Overhead of a Decentralized Gossip Algorithm on the Performance of HPC Applications

Ely Levy, Amnon Barak, Amnon Shiloh
Department of Computer Science
The Hebrew University of Jerusalem, Israel
elylevy@cs.huji.ac.il

Matthias Lieber, Carsten Weinhold, Hermann Härtig
Department of Computer Science
TU Dresden, Germany

ABSTRACT

Gossip algorithms can provide online information about the availability and the state of the resources in supercomputers. These algorithms require minimal computing and storage capabilities at each node and when properly tuned, they are not expected to overload the nodes or the network that connects these nodes. These properties make gossip interesting for future exascale systems. This paper examines the overhead of a decentralized gossip algorithm on the performance of parallel MPI applications running on up to 8192 nodes of an IBM BlueGene/Q supercomputer. The applications that were used in the experiments include PTRANS and MPI-FFT from the HPCC benchmark suite as well as the coupled weather and cloud simulation model COSMO-SPECS+FD4. In most cases, no gossip overhead was observed when the gossip messages were sent at intervals of 256 ms or more. As expected, the overhead that is observed at higher rates is sensitive to the communication pattern of the application and the amount of gossip information being circulated.

Keywords

Benchmarking, cluster management, gossip algorithm, high performance computing

1. INTRODUCTION

Management of scalable high performance compute clusters requires frequent monitoring and sharing of information about the resources of all nodes, e. g., availability, current load, free memory, and temperature. Gossip algorithms can provide such information by routinely disseminating relevant information among the nodes. These algorithms are tolerant to node failures or lost messages, which is a critically important property for large-scale systems built from hundreds of thousands to millions of components. Clearly, the rate and the amount of disseminated information should be tuned in order to provide accurate information about each node and at the same time not reduce the performance of the applications running on the system.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the authors must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ROSS '14, June 10, 2014, Munich, Germany
Copyright is held by the authors. Publication rights licensed to ACM.
ACM 978-1-4503-2950-7/14/06 ...$15.00.
http://dx.doi.org/10.1145/2612262.2612271

This paper examines the overhead of a decentralized gossip algorithm on the performance of parallel applications on supercomputers. We ran three applications using configurations ranging from 1024 to 8192 nodes on an IBM BlueGene/Q system. In each case we compare the runtime of the application with different gossip rates to the runtime of the application without gossip in the background. To the best of our knowledge, nobody has done yet any measurements of the overhead caused by gossip messages sharing the network used by HPC applications.

The specific gossip algorithm used was developed by Amar et al. [1] for sharing load balancing information among the nodes of scalable clusters. Briefly, in this algorithm each node maintains a snapshot about the state of the resources in other nodes and routinely disseminates a fixed amount of its most recently acquired information. This way, any system service in need of up-to-date information about the state of the resources in the cluster can directly obtain it locally.

The paper is organized as follows: We first discuss the background and related work for gossip-based information dissemination. Section 3 provides an overview of our gossip algorithm. Section 4 describes the hardware configurations and the parallel applications that were used for testing. Section 5 presents the benchmark results. Our conclusions are given in Section 6.

2. RELATED WORK

Randomized gossip algorithms are distributed algorithms for exchanging information through various types of network topologies. Their simplicity, low overhead and fault-tolerance make them attractive for many use cases such as membership management in peer-to-peer networks [2, 3], application data exchange in linear algebra [4] or computation of aggregate functions [5]. They are also used in sensor networks [6, 7] and for resource management in large clusters [8] and clouds [9].

The MOSIX system [10] combines gossip-based information dissemination and optimization algorithms to provide load balancing in UNIX clusters. The MOSIX nodes run daemon processes that exchange with each other information about resources (e. g., available nodes, free memory, CPU load). MOSIX exploits this information to improve the overall performance of the cluster and of individual applications by adaptive allocation and migration of processes among nodes.

Gossip Algorithm:

At a fixed point during each unit of time, **each** node:

- Updates its own entry in the locally stored *vector* with the current state of the local resources and sets the age of this information to 0;
- For the remaining vector entries, updates the current age to the age at arrival plus the time passed since;
- Immediately sends a fixed-size *window* with the most recent *vector* entries to another node, which is chosen randomly with a uniform distribution.

When a node receives a *window*, it:

- Registers the *window's* arrival time in all the received entries using the local clock;
- Updates each of its *vector's* entries with the corresponding *window* entry, if the latter is newer.

Figure 1: The gossip algorithm with fixed window sizes.

Technologies such as MOSIX are known to perform well for UNIX clusters. However, the overhead caused by MOSIX-like gossip algorithms on large-scale HPC machines is not well understood, as these systems are much more susceptible to network jitter. Menon and Kalé evaluated the performance of GrapevineLB [11], a load balancer exploiting gossip algorithms on top of the Charm++ runtime system [12]. Their paper showed that the overall performance is improved substantially, but they do not discuss the overhead caused by gossip-related messages being exchanged among the nodes. Soltero et. al. evaluated the suitability of gossip-based information dissemination for system services of exascale clusters [13]. Their simulations showed that good accuracy can be achieved for power management services with up to a million nodes. However, experiments using their prototype were emulating only 1000 nodes and did not include measurements of gossip-related network overhead on the applications.

Bhatele et al. [14] identify the contention for shared network resources between jobs as the primary reason for runtime variability of batch jobs in a large Cray system. On BlueGene systems, however, each job is assigned a private contiguous partition of the torus network, so that contention is avoided. In our measurements, we combined two applications (a gossip program and an application benchmark) in a single batch job on a BlueGene/Q system, such that network contention becomes a critical concern. We then measured the slowdown of the application due to the gossip activities.

3. THE GOSSIP ALGORITHM

Consider a cluster with a large number of active nodes. Assume that each node regularly monitors the state of its relevant resources and also maintains an information *vector* with *entries* about the state of the resources in all the other nodes. Each such vector entry includes the state of the resources of the corresponding node and the age of that information. The gossip algorithm disseminates this information among nodes.

The algorithm that is used in this paper was developed in [1]. Figure 1 shows the pseudo code. Briefly, in this algorithm, every unit of time, each node monitors the state of its resources and records it in its vector entry. Each of the nodes then exchanges a *window* containing a fixed amount of the newest information in its vector with another randomly chosen node. Thus, each node receives, on average, in every unit of time information about other nodes and each of them eventually learns about the state of all nodes. Note that the nodes are not synchronized, i. e. all the nodes use the same unit of time but run independently using their own local clocks. One relevant parameter for the algorithm's performance is the size of the window, i. e., the amount of information sent by each node. Another parameter that is studied in this paper is the unit of time, which determines the rate of the information dissemination.

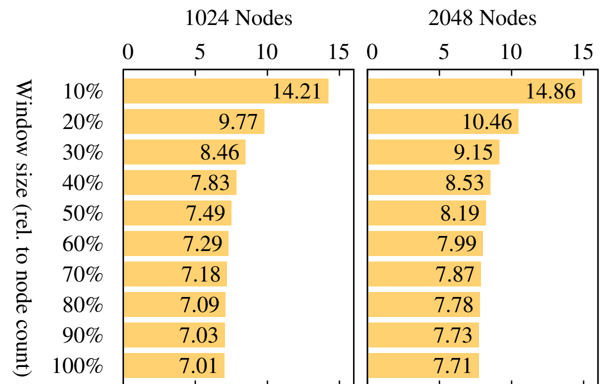

Figure 2: Average vector age (relative to the unit of time) for window sizes ranging from 10 % to 100 % of the number of nodes.

4. BENCHMARK SETUP

In a preliminary study, we measured the average age of the vector vs. the size of the circulated window for different cluster sizes. The results are depicted in Figure 2 for 1024 and 2048 nodes. Configurations with 4096 and 8192 nodes show similar behavior. From the figure it can be seen that the steepest decrease in the average age of the vector is when increasing the window size from 10 % to 20 %, whereas larger windows provide only marginal benefit at the cost of transmitting significantly more data. As we will show in Section 5.2, circulating larger gossip messages causes higher overhead than increasing the gossip rate. We therefore decided to run all experiments with a window size of 20 % of the vector size. In our experiments, we set the vector size equal to the total number of nodes, but it could also be smaller in order to save memory on the nodes. For example, MOSIX uses fixed-size vectors that are twice as large as the windows and independent of the cluster size.

4.1 BlueGene/Q Hardware

We performed measurements on the IBM BlueGene/Q system JUQUEEN installed at Jülich Supercomputing Centre, Germany, which is ranked number 8 in the November 2013 Top500 list of the largest supercomputers. The

JUQUEEN system has 28 672 nodes, each equipped with one 16-core PowerPC A2 1.6 GHz processor, resulting in a total of 458 752 cores. The 5D torus network has a peak bandwidth of 2 GB/s per link, which can send and receive at that rate simultaneously [15]. Since each node has 10 links, this leads to a theoretical peak bandwidth of 40 GB/s per node. The worst-case latency for messages in the torus network is 2.6 μs. Due to the large number of nodes and the highly scalable interconnect, the system is well suited to benchmark algorithms for future exascale systems.

4.2 Gossip Implementation

To run the gossip algorithm and measure various metrics, like age of the information and overhead on applications, we ported our gossip algorithm to the BlueGene environment. The implementation uses MPI to exchange gossip messages between the nodes. In general, gossiping does not require a reliable network and messages are sent without checking or waiting for successful transmission. We use MPI_Bsend to implement these datagram-like semantics. Whenever this MPI-based gossip implementation does not update its vector, send a message, or process a received message, it is busy waiting for a message from other nodes using MPI_Iprobe. Busy waiting is interrupted as soon as one unit of time has passed and the node has to send a gossip message. For practical application of gossip algorithms we expect OS support to avoid busy waiting, such that no compute time and energy is wasted. The percentage of runtime not spent (busy) waiting for gossip messages is thus a good indicator for the computational overhead of the gossip algorithm.

To allow the gossip implementation and MPI applications to share the same node, we linked both code bases into a single executable. This approach required minimal changes to the application, namely the HPC code's main program needs to be renamed such that it is called from the gossip program's main function. The wrapper functionality inside the gossip program configures the MPI communicator for the application such that it sees only a subset of all MPI ranks. MPI_COMM_WORLD references in the application are intercepted by wrapping MPI calls. We use an existing wrapper generator [16] for that task. For our measurements on JUQUEEN, we split MPI_COMM_WORLD such that every 16th rank runs gossip and all others run the application, i. e. each node runs one gossip process and 15 application processes. For the comparison measurements without gossip we used the same method, except that the process allocated to gossip is now idle, i. e. waiting for the benchmark to finish. In both configurations – with and without gossip – the application operates on the subset of the real MPI_COMM_WORLD of the job.

In the overhead measurements, we used the following parameters for the gossip algorithm:

- Number of nodes (vector entries): 1024, 2048, 4096, 8192.
- Size of each vector entry: 1 kB.
- Window size: 20 % of the vector size (i. e., 204, 409, 819, 1638).
- Unit of time (gossip interval): 1 ms – 1024 ms.

Note that the size of a gossip message is the window size times the size of the vector entry plus some space for the age of the information. For example with 1024 nodes, the message size is 212 164 bytes. In this case, when running with a gossip interval of 1 ms, each gossip process sends messages at a rate of 212 MB/s and receives messages at the same average rate. Neglecting message headers, this corresponds to 1.06 % of the peak bandwidth of a BlueGene/Q node.

4.3 Application Benchmarks

For the overhead measurements we used two communication-intensive MPI applications from the HPC Challenge (HPCC) benchmark suite [17, 18] and another highly scalable HPC application.

HPCC PTRANS. In the PTRANS benchmark, pairs of MPI ranks communicate with each other in order to transpose a matrix. All ranks exchange messages simultaneously, thereby utilizing the full capacity of the interconnect. The performance of PTRANS is reported in GB/s. We used a matrix size of N = 524 288 (2048 GiB), a block size of NB = 112, and the following process grids (P × Q): 32 × 480 for 1024 nodes, 32 × 960 for 2048 nodes, 64 × 960 for 4096 nodes, and 128 × 960 for 8192 nodes.

HPCC MPI-FFT. The MPI-FFT benchmark performs a parallel one-dimensional discrete Fourier transform of a double precision complex vector. This benchmark has an all-to-all communication pattern that heavily stresses the bisection bandwidth of the interconnect with large messages. We used the FFTE implementation that is included in the HPCC package and a vector length of 136 million elements (2025 GiB) for 1024 nodes and 544 million elements (8100 GiB) for larger configurations.

COSMO-SPECS+FD4. COSMO-SPECS+FD4 [19] is an atmospheric simulation model (COSMO) with detailed cloud microphysics (SPECS). It uses dynamic load balancing and model coupling techniques (FD4) to balance the workload of the microphysics model. For dynamic load balancing FD4 calculates a new SPECS partitioning every time step using space-filling curve partitioning. COSMO-SPECS+FD4 has different communication phases:

- Ghost exchange in COSMO (static, regular).
- Ghost exchange in SPECS (dynamic, irregular).
- Model coupling (dynamic, irregular, relatively small volume).
- Parallel partitioning calculation (mainly a mixture of collectives, also on sub-communicators).
- Migration as result of load balancing (highly local, mostly between neighbor ranks).

COSMO-SPECS+FD4 has been tuned for high scalability [20]; the application scales very well on 262 144 processes on JUQUEEN. We used weak scaling for our overhead benchmarks, i. e. the problem size of the simulation is proportional to the number of MPI ranks. The benchmark test case simulates the growth of a cumulus cloud, which is replicated along the two horizontal dimensions to scale the problem to arbitrary sizes. We ran 30 time steps of the simulation, which is sufficient to expose overhead due to gossip and report the net runtime, i. e., without program start-up and initialization.

5. RESULTS

In a fully integrated system, the gossip algorithm and the management services built on top would run on a dedicated service core in order to minimize execution time jitter for the application. However, in the BlueGene/Q system we used for our experiments, the service core is not available to users. We therefore ran the gossip algorithm on one of the 16 user cores of each node, while the benchmark application used the remaining 15 cores. For each application, we repeated the tests twice and calculated the average outcome. The variation between the repetitions was very low.

5.1 Application Overhead Results

The results of the overhead measurements for the three applications are shown in Figures 3, 4, and 5. In each figure, the green bar at the top shows the performance without the gossip algorithm running in background. The core usually allocated for the gossip algorithm was left idle. The blue bars below show the performance of the application for gossip intervals ranging from 1024 ms to 1 ms. Due to scalability limits, measurements with very small intervals and large number of nodes failed, refer to Section 5.3 for more details. For MPI-FFT and COSMO-SPECS+FD4, we included the runtime portion of MPI communication as red bars in the graphs. For each interval, the overhead is the difference between the obtained performance and the performance of the application without gossip.

From the figures it can be seen that in most cases, no gossip overhead was observed when using intervals higher than 256 ms. The only exception is the network sensitive MPI-FFT when running on 8192 nodes. Observe that a larger cluster size implies higher gossip overhead, due to the increase in the window size. Specifically, the maximal slowdown was 5.6 % for PTRANS, 51.6 % for MPI-FFT, and 5.5 % for COSMO-SPECS+FD4.

PTRANS Slowdown. Despite being a pure communication benchmark, PTRANS shows a moderate slowdown only, even in the worst-case. Note that no gossip overhead was noticeable when the gossip interval was equal or larger than 256 ms. The runtime of PTRANS is dominated by MPI_Sendrecv, which is used to exchange messages between ranks in a sparse pattern, determined by the parameters of the benchmark. These measurements indicate that sparse communication patterns show only a minor sensibility to network contention.

MPI-FFT Slowdown. MPI-FFT shows the highest slowdown among the tested applications. This benchmark is also the most communication intensive test; the runtime is dominated by three MPI_Alltoall collective communication calls. In each call every rank sends Ns_c/P^2 bytes to every other rank [21], where N is the vector length, s_c is the size of a complex double precision number (16 bytes), and P is the number of ranks. For example in the 8192 node configuration, every rank sends 576 bytes to every other rank, which accumulates to 1012 MiB that each BlueGene/Q node has to send and receive at the same time per MPI_Alltoall call. This dense pattern of large messages stresses the bisection bandwidth and makes this benchmark very sensible to network contention. In addition to the high MPI ratio, Figure 4 also shows that the communication part of MPI-FFT scales much worse with the number of nodes compared to the computation part. This amplifies the overhead of gossiping on the total performance of the benchmark even further.

	Slowdown for window size of			
Interval	**10 %**	**20 %**	**40 %**	**80 %**
64 ms	0.2 %	0.3 %	0.6 %	1.5 %
32 ms	0.3 %	0.4 %	0.9 %	2.6 %
16 ms	0.3 %	0.7 %	1.8 %	4.5 %
8 ms	0.7 %	1.4 %	3.2 %	8.7 %
4 ms	1.1 %	2.6 %	6.3 %	17.2 %
2 ms	1.8 %	4.7 %	–	–
1 ms	3.8 %	–	–	–

Table 1: Runtime overhead of the MPI-FFT benchmark depending on gossip rate and window size. Increasing window size usually causes more overhead than increasing the gossip rate by the same factor.

COSMO-SPECS+FD4 Slowdown. Figure 5 shows that the MPI communication contributes the substantial part of the slowdown in COSMO-SPECS+FD4. Note that runs with 2048 nodes and above show much better network performance for the ghost exchange in COSMO, which reduces overall communication time greatly. This effect is independent from background gossip.

By comparing the runtime of the five different communication phases of the application (see Section 4.3) and by analyzing the various MPI calls, we identified the last step of the parallel partitioning calculation as the dominating contributor to the slowdown. In this step, the partition vector, containing the start indices of all partitions, is distributed to all MPI ranks using MPI collectives on sub-communicators. For the highest slowdown observed at 4096 nodes with a gossip interval of 4 ms, the average runtime of this single step increases from 15 ms to 67 ms. Since load balancing is carried out after each of the 30 time steps, this accumulates to a total runtime increase of approx. 1.5 s.

The computation time in COSMO-SPECS+FD4 increases lightly due to gossip; in the worst case 0.5 % slowdown are observed. This can be attributed to the load the gossip process puts on the shared resources within a node, like memory bus and L2 cache. For example at 8192 nodes, each gossip process manages a state vector of approx. 8 MiB. Generating and processing gossip messages are memory-bound operations on this vector, which increases contention for shared memory resources.

The benchmarks show that gossip intervals between 256 and 1024 ms do not cause noticeable overhead for any of the workload configurations. This result is in line with existing systems such as MOSIX [10], which uses a gossip interval of 1 second to implement distributed load balancing.

5.2 Window Size vs. Gossip Interval

The results for MPI-FFT shown in Figure 4 indicate that the relative overhead caused by different gossip rates also depends on the number of nodes that exchange gossip messages. This behavior is especially apparent for the 4096 and 8192 node configurations. We performed additional experiments where we increased the gossip rate and the window size independently from each other while keeping the number of nodes constant. Table 1 shows the measured overheads

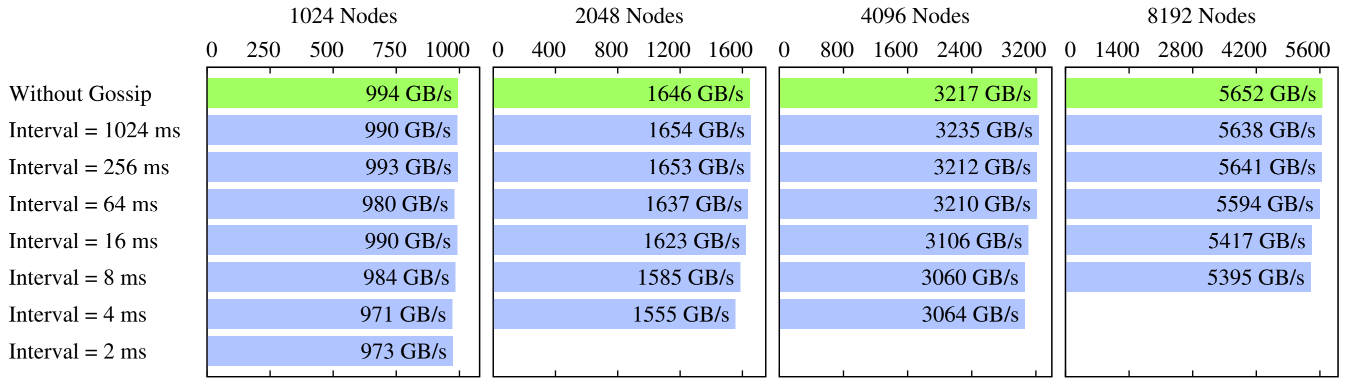

Figure 3: PTRANS performance in GB/s (higher is better).

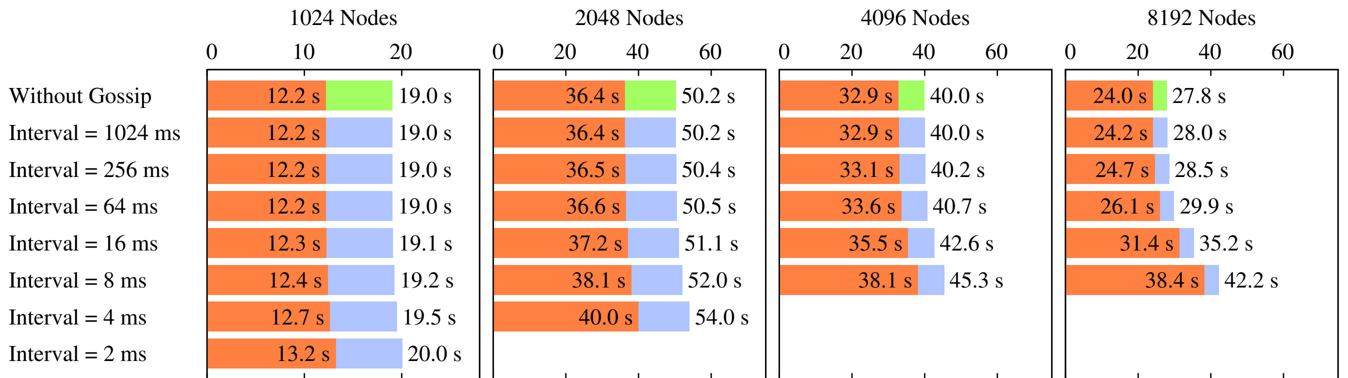

Figure 4: MPI-FFT runtime (lower is better). Inner red part indicates the MPI portion.

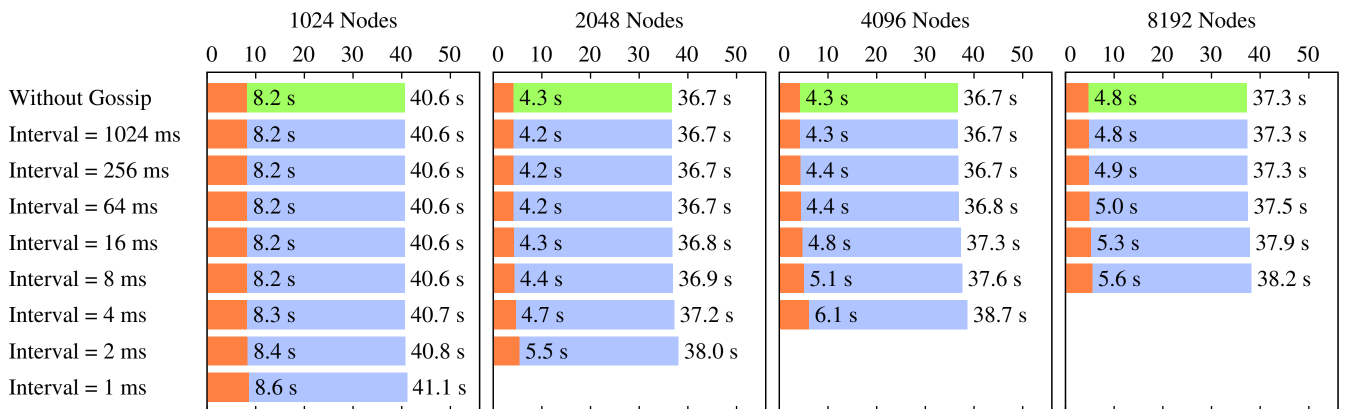

Figure 5: COSMO-SPECS+FD4 runtime (lower is better). Inner red part indicates the MPI portion.

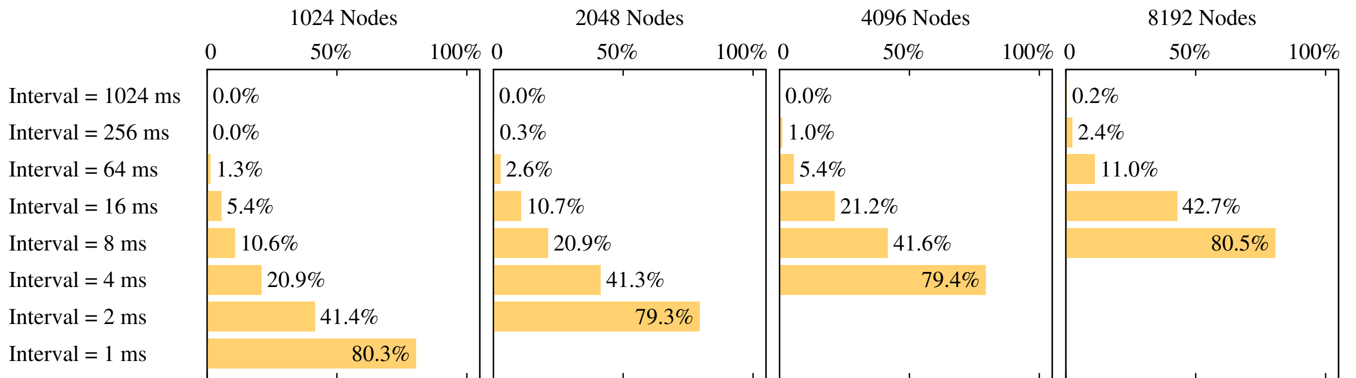

Figure 6: Runtime percentages of the gossip processes being busy generating and processing messages while running the COSMO-SPECS+FD4 overhead measurements. The rest of the time the gossip process is waiting for incoming messages.

for MPI-FFT on 1024 nodes. One can observe that increasing either the rate by a factor f or increasing the window size by the same factor f will result in different overheads for the application. This result might be unexpected, as the amount of gossip data that is circulated in a certain amount of time is the same. Further analysis is required to understand this behavior.

5.3 Scalability of Gossip

In addition to overhead caused by sharing the network, we also investigated the runtime and scalability of the gossip process implementation. This process puts load on the CPU whenever it generates and sends a gossip message and whenever it receives and processes a message from another node. The rest of the time it is waiting for messages, which we implemented as busy wait for simplicity. In practical applications, the load on the gossip processes should be minimized, such that they are waiting most of the time to process received messages immediately, without delay.

Figure 6 shows the percentages of time in which the gossip processes are not idle, while performing the overhead measurements for COSMO-SPECS+FD4. Observe the linear relationship between the gossip workload, the gossip interval and the number of nodes (which directly influences the window size). These measurements reached the scalability limits of the gossip implementation. For example with 8192 nodes, we were not able run with a gossip rate below 4 ms. In these cases, the gossip processes could not keep up with the high rate of incoming messages, leading to overflow of the send buffers and abort of the benchmark. Note that processing a received message (i. e., merging the received window with the own vector) consumes more CPU time than generating and sending a vector. These results show the importance of a highly optimized implementation of the gossip algorithm, preferably avoiding linear relationship of workload and window size, to be applicable in exascale systems.

6. CONCLUSIONS AND OUTLOOK

We presented a randomized gossip algorithm that is designed for load management and monitoring services on large-scale HPC systems. The algorithm is fault-tolerant and suitable for disseminating information among thousands of nodes. We developed an MPI-based implementation of a gossip algorithm that mimics the network usage of such a system service and measured its overhead on the performance of applications that ran on the same set of nodes. Our experiments on a BlueGene/Q system showed, that in most cases, no noticeable overhead is observed for gossip intervals of 256 ms and higher when running on 8192 nodes. Only the communication-bound MPI-FFT benchmark showed a small overhead of 2.3 %, which drops to less than 0.6 % for 4096 nodes and less. We found that increasing the gossip rate has a greater impact on the performance of communication-bound applications, when the number of gossiping nodes increases. This effect is caused by larger message sizes, as information about more nodes is transmitted across the network in each gossip round. By analyzing the communication patterns used in the applications, we identified collective MPI communication to be much more sensitive to slowdown due to network contention compared to point-to-point messages.

We investigated gossip overhead in a large-scale BlueGene/Q system. We will expand our measurements to other systems to study different types of HPC interconnects, like InfiniBand and Cray Aries. Furthermore, the work presented in this paper is now being extended [22] to be suitable for exascale systems, which are expected to consist of hundreds of thousands of nodes. For systems of this size, it is necessary to optimize the performance of the gossip implementation and to use a hierarchical approach to gossiping. Evaluating the hierarchical version of our algorithm is subject to future work.

7. ACKNOWLEDGEMENTS

This research and the work presented in this paper is supported by the German priority program 1648 ‘Software for Exascale Computing’ via the research project FFMK [22] and by the cluster of excellence ‘Center for Advancing Electronics Dresden’ (*cfaed*). The authors also want to thank the Jülich Supercomputing Centre, Germany, for access to the JUQUEEN supercomputer.

8. REFERENCES

[1] L. Amar, A. Barak, Z. Drezner, and M. Okun. *Randomized Gossip Algorithms for Maintaining a Distributed Bulletin Board with Guaranteed Age Properties.* Concurrency and Computation: Practice and Experience, January 2009.

[2] A.J. Ganesh, A.-M. Kermarrec, and L. Massoulie. Peer-to-peer membership management for gossip-based protocols. *Computers, IEEE Transactions on*, 52(2):139–149, 2003.

[3] F.M. Cuenca-Acuna, C. Peery, R.P. Martin, and T.D. Nguyen. PlanetP: using gossiping to build content addressable peer-to-peer information sharing communities. In *Proc. 12th Intl. Symp. on High Performance Distributed Computing*, pages 236–246. IEEE, 2003.

[4] Hana Straková, Wilfried N. Gansterer, and Thomas Zemen. Distributed QR Factorization Based on Randomized Algorithms. In *Parallel Processing and Applied Mathematics*, volume 7203 of *LNCS*, pages 235–244. Springer, 2012.

[5] D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *Proc. Symp. on Foundations of Computer Science*, pages 482–491, 2003.

[6] A.D.G. Dimakis, A.D. Sarwate, and M.J. Wainwright. Geographic Gossip: Efficient Averaging for Sensor Networks. *IEEE Trans. Signal Processing*, 56(3):1205–1216, 2008.

[7] P. Kyasanur, R.R. Choudhury, and I. Gupta. Smart Gossip: An Adaptive Gossip-based Broadcasting Service for Sensor Networks. In *Proc. Mobile Adhoc and Sensor Systems*, pages 91–100. IEEE, 2006.

[8] A Barak and A Shiloh. The MOSIX Cluster Operating System for Distributed Computing on Linux Clusters, Multi-Clusters and Clouds. White paper, `http://www.mosix.org`, 2014.

[9] F. Wuhib, R. Stadler, and M. Spreitzer. A Gossip Protocol for Dynamic Resource Management in Large Cloud Environments. *IEEE Trans. Network and Service Management*, 9(2):213–225, 2012.

[10] A. Barak, S. Guday, and R. Wheeler. *The MOSIX Distributed Operating System Load Balancing for UNIX*, volume 672 of *LNCS*. 1993.

[11] Harshitha Menon and Laxmikant Kalé. A Distributed Dynamic Load Balancer for Iterative Applications. In *Proc. SC '13*. ACM, 2013.

[12] Laxmikant V. Kalé and Sanjeev Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[13] Philip Soltero, Patrick Bridges, Dorian Arnold, and Michael Lang. A Gossip-based Approach to Exascale System Services. In *Proc. 3rd Intl. Workshop on Runtime and Operating Systems for Supercomputers (ROSS '13)*. ACM, 2013.

[14] Abhinav Bhatele, Kathryn Mohror, Steven H. Langer, and Katherine E. Isaacs. There Goes the Neighborhood: Performance Degradation Due to Nearby Jobs. In *Proc. SC '13*. ACM, 2013.

[15] Dong Chen, N.A. Eisley, P. Heidelberger, R.M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D.L. Satterfield, B. Steinmacher-Burow, and J.J. Parker. The IBM Blue Gene/Q interconnection network and message unit. In *Proc. SC '11*. ACM, 2011.

[16] Todd Gamblin. PMPI wrapper generator, 2013. `https://github.com/tgamblin/wrap`.

[17] HPC Challenge Benchmark Suite. `http://icl.cs.utk.edu/hpcc/`.

[18] P. Luszczek, D. Bailey, J. Dongarra, J. Kepner, R. Lucas, R. Rabenseifner, and D. Takahashi. The HPC Challenge (HPCC) Benchmark Suite. In *SC '06 Conference Tutorials*, 2006.

[19] Matthias Lieber, Verena Grützun, Ralf Wolke, Matthias S. Müller, and Wolfgang E. Nagel. Highly Scalable Dynamic Load Balancing in the Atmospheric Modeling System COSMO-SPECS+FD4. In *Proc. PARA 2010*, volume 7133 of *LNCS*, pages 131–141. Springer, 2012.

[20] Matthias Lieber, Wolfgang E. Nagel, and Hartmut Mix. Scalability Tuning of the Load Balancing and Coupling Framework FD4. In *NIC Symposium 2014*, volume 47 of *NIC Series*, pages 363–370, 2014.

[21] Franz Franchetti, Yevgen Voronenko, and Gheorghe Almasi. Automatic Generation of the HPC Challenge's Global FFT Benchmark for BlueGene/P. In *VECPAR 2012*, volume 7851 of *LNCS*, pages 187–200. Springer, 2013.

[22] FFMK Website. `http://ffmk.tudos.org`.