An Analysis of the Data Dependency Graphs of k-Means*

Francesc Gispert

9th September 2014

Abstract

High-performance computing applications have to carefully adapt their behavior to the underlying architecture. We present some graph partitioning techniques which can be applied to the data dependency graphs of parallel applications in order to map each task to the computing nodes. We analyze and compare experimental results on graphs produced by a concrete application, k-means, in order to show that the use of this partitioning step may have a significant impact on performance due to the heavy decrease in communication volume.

*This work was supported by BSC.

Contents

1 Introduction

Ever larger and more complex problems arise in many domains of research. Thus, these problems need to be addressed with parallel computers. However, parallelism adds a lot of extra complexity and performance depends on a vast number of interacting components.

For instance, the different tasks which an application is divided in can be mapped to the underlying architecture using several approaches, and the way this mapping is carried out has a huge impact on the load balance of the cores and the communication between them. That is to say, a *bad* mapping policy might lead to lots of idle cores during computation or excessive data movement between computing nodes, slowing down the execution.

Therefore, the multiple tasks have to be placed carefully on the various nodes: tasks which share a great amount of data should be executed in cores close to each other in order to share as many levels of the memory hierarchy as possible.

A parallel application can be represented with a data dependency weighted directed graph whose nodes are the tasks and whose edges represent the data dependencies between tasks. Hence, this graph could be partitioned in as many parts as available computing nodes, minimizing the edge-cut of the partition, in order to simplify the mapping phase. In other words, a much smaller graph, whose nodes correspond to the computed parts of the original graph, can be constructed, and all that is left is to map each node of this new graph to a computing node.

The aim of this paper is to show that this strategy can reduce significantly the amount of data transferred with respect to task mapping techniques which are currently used. For this purpose, we analyze the volume of communication obtained applying various graph partitioning methodologies to data dependency graphs obtained from different executions of an implementation of k-means in OmpSs.

The remainder of this paper is organized as follows. Section 2 gives an overview of the structure of the studied code. Section 3 describes the different approaches used to partition the data dependency graphs in order to take advantage of locality. Section 4 presents an experimental evaluation of the different partitioning techniques and compares the obtained results. Finally, section 5 concludes this paper.

2 k-Means

k-means clustering is a method of cluster analysis which aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean. More precisely, given n points $x_1, \ldots, x_n \in \mathbb{R}^d$, k-means clustering aims to partition this set of points into k sets $S_1, \ldots, S_k$ so as to minimize the sum

$$\sum_{i=1}^{k} \sum_{x_j \in S_i} \left\| x_j - m_i \right\|^2$$

where m_i is the mean of the points in S_i for all i.

To this end, the algorithm first initializes k centers and then runs a number of iterations to assign the points to a cluster. In each iteration, the closest center to each point is determined and a new clustering results from this. Then, the positions of the centers are updated to be the means of the points in each cluster. The procedure is repeated for a number of iterations or until the ratio of points movement across clusters is sufficiently small. Figure 1 shows a small example of the subgraph generated by a single iteration.

The parallel code in OmpSs includes four kinds of tasks. The main tasks (`CALC`) calculate the closest centers to some of the points and store these results in auxiliary vectors. Then, those partial results need to be unified after a barrier to ensure that they have actually been computed. This is done by means of tasks (`SUM1` and `SUM2`) which sum the mentioned auxiliary vectors using a tree sum algorithm and another task (`NPOS`) which computes the new positions of centers using the previous results. At the beginning of the next iteration, all the auxiliary vectors are reset to zero using another kind of tasks (`ZERO1` and `ZERO2`).

Taking this into account, the amount of data transferred between each pair of tasks (that is, the edge weights) can easily be inferred from the code. This is due to the fact that the data is always assigned to the various tasks in the same manner.

Since the used algorithm has such a simple structure, an execution of this application yields a graph divided in almost equal layers, corresponding to the various iterations. This pattern allows for incorporating additional information into the graph for further analysis; for example, implicit data dependencies hidden by barriers in the code.

The structure of such layers is shown in figure 2. The number of available SPUs determines the degree of parallelism, that is, the number of `ZERO1`, `ZERO2`, `SUM1` and `SUM2` tasks (and so the width of the graph). The number of `CALC` tasks (which form rows of nodes, as can be seen in the figure) depends entirely on the dimensions of the input of k-means.

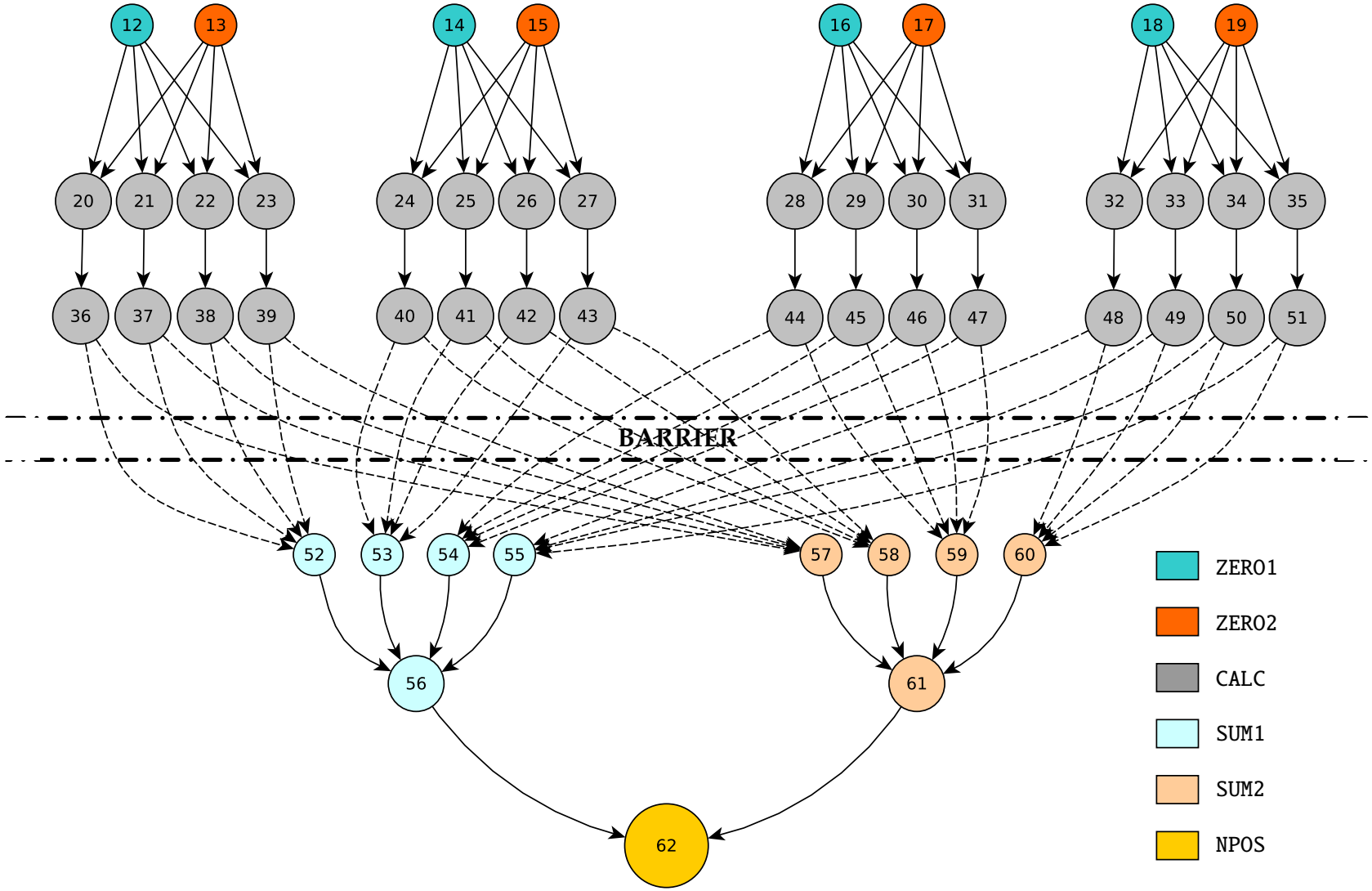


Figure 1: Example subgraph corresponding to an iteration of k-means.

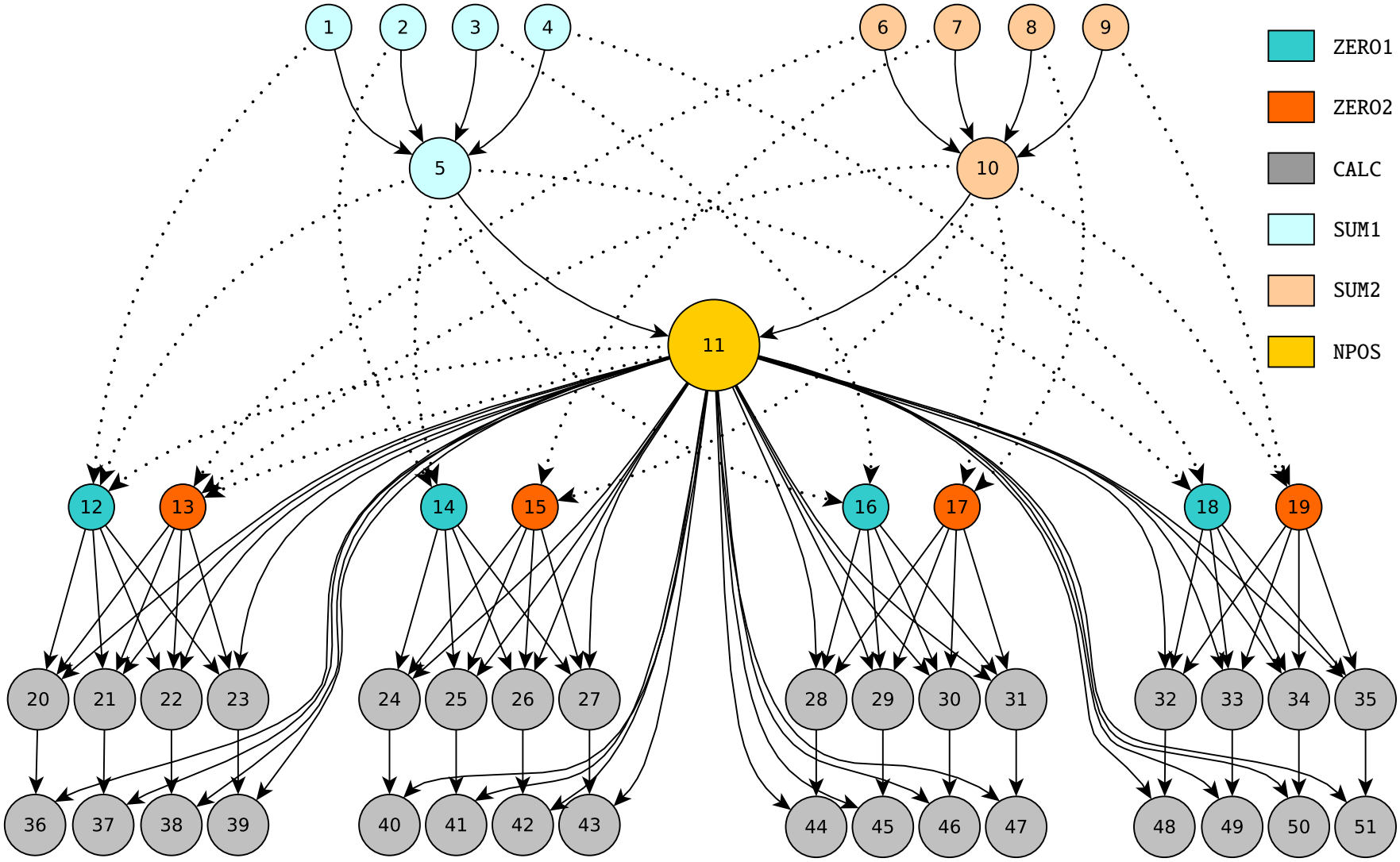


Figure 2: Example of the structure of the subgraph between two consecutive barriers.

3 Graph Partitioning

A broad spectrum of methodologies have been applied to the produced graphs. However, general partitioning of the whole graph yields bad partitions. In particular, the produced partitions tend to be very sequential as they tend to match the barriers (therefore, the nodes in one part need to be executed after the nodes in another part), so parallelism is wasted.

A better approach is to consider appropriate parts of the original graph and partition the corresponding subgraphs. Given the structure of the studied graphs, the subgraphs comprised of the nodes corresponding to each iteration (or, equivalently, the nodes between two barriers) seem to be a wise choice.

Nevertheless, this scheme poses two challenges. On the one hand, a partitioning scheme which avoids the aforementioned inconvenient partitions has to be used. On the other hand, the fact that the partitioning of the subgraph does not take into account the implicit dependencies between different subgraphs can not be ignored.

Next, we present four different techniques to partition every subgraph and three schemes to match these partitions resulting in a partition of the original graph.

3.1 Subgraph Partitioning Techniques

Once a graph has been divided into several subgraphs, each of these subgraphs has to be partitioned independently. Furthermore, to obtain a partition of the original graph with small edge-cut, the partitions of the subgraphs should also have small edge-cut. This is achieved by applying general graph partitioning techniques to every subgraph.

METIS Partitioning (`MP`). One of the most widely used graph partitioner is METIS [1]. METIS can efficiently obtain a balanced partition of a graph with small edge-cut, as desired.

Greedy Partitioning (`GP`). As explained in section 2, the studied graphs have a particular structure which can be exploited to obtain close to optimal partitions.

This greedy algorithm is based on the observation that, ideally, nodes should be on the same part as most of their neighbors. Hence, the algorithm assigns groups of nodes which have no predecessors to the parts in a balanced way. Then, the rest of the nodes are assigned to the part which most of their predecessors belong to, achieving thus a small edge-cut.

Specifically, `SUM1` and `SUM2` nodes which have no predecessors (nodes 1–4 and 6–9 in figure 2) are divided nearly evenly into the desired number of parts. Moreover, nodes which access related data are assigned to the same part if possible (for example, in figure 2, nodes 1 and 6 would be assigned to the same part as long as the number of parts is sufficiently small). Each of the other nodes is assigned to a part in a way as to minimize the edge-cut of the subgraph

composed of that node and its predecessors. However, such a partition would not be balanced in general, because all CALC nodes have the NPOS node as their predecessor (in figure 2, there is an edge from node 11 to each of the nodes 20-51). That is why the algorithm actually ignores this node when taking into consideration the predecessors of each node. Furthermore, some CALC nodes have exactly one predecessor of type ZERO1 and one predecessor of type ZERO2 (such as nodes 20-35 in figure 2), apart from the NPOS node. Therefore, such nodes are assigned to the same part as one of these two predecessors randomly in order to keep balance in the partition (that is, to avoid the imbalance produced by the possible difference of weights between edges from ZERO1 nodes to CALC nodes and edges from ZERO2 nodes to CALC nodes).

With this algorithm, due to the regularity of the graphs, the whole partition is expected to be well balanced and the obtained edge-cut is close to optimal.

Random Partitioning (RP). Another way to compute a partition is to assign each node to a part in a random manner.

The resulting partition is expected to be balanced, even though the edge-cut is not taken into consideration.

Ordered Partitioning (OP). In many architectures, tasks are mapped to nodes in a round-robin fashion. That is, tasks are ordered chronologically and then mapped to computing nodes as these become available. The ordered partitioning technique is intended to imitate this mapping scheme. Thus, the tasks are ordered sequentially and the corresponding nodes are assigned to parts alternately.

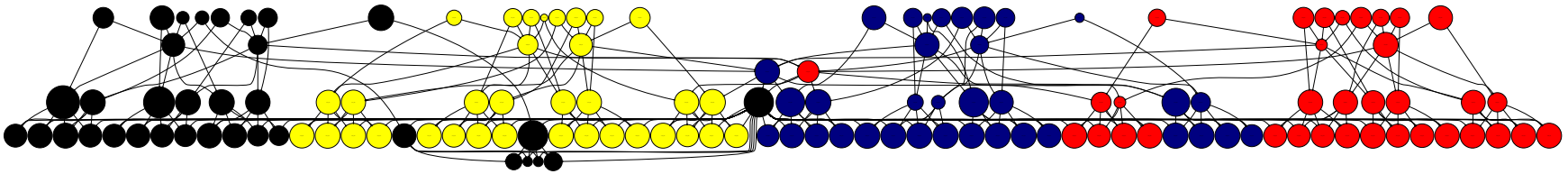
Again, the resulting partition is expected to be balanced but the edge-cut is not taken into consideration.

However, in this application, CALC nodes are organized in a matrix form with column-wise dependencies only (see figure 2). Therefore, if the number of parts in which the graph is partitioned divides the number of nodes per row, very few edges will cross the partition in this part of the graph, contributing thus to reducing the overall ratio of edge-cut.

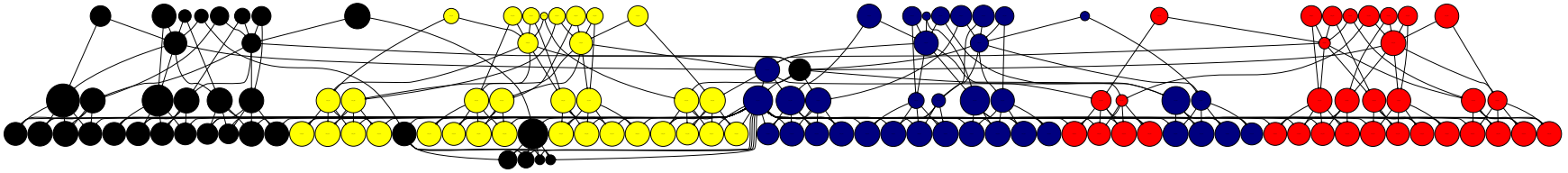
3.2 Matching Schemes

Several ways to partition the subgraphs which form the original graph have been proposed. However, distinct subgraphs are partitioned independently. Therefore, were these partitioned subgraphs merged carelessly, data locality between subgraphs would be of no avail.

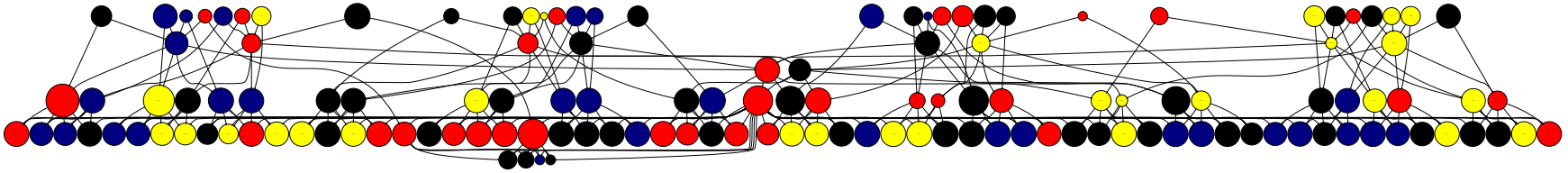
Propagating Partitions (PP). Each iteration of k-means executes the same kind of tasks with the same distribution of data. Hence, the subgraphs between barriers are all analogous. Therefore, there is no actual need to partition every subgraph independently. It is possible to partition



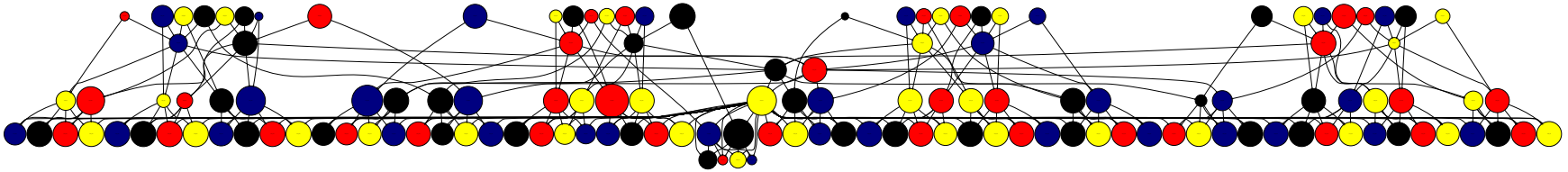
(a) MP



(b) GP



(c) RP



(d) OP

Figure 3: Sample partitions obtained applying MP, GP, RP and OP to the subgraph between two barriers. This subgraph is obtained from the data dependency graph corresponding to an execution of k-means with 65536 points in $\mathbb{R}^4$, 8 clusters and 64 SPUs and is partitioned in 4 parts, each represented by a different color.

only one subgraph and replicate the exact same partition to all other subgraphs. If the partitioning method is good, the edge-cut crossing through barriers will also be small because of the graph regular structure.

Best Permutation Matching (`BPM`). Another possibility consists in partitioning every subgraph independently and finding the best possible permutation in order to minimize the edge-cut through barriers. More precisely, after two contiguous subgraphs (separated by a barrier) have been partitioned, every possible bijection between the sets of parts of each subgraph might result in a different edge-cut, so the minimum is chosen. However, this scheme is very expensive, both in terms of time and memory.

Worst Permutation Matching (`WPM`). The worst permutation could also be chosen in order to compare the variability of the edge-cut. This serves the purpose of evaluating the possible trade-off between edge-cut and computation time dedicated to this phase.

4 Experimental Results

We analyze the obtained edge-cut with the various partitioning techniques described in section 3.

Since the used algorithms are randomized, all the experiments were carried out 100 times. All figures in this section show the ratio between the edge-cut of the partitioning and the amount of data transferred. Both the mean of the obtained results and the sample standard deviation are represented.

Despite partitioning techniques being applied to subgraphs between barriers, the edge-cut through barriers depends greatly on the partitioning technique because of the regularity of the studied graphs. That is to say, if the partition of the different subgraphs has a homogeneous structure, a good matching of the subgraphs is more likely to exist. For instance, in figure 4, we observe that `MP` and `GP` yield a lot of variability in the edge-cut through barriers and, on combining them with `PP` or `BPM`, the results are close to optimal. In addition, `PP` produces slightly better results than `BPM` in these cases, as it makes better use of the structure of the graph. On the contrary, when combined with the less refined methods, `RP` and OP, the impact of the matching scheme is practically zero. However, the obtained results are quite bad on average, having most of the data crossing between different parts of the graph.

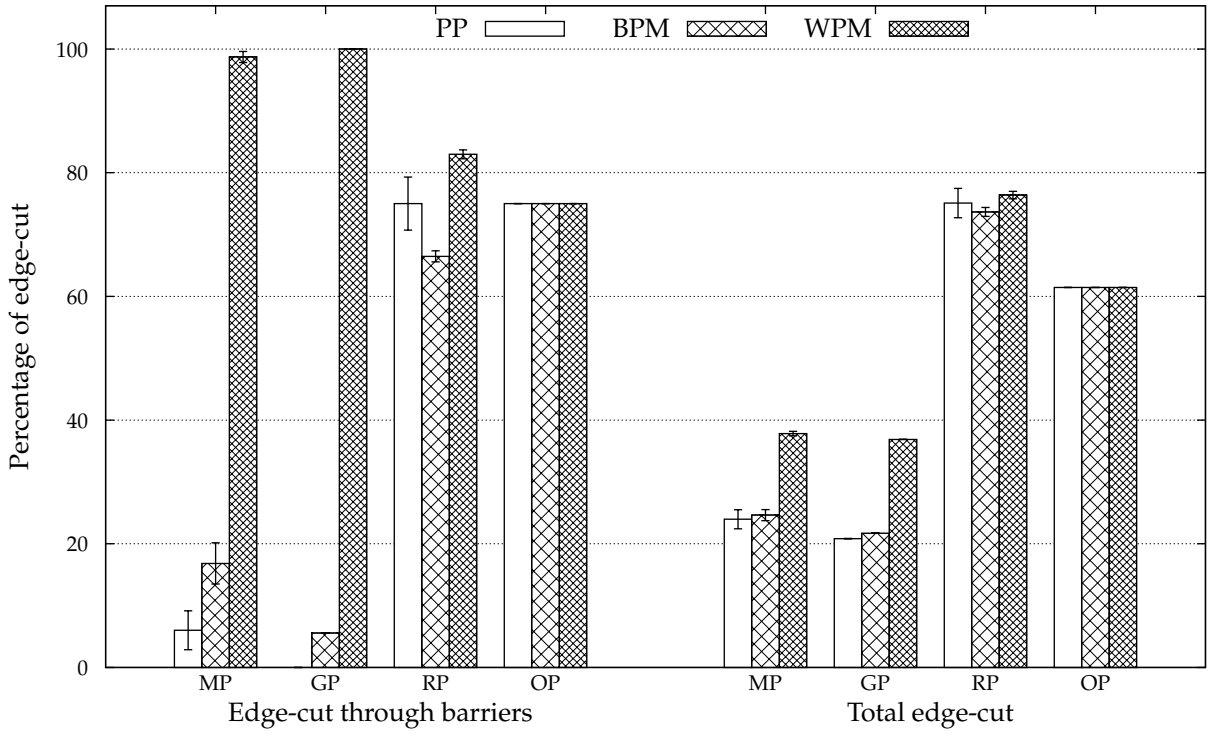


Figure 4: Edge-cut of partitions obtained using `PP`, `BPM` and `WPM` in combination with `MP`, `GP`, `RP` and `OP`. The partitioned graph is the data dependency graph corresponding to an execution of k-means with 65536 points in $\mathbb{R}^4$, 8 clusters and 64 SPUs. The graph is partitioned in 4 parts. Error bars represent the sample standard deviation.

The differences can be further appreciated in figure 5. MP and GP produce similar results and significantly outperform RP and OP, when used both with PP and with BPM. Although METIS implements partitioning algorithms of general purpose, the results achieved by MP are almost as good as the ones obtained by GP, which is designed taking into account the particular structure of the studied graphs.

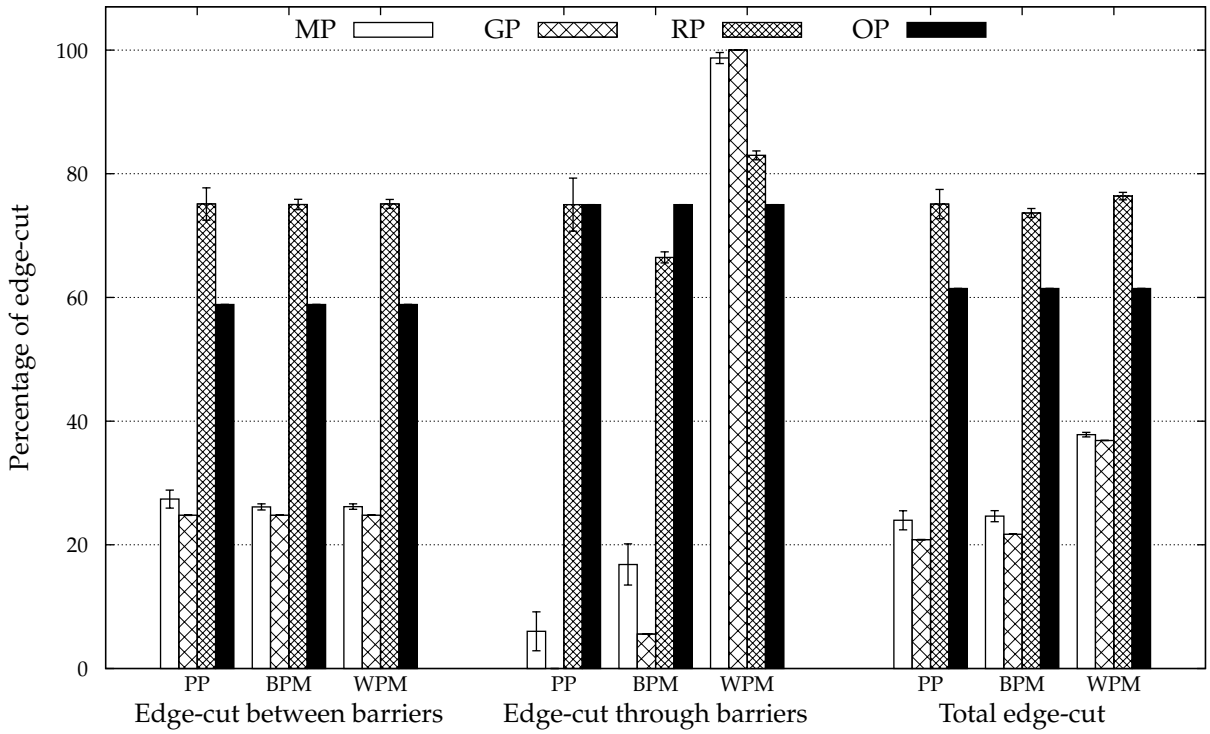


Figure 5: Edge-cut of partitions obtained using MP, GP, RP and OP in combination with PP, BPM and WPM. The partitioned graph is the data dependency graph corresponding to an execution of k-means with 65536 points in $\mathbb{R}^4$, 8 clusters and 64 SPUs. The graph is partitioned in 4 parts. Error bars represent the sample standard deviation.

The results shown so far correspond to only one representative graph. Therefore, figures 6 and 7 show how the edge-cut scales according to the sizes of graphs for four and seven parts, respectively.

Nevertheless, PP has already proven to be comparable to (or even better than) BPM in terms of edge-cut, and BPM requires significantly more time and becomes practically infeasible for larger graphs. Hence, only the results obtained combining PP with the four partitioning techniques are shown.

The relative edge-cut produced with RP roughly remains constant, as expected. In contrast, as the input size grows, the data dependency graphs have more vertices and more edges connecting these new vertices to other vertices of the graph (in particular, as the size increases, there are more rows of CALC nodes, which are all connected to the NPOS node, and most of these edges contribute to the edge-cut). That is why the relative edge-cut slightly increases for both MP and GP and also for OP when the graph is partitioned in 7 parts. However, the relative edge-

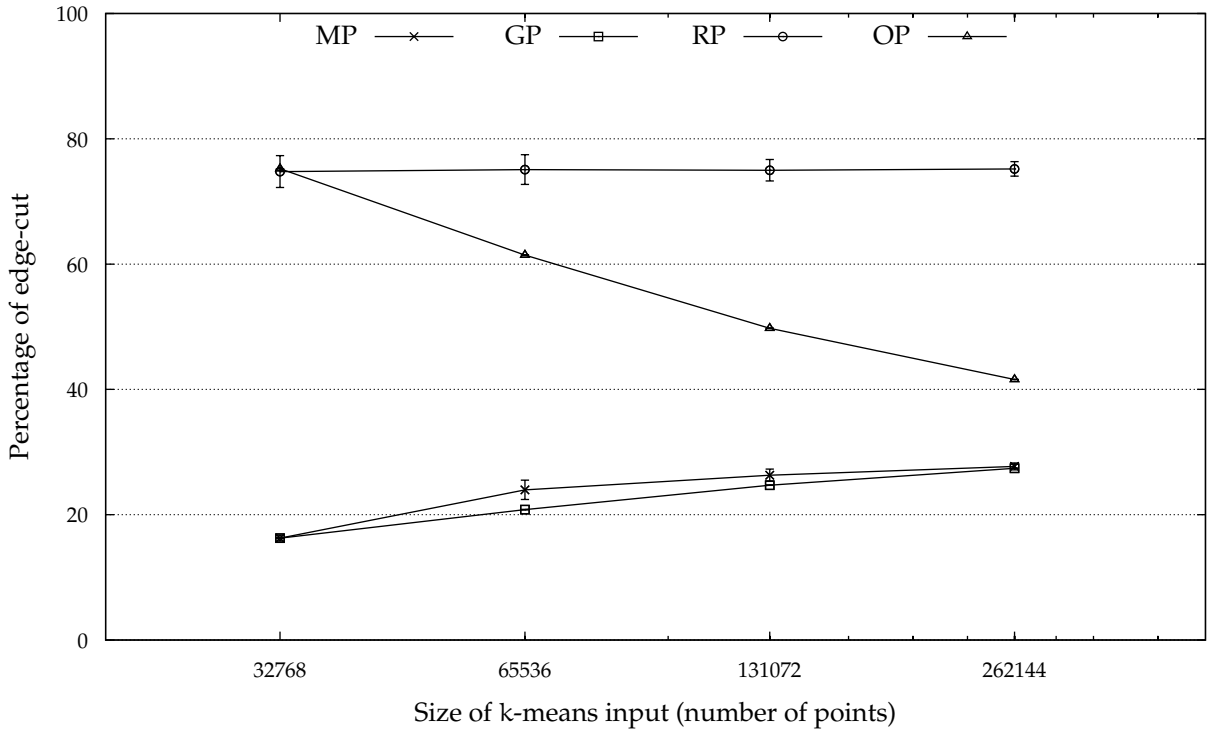


Figure 6: Edge-cut of partitions obtained using `MP`, `GP`, `RP` and `OP` in combination with `PP` on graphs of different sizes. All graphs are partitioned in 4 parts. The x axis is in logarithmic scale. Error bars represent the sample standard deviation.

cut obtained when `OP` is used to partition the graphs in 4 parts appears to decrease with the size of the graph, even though it remains worse than both `MP` and `GP`. The reason for this seemingly unexpected behavior is that, as graphs grow larger, they become more and more regular (that is, more rows of `CALC` nodes are added) and `OP` can take greater advantage of their structure (since the number of nodes per row is a multiple of 4), producing partitions which are ever more similar to those produced by `GP`.

The orders of the subgraphs between barriers produced by the studied graphs range from 150 to 600 nodes roughly. That is why the graphs have been partitioned in few parts in the previous analysis. Nonetheless, the comparison can be further extended to different numbers of parts as well. Hence, figure 8 shows the evolution of the relative edge-cut of a graph (with approximately 200 nodes per subgraph) relative to the number of parts.

As it is to be expected, the relative edge-cut of every partitioning algorithm worsens as the number of parts increases, while the relative performances remain the same. However, further differences in the behavior of these techniques can be appreciated. For instance, the edge-cut produced by `RP` and `OP` becomes stabilized because it approximates the total amount of data. In contrast, the increase in the edge-cut when passing from 16 to 32 parts is much more accentuated for `MP` and `GP`. This is due to the the number of parts approaching the number of nodes in each subgraph. Therefore, sophisticated algorithms can not achieve such good results.

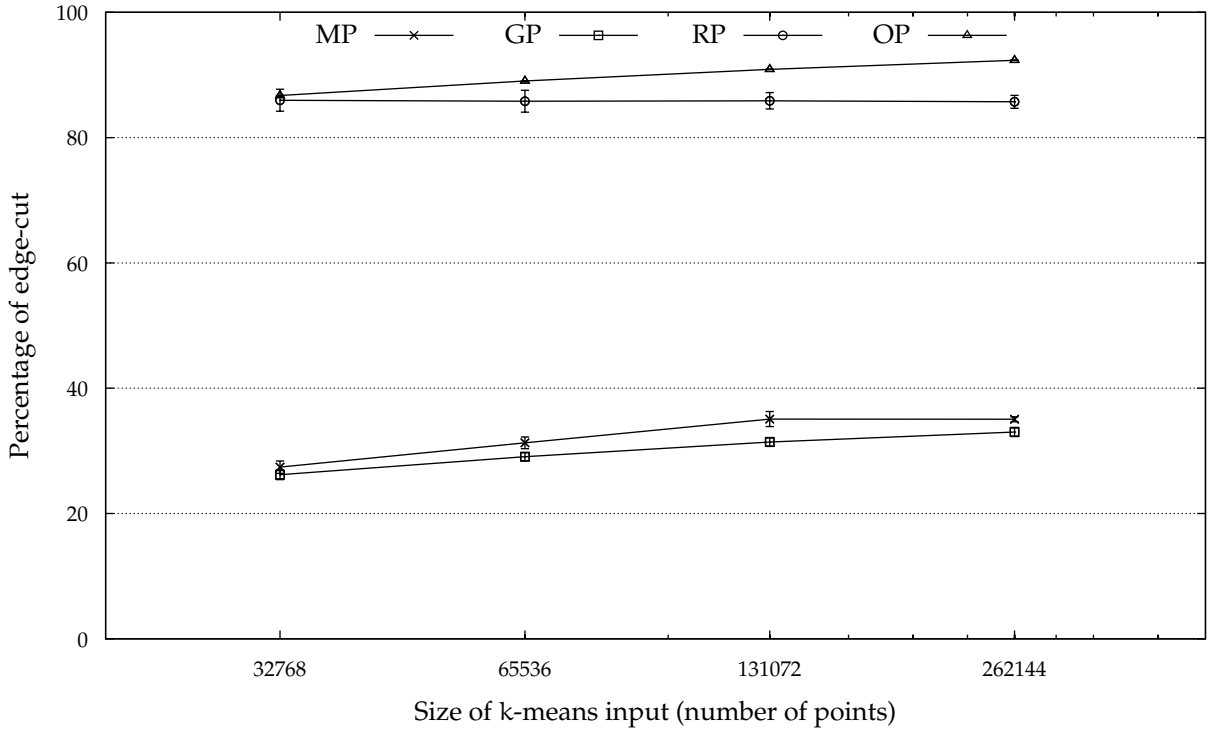


Figure 7: Edge-cut of partitions obtained using `MP`, `GP`, `RP` and `OP` in combination with `PP` on graphs of different sizes. All graphs are partitioned in 7 parts. The x axis is in logarithmic scale. Error bars represent the sample standard deviation.

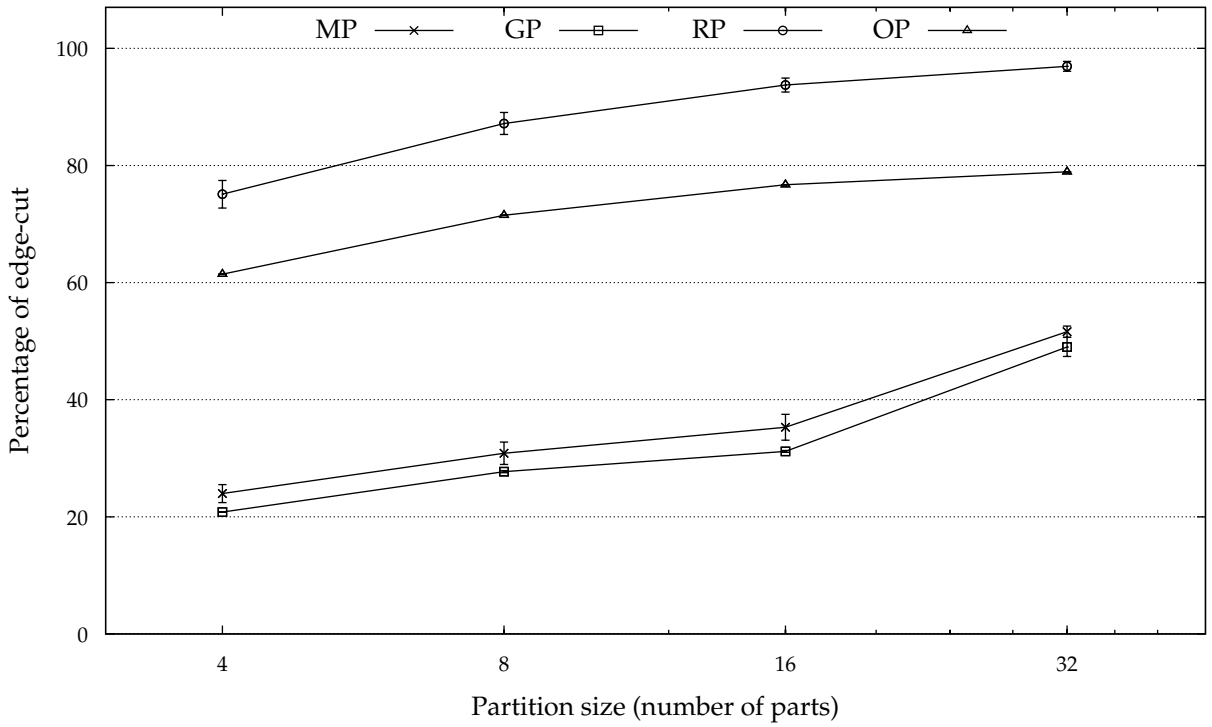


Figure 8: Edge-cut of partitions of different sizes obtained using `MP`, `GP`, `RP` and `OP` in combination with `PP`. The partitioned graph is the data dependency graph corresponding to an execution of k-means with 65536 points in $\mathbb{R}^4$, 8 clusters and 64 SPUs. The x axis is in logarithmic scale. Error bars represent the sample standard deviation.

5 Conclusions

In this work, we investigated the impact of partitioning data dependency graphs of applications previous to executing them on the amount of data which needs to be reallocated.

We analyzed the edge-cut produced by several graph partitioning techniques applied to graphs corresponding to executions of k-means in order to compare them. This analysis shows that communication between computing nodes can be effectively reduced with some preprocessing. Indeed, the amount of data which needs to be transferred between different parts of a randomly partitioned graph is up to four times more than the communication volume using a more sophisticated partitioning of the graph.

In particular, partitioning the subgraphs between barriers with METIS and propagating the obtained partition to all such subgraphs produced very good results. In spite of the greedy heuristic performing slightly better, this heuristic was especially devised for k-means. Thus, a huge effort is expected to be needed to adapt the same ideas for every application, while METIS is a much more general approach which can be applied to every application.

In conclusion, we proposed a strategy to partition data dependency graphs which is likely to reduce dramatically communication costs in many parallel applications with little additional effort, as METIS is already efficiently implemented in many platforms and data dependency graphs tend to be small. We support this claim with evident improvement over currently used strategies in an OmpSs implementation of k-means.

References

[1] KARYPIS, G., AND KUMAR, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing 20*, 1 (1998), 359–392.