Comput Sci Res Dev (2013) 28:193–201
DOI 10.1007/s00450-012-0207-3

SPECIAL ISSUE PAPER

Understanding parallelism in graph traversal on multi-core clusters

Huiwei Lv · Guangming Tan · Mingyu Chen · Ninghui Sun

Published online: 23 May 2012

Abstract There is an ever-increasing need for exploring large-scale graph data sets in computational sciences, social networks, and business analytics. However, due to irregular and memory-intensive nature, graph applications are notoriously known for their poor performance on parallel computer systems. In this paper we propose a new hybrid MPI/Pthreads breadth-first search (BFS) algorithm featuring with (i) overlapping computation and communication by separating them into multiple threads, (ii) maximizing multi-threading parallelism on multi-cores with massive threads to improve throughputs, and (iii) exploiting pipeline parallelism using lock-free queues for asynchronous communication. By comparing it with traditional MPI-only BFS algorithm, we learned several valuable lessons that would help to understand and exploit parallelism in graph traversal applications. Experiments show our algorithm is 1.9× faster than the MPI-only version, capable of processing 1.45 billion edges per second on a 32-node SMP cluster. At a large scale, our algorithm is 1.49× than the MPI-only BFS algorithm in Combinatorial BLAS Library with 6,144 cores.

Keywords Breadth-first search · Graph algorithms · Hybrid MPI/Pthreads programming · Lock-free queues

H. Lv (✉) · G. Tan · M. Chen · N. Sun
State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
e-mail: lvhuiwei@ncic.ac.cn

H. Lv
Graduate School of Chinese Academy of Sciences, Beijing, China

1 Introduction

Graphs have been extensively used to abstract complex systems and interactions in emerging "big data" applications, such as social network analysis, WWW, biological systems and data mining. With the increasing growth in these areas, petabyte-sized graph datasets are produced for knowledge discovery.

Graph applications are typical examples of irregular applications, whose performance improvements are notoriously difficult on parallel computer systems [14]. They are different from the traditional computing intensive applications in various ways. Graph computations are often completely data-driven, there is a higher ratio of data access to computation than for scientific computing applications. Moreover, the data in graph problems are typically unstructured and highly irregular, which leads to poor locality of graph algorithms.

On the other side, there is an emerging trend to use hybrid programming model on multi-core clusters. With increasing numbers of cores per node, multi-core clusters provide a natural programming paradigm for hybrid programs, where OpenMP is used for intra-node data sharing between multi-cores and MPI for communication between nodes, minimizing communication and synchronization overhead. However, whether the hybrid programming model outperforms the MPI-only depends on specific problems being considered [8, 13, 19]. It is still not clear how BFS performs with hybrid programming models, or what kinds of parallelism it can exploit.

Breadth-first search (BFS) is a basic building block for many important graph applications. The newly announced Graph500 benchmark [1], which ranks supercomputers based on their performance on data-intensive applications, chose BFS as their first representative program. In this paper we present a new hybrid MPI/Pthreads BFS algorithm

on multi-core clusters. By comparing it with the traditional MPI-only version, we also made several findings. Specifically, we make the following contributions:

- We propose a new hybrid parallel BFS algorithm on a distributed memory system with multi-core processors. The algorithm exploits both intra-node and inter-node parallelism: *core-level* and *memory-level* parallelism to improve throughput on multi-core architectures, and *pipeline-level* parallelism of asynchronous communication to improve scalability on distributed memory architectures.
- We implement our parallel BFS algorithm with hybrid MPI/Pthreads programming. Experimental results show our algorithm is 1.9× faster than the MPI-only version in Graph 500 benchmark with 32 nodes, and 1.49× than the MPI-only version in Combinatorial BLAS Library with 6,144 cores.
- We learn several valuable lessons that would help to understand and exploit parallelism in graph traversal applications on multi-core clusters.

The rest of the paper is organized as follows, Sect. 2 briefly introduces Graph500 benchmark and our experiment platform. Section 3 reports analysis of the MPI-only implementations of Graph500. Section 4 presents our new hybrid algorithm, whereas Sect. 5 shows experimental results. Section 6 describes the related works. Finally, Sect. 7 concludes this paper.

2 Background

In this section we give a brief introduction to the Graph 500 benchmark and its BFS algorithm. We also describe the architectural parameters of our multi-core cluster.

– *Graph 500.*

Graph 500 [1] is a set of large-scale benchmarks for data-intensive applications to complement the Top 500 [2]. It has gained attention from both academia and industry since the benchmark was first released for ranking supercomputers in 2010. Graph 500 use synthetic Kronecker graphs [12] which follow power law distributions. In order to save space, an adjacency array (or list) representing sparse graph is transformed into compressed sparse row (CSR). Table 1 summarizes the graph datasets used in this paper. The graph size is determined by two parameters: "Scale" and "Edge factor", where the total number of vertices N equals 2^{Scale}, and the number of edges, $M = edgefactor * N$.

In order to compare the performance of Graph 500 implementations across a variety of architectures, a new performance metric is adopted in Graph 500. Let *time* be the measured execution time for running BFS. Let m be the number of input edge tuples within the component traversed by the search, counting any multiple edges and self-loops. The normalized performance rate *traversed edges per second (TEPS)* is defined as: $TEPS = m/time$.

Table 1 Graph datasets used in this paper. Graphs are generated using synthetic Kronecker graph generator in Graph 500

Scale	Vertices number	Edge factor	Memory usage
30	1.07 billion	16	272 GB
29	537 million	16	136 GB
28	268 million	16	68 GB
27	134 million	16	34 GB
26	67 million	16	17 GB
25	34 million	16	8.5 GB

– *Serial BFS algorithm.*

A graph $G(V, E)$ is composed of a set of vertices V and a set of edges E. Given a graph $G(V, E)$ and a root vertex $r \in V$, the Breadth-First Search (BFS) algorithm explores the edges of G to traverse all the vertices reachable from r, and it produces a breadth-first tree rooted at r. In the level-based algorithm a current queue (CQ) is used to record all vertices at current level l. Each vertex is fetched from CQ and explored, that is, its neighboring vertices are visited, then are stored into a next queue (NQ) if they are visited for the first time. In some cases, the traversal path traces are recorded so that we can backtrace the BFS tree. Therefore, we set the parent of v to be u or array $P[v] \leftarrow u$ if v is identified as a neighboring vertex of u during the processing of exploring u at current level. Correspondingly, vertex v is inserted into queue NQ for the exploration of the next level. At the end of each level, we reset the current queue CQ to be the next queue NQ. The traversal finishes when there are no more vertices in NQ.

– *Multi-core cluster architecture.*

Our experiment platform is a multi-core cluster, connected by Infiniband network. In our experiments, we use two different scales: 32 and 512 nodes. Each node is an SMP architecture with two Xeon X5650 CPUs (Westmere), which are connected through Intel QuickPath Interconnect (QPI). The Xeon X5650 has six cores, each supports simultaneous multithreading (SMT) up to two threads. Table 2 summarizes its architectural parameters.

3 MPI-only BFS algorithm and analysis

In this section we first introduce an MPI-only BFS algorithm in the Graph 500 benchmark, then perform some experimental analyses to identify bottlenecks.

Table 2 Experiment platform

Node	SMP
Number of CPUs	2
Processor	Intel X5650
Number of cores	6
Number of threads	12
Core frequency	2.66 GHz
L1 cache size	384 KB
L2 cache size	1536 KB
L3 cache size	12 MB
Memory type	DDR3-1333
QPI Speed	6.4 GT/s
Interconnect	Infiniband
Rate	40 Gb/s (4X QDR)

Algorithm 1: MPI-only parallel BFS algorithm

```
   Input  : G(V, E), source vertex r
   Output: Array P[1..n] with P[v] holding the parent
           of v
   Data: outbuf[o]:Array of outgoing buffers
1  while true do
2      foreach u ∈ CQ do
3          check incoming vertices and insert them to NQ;
4          u ← Dequeue CQ;
5          foreach v adjacent to u do
6              o = getowner(v);
7              if o = rank then
8                  if P[v] = ∞ then
9                      P[v] ← u, NQ ← Enqueue v;
10                     NQ_Count ← NQ_Count + 1;
11             else
12                 outbuf[o] ← v, outbuf[o] ← u;
13                 if outbuf[o] is full then
14                     MPI_Isend(outbuf[o]);
15     flush outbuf and send finish message;
16     MPI_Allreduce(&NQ_Count, &Sum);
17     if Sum = 0 then break;
18     Swap(CQ, NQ), NQ ← ∅;
```

Algorithm 1 describes the MPI-only parallel BFS algorithm. First we partition the graph among the processes. Let each process own N/p vertices and all the outgoing edges from those vertices, where N is the number of all vertices, and p is the number of all processes. Every process only maintains the status of vertices it owns, and only the owner process of a vertex can identify whether it is newly visited or not. All the adjacencies of the vertices in the current frontier need to be sent to their corresponding owner process ($getowner(v)$). In practice, each process creates $p-1$ message buffers ($outbuf[o]$), each buffer is assigned to another process except itself. To overlap computation and communication as much as possible, message passing between processes use non-blocking *MPI_Isend* and *MPI_Irecv* (line 14, line 3).

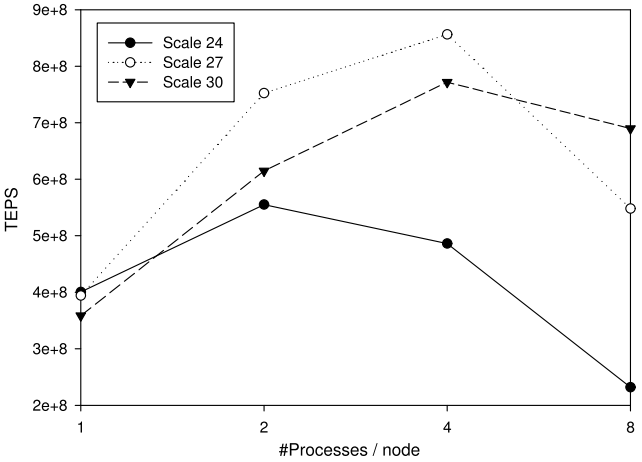

Fig. 1 Scalability of MPI-only BFS on a 32-node cluster for Kronecker graphs at different scales

Observation 1 MPI-only BFS scales poorly across multiple nodes due to extremely intensive MPI communication with long latency.

Figure 1 plots the scalability of MPI-only BFS (Algorithm 1). We fix the number of nodes and problem scale, and change the number of processes per node to see the scalability of the program. As we can see in the figure, when the number of processes per node changes from four to eight, the performance degrades. The communication cost overwhelms the benefit of increase in process numbers. Thus it is necessary to eliminate domain decomposition at node level using hybrid approach.

Observation 2 On a single multi-core node, multi-threading BFS performs better than MPI.

Next we compare the MPI-only algorithm with an OpenMP implementation of BFS program in Graph 500[1] on a single node. The OpenMP parallelism is exploited for all vertices at current level (vertices in CQ). As we can see in Fig. 2, the OpenMP-only BFS outperforms MPI-only BFS

[1] For fairness, we optimize the original program with bitmap technique as the MPI program does.

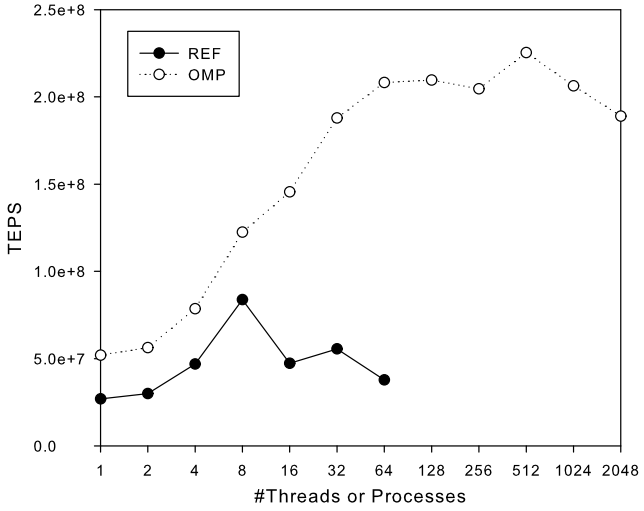

Fig. 2 MPI-only BFS (REF) vs. OpenMP-only BFS (OMP) performance on a single node

enormously, and its performance continues to increase even if its number of threads exceeds the number of cores per node. The improvement is from memory level parallelism. A deep profile finds out that the bandwidth of BFS increases as the number of threads grows (discussed in Fig. 6, Sect. 5). In fact, a similar result is observed in previous work [3] that shows Nehalem can hide the memory latency by keeping a number of read requests in flight.

Based on these observations, we will design a hybrid algorithm in the next section.

4 Hybrid BFS algorithm

In this section we describe the design of a new hybrid MPI/Pthreads BFS algorithm. *The key idea is to keep events as asynchronous as possible*. Our strategies include: (i) separating computation and communication into multiple threads to achieve overlapping; (ii) leveraging multi-threading mechanism on multi-core architecture to tolerate latency; (iii) using lock-free algorithms to efficiently execute asynchronous operations.

– *Separating communication from computation.*

The first step is to separate computation from communication. We group all operations into computation and communication, then assign them to two different threads: a *master* thread (Algorithm 2) and many *traversal* threads (Algorithm 3). The former is in charge of communication among different MPI processes, and the latter are in charge of traversing vertices at the current BFS level. We choose to use one *master* thread and many *traversal* threads based on the observation that one core is capable of saturating the lanes of the PCIe network link.

Algorithm 2 describes the work of the *master* thread: repeatedly check the incoming and outgoing buffers, insert in-

Algorithm 2: Hybrid BFS algorithm, Part 1: *master* thread

```
   Data: Sum: overall number of vertices in all NQ
   trav_finish: indicates whether all traversal threads are
   finished in this level
   barr_start: barrier for all threads
   barr_trav: barrier for all traversal threads
   barr_all: barrier for all threads
1  while true do
2      barrier_wait(&barr_start);
3      while not trav_finish do
4          check incoming vertices and insert them to NQ;
5          for o ∈ [0..size − 1] do
6              MPI_Test(&outreq[o], &flag);
7              if flag then DEQUEUE(OUTBUF);
8          for o ∈ [0..size − 1] do
9              if real_count[o] = BUFLEN then
10                 MPI_Isend(outbuf[o], BUFLEN,
                   &outreq[o]);
11     flush outbuf and send finish message;
12     MPI_Allreduce(&NQ_Count, &Sum);
13     barrier_wait(&barr_all);
14     if Sum = 0 then break;
15     Swap(CQ, NQ), NQ ← ∅;
```

coming vertices to *NQ* and send *outbuf* out. Line 7 checks the status of *outbuf*, if it is full, then sends its content to another corresponding process. At the same time, vertices received in line 6 will be inserted to *NQ*. The data structures are described in Fig. 3.

– *Fine-grained parallel traversal.*

Traversal threads first synchronize with the *master* thread at start (line 2), then partition vertices in *CQ* between themselves (line 3). After that, each *traversal* thread starts to traverse its vertices (lines 4–11). There are two different kinds of vertices here: the first kind is local vertices, they are visited and inserted to local *NQ*; the other kind is remote vertices, they will be inserted to *outbuf* and sent to other processes. After traversing all its neighbors, *traversal* threads synchronize with the *master* thread and wait for it to finish its job, then start another level (lines 12–15).

The *outbuf* is shared among all *traversal* threads and *master* thread. It is a multiple-producer single-consumer problem which involves mutual exclusion. Generally speaking, locks are sufficient when there are not too many threads. However, to leverage multi-threading mechanism of multi-core processors, high number of threads are used to maximize multi-threading parallelism (as the Observation 2 in Sect. 3 indicates). In this situation, the cost of locks would be

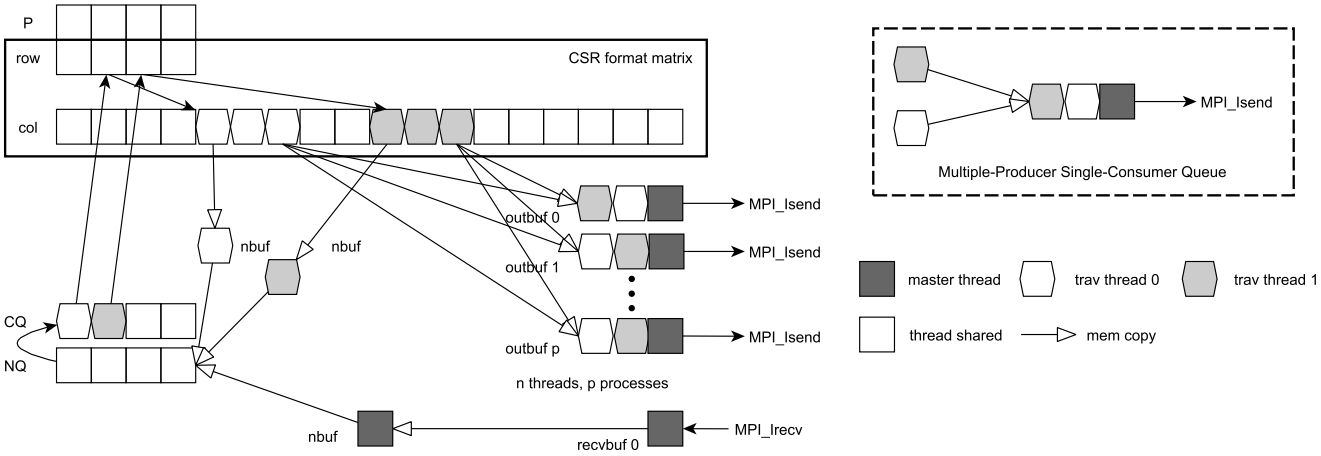

Fig. 3 Data structures used in hybrid BFS

Algorithm 3: Hybrid BFS algorithm, Part 2: *traversal* threads

```
1  while true do
2      barrier_wait(&barr_start);
3      partition CQ between threads, get my qstart and qend;
4      foreach u ∈ CQ[qstart..qend] in parallel do
5          foreach v adjacent to u do
6              o = getowner(v);
7              if o = rank then
8                  if P[v] = ∞ then
9                      P[v] ← u, NQ ← Enqueue v;
10                     NQ_Count ← NQ_Count + 1;
11             else ENQUEUE(OUTBUF[O],U,V)
12     barrier_wait(&barr_trav);
13     trav_finish = true;
14     barrier_wait(&barr_all);
15     if Sum = 0 then break;
```

Algorithm 4: Enqueue and dequeue functions for a multi-producer single-consumer lock-free queue. ENQUEUE(OUTBUF,U,V): Insert vertices u and v into queue *outbuf*. DEQUEUE(OUTBUF): Empty the queue

Input: u: vetex to be inserted
v: another vertex to be inserted
outbuf: the buffer queue, it uses two variable *count* and *real_count* to record buffer head position and actual vertex number in the queue respectively.
Function: ENQUEUE(OUTBUF,U,V)

```
1  while count ≥ BUFLEN do yield();
2  c = fetch_add(&count, 2);
3  while c ≥ BUFLEN do
4      while count ≥ BUFLEN do yield();
5      c = fetch_add(&count, 2);
6  outbuf[c] = v;
7  outbuf[c + 1] = u;
8  c = fetch_add(&real_count, 2);
```

Function: DEQUEUE(OUTBUF)

```
9  real_count ← 0;
10 count ← 0;
```

too high. Therefore, we need to find another way to implement a highly effective multiple-producer single-consumer queue. In the next subsection we will propose a lock-free multiple-producer single-consumer queue to solve this problem.

– *Lock-free based pipeline parallelism for asynchronous communication*.

The key data structure used in our hybrid BFS algorithm is a multiple-producer single-consumer lock-free queue, which brings asynchrony into parallel BFS and exploits additional pipeline parallelism for communication. As we can see in Fig. 3, multiple *traversal* threads insert outgoing vertices into the queue, while *master* thread is waiting to consume the queue by sending vertices out. The algorithm of the lock-free queue is described in Algorithm 4. The ENQUEUE(OUTBUF,U,V) function first checks whether the buffer is full by repeatedly checking (lines 2–5) the *count* variable. Atomic *fetch_add*() is used here to get a position c in the buffer. After insert the vertices (lines 6–7), *real_count* records the actual vertex count in the buffer. The DEQUEUE() function simply resets *count* and *real_count* to zero, note that *real_count* should be cleared first, then *count*.

5 Experimental results

In this section we report the experimental results of our proposed hybrid BFS algorithm, as well as the lessons we learned.

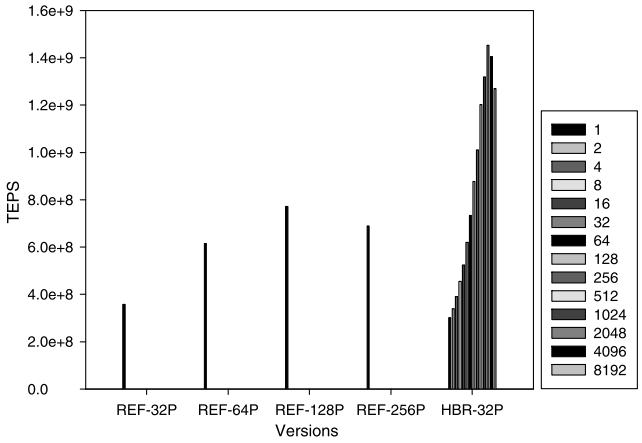

Fig. 4 HBR vs. REF on a 32-node cluster

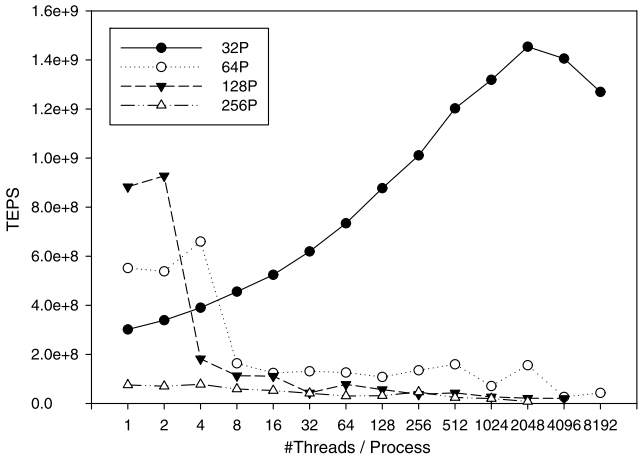

Fig. 5 *HBR* with different process and thread numbers

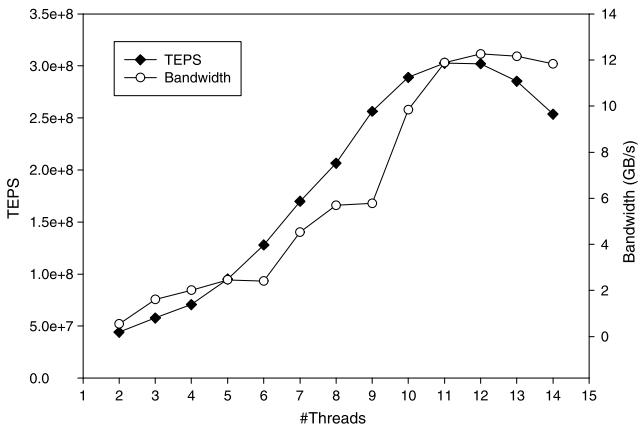

Fig. 6 Bandwidth and TEPS of *HBR* vs. Thread Numbers. Running *HBR* at scale 24 with one process on a single node

Our new hybrid algorithm is implemented with MPI (OpenMPI-1.4.2) and POSIX Pthreads on a standard Linux environment. BFS's performance is evaluated by TEPS, and the graph datasets are described in Sect. 2. Our experiment platform is described in Sect. 2. In all experiments each measurement runs BFS 64 times from different starting vertices and reports the harmonic mean of TEPS. For brevity, we use following abbreviations when plotting the figures: *REF* for the MPI-only BFS algorithm in Graph 500 [1], *HBR* for our hybrid algorithm, and *CBL* for the MPI-only BFS algorithm in Combinatorial BLAS Library [6].

Lesson 1 A careful orchestration of core level parallelism, memory level parallelism and pipeline parallelism, gives a big boost to performance.

Figure 4 plots the performance of MPI-only (REF) and hybrid (HBR) algorithms. The graph data size is scale 30 (1.07 billion vertices). On the left side of the figure are the performance results of *REF* of four different configurations. "REF-32P" denotes running *REF* with 32 processes on our 32-node cluster. The highest TEPS scores of *REF* is 7.72e+08, achieved by "REF-128P". On the right side is the performance results of *HBR* with different number of threads per process. The highest TEPS scores of *HBR* is 1.45e+09, achieved with 2048 threads per process, which is 1.9× faster than *REF*'s best performance. We use 32 MPI processes for *HBR* because our exhaustive experiments (Fig. 5) show that the best configuration is one MPI process per node.

Figure 5 presents performance trend of *HBR* with different configurations. Let process number remain the same "256P" (i.e. 8 processes per node), we see a drop of performance when the number of thread per process increases. However, with "32P" (i.e. 1 process per node), the performance continually increases until the thread number reaches 2048.

Lesson 2 Massive fine-grained thread parallelism not only takes advantage of memory level parallelism in state-of-the-art multi-core architecture, but also improves load balancing.

An important optimization is the use of *massive* number of threads (much more than the number of hardware cores). This finding echoes the results in the previous work [16]: state-of-the-art multi-core processors support memory level of parallelism, e.g., Intel Nehalem's fill buffers allow a maximum of 10 concurrent memory requests. It is one of the most notable features that make a difference between data-intensive graph applications and traditional scientific computing applications. By contrast, if programs are CPU-bound, using more number of threads than the number of physical cores will lead to poor performance. Figure 6 plots the relationship between program bandwidth and number of threads per process. Running *HBR* with one process on a single node, as the number of threads increases, the bandwidth of *HBR* increases. As a result, its TEPS increases cor-

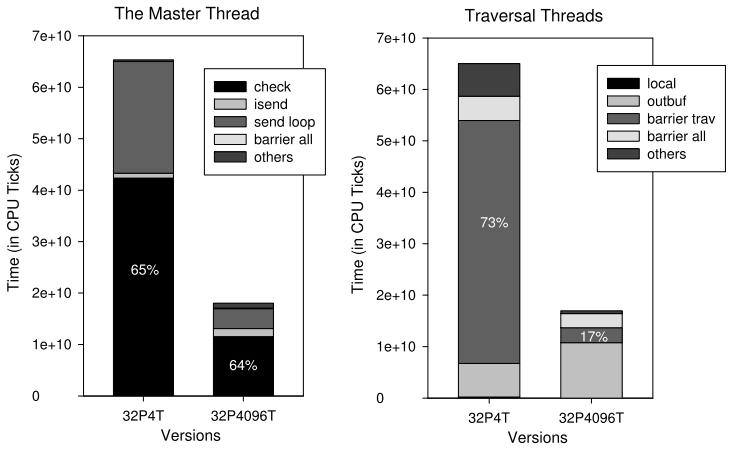

Fig. 7 Time profiling information for the *master* thread and *traversal* threads of *HBR*

respondingly benefiting from the bandwidth increase. There are knees where the performance begins to degrade when the number of threads continues to increase. When the number of threads is larger than some threshold, the overhead of thread context switch can not be mitigated by the improved memory access performance with more concurrent memory requests.

Another reason for massive threads is better load balancing. Massive threading can shorten workload gaps among different threads, i.e., with more threads, each thread is doing less and finishes more quickly. Figure 7 profiles execution time distribution of several main components in our *HBR* program. It compares the case of 4 threads and 2048 threads. On the right side of the figure, the barrier time ("barrier_trav") of *traversal* threads reduces significantly from 73 % to 17 % despite the number of threads waited at the barrier increases. Another benefit is reduced waiting time for buffers. The left side of Fig. 7 gives the time profiling information for the *master* thread. The most time consuming part "check" (lines 4–5 of Algorithm 2) reduces more than a half in absolute value when changed from 4 threads to 2048. As the total communication volume remains the same, it means that the *master* thread spends less time in waiting for the *outbuf* and incoming buffers to become ready.

Lesson 3 Hybrid programming improves scalability of the BFS algorithm.

At a small scale of 384 cores, both *REF* and *HBR* achieve good scalability. Figure 8 plots the weak scalability of *HBR* and *REF* with no more than 384 cores. Fix the graph size on each node to scale 25, experiment on 2 nodes runs scale 26, 4 node runs scale 27, and so on. *REF* use 8 processes per node and *HBR* use 1 process per node, with 1024 threads per process.

At a larger scale of 6,144 cores, *REF* did not finish weak scalability test because its exponential space cost growth in communication buffers leads to memory overflow. Instead, we compare the strong scalability of *HBR* and *REF* at a smaller problem scale in Fig. 9, along with *CBL*, an MPI-only BFS algorithm in Combinatorial BLAS Library [6]. At scale 26, both *HBR* and *CBL* scales better than *REF*.

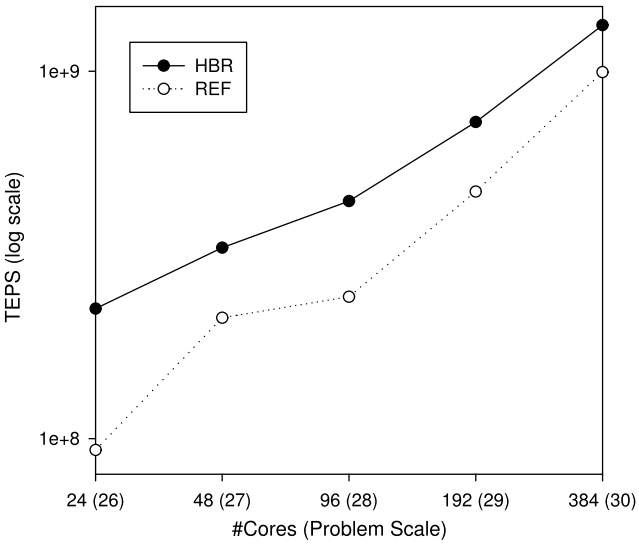

Fig. 8 *HBR* vs. *REF*: weak scalability with no more than 384 cores

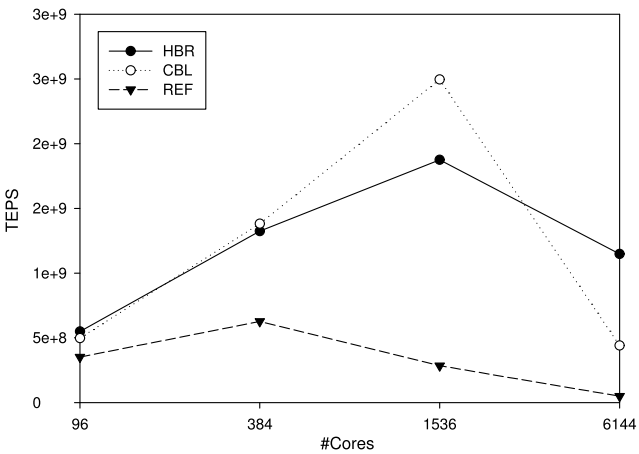

Fig. 9 *HBR*, *CBL* and *REF*: scale 26, strong scaling with 6,144 cores

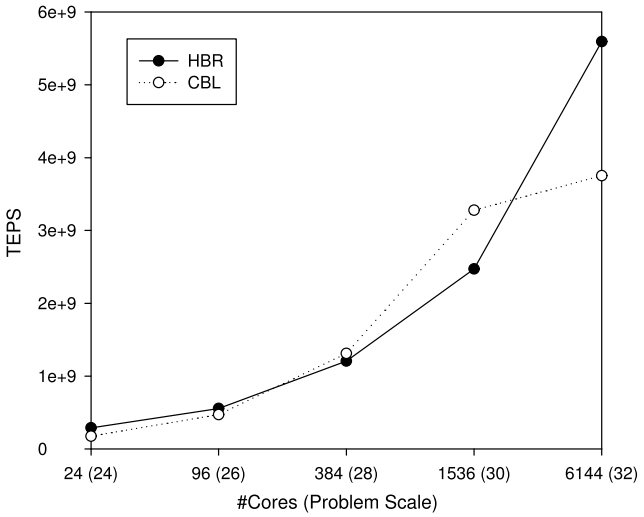

Fig. 10 *HBR* vs. *CBL*: weak scaling with 6,144 cores

Figure 10 compare the weak scalability of *HBR* and *CBL* with 6,144 cores. As we can see in the figure, *HBR* achieves 5.60e+09 TEPS with 6,144 cores at scale 32, 1.49× better than *CBL*.

6 Related works

Earlier works on Cray XMT/MTA [5, 15] and IBM Cyclops-64 [18] prove that both massive threads and fine-grained data synchronization improve BFS performance. With the recent progress of multi-core and SMT, this technique can be popularized to more commodity users. Agarwal et al. [3] achieved performances on Intel Nehalem EP and EX processors comparable to special purpose hardwares like Cray XMT and Cray MTA-2, and first identified the capability of commodity multi-core systems for parallel BFS algorithms. Our algorithm also proves the feasibility of using massive threads for BFS on commodity processors. As for asynchrony, the similar idea of asynchronous algorithm is also used in optimizing communication between SPE and SPU for running BFS on STI CELL processors [17].

Buluç and Madduri [7] managed to run hybrid BFS modified from Graph 500 on a 40,000-core machine. They use a improved 2D partition to reduce communication overhead, which based on Yoo et al.'s work [20] on BlueGene/L. It is a significant work to run BFS at such large scale. However, the performance of the 2D hybrid version is roughly the same as the 2D MPI-only version. The major disadvantage of MPI/OpenMP is its BSP like synchronization. In this paper we provide a new approach to overlap communication and computation by introducing a multiple-producer single-consumer lock-free queue between computation and communication threads.

Lock-free algorithms design requires deep understanding of an algorithm and the underlying system. Bader and Cong [4] developed a lock-free algorithm for computing the minimum spanning forest (MSF) of sparse graphs on symmetric multiprocessors. Kang and Bader [10] further developed a transactional memory (TM) algorithm for it. Leiserson and Schardl [11] also use a lock-free data structure as a key component in their work-efficient parallel BFS. Agarwal et al.'s algorithm [3] use a lock-protected single-producer single-consumer lock-free queue based on the FastForward [9] algorithm. We modify that into multiple-producer single-consumer lock-free queue using atomic instructions instead of locks.

7 Conclusion

In this paper we propose a new hybrid breadth-first search algorithm, and present a comprehensive comparison with the MPI-only version. Our experimental analysis leads to several findings, which would be valuable for other researchers who want to understand and exploit parallelism in parallel BFS on multi-core clusters. The hybrid BFS algorithm *exploits three kind of parallelism: core level, memory level, and pipeline level*. Experiments show our algorithm outperforms the MPI-only algorithm significantly. One important lesson we learned in the experiments is to use massive number of threads to take advantage of memory level parallelism in state-of-the-art multi-core processors. We chose Pthreads as our thread library and managed to run our program with thousands of threads per node. However, for fine-grained programs like BFS, the overhead of thread context switch is still too large. For future works, we believe that a user level lightweight threading mechanism will increase the performance further.

Acknowledgements The authors gratefully acknowledge Erlin Yao and the anonymous reviewers for their helpful comments on previous drafts of this work.

This work is supported by National 863 Program (2009AA01A129), the National Natural Science Foundation of China (61003062, 60925009, 60921002, 60803030, 61033009, 60921002, and 60925009) and 973 Program (2011CB302502 and 2011CB302500).

References

1. The Graph 500 List (2011). http://www.graph500.org/
2. The Linpack Benchmark (2011). http://www.top500.org/project/linpack
3. Agarwal V, Petrini F, Pasetto D, Bader DA (2010) Scalable graph exploration on multicore processors. In: Proceedings of the 2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis, SC'10. IEEE Comput Soc, Washington, pp 1–11
4. Bader DA, Cong G (2006) Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. J Parallel Distrib Comput 66:1366–1378

5. Bader DA, Madduri K (2006) Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. In: Proceedings of the 2006 international conference on parallel processing, ICPP'06. IEEE Comput Soc, Washington, pp 523–530
6. Buluç A, Gilbert JR (2011) The Combinatorial BLAS: design, implementation, and applications. Int J High Perform Comput Appl. doi:10.1.1.185.4283
7. Buluç A, Madduri K (2011) Parallel breadth-first search on distributed memory systems. Corros Rev. arXiv:1104.4518
8. Cappello F, Etiemble D (2000) MPI versus MPI+OpenMP on IBM SP for the NAS Benchmarks. In: Proceedings of the 2000 ACM/IEEE conference on supercomputing (CDROM), Supercomputing'00. IEEE Comput Soc, Washington
9. Giacomoni J, Moseley T, Vachharajani M (2008) FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In: Proceedings of the 13th ACM SIGPLAN symposium on principles and practice of parallel programming, PPoPP'08. ACM, New York, pp 43–52
10. Kang S, Bader DA (2009) An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs. ACM SIGPLAN Not 44:15–24
11. Leiserson CE, Schardl TB (2010) A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In: Proceedings of the 22nd ACM symposium on parallelism in algorithms and architectures, SPAA'10. ACM, New York, pp 303–314
12. Leskovec J, Chakrabarti D, Kleinberg J, Faloutsos C (2005) Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication. In: Jorge A, Torgo L, Brazdil P, Camacho R, Gama J (eds) Knowledge discovery in databases: PKDD 2005. Lecture notes in computer science, vol 3721. Springer, Berlin, pp 133–145
13. Loft RD, Thomas SJ, Dennis JM (2001) Terascale spectral element dynamical core for atmospheric general circulation models. In: Proceedings of the 2001 ACM/IEEE conference on supercomputing (CDROM), Supercomputing'01. ACM, New York, p 18
14. Lumsdaine A, Gregor D, Hendrickson B, Berry J (2007) Challenges in parallel graph processing. Parallel Process Lett 17(1):5–20
15. Mizell D, Maschhoff K (2009) Early experiences with large-scale Cray XMT systems. In: Proceedings of the 2009 IEEE international symposium on parallel & distributed processing. IEEE Comput Soc, Washington, pp 1–9
16. Molka D, Hackenberg D, Schone R, Muller MS (2009) Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In: Proceedings of the 2009 18th international conference on parallel architectures and compilation techniques. IEEE Comput Soc, Washington, pp 261–270
17. Scarpazza DP, Villa O, Petrini F (2008) Efficient breadth-first search on the Cell/BE processor. IEEE Trans Parallel Distrib Syst 19:1381–1395
18. Tan G, Sreedhar V, Gao G (2011) Analysis and performance results of computing betweenness centrality on IBM Cyclops64. J Supercomput 56:1–24
19. Wu X, Taylor V (2011) Performance characteristics of hybrid MPI/OpenMP implementations of NAS parallel benchmarks SP and BT on large-scale multicore supercomputers. ACM SIGMETRICS Perform Eval Rev 38:56–62
20. Yoo A, Chow E, Henderson K, McLendon W, Hendrickson B, Catalyurek U (2005) A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In: Proceedings of the 2005 ACM/IEEE conference on supercomputing, SC'05. IEEE Comput Soc, Washington, p 25

Huiwei Lv is a Ph.D. student at State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences. His research interests include High Performance Computing and Parallel Simulation of Many-core Processors.

Guangming Tan became Associate Professor at National Research Center for Intelligent Computing Systems, ICT, CAS in 2010. His research interests include Parallel algorithms and programming on Multi/Many-core architectures.

Mingyu Chen is a Professor at State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences. His research interests include HPC Architecture, Operating System and Parallel Computing.

Ninghui Sun is the Director of the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include Parallel Architecture and Distributed Operating System.