abstract*—In the MPI parallel programming model, communication remains the bottleneck that prevents applications to achieve greater performance and scalability. Due to this problem it is important to know the behavior of this communication in each application. For this reason, we propose the use of community detection algorithms to identify from the communication matrix the clusters of ranks that maximize intracluster communication and minimize intercluster communication. The aim of this project is providing another tool to identify how you can improve the performance of a MPI application.**

***Keywords*—Communication Patterns, Message-Passing, Parallel Applications, Modularity.**

I. INTRODUCTION

Many scientific applications have been developed to solve psychical, chemistry, and aerodynamic problems. They make a lot of complex mathematical calculations that require a great power of processing. For this reason, many of these applications have been parallelized with the goal of having a better performance and, therefore, they can make more complex calculations.

Many of these applications use the standard of parallel programming MPI (Message Passing Interface) for their development. This work proposes to make use of MPI to communicate different tasks that work together to accomplish a common objective [1]. Many times these applications work with the same set of data and they make use of messages to deliver the data that is required to make calculations.

Some works have found that communication is the main cause of bottlenecks in the applications that use the model MPI and, in general, in other models that use communication [2], [3], [4]. For that reason, some works have been done to make improvements on applications and implementation of the model MPI in order to reduce the role communication plays in efficiency and scalability of applications.

Based on these findings, it is important to know and identify what is the communication behavior in each application. With this information, we could have a better vision of the application and at the same time we could determine which optimizations we can make either at the application level, architecture, or implementation of the MPI standard. [1].

For this reason, it is proposed to use community detection algorithms to identify, in a communication matrix, which represents the flow of bytes transmitted between each of the ranks participating in one execution of the application, groups of these ranks that are tightly coupled. In other words, the objective is identify which clusters of ranks maximize intracluster communication and minimize intercluster communication. This is done in order to provide a new tool that allows to make decisions, when optimizing an application that utilizes MPI standard.

II. METHODOLOGY

To developed this work we extracted a communication matrix from a scientific application implemented with the protocol MPI. The communication matrix of an MPI application represents the data flow between each of the ranks that belong to the application. This matrix registers how many messages and how many bytes are transmitted between each pair of ranks of the application through the use of point to point operations and collective operations [5], [6], [7]. Then, we processed this communication matrix to extract different groups of processes based on community detection algorithms. In this section we explain in more detail how the work was developed and how it can be apply to other MPI applications.

A. Modifying mpiP to extract communication matrix

The first step was to modify a profiler application called mpiP [8]. This application lets extract some information about different MPI operations during the execution of the application, for example, how many calls of each operation were executed and also the portion of the execution time the application spent in a specific operation. Although, this information is useful, it is not possible to get the amount of bytes transmitted between each of the process during the execution of the application. This application can not obtain the communication matrix from its log because it was initially developed for analysis of communication scalability of an application [5]. To make it possible, we modified mpiP to add that functionality by changing some modules and adding some code to obtain the information necessary to build and save the communication matrix at the end of the execution.

B. Linking and extracting a communication matrix from scientific application

Once modified the mpiP profiler, the next step was to extract the communication matrix from the application MPI. To achieve this, the first thing was to link mpiP with the application to be executed. To link mpiP in the application we added some libraries and flags in the compilation command of the application. They were added to let know mpiP that the application need to be profiled in execution time. After linked these libraries in the compilation command, it was necessary to execute the application MPI with a special script that is generated after compile mpiP and it is in the bin folder of mpiP. This script specify the same flags than the normal command for executing an application MPI but instead of execute the application in the normal way the script sets environment variables that mpiP needs to work. Finally we had to wait until the execution of the application finished to get the communication matrix in a

plain text file. It is worth noting that this plain text file is always generated by mpiP in the server where is running the root node of the application MPI. We can see an example of one communication matrix in Figure 1. The communication matrix is represented using a heat map where each axis represents the ranks in the application MPI and the color represents the number of bytes transmitted between ranks. So for example the main diagonal of the matrix is completely black because the communication between a node and itself is zero.

C. Generating the graph that would be inserted in the community detection algorithms

An undirected weighted graph is the most common input for detection of community algorithms. However, our communication matrix is a directed weighted graph with edges between each pair of nodes. For that reason we converted the communication matrix in a undirected weighted graph. We will now explain why the communication matrix is a directed weighted graph and how it was converted it into an undirected weighted graph. The value at index (i, j) in the communication matrix represents the number of bytes w_{ij} sent from rank i to rank j. From the description of the values in the matrix is easy to see that the matrix is the adjacency matrix of a graph in which each vertex represents a rank and the edges represent the number of bytes transmitted between ranks.

Likewise, the graph is directed because there is one edge from node i to node j and also one edge from node j to node i. Therefore is necessary to convert this directed weighted graph into an undirected weighted graph.

We assumed that the graph $G = (V, E)$ represented the communication matrix and w_{ij} represents the number of bytes transmitted from rank i to rank j. Based on that assumption was possible to create a new graph $G' = (V', E')$ where $V' = V$ and $E' = \{(i, j) \mid i < j \wedge i, j \in V\}$. Further we defined the number of bytes transmitted between rank i and rank j in any direction as $w'_{ij} = w_{ij} + w_{ji}$. After this conversion, G' is an undirected weighted graph as required, since there is only one edge between each pair of ranks (i, j) and the weights are summed from the number of bytes sent from rank i to rank j with the number of bytes sent from rank j to rank i.

The graph G' is an ideal representation of the network for the community detection algorithms and it allows to apply the community detection algorithms to the chosen scientific application.

D. Running community detection algorithm over the graph

The last step was to run the community detection algorithm over the undirected weighted graph obtained it in the previous step. The community detection algorithms, try to find groups of nodes with more or and/or better interactions among its members than between its members and the remainder network. One of the most important concepts in this type of algorithms is *modularity*. *Modularity* is a measure to know how good a division of the network is [9]. Using this metric the community detection algorithms would maximize intracluster communication and minimize intercluster communication. To execute this type of algorithms we used a python library called igraph [10]. This library contains the most common used community detection algorithms and let manage graphs in a very easy way. In this work, we used the *fast greedy* algorithm because its complexity is very good and the results could be obtained faster than with the other algorithms. This algorithm merges individual nodes into communities in a way that greedily maximizes the modularity score of the graph. It can be proven that if no merge can increase the current modularity score, the algorithm can be stopped since no further increase can be achieved [11]. In other hand we created a python script to read the output from mpiP that contains the communication matrix and generate the communities detected by the *fast greedy* algorithm, together with its respective metric of modularity. In this script we also executed the previous step that converted the communication matrix in a undirected weighted graph.

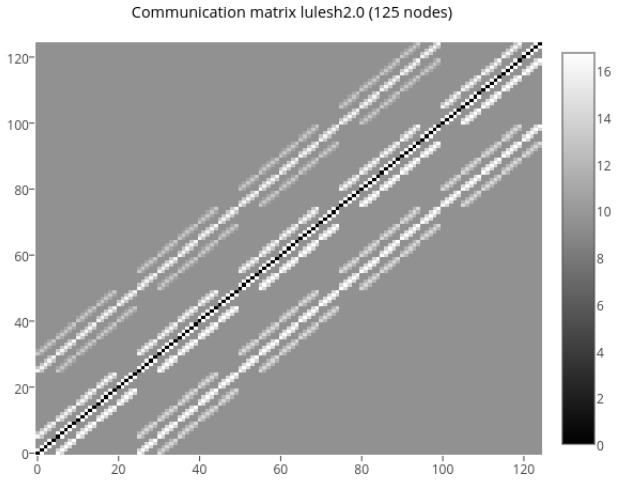

Figure 1: Communication matrix with 125 ranks in the network.

III. EXPERIMENT AND RESULTS

We chose a scientific application called *kernel conjugate gradient* to show how to generate clusters of ranks using community detection algorithms starting from a communication matrix. This application is describes next.

A. Kernel Conjugate Gradient

The application *kernel conjugate gradient* belongs to the NAS parallel benchmarks. This application solve an unstructured sparse linear system by the conjugate gradient method. A conjugate gradient method is used to compute an approximation to the smallest eigenvalue of a large sparse symmetric positive definite matrix. This kernel is typical of unstructured grid computations because it tests irregular long distance communication, employing unstructured matrix vector multiplication. This benchmark uses the inverse power method to find an estimate of the largest eigenvalue of a symmetric positive definite sparse matrix with a random pattern of nonzeros [12].

B. Detecting Communities in kernel conjugate gradient

We applied the *fast greedy* algorithm to the communication matrix of the kernel conjugate gradient to see what groups are generated by the *fast greedy* algorithm. The experiments were executed changing the number of processors to know what was the effect of that. The results are shown in Figure 2. Each element in the axis represents a rank of the application and each color in

the figure represents a community detected within the network by the algorithm.

It can be seen in Figure 2 that the elements of each community are consecutive ranks. Also, it is notable that these communities are very symmetric because each of them has pretty much the same number of elements. The only exception to this rule is the detection of communities for 512 ranks. In this case, there are some communities with different size. This is shown in Figure 3, where x-axis represent the number of the community and y-axis represent the number of elements in the community.

Based on Figures 2 and 3 is clear that the communities are well define and is clear how could be the split of the network.

Number of processors	Modularity	Number of communities
64	0.65	8
128	0.69	8
256	0.75	16
512	0.77	16

Table I: Modularity and number of communities to each number of processors in kernel conjugate gradient.

Table I shows the modularity and number of communities detected by the *fast greedy* algorithm for different number of processes. The results shown that when we added more processes in the application, the number of groups and the modularity grew up. Because the community detection algorithms use modularity, this measure can be use to compare the results. Some authors have mentioned that a good value of modularity is usually between 0.3 and 0.7 [9], [11]. As the results are in that range these groups are good to describe a partition of the network that maximize intracluster communication and minimize intercluster communication.

IV. Conclusion

We applied the *fast greedy* community detection algorithm to the communication matrix of *kernel conjugate gradient*. The analysis showed well define communities that can help to split the network in groups of ranks that could be together to improve latency of the communication. Also, some of the partitions of this network had a good modularity between 0.6 and 0.7.

The results showed that the *fast greedy* algorithm detected communities with good modularity and structure. Further, this algorithm is stable because it detected good communities when the number of processes in the network was changed.

The application showed well defined communities and it was easy to find this communities with detection algorithms. Besides, in some cases the communities detected were symmetric because they had the same number of elements.

The communication matrix was proven to give insight into communication in parallel applications.

V. Future Work

One important thing to do is to use the kernel conjugate gradient and apply the *fast greedy* algorithm in a network with more nodes to see if the behavior is maintained. On the other hand, it is important to explore more applications and apply these methods in real applications that are been using in real time to know if the work proposed can be applied in the real life.

After applying these algorithms to a real applications, it might be interesting to implement a special network that could represent exactly the communities detected by the algorithms and see if the split of the network can help to improve the application performance.

In the future, it would be interesting to add more point-to-point and collective operations to identify how this can affect the detection of communities. Further, in some cases it would be interesting to filter the operations used to detect the communities to identify if some could be feeding noise to the algorithms.

Finally, it is important to try with more community detection algorithms to compare their results and see if it is possible to get better or different communities and have different point of views of how the network can be divided in groups of processes to achieve better performance in communication.

References

[1] (1993) mpi: A standardized and portable message-passing system. [Online]. Available: http://mpi-forum.org/

[2] A. Ovcharenko, O. Sahni, C. D. Carothers, K. E. Jansen, and M. S. Shephard, "Subdomain communication to increase scalability in large-scale scientific applications," *Proceedings of the 23rd International Conference on Supercomputing*, pp. 497–498, 2009.

[3] N. Jain and Y. Sabharwal, "Optimal bucket algorithms for large mpi collectives on torus interconnects," *Proceedings of the 24th ACM International Conference on Supercomputing*, pp. 27–36, 2010.

[4] S. Li, C. Hu, J. Zhang, and Y. Zhang, "Automatic tuning of sparse matrix-vector multiplication on multicore clusters," *Science China Information Sciences*, vol. 58, no. 9, pp. 1–14, 2015.

[5] P. C. Roth, J. S. Meredith, and J. S. Vetter, "Automated characterization of parallel application communication patterns," *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pp. 73–84, 2015.

[6] E. Meneses and L. V. Kalé, "A fault-tolerance protocol for parallel applications with communication imbalance," *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2015.

[7] A. Mazaheri, A. Jannesari, A. Mirzaei, and F. Wolf, "Characterizing loop-level communication patterns in shared memory," *Parallel Processing (ICPP), 2015 44th International Conference on*, pp. 759–768, 2015.

[8] (2013) mpip: Lightweight scalable mpi profiling. [Online]. Available: http://mpip.sourceforge.net/

[9] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical Review*, vol. E 69, no. 026113, 2004.

[10] (2015) igraph: The network analysis package. [Online]. Available: http://igraph.org/

[11] A. Clauset, M. E. J. Newman, and C. Moore, "Finding community structure in very large networks," *Phys. Rev. E*, vol. 70, p. 066111, Dec 2004.

[12] (2012) Nas parallel benchmarks. [Online]. Available: https://www.nas.nasa.gov/publications/npb.html

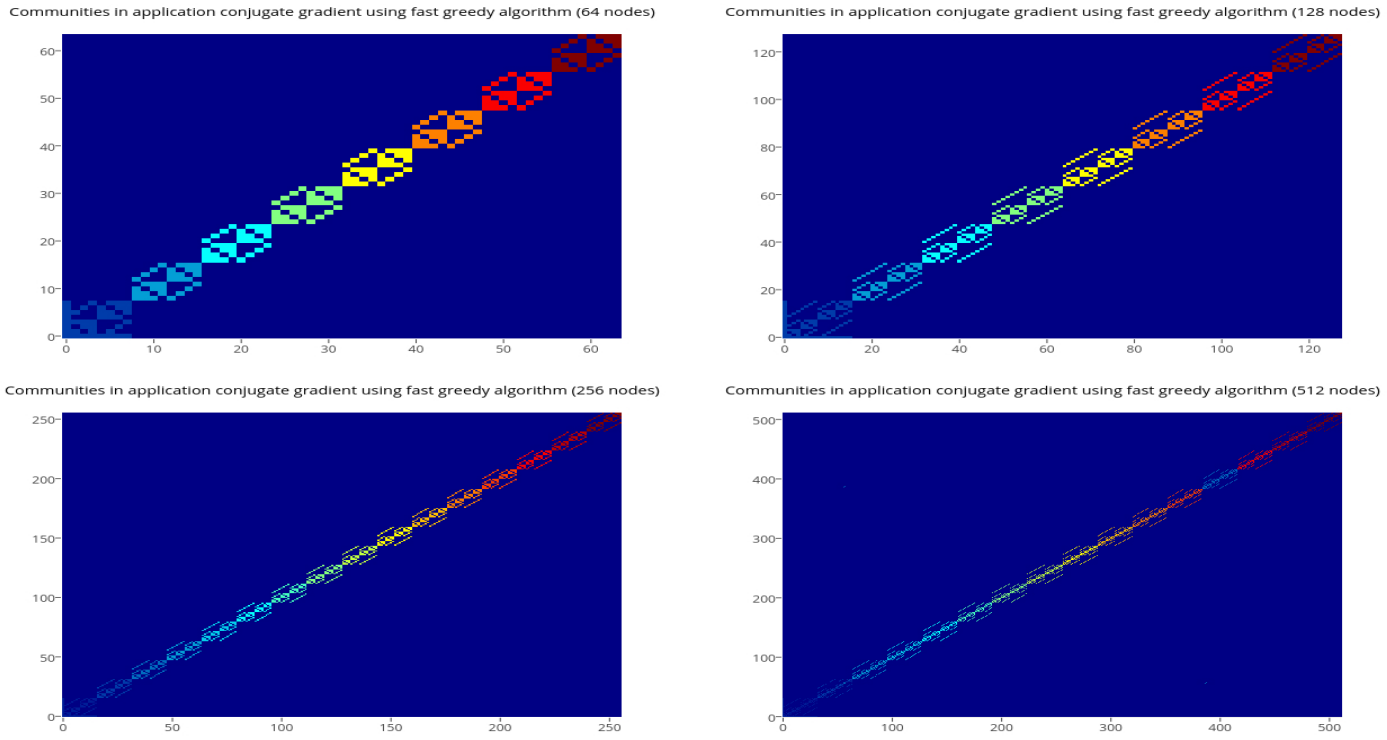

Figure 2: Communities detected by the fast greedy algorithm on different number of processors in conjugate gradient.

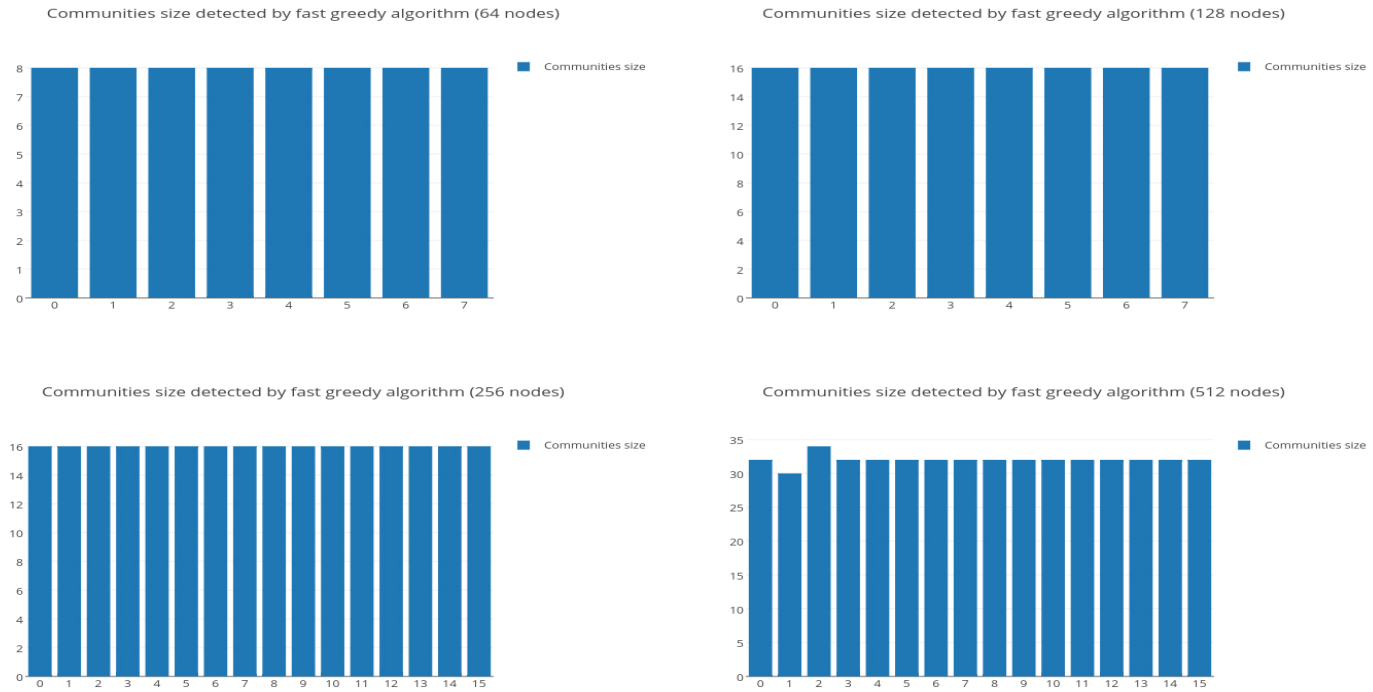

Figure 3: Communities's size in conjugate gradient using fast greedy algorithm.