Scaling the PageRank Algorithm for Very Large Graphs on the Fugaku Supercomputer

Maxence Vandromme[1], Jérôme Gurhem[2], Miwako Tsuji[3], Serge Petiton[1,2], and Mitsuhisa Sato[3]

[1] Univ. Lille, UMR 9189 - CRIStAL, CNRS
F-59000 Lille, France
[2] USR 3441 - Maison de la Simulation, CNRS
Saclay, France
[3] RIKEN Center for Computational Science
Kobe, Japan

Abstract. The PageRank algorithm is a widely used linear algebra method with many applications. As graphs with billions or more of nodes become increasingly common, being able to scale this algorithm on modern HPC architectures is of prime importance. While most existing approaches have explored distributed computing to compute an approximation of the PageRank scores, we focus on the numerical computation using the power iteration method. We develop and implement a distributed parallel version of the PageRank. This application is deployed on the supercomputer Fugaku, using up to one thousand compute nodes to assess scalability on random stochastic matrices. These large-scale experiments show that the network-on-chip of the A64FX processor acts as an additional level of computation, in between nodes and cores.

1 Introduction

The PageRank algorithm was originally developed to rank Web pages by importance, as the main component powering Web search engines [16]. It generally uses the power iteration method to compute the dominant eigenvector of a stochastic matrix efficiently. In this context, the Web pages and the links between them form a graph that can be represented by a sparse adjacency matrix, on which the PageRank is applied. Beyond its historic roots, this algorithm saw widespread use in applications where data is organized in a graph structure, such as citation networks [15] or traffic grids [17]. More recently, the (personalized) PageRank has been used as a tool to weigh communication between nodes in Graph Neural Networks [12].

The most common method to compute the PageRank exactly is the power iteration, which relies on iterative sparse matrix-vector multiplication (SpMV) as its kernel. In this regard, the PageRank bears a resemblance to other linear algebra methods, such as the conjugate gradient, which is notably used in the HPCG benchmark to measure the performance of supercomputers with a focus on memory and interconnections [6, 10]. Indeed, the main component of

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_28

the conjugate gradient is also sequences of sparse matrix-vector multiplication, and such methods have direct applications for real-life problems. However, the HPCG benchmark uses sparse diagonal matrices, that represent the kind of computational problems arising from physical applications. Such matrices are a best-case scenario for distributed and parallel computing, since the data is organized to minimize cache misses. This benchmark is well-suited to measure the peak optimal performance of HPC systems on graph problems.

In this paper, we focus instead on the PageRank algorithm and its application to data closer to what can be found in networks or Web graphs. These graphs are usually much less structured, and as a result, their underlying adjacency matrices stray quite far from the ideal case of a regular diagonal matrix. Given the very large size of Web graphs, we need to efficienctly scale the PageRank algorithm on graphs up to billions of nodes. To this end, we present a parallel and distributed implementation of the PageRank algorithm, we deploy it on the top-end supercomputer Fugaku, and measure the scalability of the algorithm up to very large graph sizes on hundreds of compute nodes.

The rest of this paper is organized as follows:

- in Section 2, we review the relevant existing work on the PageRank algorithm for distributed settings, and put it in perspective with the recent evolution of HPC systems
- in Section 3, we present the implementation of the parallel and distributed PageRank algorithm, and the distributed generation of sparse stochastic matrices
- in Section 4 we detail the experiments settings on the supercomputer Fugaku, and discuss the results and insights gained from the experiments. We analyze the impact of the network-on-chip on the performance by using different MPI configurations
- in Section 5 we summarize the findings and propose further work to do for the future

2 Background and related work

The PageRank was designed with scalability as a primary goal. The original study dealt with a database of 75 million URLs and more than 300 million links between them, and the size has grown to billions of elements or more since then. Nevertheless, the number of iterations of the power method remains small compared to the graph size, and the algorithm usually converges in less than 100 iterations with standard parameters [16].

Despite this efficiency, there has been a growing need to design distributed versions of the algorithm to keep up with the size increase and better take advantage of modern HPC systems with many compute nodes. Applying the standard algorithm in a distributed environment implies splitting the matrix in parts, running partial computation independently on each node, then aggregating and distributing the result vector across all nodes between two iterations of the method. A number of previous studies have noted that the overhead induced

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_28

by the communications between the nodes would be a major problem [11, 18]. Indeed, computational applications on real-world graphs suffer from poor locality, since the graph structure is irregular, and high data access to computation ratio, meaning that the performance will be bounded by the data access speed of the system [14]. On this basis, research in this area mainly focused on distributed implementations that did not require this type of broadcast communications.

Regarding the PageRank, an original study outlined the future research directions by first providing a solid definition of the algorithm [4]. It then pointed out that a fully centralised algorithm on a distributed system would incur significant congestion on the communication network, and that it was not suited for graphs that evolve over time due to the synchronization costs. The authors then proposed several versions of distributed algorithms, relying on the parallel execution of random walks to iteratively approximate the steady state distribution of the dominant eigenvector underlying the PageRank. Later studies expanded on these ideas, and notably likened the random walk to a Markov process in order to reduce the amount of information accumulated from walks and communicated to other nodes [18]. In another study, the asynchronous updating process was extended by considering Web pages as agents of a multi-agents system, updating its PageRank value and passing the information to its outbound links [11]. This method uses randomization on the information communicated. Other usually select nodes randomly, from which a walk is started [3].

A related issue is the Personalized PageRank (PPR), where the goal is not to find the importance of a node i among the whole graph, but the importance of a node i relative to another node j. Full naive computation of the PPR on a graph with n nodes requires running the PageRank algorithm n times, which is irrealistic for large graphs, and implies storing a dense matrix of size $n \times n$, which is a major issue as well. Therefore, methods have been proposed that also use Monte Carlo random walks adapted for the PPR [13]. A study used a *Graph Partition Algorithm* to distribute the graph on different compute nodes in order to minimize the links between the subgraphs, and therefore the communications. Each node would compute its local PageRank on the subgraph, which would then be used to build the final result vector. Interestingly, it could also be applied recursively to a hierarchy of subgraphs, allowing for high scalability [7].

The supercomputer Fugaku [5] is #1 in the TOP500 list at the time of the study. Supercomputers have been shifting towards an increase in the number of compute nodes rather than increasing the nominal performance of each node. As a prime example, Fugaku uses more than 150,000 processors. Therefore, particular attention has been brought to the communication network between the nodes, in order to alleviate the limitations that communications may place on computational performance. It uses A64FX processors at 2.2GHz with 32 GB of HBM2 memory and a memory bandwidth of 1GB/s. Each processor contains 48 compute cores, split into four groups of 12. Each such group is called Core Memory Group (CMG), and has its own L2 cache and memory. The CMGs inside a processor are linked by a network-on-chip that handles the communications between them. The processors are linked by a 6D topology Tofu interconnect [1].

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_28

Given the recent developments on these networks, we want to evaluate whether the communications are still a roadblock to scalability in a highly distributed environment, using the PageRank application as a test case.

Recent work has been done about the scalability of the sparse matrix-vector product on the A64FX processor. This operation is the main kernel of the PageRank power iteration method, so it is of prime importance for our study. These past studies showed great performance on regular diagonal matrices, when using an adequate storage format taking advantage of the SIMD capabilities of the processor [2]. The performance decreases quickly as the nonzero elements deviate from the diagonal, due to the aforementioned cost of irregular data access [8]. These studies used only one or two compute nodes. In this study, we aim to extend the experiments using many compute nodes, in order to identify the barriers to scalability for large size problems. We also focus on sequences of SpMV rather than SpMV alone, since these are more directly useful for applications.

3 Parallel and distributed implementation

In this Section, we describe the PageRank algorithm in a distributed parallel computing environment, and the sparse matrix data storage formats used.

3.1 PageRank

Regular algorithm We focus here on matrices extracted from graphs, where graphs are objects $G = (V, E)$ where a set of vertices V are connected by a set of edges E. A graph can be represented by its adjacency matrix $M \in \mathbb{R}^{n \times n}$, where n is the number of vertices (or nodes) in the graph. In such cases, M is a binary matrix, i.e. all of its elements m_{ij} are either 0 or 1. Given the shape of large graphs, their adjacency matrices are usually very sparse; that is, the number of non-zero values is much smaller than the total number of elements n^2, and usually of the order of n.

The PageRank algorithm outputs a unique score for each node of the graph, based on the graph structure, corresponding to the dominant eigenvector of the normalized adjacency matrix. A higher score means that the node is more important in the graph. The most common way to compute this vector exactly is through the power iteration method, which is essentially a sparse matrix-vector product on the column-normalized transpose of the adjacency matrix, repeated until the result varies by less than a threshold ϵ from one iteration to the next. During each iteration $t + 1$, the SpMV operation is performed on the vector b_t computed in the previous step t, with the initial b_0 being a vector of ones. b_{t+1} is modified to add an uniform probability of teleportation to any node of the graph, using the β parameter, and then normalized. The PageRank algorithm can be seen as a random walk over the graph. In this context, the teleportation corresponds to the probability, at each step, to restart the random walk at another node of the graph. This mechanism is mainly used to avoid getting stuck in sink nodes, i.e. nodes without outbound edges.

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_28

Distributed implementation The iterated sparse matrix-vector multiplication (SpMV) is the most computationally expensive part of the algorithm, therefore an efficient distributed implementation is required to scale on many compute nodes of modern HPC systems.

We implement the application in C++ with MPI and OpenMP. Each MPI process computes the operation on a part of the matrix, then the results are aggregated and shared between processes at the end of each iteration. OpenMP is used on each process to parallelize the computations for increased efficiency.

3.2 Sparse matrices

Since each MPI process performs the computations only on a part of the matrix, we need to split the data matrix so that each process has access to the part it uses. More precisely, the matrix is generated directly in a distributed manner, with each process creating and storing its own block of the matrix based on its process rank.

The block distribution consists of both a distribution by rows and a distribution by columns. The $Nc \times Nr$ matrix is split in $Ngc \times Ngr$ sub-matrices. The sub-matrices are stored in a sparse storage format locally. In this case, the input vector is split across the columns of the matrix and the sub-vectors are duplicated on the sub-rows of the same column. The resulting vector has the same size as the number of rows in the sub-matrices of the corresponding row. However, each computing resource contains a part of a sub-vector. To obtain the global result, all the distributed results of the same row have to be summed then each row has to be gathered if the full result vector is needed in one place.

We use three standard storage formats for sparse matrices: CSR, ELLPACK, and SCOO [9]. The computation of the SpMV for these three formats is detailed in the Algorithms 1, 2 and 3 below. CSR and ELLPACK are standard formats used in many applications and frameworks. SCOO is a variant of COO where the matrix is split into blocks, then each block is stored in a COO format. It allows for better locality than COO in distributed settings, and therefore better performance on operations.

4 Experiments

In this Section, we first detail the parameters of the experiments performed to study the scalability of the distributed PageRank algorithm on the supercomputer Fugaku. We present the computing environment and some specificities of the processors. Then, we describe the matrices used for the experiments and present the results.

4.1 Parameters

SVE implementation A major feature of the recent ARM-based processors, including the A64FX, is the Scalable Vector Extension (SVE) that enables support for SIMD operations with per-lane prediction, which allows for efficient

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_28

Algorithm 1: CSR format data structure and matrix vector product. *idx* is the vector of indexes for the start of each row (of size $n + 1$). *col* and *val* are the vectors of columns and values (of size *nnz* each). *fr* (*fc*) is the index of the first row (column) of the block of data stored on this process, in the context of the whole data matrix

```
Function spmv_csr()
    Data: m : MatrixCSR, v : Vector
    Result: r : Vector
    for i ← 0 to m.idx.size() − 1 do
        for j ← m.idx[i] to m.idx[i + 1] − 1 do
            r[i] += m.val[j] * v[m.col[j] - m.fc]
```

Algorithm 2: ELL format data structure and matrix vector product. *col* and *val* are the vectors of columns and values (of size $n \times max\_col$ each)

```
Function spmv_ell()
    Data: m : MatrixELL, v : Vector
    Result: r : Vector
    for i ← 0 to m.lrs − 1 do
        for j ← 0 to m.max_col − 1 do
            r[i + m.rpos] += m.val[i * m.max_col + j] * v[m.col[i * m.max_col
             + j]- m.fc]
```

Algorithm 3: SCOO format data structure and matrix vector product. *row*, *col* and *val* are the vectors of rows, columns and values (of size *nnz* each)

```
Function spmv_scoo()
    Data: m : MatrixSCOO, v : Vector
    Result: r : Vector
    for i ← 0 to m.val.size() − 1 do
        r[m.row[i] - m.fr] += m.val[i] * v[m.col[i] - m.fc]
```

vectorization [19]. The vector length can be specified as a multiple of 128 bits. We use a vector of 512 bits, which is the default for the processor and allows for the simultaneous computation of 8 double-precision values (64 bits each). The SVE instructions can be generated automatically by the compiler for simple functions. However, the compilers supporting the SVE instructions sometimes fail to vectorize the loops in the SpMV since they require indirect access to store arrays. Instead, we used the ARM SVE intrinsic functions to implement a vectorized version of the SpMV on different matrix formats. [4]

[4] Code is available at https://github.com/jgurhem/TBSLA/tree/dev_array

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_28

Computing environment The code is implemented in C++ with MPI and OpenMP. The program is compiled using the Fujitsu compiler in Clang mode, with flags '-Kopenmp -fPIC -Ofast -mcpu=native -funroll-loops -fno-builtin -march=armv8.2-a+sve'.

Input matrices The PageRank takes as input a stochastic matrix, representing the normalized transpose of the adjacency matrix of a graph, and outputs a single result vector. As described in Section 3, each (MPI) process generates a subpart of the matrix, i.e. a block of rows and columns. This block of data is only accessed by the related process, which handles the computation on this part of the matrix. The matrix is therefore never stored in full in a single data structure.

We choose to build the adjacency matrix of graphs where each node has nnz edges linking to other nodes, with these nodes being chosen randomly with uniform distribution (forbidding duplicate edges to ensure a constant number of edges per node). This way, we have a matrix where data access is very irregular, which is often the case in graph-based applications [14]. One could argue that graphs also usually have nodes with different degrees, and regions more or less densely connected. We choose to ignore this parameter here and assign a fixed number of edges originating from each node. This is done in order to simplify the observations and not introduce uncertainty due to the load balancing issues.

4.2 Results

Weak scaling on the matrix density Since we want to evaluate how the PageRank algorithm scales on Fugaku, it makes sense to perform weak scaling experiments, where we increase the size of the problem along with the amount of computing resources. This also allows us to work on very large matrices, which would not be possible with strong scaling experiments since the instances would have to fit on one single processor in this case. There are two possibilities to increase the problem size: either a) increase the size n of the matrix, or b) increase the density of the matrix by increasing the number of nonzero elements per row. These two can be combined, but since the goal of these weak scaling experiments is to keep the same load per computing resource, it is simpler to do so by only changing either the size n or the density nnz. In this study, we choose the second option and increase the density of the matrix along with the number of compute nodes, while keeping the same matrix size. The first option (increasing the matrix size) implies further limitations that will be discussed at the end of this Section.

As a baseline, we consider a (square) matrix of size $4,000,000 \times 4,000,000$ with either 50 or 100 nonzero elements per row. Therefore, when using 16 compute nodes, the size of the matrix remains $4,000,000 \times 4,000,000$, but the number of nonzero elements per row goes up to 800 and 1600 respectively. As explained in a previous section, the sparse matrix is split in blocks and distributed on the compute nodes. Thus, using additional compute nodes for a matrix of the same

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_28

size means splitting the matrix in increasingly small blocks. However, since the number of nonzero elements per row increases accordingly, the load per process remains constant. Since the complexity of the PageRank algorithm (using the power iteration method) depends primarily on this number of nonzeroes, this allows us to correctly assess the scalability using this experimental process.

This size was chosen to fit within the memory limits of one processor for all three sparse matrix storage formats. We use two different base densities (50 and 100) to get more insight on how this parameter affects the performance of the application. We increase the number of compute nodes progressively from 1 to 1024, doubling each time. Thus, in the largest case with 1024 compute nodes, the matrix contains $4M$ rows with 102400 nonzero elements each, i.e. more than $400B$ nonzeroes in total. For a given number of processes p, we choose to arrange them on a grid of $p/2$ (horizontally) by 2 (vertically). For example, with 16 processes, the matrix is split into 8 chunks horizontally and 2 chunks vertically, resulting in a sub-matrix of size $500,000$ by $2,000,000$ on each process. Different grid configurations have different impacts on the runtime, but the study of this behavior is outside the range of our present work.

In addition, we experiment on two different MPI/OpenMP configurations:

- 1 MPI process per compute node: each process uses 48 threads, one per processor core. The communications inside the processor are therefore handled by OpenMP, and the communications between nodes by MPI.
- 1 MPI per CMG: each process uses 12 threads, and there are 4 MPI processes per processor. With this setup, the communications between the 4 CMGs inside one processor are also handled by MPI.

The results in Tables 1 and 2 show that in absolute terms, the PageRank only takes a few seconds even for large instances.

nodes	CSR		ELL		SCOO	
	node	CMG	node	CMG	node	CMG
1	1.89	0.91	1.30	0.91	5.19	2.17
2	2.17	0.76	1.41	0.79	4.24	2.01
4	1.98	0.69	1.33	0.71	3.28	1.84
8	1.58	0.54	1.02	0.55	2.57	1.47
16	1.39	0.47	0.98	0.48	2.24	1.28
32	1.39	0.46	0.88	0.54	2.25	1.33
64	1.40	0.46	0.93	0.47	2.24	1.32
128	1.20	0.43	1.22	0.40	1.89	1.10
256	1.00	0.36	1.02	0.35	1.56	0.95
512	1.00	0.35	1.00	0.35	1.13	0.84
1024	1.00	0.37	1.01	0.37	1.16	0.86

Table 1: Median runtime for the PageRank, scaling the nnz per row, from a base of $nnz = 50$

nodes	CSR		ELL		SCOO	
	node	CMG	node	CMG	node	CMG
1	5.90	1.28	2.59	1.30	5.59	3.53
2	3.92	1.15	2.46	1.19	4.55	3.24
4	3.14	0.90	1.89	0.92	4.37	2.59
8	2.75	0.78	1.70	0.79	3.77	2.27
16	2.78	0.77	1.69	0.81	3.83	2.27
32	2.77	0.77	1.83	0.81	3.81	2.31
64	2.38	0.67	2.20	0.70	3.26	1.99
128	1.98	0.56	2.00	0.58	2.68	1.63
256	1.99	0.56	2.00	0.56	2.68	1.64
512	1.96	0.56	1.96	0.57	2.26	1.55
1024	1.98	0.59	1.97	0.59	2.28	1.58

Table 2: Median runtime for the PageRank, scaling the nnz per row, from a base of $nnz = 100$

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_28

However, since the convergence of the power iteration method depends on the input matrix, the number of iterations varies from one configuration to another. Therefore, the runtime itself may not be best for assessing the scalability of the application. For more rigorous comparison, we report the median performance in GFlop/s corresponding to the weak scaling results of the above two tables. Figures 1 and 2 show the results with $nnz = 50$ elements per row as basis for matrix density (scaling with the number of compute nodes), using one MPI process per compute node and one MPI process per CMG, respectively.

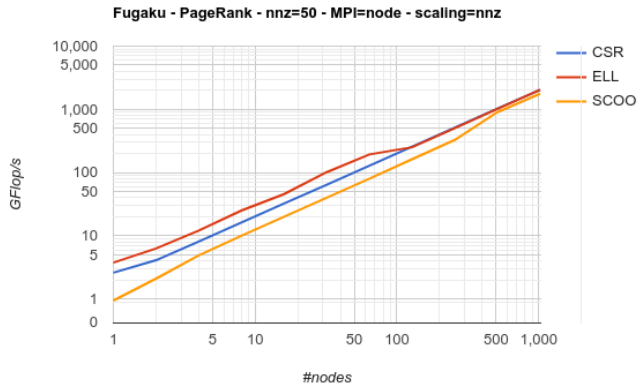

Fig. 1: Median performance for the PageRank, scaling the number of nonzero elements, from a base of $nnz = 50$, with 1 MPI per node

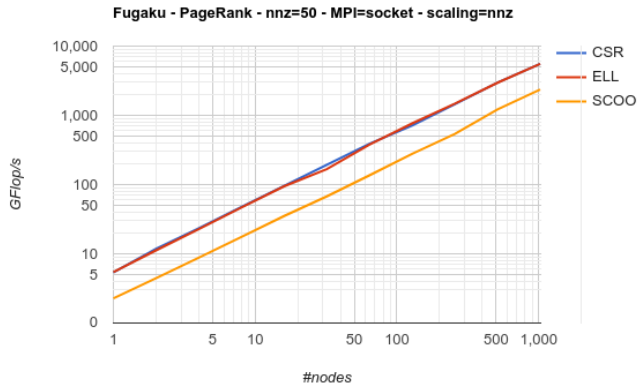

Fig. 2: Median performance for the PageRank, scaling the number of nonzero elements, from a base of $nnz = 50$, with 1 MPI per CMG

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_28

Figures 3 and 4 show the results with $nnz = 100$ elements per row as basis for matrix density, using one MPI process per compute node and one MPI process per CMG, respectively.

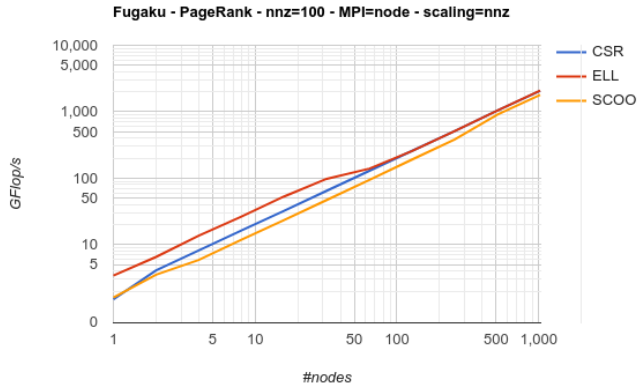

Fig. 3: Median performance for the PageRank, scaling the number of nonzero elements, from a base of $nnz = 100$, with 1 MPI per node

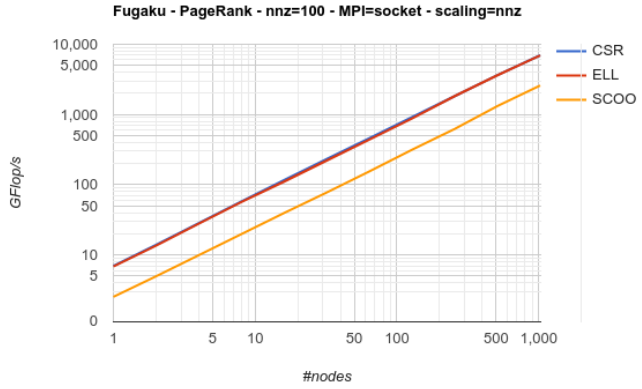

Fig. 4: Median performance for the PageRank, scaling the number of nonzero elements, from a base of $nnz = 100$, with 1 MPI per CMG

Memory usage Compared to other supercomputers, Fugaku uses processors with limited memory. Whereas an A64FX processor has 32GB of HBM2 memory, only 28GB of allocable RAM can be used, which poses constraints on the size of the data contained in each one. In Figure 5, we show the memory used per

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_28

node as the number of nodes increases (using the larger case with base density $nnz = 100$). The patterns are similar with the other MPI configuration (1 MPI process per CMG).

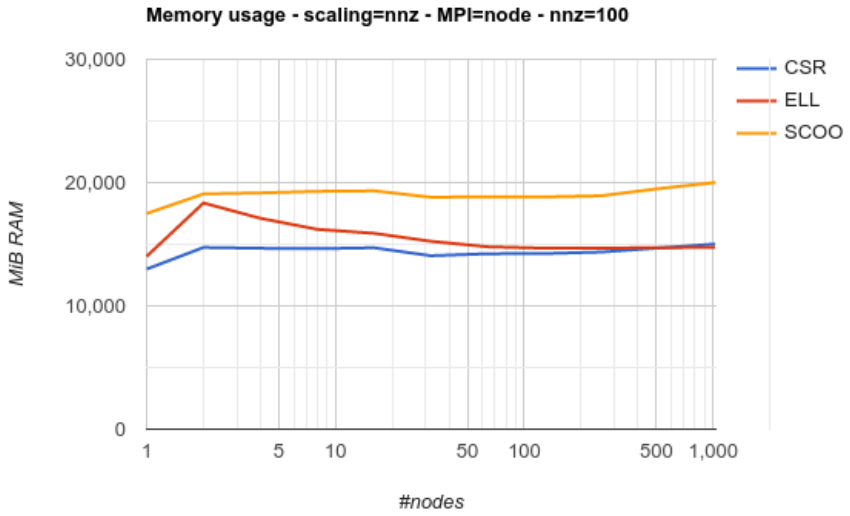

Fig. 5: Memory usage for the PageRank, scaling the number of nonzero elements, from a base of $nnz = 100$, with 1 MPI process per node

4.3 Discussion

Weak scaling results The distributed PageRank shows excellent scalability from 1 to 1024 compute nodes, reaching more than 5TFlop/s on the largest experiments. The runtime per iteration of the power method remains about constant, giving linear speedup on the performance with the number of nodes used, with no signs of faltering. The performance patterns are similar for the two base matrix densities ($nnz = 50$ and $nnz = 100$), although the numbers are higher for the larger case. There are noticeable difference in performance depending on the sparse matrix storage format used, with SCOO performing overall worse than both CSR and ELL. Still, we can observe good scalability with all of these formats.

Another interesting point is the difference between the two MPI configurations; that is, either using 1 MPI process per node, or 1 MPI process per CMG. This parameter has an impact on two aspects. First, using one MPI process per CMG significantly increases performance, as can be observed by comparing Figure 2 to Figure 1, and Figure 4 to Figure 3. The performance overall is about doubled when using this configuration of 1 MPI per CMG. This indicates

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_28

that using MPI for communications between the CMGs of a same processor is more efficient than using a pattern of shared memory across the 4 CMGs and using OpenMP for intra-processor communications. Second, the MPI configuration changes the relative behaviors of the different storage formats. When using 1 MPI per node (Figures 1 and 3), the differences between the storage formats tend to diminish as the number of compute nodes increases. On the contrary, with 1 MPI per CMG (Figures 2 and 4), there is a larger gap between SCOO on one hand, and CSR and ELL on the other, with these last two being nearly equal.

Memory usage and roadblocks Figure 5 show the memory usage per node of our application, scaling with the number of nodes. We can see that this usage remains nearly constant, and below the limit of 28GB of allocable RAM per node.

An analysis of the space complexity of our implementation of the PageRank algorithm show that it uses two vectors of n_col elements to store the iterated result of the computation, and two vectors of ln_row elements (all in double-precision), in addition to the matrix storage itself. n_col is the number of columns of the full data matrix, and ln_row is the number of rows in the local matrix block for this process.

In the case of the experiments presented here, where we increase the density of the matrix, the full matrix size remains constant, and the local size may even decrease depending on how the matrix is split between processes. Therefore, the constant memory usage for both the matrix storage and the computation, observed in these two Figures, is normal.

However, these two vectors of size n_col are an issue when scaling on the matrix size, as mentioned previously. At the start of each iteration, each process needs to have the full result vector from the previous iteration in order to perform the computation. At some point, the memory required to store this full result vector on each process becomes larger even than the memory used to store the sparse matrix block. This problem is magnified when using more than 1 MPI process per node, since each process has its copy of the result vector. In our preliminary experiments, we hit the memory limit of 28GB with 256 nodes when using 1 MPI per node, and with 64 nodes when using 1 MPI per CMG (i.e. 4 MPI per node). This amounts to vectors of more than 1 billion double-precision elements. Consequently, scaling for sparse matrices representing graphs of billions of nodes would require a different implementation of the PageRank algorithm, which does not require the full result vector to be stored on each process.

5 Conclusion

We have presented an implementation of the PageRank algorithm that uses a distributed and parallel sparse matrix vector product as its kernel, in order to scale the exact computation of the PageRank to very large data sets. We have

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_28

performed weak scaling experiments for this application on the supercomputer Fugaku, increasing the density of a sparse stochastic matrix with the number of compute nodes. The experiments used up to 1024 compute nodes (49152 compute cores), for matrices with hundreds of billions of nonzero elements. The matrix was generated in a distributed setting, with the nonzero elements chosen randomly in order to avoid any structure, and to present the worst possible case for data access patterns. We observed linear scalability for this PageRank algorithm in this context. We compared two MPI configurations and found that using MPI for communications between CMGs on the A64FX processor leads to noticeable improvements in performance, compared to using one MPI per compute node. Whereas the HPCG benchmark uses diagonal matrices, we studied irregular matrices, which are more representative of real-life matrices such as Web graphs or networks commonly used as input for the PageRank algorithm. On these very large irregular matrices, the network-on-chip therefore induces a different programming paradigm, with the communications between CMG being paramount to the performance. The sparse matrix storage formats also exhibit different performance patterns depending on the MPI configuration. Further analysis at a lower level, and profiling of the MPI communications, would be required to investigate these differences in more detail, as well as the discrepancies observed between the different sparse matrix storage formats. Besides that, the main perspective would be to design a PageRank implementation that does not require to store the full result vector on each process, which leads to memory issues when scaling on very large matrices (beyond 1 billion in size). Such an implementation would likely use additional MPI communications to gather the necessary input at each iteration. Therefore, an analysis of the performance tradeoff in this case would be insightful for future applications.

References

1. Ajima, Y., Kawashima, T., Okamoto, T., Shida, N., Hirai, K., Shimizu, T., Hiramoto, S., Ikeda, Y., Yoshikawa, T., Uchida, K., Inoue, T.: The tofu interconnect d. In: 2018 IEEE International Conference on Cluster Computing (CLUSTER). pp. 646–654 (2018). https://doi.org/10.1109/CLUSTER.2018.00090
2. Alappat, C.L., Laukemann, J., Gruber, T., Hager, G., Wellein, G., Meyer, N., Wettig, T.: Performance Modeling of Streaming Kernels and Sparse Matrix-Vector Multiplication on A64FX. CoRR **abs/2009.13903** (2020), https://arxiv.org/abs/2009.13903
3. Dai, L., Freris, N.M.: Fully distributed pagerank computation with exponential convergence. arXiv preprint arXiv:1705.09927 (2017)
4. De Jager, D.: Pagerank: Three distributed algorithms. Master's thesis, Imperial College London, London, pubs. doc. ic. ac. uk/pagerank-algorithms (2004)
5. Dongarra, J.: Report on the Fujitsu Fugaku system. University of Tennessee-Knoxville Innovative Computing Laboratory, Tech. Rep. ICLUT-20-06 (2020)
6. Dongarra, J., Heroux, M.A., Luszczek, P.: High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. The International Journal of High Performance Computing Applications **30**(1), 3–10 (2016)

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_28

7. Guo, T., Cao, X., Cong, G., Lu, J., Lin, X.: Distributed algorithms on exact personalized pagerank. In: Proceedings of the 2017 ACM International Conference on Management of Data. pp. 479–494 (2017)
8. Gurhem, J., Vandromme, M., Tsuji, M., Petiton, S.G., Sato, M.: Sequences of sparse matrix-vector multiplication on fugaku's a64fx processors. In: 2021 IEEE International Conference on Cluster Computing (CLUSTER). pp. 751–758. IEEE (2021)
9. Hugues, M.R., Petiton, S.G.: Sparse matrix formats evaluation and optimization on a gpu. In: 2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC). pp. 122–129. IEEE (2010)
10. Ihde, N., Marten, P., Eleliemy, A., Poerwawinata, G., Silva, P., Tolovski, I., Ciorba, F.M., Rabl, T.: A survey of big data, high performance computing, and machine learning benchmarks
11. Ishii, H., Tempo, R., Bai, E.W.: A web aggregation approach for distributed randomized pagerank algorithms. IEEE Transactions on automatic control **57**(11), 2703–2717 (2012)
12. Klicpera, J., Bojchevski, A., Günnemann, S.: Predict then propagate: Graph neural networks meet personalized pagerank. arXiv preprint arXiv:1810.05997 (2018)
13. Lin, W.: Distributed algorithms for fully personalized pagerank on large graphs. In: The World Wide Web Conference. pp. 1084–1094 (2019)
14. Lumsdaine, A., Gregor, D., Hendrickson, B., Berry, J.: Challenges in parallel graph processing. Parallel Processing Letters **17**(01), 5–20 (2007)
15. Ma, N., Guan, J., Zhao, Y.: Bringing pagerank to the citation analysis. Information Processing & Management **44**(2), 800–810 (2008)
16. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: Bringing order to the web. Tech. rep., Stanford InfoLab (1999)
17. Pop, F., Dobre, C.: An efficient pagerank approach for urban traffic optimization. Mathematical Problems in Engineering **2012** (2012)
18. Sarma, A.D., Molla, A.R., Pandurangan, G., Upfal, E.: Fast distributed pagerank computation. In: International Conference on Distributed Computing and Networking. pp. 11–26. Springer (2013)
19. Sato, M., Ishikawa, Y., Tomita, H., Kodama, Y., Odajima, T., Tsuji, M., Yashiro, H., Aoki, M., Shida, N., Miyoshi, I., Hirai, K., Furuya, A., Asato, A., Morita, K., Shimizu, T.: Co-design for A64FX Manycore Processor and "Fugaku". In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–15 (2020). https://doi.org/10.1109/SC41405.2020.00051

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_28