Characterizing the Communication Demands of the Graph500 Benchmark on a Commodity Cluster

Pablo Fuentes, José Luis Bosque, Ramón Beivide
University of Cantabria, Santander, Spain
{pablo.fuentes, joseluis.bosque, ramon.beivide}@unican.es

Mateo Valero
Barcelona Supercomputing Center, Barcelona, Spain
mateo@bsc.es

Cyriel Minkenberg
IBM Zurich Research Laboratory, Zurich, Switzerland
sil@zrl.ibm.com

***Abstract*—Big Data applications have gained importance over the last few years. Such applications focus on the analysis of huge amounts of unstructured information and present a series of differences with traditional High Performance Computing (HPC) applications. For illustrating such dissimilarities, this paper analyzes the behavior of the most scalable version of the Graph500 benchmark when run on a state-of-the-art commodity cluster facility. Our work shows that this new computation paradigm stresses the interconnection subsystem.**

In this work, we provide both analytical and empirical characterizations of the Graph500 benchmark, showing that its communication needs bound the achieved performance on a cluster facility. Up to our knowledge, our evaluation is the first to consider the impact of message aggregation on the communication overhead and explore a tradeoff that diminishes benchmark execution time, increasing system performance.

***Keywords*-Graph500, cluster supercomputing platforms, communication characterization, message aggregation**

I. Introduction

Over the last few decades there has been an exponential rise in the amount of data to be processed in multiple human activities. Furthermore, in parallel to this data increase there has been a fast growth in its complexity. Both factors originate a need for higher computational capacities and introduce several algorithmic challenges. Such challenges can be summarized in a need to discover patterns in the data, to create a framework to analyze those patterns facing time and resource restrictions, and to predict future behaviors upon those patterns. These challenges are valid in fields as diverse as social networks, medical informatics, banking and cybersecurity among others. One example is the use of the Facebook social network, whose number of active monthly users has grown exponentially in last years before achieving more than 1.2 billion as of late 2013.

The core idea behind these Big Data problems is to extract knowledge from a huge range of unstructured data, to ease analysis and decision taking. A convenient model for this information is a graph, which permits to reorganize data by searching spanning trees embedded in the graph.

The size of these Big Data applications requires high amounts of memory and computing capabilities. The state-of-the-art computing server performance does not meet such requirements and does not scale at their growth pace. The only realistic option is to run such applications in parallel, partitioning the graph and its associated computation across several processes. Parallel computers have been constantly used in other computing fields for decades but are not optimized for this new set of problems.

In this context, Graph500 [1] organization arises to gather international High Performance Computing (HPC) experts from the industry and academia, with the aim to determine the capacity of current computing systems to run graph-based applications. Its main contribution is a large-scale benchmark which performs a typical concurrent tree search algorithm called Breadth-First Search (BFS). Graph500 proposes a new performance metric, named the number of Traversed Edges Per Second (TEPS). TEPS are calculated as the division of the number of edges traversed in the graph by the execution time for kernel 2. The Graph500 benchmark consists of various kernels working with large graphs:

1) Construction of a graph from a random edge list (kernel 1).
2) Ancestors tree computation for a random sample search key through a BFS algorithm (kernel 2).
3) Validation of the parent tree from kernel 2, through an assertion of accomplished properties.

Due to the extended and fast-rising relevance of Big Data applications, it is essential to determine the existence of any possible performance losses, and identify their sources. In this regard, a comprehensive study of the specific algorithm behavior can be indispensable to locate and minimize inefficiencies in the code execution, and to optimize resource usage. The variety of these applications turns unfeasible to repeat such scrutiny over each of them. The Graph500 benchmark can be a good start point, as it represents a subset of graph-based, data-intensive applications. A more ambitious, longterm proposal would be to translate the lessons learned analyzing the Graph500 into new computer architectures and enhanced software stacks.

In this work, we analyze the Graph500 benchmark code to establish a thorough understanding of its behavior, and characterize its parallel execution in a cluster facility intended to run HPC applications. Our main contributions are:

- We characterize the communications in the Graph500 benchmark, determining their spatial and temporal homogeneity.
- We evaluate the impact of the interconnection network on the Graph500 performance, determining sources of performance losses. More specifically, we focus on the use of message aggregation and explore the existence of an optimal value that improves overall performance.

The remainder of this paper is organized as follows: first, we introduce the most relevant related work. Next, we will analyze the BFS algorithm, with special focus on characterizing the communications originated in its parallel execution. Then, we will present empirical results collected by direct execution to evaluate our hypothesis. Finally, we will summarize our main findings and consider future work.

II. Related Work

The Graph500 benchmark is based on a Breadth-First Search (BFS), a graph search algorithm [1]. Several authors have proposed efficient and scalable shared memory implementations of BFS algorithms on commodity multicore processors. Agarwal et. al. [2] have presented a multi-core implementation with several optimizations. A very interesting work has been developed by Beamer, Asanovic and Patterson in [3], where they propose a hybrid approach that is advantageous for low-diameter graphs, combining a conventional top-down algorithm along with a bottom-up one. The bottom-up algorithm reduces the number of edges visited, which in turn accelerates the search as a whole.

Checconi et al. [4] described a family of highly-efficient Breadth-First Search (BFS) algorithms, optimized for their execution on IBM Blue Gene/P and Blue Gene/Q supercomputers. Alternatively, Bulu and Madduri [5] conducted a performance evaluation of a distributed BFS using 1D and 2D partitioning on Cray XE6 and XT4 systems.

Finally, the BFS algorithm has been implemented in architectures with hardware accelerators. For instance, Hong et al. in [6] presented a hybrid method which dynamically decides the best execution method for each BFS-level iteration, shifting between sequential execution, multi-core CPU-only execution, and GPUs. Tao, Yutong and Guang [7] developed two different approaches to improve the performance of BFS algorithm on an Intel Xeon Phi coprocessor.

In this paper, we concentrate on a pure MPI BFS benchmark executed on a state-of-the-art cluster supercomputer facility using current Xeon-based servers and Infiniband network technology. This tries to represent a highly scalable platform with a good performance/cost ratio. To our knowledge, the only previous Graph500 benchmark characterization has been conducted by Suzumura et. al. in [8]. In the current paper, we compare their estimations to empirical data obtained from our experiments. In addition, and more importantly, we do not know any previous work considering the impact on performance of message aggregation.

III. The Breadth-First Search Algorithm

Breadth-First Search is a strategy to traverse a graph that organizes all the elements of a graph in a tree, starting by a given root vertex. The search of the tree is conducted in multiple stages or ‘graph levels’, by traversing all the edges that are connected to each of visited vertices in the previous level. The vertices are analyzed in a FIFO-queue fashion. This behavior enforces vertices to be searched in order of their distance from the root vertex. One of BFS uses is to find the path which traverses the lowest number of edges between two specific vertices in a graph. More details about BFS and some alternative implementations are given in [9].

A. BFS Implementation

BFS receives two parameters (scale and edgefactor) and returns a series of statistics with the time employed for the BFS execution, and number of TEPS. ’Scale’ refers the base two logarithm of the number of vertices in the graph. The ‘edgefactor’ is the half of the number of edges per graph vertex, where an ‘edge’ is a link that joins a pair of vertices. In this paper we will refer the ‘problem size’ as the size of the graph, given by the values of scale and edgefactor.

Each step of the BFS algorithm (BFS level) can be seen as a Sparse-Matrix Vector multiplication where the matrix is the graph’s adjacency matrix recording the connectivity between pairs of vertices and the vector corresponds to the list of vertices that have to be visited in such BFS level.

In this work we will focus on the ‘simple’ implementation of the Graph500 benchmark. This version is expected to achieve higher performance than other implementations when the problem size is big. Additionally, it presents more memory scalability: each process only operates with the visited vertices of its own part of the graph, instead of storing the complete array of visited vertices. The drawback of this version is that every process needs to send visited vertices to their host processes, resulting in a higher exchange of data that increases the network impact on the performance. The ‘simple’ version is considered to be a better approach for relatively big problem sizes with a high number of processes employed. Ueno [10] describes with more detail the differences and communication needs of the different implementations of the algorithm.

We have analyzed the behavior of the ‘simple’ version of the algorithm, and modelled it through a pseudocode described in Fig. 1 which combines computation and communication phases. This helps to understand how it will perform and to identify potential sources of inefficiency.

B. Analysis of the communications

In the pseudocode of the algorithm in Fig. 1 we can see the presence of two send calls, both implemented through asynchronous MPI_Isend functions. The first send (in line 8) can

```
1: $visited = current = 0$
2: $next = \{root\}$
3: repeat
4:    $current = next$
5:    for $vertex$ in $current$ do
6:       for $neigh$ in Neighbors($vertex$) do
7:          if $neigh$ hosted in another process then
8:             MPI_Isend [$vertex$,$neigh$] to host process
9:             while not MPI_Test (outgoing messages completed) do
10:               if outgoing message completed then
11:                  clear sending buffer
12:               if MPI_Test (incoming messages) then
13:                  [$vertex$,$neigh$] = receive()
14:                  if $neigh$ not visited then
15:                     visit $neigh$ and add it to $next$
16:         else
17:            if $neigh$ not visited then
18:               visit $neigh$ and add it to to $next$
19:    for all process do
20:       MPI_Isend (this process has stopped sending)
21:    repeat
22:       if MPI_Test (incoming messages) then
23:          [$vertex$,$neigh$] = receive()
24:          if $neigh$ not visited then
25:             visit $neigh$ and add it to to $next$
26:    until rest of processes have stopped sending
27:    MPI_Allreduce($next$)
28: until $next$  is empty
```

Fig. 1. BFS pseudocode with communications

have different message size for communications aggregation: it starts accumulating queries to another processes and only generates a message when the amount of queries exceeds a threshold called ”coalescing size”. When the graph level has been traversed, those queries that have not completed a full message are dispatched regardless of their size. The second send (line 20) is employed for each process to communicate to the rest that it has ended dispatching messages for this level.

The number of messages originated in the second send is much lower than in the first, and its contribution to total traffic can be neglected. We can estimate the amount of messages that will be sent per process by the first MPI_Isend function as:

$$\#\ {}^{messages}\!/_{process} = \frac{2^{Scale+1}\cdot edgefactor}{coalescing\ size}\cdot\frac{n-1}{n^2}, \tag{1}$$

where n represents the number of processes employed. This estimation can be explained as follows: the algorithm has to traverse all the edges in the graph, $2^{Scale}\cdot edgefactor$. Since the graph is undirect, edges are traversed in both senses, leading to a factor of 2. Processes generate a query per every vertex to visit that is hosted by another process. Considering that graph distribution is homogeneous, $1/n$ of the visited vertices will be located in the same process and will not originate a query, leading to a factor of $(n-1)/n$. Additionally, each process only traverses its own part of the graph. Finally, message aggregation forces multiple queries to be joined into a single message, with $coalescing\ size$ adjusting the amount of queries per message.

Additionally, a query consists of 2 vertices (the vertex to visit and its possible tree parent), with each vertex being an integer of 8 bytes. As we have stated, one message consists of $coalescing\ size$ queries. This gives a total amount of exchanged data in the network estimated by

$$Data\ sent = 2^{Scale+5}\cdot edgefactor\cdot\frac{n-1}{n}. \tag{2}$$

Last equation matches the estimation given by Suzumura et al. in [8], and will be later contrasted with empirical data from our experiments in section IV-B.

At the beginning of this subsection we have described the function calls in the code that originate communications. A second analysis concerns the behavior at the reception side, which is composed by three test calls at lines 9, 12 and 22, and one All-reduce at line 27 in Fig. 1.

Test in line 9 determines when an outgoing message has been dispatched, to flush the corresponding buffer. Both tests in lines 12 and 22 check the arrival of messages from other processes, and, for our purposes, they will be evaluated as a single call. All these test calls are asynchronous functions which are executed a high number of times, and introduce a significant overhead over execution time. Specifically, test in line 22 is placed inside a loop at the end of each graph level, acting as a polling for synchronization between processes. This test clearly impacts on performance, because processes enter such loop when they do not have any remaining messages to dispatch, and its execution is overlapped with computation only when a message is received. We will not consider the impact of reception functions, as they are asynchronous and will be only executed upon a positive test.

Finally, the All-reduce is a collective blocking call that will stall those processes that arrive first. However, it is used after the polling and will find processes with a high level of synchronization, so we expect it to have a minor impact on the overall behavior of the algorithm.

IV. Empirical Evaluation of the BFS kernel

A. System description

Our evaluations have been conducted through the ”Altamira” supercomputer cluster, located at the University of Cantabria. This platform consists of IBM-idataplex dx360m4 nodes, each one with two Intel Sandybridge Xeon E5-2670 processors, 8 cores per processor, and 64 GB of RAM. These nodes are interconnected by a folded Clos topology using Infiniband FDR10 network technology. The results shown in this paper correspond to the execution of the benchmark code in its version 2.1.4 [11]. This code was compiled with GCC version 4.4.7 [12] and the OpenMPI 1.6.5 library [13], which was the latest stable release version at the time.

B. Communication pattern

Communications in the benchmark execution are originated by two asynchronous MPI send calls: one to exchange queries for the graph traversal (line 8 in Fig. 1) and the other to communicate the end of the graph level traversal (line 20). For simplicity, we will refer to them as ‘send1’ and ‘send2’, respectively. Fig. 2 displays the total number of messages sent for these two calls, for an execution with scale 25, edgefactor 16, and 128 processes. Results show that ‘send1’ has been called 30 times more than ‘send2’. The amount of those messages, detailed in (1), is inversely related to the ‘coalescing size’ parameter described in section III-B, which in this case has been set to 256. It should be noted that employing these actual parameters in (1), a number of approximately 2^{15}

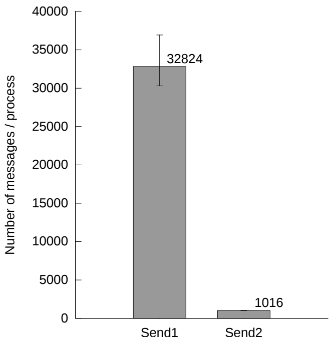


Fig. 2. Number of messages sent per MPI process. Results correspond with a problem of scale 25 and edgefactor 16, ran with 128 processes. Bar value represents average value, errorbar shows standard deviation between processes.

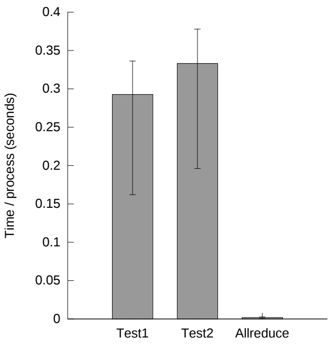


Fig. 3. Total execution time per process of each reception-related call. Results correspond with a problem size of scale 25 and edgefactor 16, with 128 processes. Bar values represent average values, error bars show minimum and maximum values among all processes.

messages per process is obtained which perfectly match with the empirical data obtained from direct execution. A number of 32512 messages per process is obtained, with the actual value (32824 messages) diverging less than 1%. This difference is explained by the message aggregation and the random nature of the algorithm: in every graph level, there will be queries that do not complete a 256-queries message, thus increasing the number of messages that are exchanged in total. This variability completely disappears when we consider total data exchanged in 'send1', as the number of queries is fixed by the problem size.

As we explained in the code description (Section III-A), graph data distribution is uniform across processes. Since all the processes must execute the computations on the vertices they host, this should lead to an homogeneous spatial distribution of messages. The results from our evaluations have proven the existence of a symmetry in the communication matrix for the messages dispatched in 'send1' during the execution of the BFS key search. This symmetry does not correspond to a petition-response model, but to the undirected nature of the graph traversed. Searches on undirected graphs imply that all the edges must be traversed twice, once per edge direction. This communication distribution is widely known and acts as a validation step for the accuracy of our evaluation platform, which will be used to achieve the more comprehensive information explained in further subsections of the paper.

C. Behavior of the synchronization points

As it was stated in Section III-B, there are four points in the code which can lead to communication waiting times: three MPI_Test calls and one MPI_Allreduce. We can observe that tests in lines 9 and 22 of Fig. 1 are originated in polling loops, in which processes repetitively execute these functions until they return a positive match (confirming that a message has been sent or received, upon each case). For the rest of the paper, we will refer to these synchronization points as 'test1' (tests for incoming messages, grouping the execution of tests in lines 12 and 22), 'test2' (tests for dispatched messages) and 'allreduce' (to signal the BFS level completion).

Fig. 3 reporting empirical values confirms our description, showing a significantly lower amount of time spent on 'allreduce' call because it is executed after the polling loops. 'Test2' is in charge of confirming message dispatching to flush send buffers. 'Test1' determines the existence of incoming communications from other processes. Although 'test2' time is slightly higher than 'test1', the impact of 'test1' on benchmark execution time is more important because a significant part of it happens when processes are stalled at the end of a graph level. Hence, diminishing that time would have an immediate effect on reducing the whole benchmark execution time. This analysis is extended on the following section.

D. Load balancing analysis

We have characterized the benchmark communications in section IV-B as an application with high spatial homogeneity. In this section, we will evaluate the time homogeneity of those communications. Based on the code analysis in Section III-A we can conclude that transmissions will be homogenously distributed between processes across execution time. Nevertheless, these processes will synchronize at the end of every graph level, which can lead to unproductive stalls in those processes that end their graph level analysis first.

Fig. 4 plots the execution of 1 BFS key search with three phases: communications between processes interleaved with several small computation blocks, polling loops, and 'allreduce' call executions. There is also computation during the polling loops, but only when a message is received, which occurs significantly less than during communications. Those polling loops are placed between the end of every communications phase and the 'allreduce' execution, and act as a synchronization point between all the processes, continuing as long as any process is still dispatching messages. The number of 'allreduce' calls equals the number of graph levels plus one last empty iteration, as can be observed in the pseudocode in Fig. 1. For this problem size, there are 7 'allreduce' calls.

The execution is divided into several stages, one per graph level. The first and last phases show an almost nonexistent amount of communications and little gap with 'allreduce' calls. Two intermediate phases (the middle graph levels) concentrate around 90% of the execution time as it can be better appre-

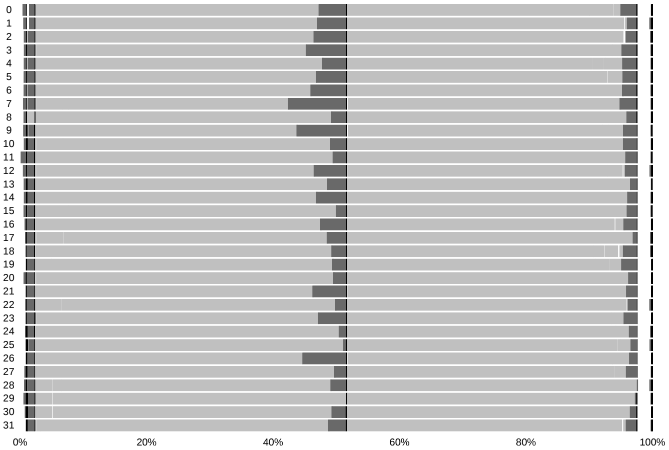


Fig. 4. Snapshot of 1 BFS search execution for a problem with scale 22 and edgefactor 16, executed with 32 MPI processes. X-axis represents the execution time, scaled to total BFS kernel execution time. Numbers in Y-axis refer every MPI process employed. Light grey bars correspond to communication phases. Dark grey bars correspond to polling loops for process synchronization. Black bars correspond to MPI_Allreduce calls which act as synchronization barriers at the end of every graph level. During white gaps and interleaved with the communication in the light grey parts are multiple small computation phases. In the polling loops (dark grey bars) there are also computation phases, although to a much smaller ratio.

ciated in the figure. These stages are the most interesting in terms of interprocess communications and clearly show a load imbalance problem. The first of these stages is bounded by the communications of processes 25 and 29, whereas processes 7, 9 and 26 show highest polling time awaiting for incoming messages. There are less processes on the communications phase when they get closer to the 'allreduce' call, diminishing the probability of an incoming message and reducing the ratio of effective execution. In the second of these two phases a similar behavior can be observed.

This behavior originates because the tree search does not involve an equal number of vertices to be traversed per process and, more importantly, because the graph is not regular. This lack of regularity implies that some vertices will have a significantly higher degree than others, and is representative of the common structure of graphs in Big Data applications. This evaluation proves the impact of communications on this benchmark, because most of the execution time corresponds with message delivery and tests for the completion of either side of the transmission. It also demonstrates a temporal uniformity in the communications pattern.

E. Impact of message aggregation

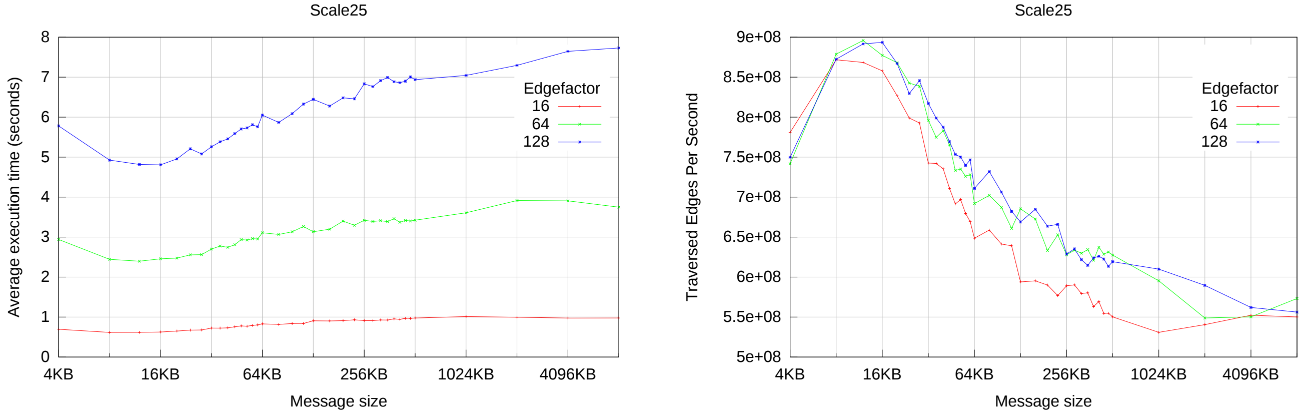
As stated in previous sections, 'Send1' constitutes the core of the benchmark communications. The number of messages between processes can be modulated by aggregating queries, with coalescing size being the threshold that determines the granularity of the aggregation. This threshold is affected by two different tradeoffs.

One is a technological tradeoff and is related to the employed communications network: if the network latency is low, it is interesting to have a fine granularity which translates into sending more messages with less queries on them. On the other hand, each message has an overhead caused by the message creation that can not be neglected, and that enforces a higher level of aggregation, to take advantage of the network bandwidth. This technological tradeoff will obviously vary from one computing system to another, and enforces to conduct a detailed evaluation of the coalescing size parameter for each system employed.

The other tradeoff is driven by the algorithm implementation. Message sending is asynchronous, so we need to prevent sending buffer from being altered while a message is being dispatched. To guarantee an atomic access to this buffer, processes enter a polling loop using 'test2' to determine the dispatching completion. This loop is interleaved with another test that verifies the reception of any incoming messages, and its subsequent computation, reducing the amount of unproductive CPU usage. Increasing the value of the coalescing size will lower the number of messages exchanged and the time consumed in their delivery, but it will also increase the time employed in message reception and sending buffer release because messages are larger. Additionally, the number of tests for incoming messages conducted during the traversal of the graph level will be reduced, delaying a higher proportion of the computation associated to incoming queries towards the end of every graph level. Moreover, this can potentiate the imbalance between processes that was observed in previous subsection.

By default, the coalescing size value is set to 256 queries per message in the original code. Every query consists of two vertices (vertex to visit and its ancestor) stored as 8-byte unsigned integers, giving a total message size of 4KB. By finely tuning this value we can adequately exploit the particular network infrastructure and achieve a compromise between time dedicated to sending messages and time spent awaiting for other processes to synchronize. This compromise would reduce the total amount of time spent in the execution of the BFS algorithm. Fig. 5 shows the mean answer time and TEPS for different message sizes, with a problem size of scale 25, and three edgefactor values: 16, 64 and 128. The coalescing size ranges from the default value of 256 to 524288, corresponding with message sizes from 4KB to 8MB. All the experiments were conducted running 128 processes, fully populating 8 nodes.

Fig. 5 portrays the execution time for a BFS search and the number of Traversed Edges Per Second (TEPS). TEPS is the selected metric for the Graph500 benchmark, and is transparent to the edgefactor because a higher connectivity implies longer execution times but also larger number of edges to be traversed. This explains the similarity between the 3 edgefactor curves in the TEPS metric. The value of curves in Fig. 5(b) can be calculated as the division of the number of graphs edges by the execution time shown in corresponding curve of Fig. 5(a). Curves in Fig. 5(a) show the presence of a minimum execution time for a message size of 12KB, corresponding to a coalescing size of 768. The impact of this time reduction is more visible when the edgefactor is higher, since it increases the graph connectivity and thus the number of messages to be sent, as stated in (1). After this minimum, execution time rises and overpasses the achieved time for the



(a) Execution time Scale 25

(b) TEPS Scale 25

Fig. 5. Average execution time and TEPS for a problem size of scale 25 and different edgefactors, with several message sizes. Every curve represents an edgefactor value. Results correspond to the average of 10 executions with 32 BFS key searches per execution. All the executions employ 128 processes distributed across 8 computing nodes. Each message size is defined from a coalescing size value, multiplied by a factor of 16.

default coalescing size. Finally, the curves reach an asymptote which marks the upper bound for the execution time. This behavior unveils the predicted existence of a tradeoff between time spent dispatching messages and the time consumed in polling loops awaiting for incoming communications. Greater connectivity levels provoke a higher number of edges to be traversed in the graph, and explain the difference between the three curves. This evaluation confirms the big impact of the coalescing size on performance, and the need to carefully select this value to achieve optimal performance.

V. Conclusions

Big Data applications have gained attention from industry and academia in the past few years. These applications present high demands for memory and network subsystems due to their huge data sizes with fast growth rhythms. Most HPC supercomputer systems are not designed to address the specific needs of such applications. Graph500 has proposed a benchmark to rank computing systems upon their suitability for Big Data applications, promoting the optimization of new systems for data-intensive workloads. This benchmark consists of a parallel Breadth-First Search over a graph.

This paper concentrates on two main contributions. The first is the analysis of the highly scalable simple Graph500 benchmark behavior, focusing on characterizing its communications demands. This analysis is followed by an empirical evaluation of the Grap500 performance obtained from direct monitoring of real executions. Our experiments reveal the spatial and temporal uniformity of the communications, and their big impact over the execution time.

The other main contribution is the evaluation of message aggregation for the communications between processes and its impact on benchmark performance. As far as we are concerned, this analysis has not been conducted previously. A tradeoff can be achieved for the two effects that this aggregation causes: time spent for message generation and dispatching, and time consumed awaiting incoming communications. This tradeoff is related both to the network technology and to the algorithm behavior. In our experiments, a finely tuning of the coalescing size value has permitted significative performance improvements of more than 10% over the default value.

In forthcoming works we will extend our analysis of the Graph500 benchmark to other parts of the system infrastructure apart from the interconnection network. We will also consider repeating some of our experiments in different computing platforms, as well as evaluating other BFS implementations.

Acknowledgment

This work has been supported by the Spanish Science and Technology Commission (CICYT) under contract TIN2010-21291-C02-02, the European Unions FP7 under Agreements ERC-321253 (RoMoL) and ICT-288777 (Mont-Blanc) and by the European HiPEAC Network of Excellence. This project has also been partially funded by the JSA no. 2013_119 as part of the IBM/BSC Technology Center for Supercomputing agreement.

References

[1] R. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, “Introducing the Graph 500,” *Cray User’s Group (CUG)*, May 5, 2010.

[2] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, “Scalable graph exploration on multicore processors,” in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010, pp. 1–11, doi:10.1109/SC.2010.46.

[3] S. Beamer, K. Asanovic, and D. Patterson, “Direction-optimizing breadth-first search,” in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. IEEE, 2012, pp. 1–10, doi:10.1109/SC.2012.50.

[4] F. Checconi, F. Petrini, J. Willcock, A. Lumsdaine, A. R. Choudhury, and Y. Sabharwal, “Breaking the speed and scalability barriers for graph exploration on distributed-memory machines,” in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. IEEE, 2012, pp. 1–12, doi:10.1109/SC.2012.25.

[5] A. Buluç and K. Madduri, “Parallel breadth-first search on distributed memory systems,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 65, doi:10.1145/2063384.2063471.

[6] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core cpu and gpu," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 2011, pp. 78–88, doi:10.1109/PACT.2011.14.

[7] G. Tao, L. Yutong, and S. Guang, "Using mic to accelerate a typical data-intensive application: the breadth-first search," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE, 2013, pp. 1117–1125, doi:10.1109/IPDPSW.2013.197.

[8] T. Suzumura, K. Ueno, H. Sato, K. Fujisawa, and S. Matsuoka, "Performance characteristics of Graph500 on large-scale distributed environment," in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*. IEEE, 2011, pp. 149–158, doi:10.1109/IISWC.2011.6114175.

[9] R. Sedgewick, "Algorithms in C, part 5: Graph algorithms," 2002.

[10] K. Ueno and T. Suzumura, "Highly scalable graph search for the Graph500 benchmark," in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '12, ACM. New York, NY, USA: ACM, 2012, pp. 149–160, doi:10.1145/2287076.2287104.

[11] (2014, Jun.) Graph500 benchmark. [Online]. Available: http://www.graph500.org/

[12] GNU Compiler Collection (gcc). [Online]. Available: http://gcc.gnu.org/

[13] OpenMPI. [Online]. Available: http://www.open-mpi.org/