Evaluation and Optimization of Breadth-First Search on NUMA Cluster

Zehan Cui[1,2], Licheng Chen[1,2], Mingyu Chen[1], Yungang Bao[1], Yongbing Huang[1,2], Huiwei Lv[1,2]
[1] State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
[2] Graduate School of Chinese Academy of Sciences
{cuizehan, chenlicheng, cmy, baoyg, huangyongbing, lvhuiwei}@ict.ac.cn

Abstract— **Graph is widely used in many areas. Breadth-First Search (BFS), a key subroutine for many graph analysis algorithms, has become the primary benchmark for Graph500 ranking. Due to the high communication cost of BFS, multi-socket nodes with large memory capacity (NUMA) are supposed to reduce network pressure. However, the longer latency to remote memory may cause problem if not treated well. In this work, we first demonstrate that simply spawning and binding one MPI process for each socket can achieve the best performance for MPI/OpenMP hybrid programmed BFS algorithm, resulting in 1.53X of performance on 16 nodes. Nevertheless, we notice that one MPI process per socket may exacerbate the communication cost. We propose to share some communication data structure among the processes inside the same node, to eliminate most of the intra-node communication. To fully utilize the network bandwidth, we make all the processes in a node to perform communication simultaneously. We further adjust the granularity of a key bitmap for better cache locality to speed up the computation. With all the optimizations for NUMA, communication and computation together, 2.44X of performance is achieved on 16 nodes, which is 39.2 Billion Traversed Edges per Second for an R-MAT graph of scale 32 (4 billion vertices and 64 billion edges).**

Keywords-Graph; BFS; NUMA; MPI/OpenMP; Allgather

I. INTRODUCTION

Exploration of large-scale graphs is commonly used in many areas, such as bioinformatics, astrophysics, data mining, social network analysis, and so on. Breadth-First Search (BFS) is a key building block for many graph analysis algorithms, such as finding spanning tree, shortest path, connected component and max flow. The recently announced Graph500 ranking [1] as opposite to Top500, is aimed to rank computers according to their capability of processing data-intensive tasks. BFS has been chosen as the primary benchmark for Graph500.

Due to the integration of memory controller into microprocessor chip for reducing memory access latency as well as the high-speed interconnects such as HT and QPI connecting multiple processors, non-uniform memory access (NUMA) architecture has been the dominated computer architecture in server machines, where local memory is directly attached to each processor instead of shared bus and can be accessed through cross-chip interconnects from other processors. For example, the newly announced processors such as Intel Xeon 7500 / E7-8800 series (Nehalem-EX/ Westmere-EX) begin to support up to eight sockets in a single node without the help of third-party node controller.

The BFS of large-scale graph has unique features compared with previous high performance computing applications. It suffers little locality in all hierarchies – cache, local memory and global distributed memory. The partition of a graph is still a very difficult problem. So far no partition methods can radically cut down the communication cost, which is a dominated issue for BFS. A promising way is to use fewer nodes to lighten the communication traffics, with each node of higher memory capacity and capable of processing larger part of graph. NUMA node with 4 or more CPUs can satisfy such need for memory and processing capacity.

Though NUMA can provide a much powerful node with more than one processor - more cores and memory, it can be a significant problem for applications due to the congestion on cross-chip interconnects, long latency and potential bandwidth saturation of remote memory accesses [10, 28-30, 32]. For example, it is shown in [32] that the performance of a multi-threaded program when running on 4 cores (1 socket) and 8 cores (2 sockets) are almost the same. More cores do not result in better performance due to the NUMA effect.

A number of studies have been done for BFS on cluster system [11, 40, 48], but few of them have taken the effect of NUMA architecture of cluster nodes into account. Other studies [7, 9, 19, 47] try to maximize the parallelism of BFS on multi-core platform, which can be adopted as the intra-node scheme in cluster system. An approach has been proposed in [7] to avoid random memory accesses and atomic memory updates across sockets by modifying the algorithm, but its performance on cluster system has not been evaluated.

On the other hand, parallel BFS algorithms for cluster are always implemented as MPI/OpenMP hybrid programmed programs [40] to utilize the hierarchical hardware. How to map such hybrid programmed applications to NUMA cluster node has been studied in [36, 44], but BFS has not yet been studied in this scope. In [36], the authors analyzed different mapping methods but didn't give an universal optimal method, for it is related to both the machine topology and application behavior. In [44], using MPI across sockets and OpenMP within sockets is recommended through evaluation of two NPB-MZ benchmarks [45] and a real application.

In this paper, we investigate how to achieve better performance for BFS on a multi-socket NUMA cluster. We choose a BFS algorithm, which we believe to be highly efficient, and optimize the algorithm through analyzing both computation and communication characteristic on NUMA architecture. We first demonstrate that the performance of BFS on an 8-socket NUMA node can be improved by up to

978-0-7695-4807-4/12 $26.00 © 2012 IEEE
DOI 10.1109/CLUSTER.2012.29

74.4% by spawning one process per socket instead of one process per node to exploit computing power. Nevertheless, simply spawning one process per socket exacerbates the cost of collective communication, especially when the number of sockets in a node is large. To solve this problem, we take advantage of the read-only nature of some data structure to share them among the multiple processes inside a node, and employ a scheme to perform collective communication in parallel to fully utilize the network bandwidth. Furthermore, we find that choosing an appropriate granularity for a frequently accessed bitmap structure according to its sparsity can improve the algorithm's performance. By utilizing a relative small system – 16 NUMA nodes (1,024 cores), we achieve 39.2 billion traversed edges per second (TEPS) for an R-MAT graph with 4 billion vertices and 64 billion edges, which ranked No. 16 in the Graph500 list of June 2012. Our main contributions are:

- We demonstrate that simply spawning one process per socket to reduce cross-socket memory accesses can improve performance for BFS by about 74.4% for single node and 52.5% for 16 NUMA nodes, compared to one process per node with uniformly data distribution.
- We propose two approaches to optimize collective communication for NUMA cluster, sharing communication buffer and parallelizing allgather, which can boost the performance by 69.0% more for 16 nodes.
- By selecting an appropriate granularity for a key bitmap, we can further improve the performance by 22.6% to 39.2 billion traversed edges per second for 16 nodes.
- Using these methods overall 2.44X performance is achieved. We believe these approaches can be migrated to other applications with similar characteristic.

The following of paper is organized as below: Section II introduces the problems of BFS and NUMA optimization; our optimizations for both communication and computation are presented in Section III; Section IV performs detailed evaluation; some related works are reviewed in Section V. Section VI gives a conclusion.

II. Background and Motivation

A. BFS Algorithms

A graph $G(V,E)$ is composed of a set of vertexes V and a set of edges E. Given a particular source vertex $r \in V$, BFS explores the graph G level by level to discover all the vertices that can be reached from vertex r, and generates a BFS tree rooted at vertex r. The set of vertices reached in a level is called frontier.

In each level, the current frontier is explored to find all adjacent and unreached vertices, which form the next frontier. The exploration of frontier can be performed in two ways: 1) the top-down approach – for each vertex in the current frontier, its adjacent vertices are checked and unvisited ones are put into the next frontier; 2) the bottom-up approach – for each unvisited vertex in the graph, it is put into the next frontier only if at least one of its adjacent vertices is in the current frontier.

The two approaches have their respective advantages, which has been well examined in [9]: the top-down approach performs well when the frontier is small, while the bottom-up approach is more efficient when the frontier is large.

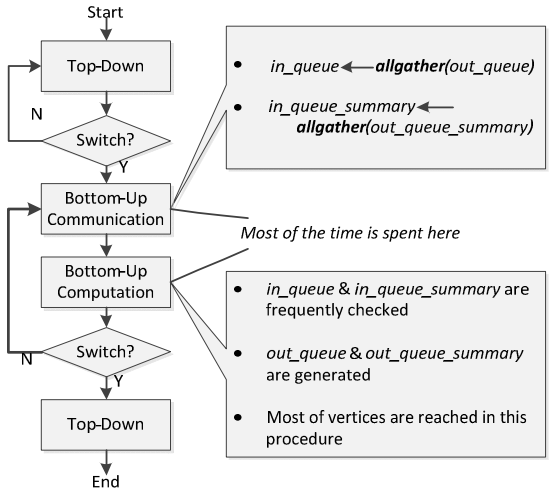


Figure 1. Overview of a parallel implementation of *hybrid* BFS algorithm. *in_queue* is the current frontier, *out_queue* is the next frontier, and both are bitmaps for all vertices. Each MPI process owns an copy of *in_queue*, but only part of *out_queue*, so all processes need to perform *allgather* to construct the next frontier. The **_summary* are the bitmaps for *in_queue* and *out_queue* respectively.

Utilizing the complementation of the two approaches, a *hybrid* approach is proposed in [9], which uses top-down approach to explore the frontier when it is small, and uses bottom-up approach when the frontier is large. This is considered to be the most efficient algorithm of BFS on multicore platform. Our experiment result on a 64-core platform shows that the *hybrid* approach (8 MPI processes, each of 8 OMP threads) is 27.3 times faster than the top-down approach (pure MPI, 64 MPI processes) and 4.7 times faster than the bottom-up approach (8 MPI processes, each of 8 OMP threads) using the evaluation method of Graph500.

The Graph500 reference codes [1] contain several parallel implementations of BFS for distributed-memory clusters, which has been evaluated in [40]. The *mpi_simple* version is an implementation of the top-down approach, while the *mpi_replicated* version is an implementation of the bottom-up approach. To perform BFS in parallel on a distributed-memory cluster, the entire graph is partitioned into *np* parts, where *np* is the number of MPI processes; each MPI process searches one part and communicates with other processes when needed.

Based on the Graph500 reference codes [1], we implement the *hybrid* BFS algorithm [9] for distributed-memory clusters, by adding a data conversation procedure and a switching function, and using hybrid MPI/OpenMP programming.

The graph is generated using R-MAT algorithm [13], the distribution of which is scale-free. So the frontier of each level first ramps up and then down exponentially [9], which results in a three-phase BFS procedure: first top-down, then bottom-up, and finally top-down. Fig. 1 illustrates the procedure of our parallel implementation of the *hybrid* BFS algorithm.

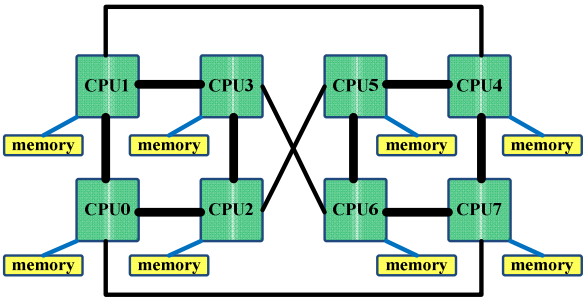


Figure 2. Topology of an eight-socket node. Eight Intel Xeon X7550 CPUs are connected through QPI.

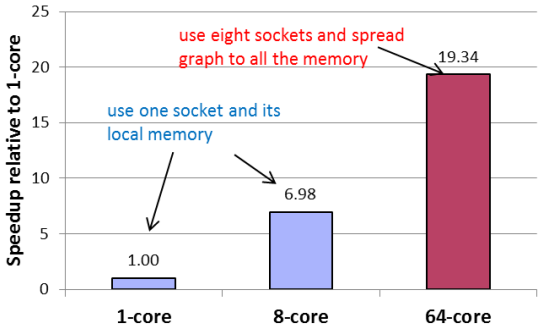


Figure 3. Speedup of BFS algorithm when running on 1 core, 8 cores and 64 cores. One thread per core is used in all three cases.

B. Characteristics of hybrid BFS algorithm

Fig. 1 gives an overview of *hybrid* algorithms: most of vertices are reached in the bottom-up procedure [9], which consumes most of the time, as illustrated in Fig. 11. So this work only focuses on the bottom-up procedure, the characteristics of which are listed as follow: (for detail information about *hybrid* algorithm and implementation of bottom-up procedure, please refer to [9] and [40] respectively.)

1) Separated Computation and Communication Phases

It can be clearly seen from Fig. 1 that computation and communication phases in bottom-up procedure are separated. In the computation phase, the access to the vertices and edges are randomly across large data set and little data is reused, resulting in low locality of both spatial and temporal.

The communication phase is quite simple - two ***allgather*** operations are performed. The first ***allgather*** for *in_queue* consumes most of the time, for the size of *in_queue* is 64 times of *in_queue_summary*.

2) in_queue_summary can speed up the check operation of in_queue

in_queue is the bitmap for all vertices (local and remote). The corresponding bit of a vertex in *in_queue* is 1 means this vertex is visited in last level. For each vertex traversed in the computation phase, its corresponding bit in *in_queue* is checked [40].

in_queue_summary, which is the bitmap for *in_queue*, is used to speed up the check operations. Generally, one bit in *in_queue_summary* represents 64 bits of *in_queue* as in the Graph500 reference codes, which means only when the contiguous 64 bits of *in_queue* are all 0, the corresponding bit of *in_queue_summary* will be 0.

In general, *in_queue_summary* has much better cache locality than *in_queue* due to its much smaller size, so if we check both of them simultaneously, the result of *in_queue_summary* will return faster – if it is 0, there is no need to wait for the result of *in_queue*, which will definitely be 0. The check operations are speeded up in this way. If the scale of graph is 32 (2^{32} vertices), the size of *in_queue* and *in_queue_summary* are 512 MB and 8 MB respectively.

3) in_queue & in_queue_summary are Read-only and Identical among all processes in Computation Phase

In the parallel implementation, each MPI process has its own copy of *in_queue* and *in_queue_summary*. These two data structures are frequently checked (read-only) in the computation phase and updated by the ***allgather*** operations in the communication phase. More clearly, all processes will have the same data after ***allgather*** operations, and will not change them until next ***allgather***.

C. NUMA Architecture and its influence on BFS

NUMA architecture is becoming the dominating node architecture in today's commodity cluster. Each CPU has its own attached memory and can access other CPUs' memory through high-speed interconnects, such as QPI and HT. The memory accesses of such system are non-uniform – local memory accesses are of much lower latency and higher bandwidth. With the new high-end CPUs such as Nehalem-EX/Westmere-EX, systems with up to eight sockets can be achieved. Fig. 2 is an example of eight-socket node.

NUMA has provided much larger capacity of shared memory or even shared cache, but remote memory accesses suffer much longer latency and lower bandwidth than local memory accesses, which, if not properly handled, may lead to serious performance reduction [10, 28-30, 32].

For BFS, large portion of memory accesses are to the graph, which may go into the remote memory and seriously degrade the performance. For example, if we use multi-thread on the platform shown in Fig. 2 and spread the graph across the eight sockets, each thread may randomly access the entire graph, which means about 7/8 of the accesses will go into remote memory through QPI. Fig. 3 shows that when all accesses are uniform to local memory, 8 cores can improve the performance by 6.98 times compared to 1 core; however, when NUMA effect is present, 64 cores can only improve the performance by 2.77 times compared to 8 cores.

D. A Method To Better Use NUMA – Pros and Cons

For MPI/OpenMP hybrid programmed application, the authors in [44] has recommended to use MPI across sockets and OpenMP within the socket. The method can be achieved by spawning one MPI process per socket, and binding them to each socket. The binding to socket operation is supported by both Open MPI [3] and MVAPICH2 [2] by using *"-bind-to-socket -bysocket"* and *"-genv MV2_ENABLE_AFFINITY 1"* flags respectively, or shell script in [44].

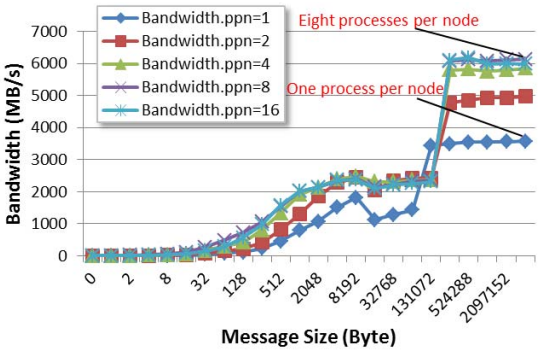


Figure 4. Communication bandwidth between two nodes (dual Infiniband ports each). More processes per node communicating simultanuously results in higher bandwidth. *ppn* is short for "number of processes per node".

1) Pros: eliminate cross-socket accesses in computation phase and fully utilize inter-node network bandwidth in communication phase

This scheme results in better locality since all threads only access their local memory and remote accesses only happen during the MPI communication.

On the other hand, multiple MPI processes performing inter-node communication can take advantage of the bi-directional link bandwidth, especially when multiple network ports exist. Fig. 4 shows the achieved bandwidth between two nodes using OSU micro-benchmark [4] when different number of MPI processes perform inter-node communication. It shows that when eight processes per node are simultaneously participating in communication, the highest bandwidth is achieved, while one process per node can only utilize about half.

2) Cons: multiple processes per node will exacerbate the cost of communication

The cost of communication will increase due to increased number of processes – multiple processes per node will introduce extra intra-node communication. Though intra-node communication through shared memory is faster than inter-node, the communication time spent within nodes may take an unexpectedly high percentage [23]. Some work have been done to optimize intra-node communication, such as improving cache efficiency [12, 21], topology-aware process affinity [27] and overlapping with inter-node communication [26, 31].

In the following, we take ***allgather*** for example to estimate its impact on collective communication. Given the total data size is *m* bytes, and the number of MPI processes is *np*, then each process owns *m/np* bytes of data. The total amount of data transmitted during ***allgather*** is

$$(m/np)*(np-1)*np = m*(np-1) \quad (1)$$

where *(np-1)* means each process need to receive data from the other *(np-1)* processes; *np* means all the *np* processes need to receive the same amount of data.

If the total data size does not change, the total amount of data transmitted is about proportional to the number of spawned processes. If one MPI process is spawned per socket instead of per node for system in Fig. 2, the amount of data transmitted will increase by 8 times.

3) Apply to BFS

The *hybrid* BFS algorithm presented in Fig. 1 can fit this method well for two reasons: 1) the graph is intrinsically partitioned to all processes, so each MPI process will only access its local graph inside the socket; 2) the data transmitted through MPI across sockets, which are bitmaps for vertices, are much smaller than the graph.

The computation phase of *hybrid* BFS algorithm will benefit from the restriction of accesses of graph to the local memory. If we apply the method to the example in Fig. 3, speedup of 6.31X can be achieved instead of 2.77X on 64 cores relative to 8 cores. Our evaluation in Section IV has demonstrated that spawning one MPI process per socket and binding it to socket can improve performance of *hybrid* algorithm for 74.4% on single node, compared to the best result of one MPI process per node.

On the other hand, the *hybrid* BFS algorithm will also suffer from the increasing communication overhead of the two ***allgather*** shown in Fig. 1. Our evaluation in Section V has shown that when 8 nodes are in use, spawning one process per socket results in 2.34 times of execution time in each bottom-up communication phase, compared to one process per node. This leads to that 54% of total time is spent on bottom-up communication.

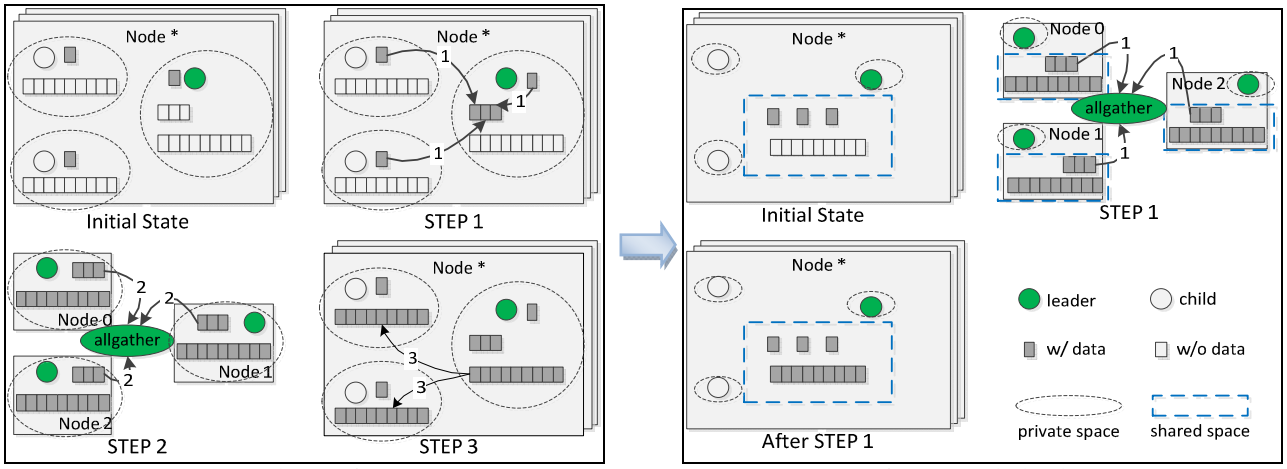
Our goal is to refine the method to maintain its benefits for computation phase and minimize the cost of the ***allgather*** in Fig. 1, by taking advantage of certain characteristic of the *hybrid* BFS algorithm.

III. OPTIMIZATIONS

In this section, we explain several optimizations we have done for both communication and computation: the first optimization will achieve our goal proposed in Section II by sharing some data structure among processes inside a node; the second optimization can further improve the performance of communication by fully utilizing the network bandwidth; the third optimization choose an appropriate granularity for the bitmap *in_queue_summary* to maximize the speedup of computation.

A. Sharing Communication Data among Processes

Prior work has done some optimization for ***allgather*** when there are multiple processes in one node. For example, leader-based approach [31] selects one process per node as a leader and the other processes inside the same node are children, the ***allgather*** is divided into 3 steps: 1) aggregate data to leader process; 2) perform allgather between leaders; 3) distribute data to child processes. Fig. 5a illustrate the mechanism of this approach, which can eliminate extra memory copies between different channel buffers and allow overlap of intra- and inter- node communication [31]. We implement this approach by using ***gather*** from children to leader for step 1, ***allgather*** between leaders for step 2, and ***broadcast*** from leader to children for step 3. Fig. 6 shows the time spent on each step, which is normalized to the default implementation of ***allgather*** in Open MPI 1.5.5.



(a) Prior work (b) Our optimization

Figure 5. Prior work and our optimization for allgather. (a) Prior work: leader-based allgather. Initial state, one process owns one part of the data; STEP 1, aggregate data to leader process; STEP 2, perform *allgather* between leaders; STEP 3, distribute data to child processes. **(b) Our optimization:** share data among processes inside the same node**.** Initial state, one process conrresponds to only one part of the data in the shared space; STEP 1, the leaders perform *allgather* for the data in the shared space; after step 1, all processes can see and directly use the result from the shared space.

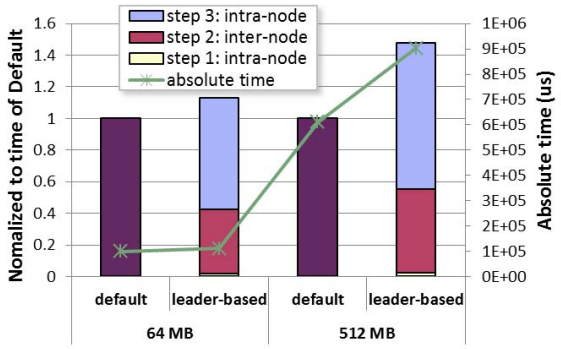


Figure 6. Excuation time of default implementation in Open MPI 1.5.5 and leader-based implementation for *allgather*. 16 eight-socket nodes and totally 128 processes are used to *allgather* 64/512 MB data, which is exactly the size of *in_queue* at scale 29/32.

Fig. 6 shows that the intra-node communication actually takes a much larger portion of time than inter-node communication in this case. This means that even the best way to overlap intra- and inter- node communication cannot hide the extra intra-node communication cost introduced by spawning one process per socket.

Fig. 5b illustrates how we share some data structures among processes inside the same node to eliminate intra-node communication of conventional leader-based ***allgather***. The sharing among processes is implemented through ***mmap***.

1) shared in_queue to eliminate step 3

As discussed in Section II, the *in_queue* is actually read-only and is the same among all processes. This motivates us to share one *in_queue* between leader and its children, so that time consumed by *broadcast* (step 3 of Fig. 5a) is completely eliminated. Although the shared *in_queue* will bring certain memory accesses across sockets, it won't cause severe problem for several reasons:

a) smaller data size: the key point that we can tolerant cross-socket accesses for *in_queue* but not for the graph is that the data size of *in_queue* is very small compared to the graph itself, usually less than one thousandth.

b) larger cache size: sharing one *in_queue* among multiple sockets will equivalently enlarge the usable cache capacity for *in_queue*, which may achieve better cache locality.

c) higher access frequency: in_queue is now more frequently accessed by all the cores of multiple sockets, which may result in higher possibility to be cached.

d) faster remote cache access: if *in_queue* is present in the remote cache, it can be accessed even faster than from local memory. The latency of remote cache is considered to be lower than local memory [35]. Even if the data is in local memory, the CPU still needs to snoop remote cache to check if the required data is present there for Intel Nehalem architecture [39].

2) shared out_queue to eliminate step 1

Further on, we can make all processes inside the same node share *out_queue* too; then the leaders can directly access the other processes' *out_queue* and perform ***allgather***, so that the time spent on step 1 in Fig. 5a is eliminated. The sharing scheme is a little different with that of *in_queue*. The *out_queue* of each process is different with each other, so there are multiple *out_queue* in the shared space among processes. Each process is in charge of the update of one *out_queue*, but can read the other *out_queue*. So the leaders can directly perform ***allgather*** without aggregation of data.

The *in_queue_summary* and *out_queue_summary* can be dealt in the same way.

B. Parallelizing Allgather

In Fig. 5b, only one process per node participates in the inter-node ***allgather*** operation, the inter-node network bandwidth cannot be fully used as shown in Fig. 4. This motivates us to let multiple processes perform ***allgather*** simultaneously to decrease inter-node communication time.

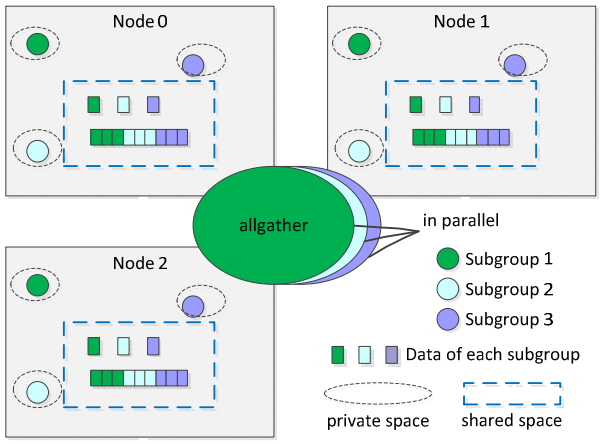


Figure 7. Parallelizing *allgather* – refinement of step 1 in Fig. 5b. Processes are divided into several subgroups, labeled with different colors. Each subgroup performs *allgather* for part of data in parallel. All intermediate results of each subgroup joint into the final result.

Fig. 7 illustrates our mechanism of utilizing multiple processes to perform ***allgather*** in parallel:

- All the processes are divided into several subgroups, labeled with different colors. Each process in a node belongs to exactly one subgroup.
- Each subgroup performs ***allgather*** for the data in the subgroup, also labeled with different colors.
- All the subgroups perform ***allgather*** in parallel, each resulting in only part of the final data.
- The intermediate results of all subgroups joint into the final result. The joint phase does not need any operation actually – the results of each subgroup are mapped to a contiguous virtual space during their allocation, so the final result is already there after ***allgather***.

Though multiple processes participate in ***allgather***, the amount of data transmitted does not change, and the network bandwidth will be better utilized. Suppose that the total data size is still *m* bytes, the number of MPI processes per node is 8, and the number of all processes is *np*, then according to (1), the total amount of data transmitted can be calculated as

$$8 * (m/8)*(np/8-1) = m*(np/8-1) \qquad (2)$$

where *8* is the number of subgroups performing ***allgather***, *(m/8)* is the data size of each subgroup, *(np/8)* is the number of processes in each subgroup. It indicates that, the amount of data transmitted is exactly the same with that of one process per node responsible for allgather all data.

The parallelizing allgather scheme is similar with multi-leader-based allgather algorithm [21], the key difference is that the multi-leader-based allgather scheme is aimed to optimize intra-node communication by reducing cache contention [21], and each leader still need to obtain all the data, which result in more data transmission than our parallelizing allgather.

C. Granularity of Bitmap

As discussed in Section II, the bitmap for *in_queue* – *in_queue_summary* can speed up the check operations when the corresponding bit is 0, due to its better cache locality and lower access latency.

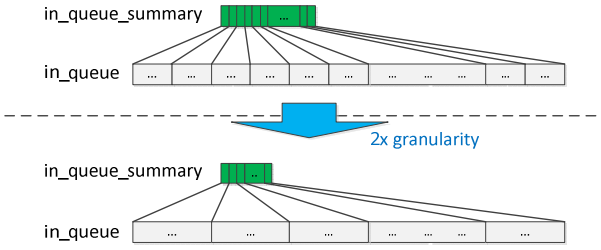


Figure 8. Increase the granularity of *in_queue_summary*, resulting in a much smaller bitmap.

However, as the scale of graph gets larger, the size of the bitmap gets larger too, resulting in lower cache locality.

The conventional granularity for the bitmap is 64 – the size of "unsigned long". If we increase the granularity – let one bit in the bitmap represents more bits in the *in_queue* as shown in Fig. 8, there will be advantages and disadvantages.

1) advantages: smaller size, better cache locality

As the granularity increasing, the size of bitmap gets smaller, thus, the bitmap will have better cache locality and higher possibility to be cached in faster high-level caches.

2) disadvantages: less zeros, less speedup

On the other hand, the proportion of zeros will decrease, since as long as there is one bit 1 in the *in_queue*, the corresponding bit in the bitmap will be 1, no matter what the other bits in the *in_queue* are. For example, if *in_queue* is 00010010_b, for granularity of 2, the bitmap will be 0101_b with 50% zeros; for granularity of 4, the bitmap will be 11_b with no zero. As discussed before, the bitmap is only useful when it is 0, so less zeros means less opportunity of speedup. It is worth to mention that, the bitmap is only used in bottom-up procedure, during which most of the vertices are visited and there are lots of 1s in *in_queue*, so the granularity cannot be very large.

The bitmap can tolerate the increase of granularity to some extent, since as the scale of graph gets larger, there will be more contiguous zeros in the bitmap.

So there may be a trade-off point for the granularity of bitmap: the proportion of zeros does not drop too much while reducing its size for better cache locality.

IV. EVALUATIONS

A. Experimental Setup

We evaluated the *hybrid* BFS algorithm on a "thousand-core" platform, which consists of 16 eight-socket NUMA nodes. The detail configuration of each node is illustrated in Table I, and Fig. 2 gives the topology of the eight CPUs. The SMT is disabled, so there are exactly 1024 cores; the DVFS is also disabled. The two Infiniband ports of each node are connected to one 36-port switch. It is worth to mention that there is one weak node in the 16 nodes, the communication performance of which is weak compared to other nodes due to unknown reason.

TABLE I. NODE CONFIGURATION

CPUs	Eight Intel Xeon X7550 processers each contains: 8 cores, running at 2.0GHz, SMT disabled 32KB/32KB private L1 D/I Cache per Core 256KB private L2 Cache per Core 18MB shared L3 Cache per CPU Four 6.4GT/s fully-width QPI Four 6.4GT/s Intel SMI channels, expanded to eight DDR3 channels after Intel SMB
Memory	Half (four) of the eight DDR3 channels per CPU are populated with 8GB DDR3-1066 DRAMs 17.1GB/s[1] peak memory bandwidth per CPU Total 256GB memory per node
Network	2x 40Gbps Infiniband Ports per node

1. Intel SMB can only support DDR3 burst length of four, so only half of the theoretical bandwidth of one DDR3 channel can be achieved. [6]

The *hybrid* BFS algorithm (Fig.1) is implemented according to [9], which is MPI/OpenMP hybrid programmed. Open MPI 1.5.5 [3] is used as implementation of MPI.

The evaluation method of Graph500 [1] is adopted. The size of input graph is defined by its SCALE, which is the logarithm base two of the number of vertices. In each single test, 64 different vertices are random selected as the roots of 64 BFS iterations. Each iteration reports its TEPS (Traversed Edges per Second), and the final result is calculated as the harmonic mean of the TEPS of 64 iterations. Besides the final TEPS, we perform detailed profiling for each test. When the profiling results are given, they are the average of 64 BFS iterations.

We perform most of the evaluations by spawning one process per socket (eight processes per node) and binding it to socket as discussed in Section II; otherwise indicated.

B. Overview

Fig. 9 gives an overview of the performance of all the optimizations for *hybrid* BFS algorithm on 16 nodes, where "+ xx" means the corresponding optimization explained in Section III is implemented on the previous version.

We demonstrate that by simply spawning 1 process per node and binding it to socket, 1.53X performance can be achieved for "*Original*" implementation. With all our optimizations together, the speedup is up to 2.44X relative to "*Original.ppn=1*" and 1.60X relative to "*Original.ppn=8*". When compared to "*Original.ppn=8*", "*Share in_queue*" improves the performance by 34.1% for its great reduction of communication cost; "*Share all*" and "*Par allgather*" can give another 6.5% and 4.6% of speedup respectively; "*Granularity*" can further improve the performance by 14.8%.

C. NUMA Optimization

We evaluate the "*Original*" implementation under scale 28 on single node when spawning different number of processes with different *mpirun* and *numactl* flags, and varying the number of OpenMP threads. The OpenMP

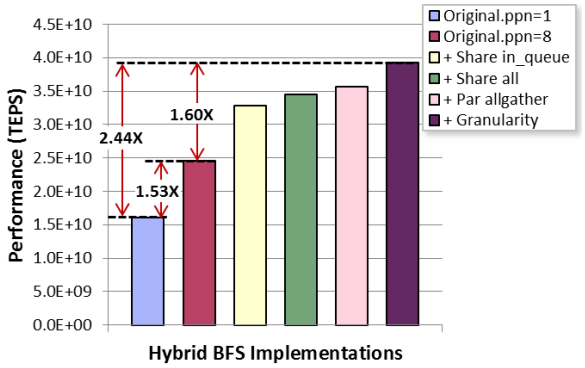


Figure 9. Overview of all optimizations. Graph scale of 32 is evaluated on 16 nodes. "***Original***" means the initial implementation. All versions spawn 8 procs per node and bind them to sockets, except "***Original.ppn=1***" with 1 proc per node. "***Share in_queue***" means *in_queue* is shared; "***Share all***" means *in_queue*, *out_queue*, *in_queue_summary,* and *out_queue_sumamry* are all shared; "***Par allgather***" means *allgather* for *in_queue* is done in parallel; "***Granularity***" means the best result of all tested granularities.

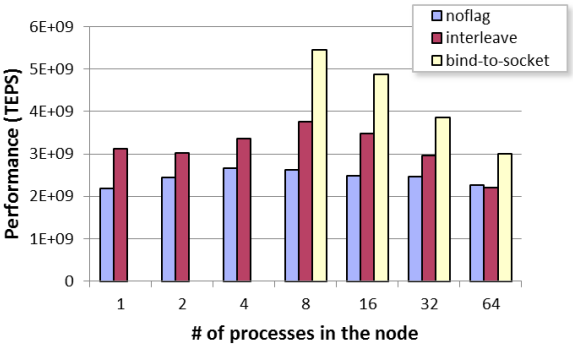


Figure 10. Performance of "*Original*" implementation with various execution policy. Graph scale of 28 is evaluated on 1 node**.** *"**noflag**"* means just simply execution of the program without special *numactl* or *mpirun* flags; *"**interleave**"* means using the flag *"--interleave=all"* for *numactl*; *"**bind-to-socket**"* means using the flag *"--bind-to-socket --bysocket"* for *mpirun*. *"bind-to-socket"* only works when more than 8 processes are spawned, otherwise partial of the 8 CPUs will be idle.

dynamic scheduler [5] is used to avoid load-balance problem, so when we run 64 threads total (equal to core number), the results are almost the best. Fig. 10 only illustrates the best results among various thread numbers on single node.

For single node**, the performance of "*ppn=8.bind-to-socket*" is best** as shown in Fig. 10, which is 1.74X of "*ppn=1.interleave*" and 2.08X of "*ppn=8.noflag*". "*ppn=1.interleave*" has better performance than "*ppn=1.noflag*", because the graph is equally interleaved across all sockets in the former, so all the memory bandwidth of all sockets can be utilized. The low performance of "*ppn=8. noflag*" indicates that the multiple threads of one process are spread across multiple sockets if binding is not used, which results in cross-socket memory accesses.

Fig. 11 shows the detailed profiling results of the "*Original*" implementation to show how "*ppn=8.bind-to-socket*" affects different phases of BFS. It can be clearly seen that ***"ppn=8.bind-to-socket"* greatly speeds up both the top-down and bottom-up computation phase.** The 1.58x

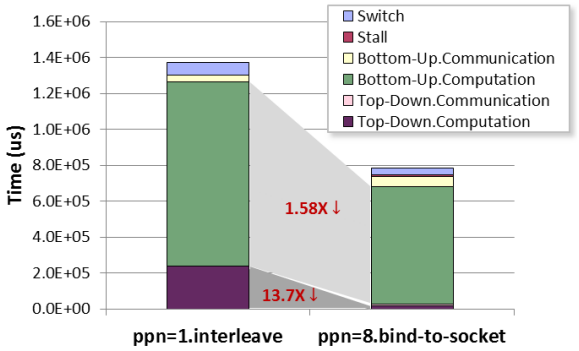


Figure 11. Execution time breakdown and speedup of computation procedure for the "*Original*" implementation. Graph scale of 28 is evaluated on single node. *Switch* represents the time consumed on switching top-down to bottom-up and vice versa, which need to convert corresponging data structure. *Stall* represents the idle time due to load unbalance.

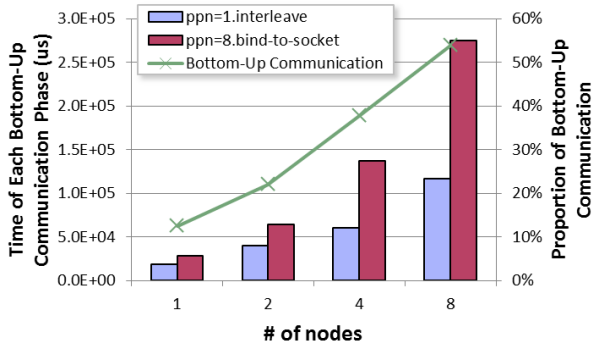


Figure 12. Communication cost of the "*Original*" implementation when weak scaling from 1 node to 8 nodes. The **bars** represent the absolute time of each communicaiton phase in bottom-up procedure for *"ppn=1.interleave"* and *"ppn=8.bind-to-socket"*. The **curve** represents the proportion of all the bottom-up communicaiton time relative to total execution time for *"ppn=8.bind-to-socket"*. The scales of graph for 1, 2, 4, and 8 nodes are 28, 29, 30, and 31 respectively.

speedup of bottom-up computation can be fully attributed to the reduction of remote memory access, because unlike [7], there are no atomic memory updates in bottom-up procedure. In *"ppn=1.interleave"*, one socket owns 1/8 of the graph, so 7/8 of the memory accesses to the graph will go into the remote memory. However, in *"ppn=8.bind-to-socket"*, the graph is naturally partitioned into 8 parts, and the remote memory accesses to the graph in the computation phase is now replaced by the two ***allgather*** for *in_queue* and *in_queue_summary* in communication phase as shown in Fig. 1. Although the intra-node ***allgather*** still needs memory accesses across sockets, the amount of data accesses is much smaller than that of graph.

For the 16-node cluster, "*ppn=8.bind-to-socket*" can still provide 1.53X performance than "*ppn=1.interleave*", as shown in Fig. 9.

1) Communication Cost

In Section II, we discuss the amount of data transmitted in communication phase in theory. We measure the actual communication time here. Fig. 12 illustrates the cost of communication in bottom-up procedure of the "*Original*" implementation when weak scaling from 1 node to 8 nodes.

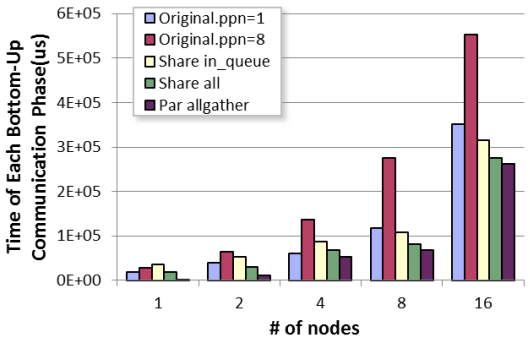


Figure 13. Reduction of average time spent on each bottom-up communicaiton phase. The results of 1, 2, 4, 8 and 16 nodes correspond to the graph scale of 28, 29, 30, 31, and 32.

It can be seen from Fig. 12 that the cost of each communication phase grows exponentially as weak scaling; besides, **the cost of spawning eight processes per node is much larger than one process per node**. The cost of communication phase is the two ***allgather*** for *in_queue* and *in_queue_summary*. As weak scaling, the size of these two data structures grows exponentially, which makes the cost of each communication phase grow exponentially too. Besides, since *"ppn=8.bind-to-socket"* has 8 times of processes, its communication cost is much larger than *"ppn=1.interleave"*, about 2.34 times when 8 nodes are in use.

As a result, **the proportion of time spent on communication quickly grows from 12% for one node to 54% for 8 nodes**. The size of graph that one node needs to search stays the same as weak scaling, so the time spent on computation phase does not changes too much. Due to the exponentially growing communication cost as weak scaling, the proportion it occupies will increase too, which will finally dominate the execute time as continuously scaling.

In summary, although *"ppn=8.bind-to-socket"* has great advantage on computation, its high communication cost makes it less scalable.

D. Communication Optimization

In this subsection, we evaluate the effect of the two optimizations we propose in Section III – "*share data*" and "*parallelizing allgather*" on communication cost.

1) Absolute Time

Fig. 13 shows the average time of each communication phase in bottom-up procedure as weak scaling, which is mainly the two ***allgather*** as shown in Fig. 1. The result of 16 nodes does not have much meaning when compared to 8 nodes or others, since there is one weak node whose Infiniband performance is weak as mentioned before.

Fig. 13 proves that **the abovementioned optimizations have greatly reduced the communication cost, 4.07X for eight nodes**. "*Share in_queue*" has the most significant effect, which can cut off about half of the communication cost. This is because most of the communication cost is spent on ***allgather*** for *in_queue* due to its larger size, and the cost of step 3 in Fig. 5a can be fully eliminated. *"Share all"* can eliminate the cost of step 1 and step 3 in Fig. 5a for the

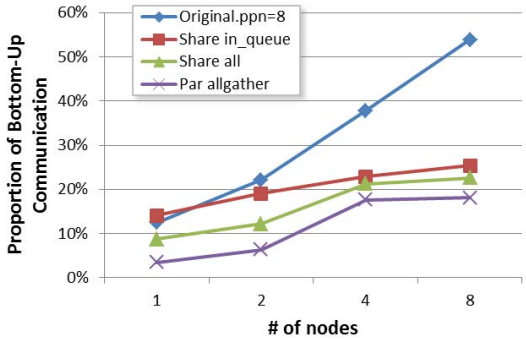


Figure 14. Reduction of bottom-up communication's proportion relative to the total execution time in *hybrid* BFS algorithm. The results of 1, 2, 4, and 8 nodes correspond to the graph scale of 28, 29, 30, and 31. We do not give the result of 16 nodes, because it does not make much sense due to the one ill-performed node.

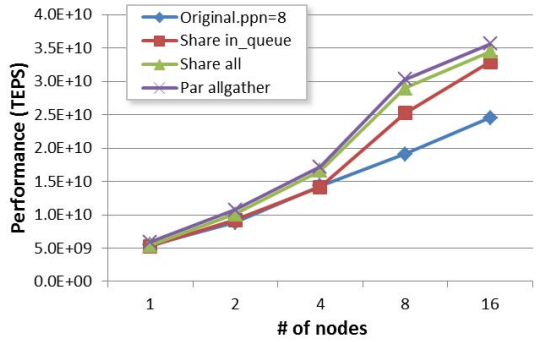


Figure 15. Weak scalability of different implementations under *"ppn=8.bind-to-socket"*. The results of 1, 2, 4, 8 and 16 nodes correspond to the graph scale of 28, 29, 30, 31, and 32.

both ***allgather*** for *in_queue* and *in_queue_summary*. "*Par allgather*" can further reduce the cost of step 2 in Fig. 5a (or step 1 in Fig. 5b) by better utilizing the two Infiniband ports.

2) Proportion and Weak Scalability

If the time spent on communication can be reduced, the *hybrid* BFS algorithm will be scaled to much more nodes, because the time spent on computation does not change too much as discussed before. Fig. 14 illustrates the proportion of time spent on bottom-up communications relative to the total execution time. The proportion of time spent on top-down procedure, stall and switch together occupy less than 20% of total time, even in the case that the bottom-up communication time has been optimized to minimum. The optimization for these cost are out of scope for this paper.

It can be seen from Fig. 14 that **the proportion of time spent on bottom-up communication has been greatly reduced**. For example, the proportion on eight nodes can be reduced from 54% without any optimizations to 18% with all the optimizations for communication implemented. This will lead to much better scalability.

Fig. 15 illustrates the weak scalability from 1 node to 16 nodes. The optimizations for communication result in better scalability than "*Original.ppn=8*". The inferior scalability from 8 nodes to 16 nodes is due to the influence of the weak node.

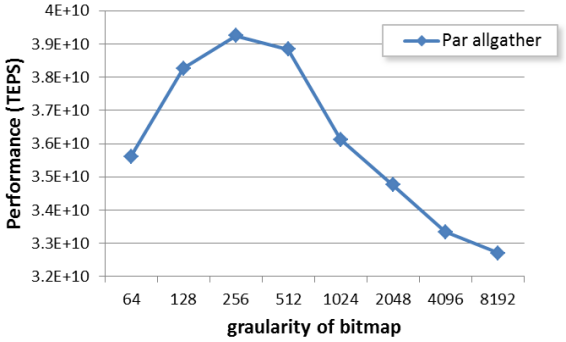


Figure 16. Performance of different granularities for the bitmap - *in_queue_summary*. Graph scale of 32 is tested on 16 nodes.

E. Computation Optimization

As is discussed before, *in_queue_summary*, the bitmap of *in_queue* can speed up the computation phase in bottom-up procedure. Its granularity can further influence the cache locality and then the performance of computation. Fig. 16 illustrates the performance of *hybrid* BFS algorithm when different granularities are chosen for the bitmap based on *"Par allgather"* implementation.

As is shown in the Fig. 16, **granularity of 256 can achieve the highest performance**, which is 10.2% higher than that of 64. However, as the granularity continues to increase, the benefit starts to decrease, and the performance is even lower than that of 64. This is due to the decreasing proportion of 0 in the bitmap.

V. RELATED WORK

BFS: Since BFS on large, distributed graphs has been gaining ever-increasing importance, a large body of work has been contributed to optimize BFS. Agarwal et al. [7] observed that atomic updates could not scale efficiently across multiple sockets. To mitigate it, they proposed a socket-aware channel communication mechanism, they partitioned graph among sockets and adopted batching updates, which achieved good scaling across multiple Nehalem EX sockets. Our work adopts a one-process-per-socket mapping, which could achieved nearly the same result as socket-channel, however our approach is much easier to implement cause we do not need to modify any code. Beamer et al. [9] proposed a hybrid top-down along with bottom-up algorithm to accelerate BFS in a quad-socket node, which could significantly reduce the number of edges examined, and it was the basis of our work. Hong et al. [19] proposed a similar hybrid Read and Queue method to remove atomic instructions and make the memory access pattern more sequential, they further extended BFS for GPU to gain better performance. However all of the above work just optimized BFS performance inner-node, and did not consider the network communication overhead, which would increase up to 54% of the whole execution time of BFS with only 8 nodes (please refer to section IV for more detail), thus our work is mainly focus on mitigating the communication overhead. Aydin et al. [11] proposed a two-dimensional partitioning-based approach for BFS on distributed memory

systems. When coupled with intra-node multithreading, they could reduce the communication overhead by a factor of 3.5. Although our work also focus on reducing communication overhead, there exist many differences: (1) their work was based on the traditional bottom-up algorithm, while this paper adopts the hybrid BFS algorithm [9]; (2) their work did not consider the impact of NUMA architecture, especially on communication overhead; (3) their work proposed a new BFS algorithm, our work mainly focuses on how to effectively implement inter-node communication, and reduce or even remove the extra intra-node communication among multiple sockets. Actually, they are orthogonal – our implementation could be applied to 2-D partition algorithm to further reduce its communication overhead. Xia and Prasanna [47] proposed a topologically adaptive parallel BFS, which dynamically adjust the number of active threads based on the estimated scalability to reduce the synchronization overhead in one multicore node. There has been also much research conducted on implementing efficient BFS on specific architectures, such as GPU [8, 18, 19, 24, 33, 34], Cray system [8, 34], Cell/BE processor [38], BlueGene/L system [48].

Collective Communication: Thakur and Gropp [41] used multiple algorithms depending on the message size to improve the performance of collective operations in MPICH, take allgather operation for example, they suggested to use recursive-doubling algorithm for short and medium size messages (<512KB) and the ring algorithm for long messages (>=512KB). Ma et al. [26] proposed new MPI collective communication algorithms that took NUMA into account to avoid memory contention, and they further adopted topology and NUMA-aware KNEM collectives with pipelined strategies. Ma et al. [27] also took NUMA architecture into account, and proposed a distance-aware process placement mechanism which would construct optimal runtime topologies to optimize intra-node collective algorithms. HierKNEM [25] is a kernel-assisted topology-aware collective framework, which could make intra- and inter- node communication overlap. It adopts leaders to participate in the inter-node collective topology and offloading memory copies to non-leader processes for intra-node communication. However, if the intra-node communication cost is even higher than that of inter-node (please refer to Fig. 6), overlapping will not help. Leader based hierarchical algorithms [16, 31, 42] chose a single leader process on each node, only the leaders were involved in the inter-node communications, and the intra-node communications were between the leader and other non-leader processes. However it might face high shared memory contention as the number of cores per node increased. To mitigate the memory contention for single-leader designs, Kandalla et al. [21] proposed a scalable multi-leader based hierarchical Allgather design for NUMA machines, they suggested one process per socket as the leader for shared memory approach would achieve the best performance. However, their multi-leader approach would only apply for small and medium size messages, not suitable for large messages. Other Collective Communication optimizations, includes: Topology-Aware [15, 22, 37], process placement [14, 43] and non-blocking collective operations [17].

Hybrid MPI/OpenMP programming model: In [36], Rabenseifner et al. evaluated various programming models on hierarchically hardware, including pure MPI, pure OpenMP, and hybrid MPI/ OpenMP. They found that hybrid programming model was the superior solution, and they suggested taking hardware topology into account. Tsuji and Sato [44] evaluated performance of hybrid MPI/ OpenMP programming model on a large scale multi-core multi-socket NUMA cluster, they found that using MPI across sockets and OpenMP within sockets would gain the best performance. Jeannot and Mercier [20] proposed a novel process placement algorithm to reduce communication cost on NUMA nodes, which took hierarchy topology and communication pattern into account. Wu and Taylor [46] implemented hybrid MPI/OpenMP versions of SP and BT benchmarks on large scale multicore supercomputers, in which they adopted MPI between nodes and OpenMP multithreads within a node mechanism.

VI. CONCLUSION

In this work, we demonstrate that using MPI across sockets and OpenMP inside socket performs better for the *hybrid* BFS algorithm on NUMA architecture. For a NUMA cluster with 16 eight-socket nodes, this method results in 1.53X performance without modifying the algorithm. However, we have noticed that the communication cost of this method is very high due to more MPI processes, which will make it unable to scale to more nodes.

We introduce optimizations to the algorithm to reduce its high communication cost while maintaining its advantage on computation. Sharing data among processes can eliminate the need of intra-node communication; parallelizing allgather can fully utilize the bandwidth of the two Infiniband ports to speed up inter-node communication. We successfully reduce the communication cost from 54% of total time on eight nodes to 18%, which will result in better scalability. We further change the granularity of a key bitmap, which can speed up the check operations in the computation phase of hybrid BFS algorithm. These together will result in 2.44X speedup relative to the original.

Our result shows although using socket-private data structure can reduce cross-socket memory accesses, directly mapping and sharing certain data structure can also reduce communication cost. To what extent data should be shared on NUMA platform need to be considered carefully.

ACKNOWLEDGEMENT

The authors thank the anonymous reviewers for their constructive suggestions. This research is supported by the National Natural Science Foundation of China (NSFC) under grant numbers 60903046, 60921002, 60925009, 61003062 and the National Basic Research Program of China (973 Program) under a grant number 2011CB302502.

REFERENCES

[1] Graph500 Home Page. Available: www.graph500.org

[2] MVAPICH Home Page. Available: http://mvapich.cse.ohio-state.edu

[3] Open MPI Home Page. Available: www.open-mpi.org

[4] OSU Micro-Benchmarks 3.6. Available: http://mvapich.cse.ohio-state.edu/benchmarks

[5] "OpenMP Application Program Interface Version 3.0," ed: OpenMP Architecture Review Board, 2008, pp. 38-46.

[6] Intel® 7500/7510/7512 Scalable Memory Buffer Datasheet: Intel, 2011.

[7] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable Graph Exploration on Multicore Processors," in SC, 2010, pp. 1-11.

[8] D. A. Bader and K. Madduri, "Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2," in ICPP, 2006, pp. 523-530.

[9] S. Beamer, K. Asanovi , and D. A. Patterson, "Searching for a Parent Instead of Fighting Over Children: A Fast Breadth-First Search Implementation for Graph500," Technical Report No. UCB/EECS-2011-1172011.

[10] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova, "A case for NUMA-aware contention management on multicore systems," presented at the USENIX ATC, Portland, OR, 2011.

[11] A. Buluc and K. Madduri, "Parallel breadth-first search on distributed memory systems," in SC, 2011.

[12] L. Chai, A. Hartono, and D. K. Panda, "Designing high performance and scalable MPI intra-node communication support for clusters," in Cluster, 2006, pp. 1-10.

[13] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in SIAM International Conference on Data Mining, 2004, p. 541.

[14] H. Chen, W. Chen, J. Huang, B. Robert, and H. Kuhn, "MPIPP: an automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters," in SC, 2006, pp. 353-360.

[15] C. Coti, T. Herault, and F. Cappello, "MPI applications on grids: A topology aware approach," in Euro-Par, 2009, pp. 466-477.

[16] R. Graham and G. Shipman, "MPI support for multi-core architectures: Optimized shared memory collectives," Lecture Notes in Computer Science, vol. 5205, pp. 130-140, 2008.

[17] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and performance analysis of non-blocking collective operations for MPI," in SC, 2007, pp. 1-10.

[18] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in PPoPP, 2011, pp. 267-276.

[19] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core CPU and GPU," in PACT, 2011, pp. 78-88.

[20] E. Jeannot and G. Mercier, "Near-optimal placement of MPI processes on hierarchical NUMA architectures," in Euro-Par 2009 Parallel Processing, 2010, pp. 199-210.

[21] K. Kandalla, H. Subramoni, G. Santhanaraman, M. Koop, and D. K. Panda, "Designing multi-leader-based allgather algorithms for multi-core clusters," in IPDPS, 2009, pp. 1-8.

[22] K. Kandalla, H. Subramoni, A. Vishnu, and D. K. Panda, "Designing topology-aware collective communication algorithms for large scale InfiniBand clusters: Case studies with Scatter and Gather," in IPDPSW, 2010, pp. 1-8.

[23] C. Lei, G. Qi, and D. K. Panda, "Understanding the Impact of Multi-Core Architecture in Cluster Computing: A Case Study with Intel Dual-Core System," in CCGrid, 2007, pp. 471-478.

[24] L. Luo, M. Wong, and W. Hwu, "An effective GPU implementation of breadth-first search," in DAC, 2010, pp. 52-55.

[25] T. Ma, G. Bosilca, A. Bouteiller, and J. J. Dongarra, "HierKNEM: An Adaptive Framework for Kernel-Assisted and Topology-Aware Collective Communications on Many-core Clusters," in IPDPS, 2012.

[26] T. Ma, G. Bosilca, A. Bouteiller, B. Goglin, J. M. Squyres, and J. J. Dongarra, "Kernel Assisted Collective Intra-node MPI Communication Among Multi-core and Many-core CPUs," in ICPP, 2011, pp. 532-541.

[27] T. Ma, T. Herault, G. Bosilca, and J. J. Dongarra, "Process Distance-aware Adaptive MPI Collective Communications," in Cluster, 2011, pp. 196-204.

[28] Z. Majo and T. R. Gross, "Memory management in NUMA multicore systems: Trapped between cache contention and interconnect overhead," in ISMM, 2011.

[29] Z. Majo and T. R. Gross, "Memory system performance in a NUMA multicore multiprocessor," in SYSTOR, 2011.

[30] Z. Majo and T. R. Gross, "Matching Memory Access Patterns and Data Placement for NUMA Systems," in CGO, 2012, p. 25.

[31] A. Mamidala, A. Vishnu, and D. Panda, "Efficient Shared Memory and RDMA Based Design for MPI_Allgather over InfiniBand," Lecture Notes in Computer Science, vol. 4192, pp. 66-75, 2006.

[32] C. McCurdy and J. Vetter, "Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms," in ISPASS, 2010, pp. 87-96.

[33] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in PPoPP, 2012, pp. 117-128.

[34] D. Mizell and K. Maschhoff, "Early experiences with large-scale Cray XMT systems," in IPDPS, 2009, pp. 1-9.

[35] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller, "Memory performance and cache coherency effects on an intel nehalem multiprocessor system," in PACT, 2009, pp. 261-270.

[36] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes," in PDNP, 2009, pp. 427-436.

[37] P. Sack and W. Gropp, "Faster topology-aware collective algorithms through non-minimal communication," in PPoPP, 2012, pp. 45-54.

[38] D. P. Scarpazza, O. Villa, and F. Petrini, "Efficient breadth-first search on the cell/be processor," Parallel and Distributed Systems, IEEE Transactions on, vol. 19, pp. 1381-1395, 2008.

[39] R. Singhal, "Inside Intel next generation Nehalem microarchitecture," in Hot Chips, 2008.

[40] T. Suzumura, K. Ueno, H. Sato, K. Fujisawa, and S. Matsuoka, "Performance characteristics of Graph500 on large-scale distributed environment," in IISWC, 2011, pp. 149-158.

[41] R. Thakur and W. Gropp, "Improving the performance of collective operations in MPICH," Lecture Notes in Computer Science, vol. 2840, pp. 257-267, 2003.

[42] J. Träff, "Efficient allgather for regular SMP-clusters," Lecture Notes in Computer Science, vol. 4192, pp. 58-65, 2006.

[43] J. L. Traff, "Implementing the MPI process topology mechanism," in SC, 2002, pp. 28-28.

[44] M. Tsuji and M. Sato, "Performance Evaluation of OpenMP and MPI Hybrid Programs on a Large Scale Multi-core Multi-socket Cluster, T2K Open Supercomputer," in ICPPW, 2009, pp. 206-213.

[45] R. F. Van der Wijngaart and H. Jin, "Nas parallel benchmarks, multi-zone versions," NASA Ames Research Center, Tech. Rep. NAS-03-010, 2003.

[46] X. Wu and V. Taylor, "Performance characteristics of hybrid MPI/OpenMP implementations of NAS parallel benchmarks SP and BT on large-scale multicore supercomputers," ACM SIGMETRICS Performance Evaluation Review, vol. 38, pp. 56-62, 2011.

[47] Y. Xia and V. Prasanna, "Topologically Adaptive Parallel Breadth-First Search on Multicore Processors," in PDCS, 2009.

[48] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, "A scalable distributed parallel breadth-first search algorithm on BlueGene/L," in SC, 2005, pp. 25-25.