International Journal of
HIGH PERFORMANCE
COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
1–12
© The Author(s) 2014
Reprints and permissions:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/1094342014524240
hpc.sagepub.com
SAGE

Using the Intel Many Integrated Core to accelerate graph traversal

Tao Gao, Yutong Lu, Baida Zhang and Guang Suo

Abstract
Data-intensive applications have drawn more and more attention in the last few years. The basic graph traversal algorithm, the breadth-first search (BFS), a typical data-intensive application, is widely used and the Graph 500 benchmark uses it to rank the performance of supercomputers. The Intel Many Integrated Core (MIC) architecture, which is designed for highly parallel computing, has not been fully evaluated for graph traversal. In this paper, we discuss how to use the MIC to accelerate the BFS. We present some optimizations for native BFS algorithms and develop a heterogeneous BFS algorithm. For the native BFS algorithm, we mainly discuss how to exploit many cores and wide-vector processing units. The performance of our optimized native BFS implementation is 5.3 times that of the highest published performance for graphics processing units (GPU). For the heterogeneous BFS algorithm, the performance of the general processing unit (CPU) and MIC cooperative computing can gain an increase in speed of approximately 1.4 times than that of a CPU for graphs with 2M vertices. This work is valuable for using a MIC to accelerate the BFS. It is also a general guidance for a MIC used for data-intensive applications.

Keywords
Many Integrated Core, MIC, breadth first search, BFS, graph traversal, heterogeneous computing, Graph 500

1. Introduction

In the last several decades, with the comprehensive intrusion of computer technology into human life, the information explosion has produced a large amount of data. There exists a huge challenge in the field of “big data” analytics (Manyika et al., 2011). Graph data structure is used to model the relationships between data. As a result, graph algorithms, a challenging problem (Lumsdaine et al., 2007), attract the attention of many researchers. As basic graph algorithms, graph traversal is widely used in numerous applications, for example in cybersecurity, medical information, data enrichment, social networks, symbolic networks and so on (Brette et al., 2007; Albert et al., 1999; Dodds et al., 2003). The breadth-first search (BFS) is a typical graph traversal algorithm. Recently, the BFS is used as the kernel component of the Graph 500 benchmark, which is used to rank the performance of supercomputers used for data-intensive applications (Murphy et al., 2010).

Because graph traversal is a key component for many of these applications, the Graph 500 benchmark focuses its attention on the BFS search of a particular class of graphs, the recursive matrix (R-MAT) scale-free graphs (Chakrabarti et al., 2004; Leskovec et al., 2010). In this benchmark, the *SCALE* and *edgefactor* parameters can be assigned. The generated graphs have 2^{scale} vertices and $edgefactor \bullet 2^{scale}$ undirected edges. As the graphs are undirected, the average degree is $2 \bullet edgefactor$. It uses the traversed edges per second (TEPS) to measure the performance.

Heterogeneous integration system architecture, which contains both general processing units (CPU) and acceleration units (graphics processing units (GPU), field-programmable gate array (FPGA), many integrated core (MIC) and so on), is an important innovation in the field of high-performance computing (Yang et al., 2010). While heterogeneous integration systems can increase the performance, it brings some challenges for program design. This article focuses on the efficient implementation of the BFS based on heterogeneous integration systems with CPUs and MICs.

The Intel MIC architecture was announced in 2010 as a massive parallel co-processor (Duran and Klemm,

State Key Laboratory of High Performance Computing, China
School of Computer Science, National University of Defense Technology, China

Corresponding author:
Yutong Lu, State Key Laboratory of High Performance Computing/School of Computer, National University of Defense Technology, Changsha, Hunan, 410073, China.
Email: ytlu@nudt.edu.cn

2011). Knights Ferry, the first generation MIC product, follows different design principles than other accelerators. While other accelerators are based on specialized hardware, the Intel MIC architecture is a many-core co-processor based on the Intel architecture (IA). Knights Corner is the second generation of the Intel MIC product family. It comprises up to 62 IA cores connected by a high-performance on-die bi-directional interconnect. In addition to the IA cores, there are 8 memory controllers supporting 16 graphics double date rate, version 5 (GDDR5) channels expected to provide a theoretical bandwidth of 352 gigabytes per second (GB/s) and special function devices including the PCI Express system interface.

In the last few years, many studies about the BFS have been conducted on distributed-memory systems (Yoo et al., 2005; Buluç and Madduri, 2011; Satish et al., 2012; Checconi et al., 2012), shared-memory systems (Bader and Madduri, 2006; Chhugani et al., 2012; Agarwal et al., 2010; Beamer et al., 2012) and GPUs (Merrill et al., 2012). In this study, we concentrate on using a MIC to accelerate the BFS. Some optimizations are presented for the native BFS algorithm which uses only a MIC. In addition, we also propose a heterogeneous BFS algorithm which supports the cooperative computing of the CPU and accelerators.

The specific contributions of this paper are as follows:

- We present some optimizations of the BFS on a MIC, which has many cores and wide-vector processing units. We mainly discuss how to make full use of 512-bits single instruction multiple data (SIMD) instructions to gain a performance improvement on the MIC.
- We propose a heterogeneous BFS algorithm, which combines the computing of the CPU and accelerators. The MIC is used to evaluate the heterogeneous algorithm. The experimental result proves that this algorithm can increase the speed of large-scale graphs.
- Our work is a general guidance for those who want to use a MIC to accelerate data-intensive applications.

This paper is organized as follows: section 2 discusses the general parallel BFS algorithms, section 3 describes optimizations for native BFS algorithms, section 4 describes the heterogeneous BFS algorithm, section 5 shows and analyzes the experimental results, related work is discussed in section 6 and the conclusion and future work are discussed in section 7.

2. BFS algorithms

A graph $G(V, E)$ is composed of a vertex set V and an edge set E. Given a graph $G(V, E)$ and a source vertex $r \in V$, the BFS algorithm explores all vertices reachable from r and produces a breadth-first spanning tree.

A graph can be represented by an adjacency matrix A, whose rows are the adjacency lists A_i. The well-known compressed sparse row (CSR) or compressed sparse column (CSC) sparse matrix format can be used to store the graph in the memory. The CSR storage format consists of two arrays: *rowstarts* and *column*, in which *column* is formed from the set of the adjacency lists concatenated into a single array and *rowstarts[i]* is the index of adjacency list A_i in the column. The CSC has a similar storage format, except that it compresses the sparse matrix according to the column. For an undirected graph, the CSR and CSC storage formats are the same.

Most parallel BFS algorithms are level-synchronized; all vertices at one level are processed before any vertices further from the source vertex. The algorithm keeps an active vertex set called the *frontier* and for each iteration explores all the vertices reached from the frontier in one step, which will form the frontier for the next iteration. The level-synchronized BFS algorithm framework is shown in algorithm 1. The frontier of this level and the frontier of the next level are represented by *in* and *out* respectively. To reduce the working set size, a bitmap *vis* is used to mark the vertices during the visit (Agarwal et al., 2010). The spanning tree is stored in the predecessor map p, with $p[v_0]$ recording the predecessor of vertex v_0.

The exploration begins initializing the frontier with the source vertex (line 1); it also marks the visited vertex (line 2) and the predecessor is recorded (line 3, we define the source vertex's predecessor as itself similar to the Graph 500 benchmark). Then, the algorithm iterates until the frontier becomes empty (lines 4–7). For each iteration, two strategies can be used to expand the frontier (***one-level-step*** function in line 6): (1) *top-down* and (2) *bottom-up*.

2.1. Top-down BFS

The *top-down* method is shown in algorithm 2. All vertices adjacent to the vertices in the frontier are scanned (lines 1 and 2). If an unvisited vertex is found (line 3), it

Algorithm 1: Level-synchronized BFS

```
Input : $G(V,E)$, $r$ // source vertex
Output : $p$ // predecessor map
1 $in \leftarrow \{r\}$
2 $vis \leftarrow \{r\}$
3 $p[r] \leftarrow r$
4 while $in \neq \emptyset$ do
5     $out \leftarrow \emptyset$
6     one-level-step ($in, out, vis, p$)
7     swap ($in, out$)
```

Algorithm 2: top-down-step(*in, out, vis, p*)

```
1 for $v_0 \in in$ do
2   for $v_1$ is adjacent to $v_0$ do
3     if $v_1 \notin vis$ then
4       $out \leftarrow out \cup v_1$
5       $vis \leftarrow vis \cup v_1$
6       $p[v_1] \leftarrow v_0$
```

Algorithm 3: bottom-up-step(*in, out, vis, p*)

```
1 for $v_1 \in vis$ do
2   for $v_0$ is connected to $v_1$ do
3     if $v_0 \in in$ then
4       $out \leftarrow out \cup v_1$
5       $vis \leftarrow vis \cup v_1$
6       $p[v_1] \leftarrow v_0$
7       break
```

is marked as visited (line 5) and added to *out* (line 4). The predecessor is also labeled (line 6). To access the neighbors of a vertex quickly, the CSR format is often used to store the graph. Most work of this algorithm is in checking the edges that started from vertices in the frontier. Every edge in the connected component containing the source vertex should be visited once.

2.2. Bottom-up BFS

The *bottom-up* method is shown in algorithm 3. Instead of vertices in the frontier attempting to become the predecessor of its neighbors, each unvisited vertex tries to find any predecessor among its neighbors (Beamer et al., 2012). For each unvisited vertex, the edges connected to it are scanned (lines 1 and 2). If the other endpoint of the edge is in the frontier (line 3), the vertex is marked as visited (line 5) and added to *out* (line 4). The predecessor is also labeled (line 6). We often use a bitmap to represent the frontier for testing the vertex in line 3 quickly. To access the edges connected to a vertex quickly, the CSC format is often used to store the graph. The algorithm is efficient if it can break as soon as possible in line 7. It has the potential to scan fewer edges than the *top-down* algorithm.

2.3. Hybrid BFS

The *top-down* method and *bottom-up* method are complementary; when the size of the vertex frontier is small, the *top-down* method is better than the *bottom-up* method, and vice versa (Beamer et al., 2012). For graphs with *small-world* (Watts and Strogatz, 1998) and *scale-free* (Barabási and Albert, 1999) properties,

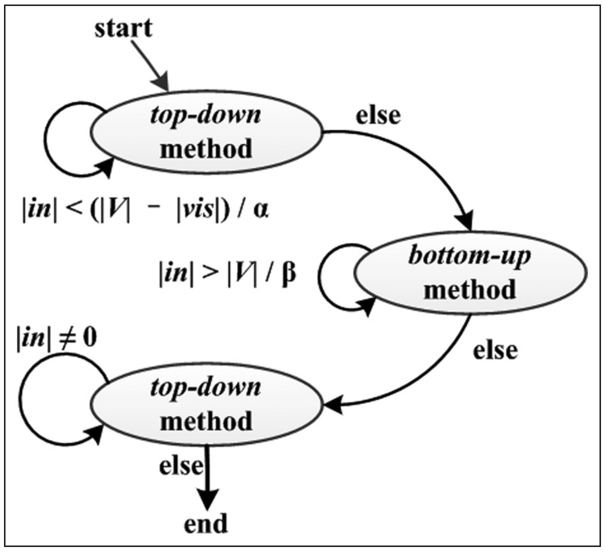

Figure 1. The state machine of the *hybrid* BFS algorithm.

the size of the frontier ramps up and down exponentially during the BFS exploration (Beamer et al., 2012). As a result, the *hybrid* BFS algorithm is proposed as shown in Figure 1.

The $|V|$, $|in|$ and $|vis|$ represent the vertex number of the graph, the frontier and *vis* respectively. When the frontier size is larger than $(|V| - |vis|) / \alpha$, the algorithm converts from *top-down* to *bottom-up*; when the frontier size is smaller than $|V| / \beta$, the algorithm converts from *bottom-up* to *top-down*.

Both bitmaps and queues can be used to represent *in* and *out*; we use the bitmap-based method, as it often has better memory access locality. The algorithmic complexity of the queue-based method is close to $O(|V| + |E|)$ (Buluç and Madduri, 2011), while the complexity of the bitmap-based method may reach $O(|V|^2)$ when the graph is a line. However, for small-world graphs, because the diameter is small, the bitmap-based method is practically effective.

3. Native BFS algorithm optimizations

In this section, we describe native BFS algorithm optimization methods. The optimizations for both *top-down* and *bottom-up* BFSs are discussed.

3.1. Top-down BFS optimizations

In the *top-down* BFS algorithm, at least two levels of parallelism can be exploited as follows: (1) all vertices in the frontier can be processed simultaneously and (2) the adjacencies of each vertex can be inspected in parallel (Bader and Madduri, 2006). We can use multi-thread to exploit the former parallelism and SIMD to make use of the latter fine-grained parallelism.

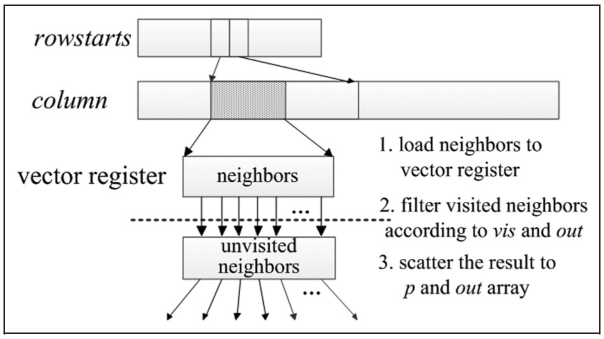
rowstarts
column
vector register
neighbors
unvisited neighbors
1. load neighbors to vector register
2. filter visited neighbors according to *vis* and *out*
3. scatter the result to *p* and *out* array

Figure 2. The sketch of the top-down BFS with SIMD.

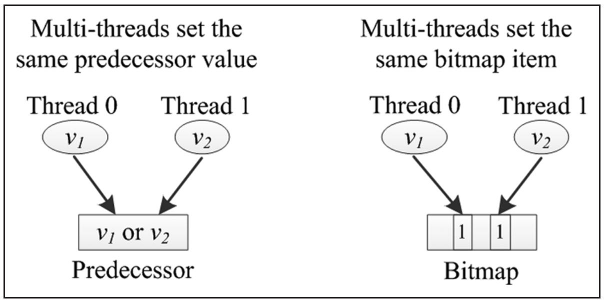

Figure 3. The two data race conditions.

Algorithm 4: atomic-top-down-step(in, out, vis, p)

```
1 for $v_0 \in in$ parallel do
2    for $v_1$ is adjacent to $v_0$ do
3       if $v_1 \notin (vis \cup out)$ then
4          if $\_\_fetch\_and\_or(out, v_1) = 0$ then
5             $p[v_1] \leftarrow v_0$
6 $vis \leftarrow vis \cup out$
```

The multi-thread implementation of the top-down BFS is shown in algorithm 4. All vertices in the frontier are processed simultaneously as shown in line 1. When modifying *out*, the atomic operation is used to make sure it is correct (line 4) (Agarwal et al., 2010). To reduce the atomic operations, *vis* is set last (line 6). By first checking whether the vertex has already been visited in line 3 (Agarwal et al., 2010), the potentially expensive atomic operation can be reduced up to 95%.

As we know, there is no SIMD atomic instruction in a MIC, so the use of SIMD instructions need further consideration. Inspecting the adjacencies of each vertex can be divided into three steps: (1) load the neighbors, (2) filter the visited vertices and (3) set the results. The sketch of the SIMD implementation is shown in Figure 2. The first two steps can use SIMD directly. However, there exists data race in step 3.

In step 3, *p* and *out* are set. The predecessor value is represented by an integer and *out* is represented by a bitmap. The two different data race conditions are shown in Figure 3. If 8/16/32/64-bits are used to represent the predecessor values, the update to *p* is always consistent (Chhugani et al., 2012) as shown in the left part of Figure 3. The data race to the bitmap, as shown in the right part of Figure 3, may result in an incorrect result, since the bit for v_1 or v_2 may be lost. However, the predecessor map is always a valid spanning tree, although it may be different in different travels. As a result, after scanning the neighbors using the SIMD, the predecessor map is valid; however, the bitmap *out* is not correct. We can label the newly discovered vertices by storing the negative numbers of their predecessor values. At last, this information is used to restore *out*.

Table 1. The SIMD usage rate (1M vertices with different edgefactors).

level \ edgefactor	8	16	32	64
1	95.72%	99.60%	99.77%	99.84%
2	98.04%	95.91%	96.42%	97.05%
3	68.43%	51.88%	50.67%	48.94%
4	11.23%	7.20%	6.68%	6.43%
5	6.33%	6.25%	6.25%	6.25%

Algorithm 5: restore the out array

```
1 for $v \in out$ do
2    for $v_1$ whose bitmap bit in the same integer with $v$ do
3       if $p[v_1] < 0$ then
4          $out \leftarrow out \cup v_1$
5          $p[v_1] \leftarrow -p[v_1]$
```

The restoring process of *out* is shown in algorithm 5. The predecessor map is used to restore *out*. The vertices in *out* are scanned (line 1). The predecessor value of the vertex whose bit in the same integer in the bitmap are checked (the data race only influences the bits in the same integer) (line 2). If the predecessor value is negative (line 3), *out* and *p* are restored (line 4 and 5).

The SIMD implementation can increase the speed for three reasons. First, the filter operation in step 2 eliminates most of the duplicate vertices (more than 95%) which have been visited in the previous levels or by other threads in this level. This greatly reduces the data race. Second, the *prefetch* operations for vector components can reduce the memory access latency, and finally the restoring process efficiently accesses the memory in sequence.

The SIMD usage rate can be defined as the effective operation rate, because some vertices do not have enough neighbors to fully utilize the 16-way vector registers (the SIMD is 512-bits wide and 32-bits integers are used to represent vertices). For instance, if a vertex has only one neighbor, then the SIMD usage rate is

Table 2. The number of the edges scanned per visited vertex.

level \ method	Bottom-up	Top-down
1	524.62	3.62
2	4.49	33.66
3	1.09	125.00
4	1.18	259.06
5	47.98	307.50
6	7201	219.00

6.25% (1/16). Table 1 shows the SIMD usage rate for an example graph with 1M vertices (The usage rate of level 0 is discarded, since it totally depends on the source vertex). From this table, it is clear that the first several levels have extremely high usage rates, while the last few levels have very low usage rates. This is because a vertex has a higher probability of being found in the first several levels if it has more neighbors. As a result, we can only apply the SIMD instructions to the first several levels.

3.2. Bottom-up BFS optimizations

The *bottom-up* BFS algorithm can be paralleled easily. Bitmaps are used to represent *in* and *out*. Every thread handles different vertices. If any neighbors of an unvisited vertex are in the frontier, the vertex is marked as visited. The thread-level parallelism can be exploited in this way. However, making use of the SIMD instructions to gain an increase in the performance improvement needs further discussions.

In level-synchronized BFS, a metric (*the number of edges scanned per visited vertex*), which is the total checked edge count divided by the visited vertex number in one level, can be used to estimate algorithm efficiency for each level. When the metric value is smaller, the algorithm is more efficient. Table 2 is the average value for 64 times with 1M vertices and 16M undirected edges. From this table, we can see that the *bottom-up* BFS is quite efficient in the middle levels, while quite inefficient in the first and last few levels. As a result, the *hybrid* algorithm can combine the advantages of these two strategies.

In fact, the metric is quite small (less than 5) in the middle levels, which means that most vertices find predecessors within a few iterations. Also, the middle levels account for most of the total expanded vertices (more than 90%), which implies many vertices are visited in one level. Based on these observations, SIMD instructions can be used to handle 16 vertices simultaneously in the middle levels.

The basic idea is shown in Figure 4. Each time, the SIMD is used to handle 16 vertices. The unvisited vertices are handled in a *while* cycle. In each iteration, two steps are performed as follows: (1) neighbors of the unvisited vertices are gathered to a vector register and

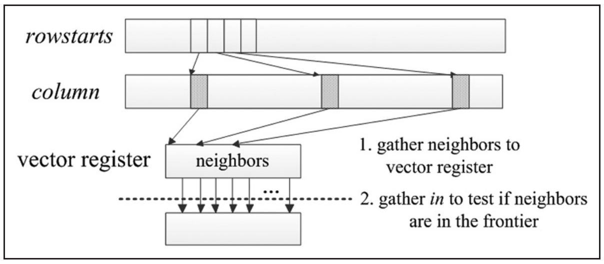
rowstarts
column
vector register
neighbors
1. gather neighbors to vector register
2. gather *in* to test if neighbors are in the frontier

Figure 4. The sketch of the bottom-up BFS with SIMD.

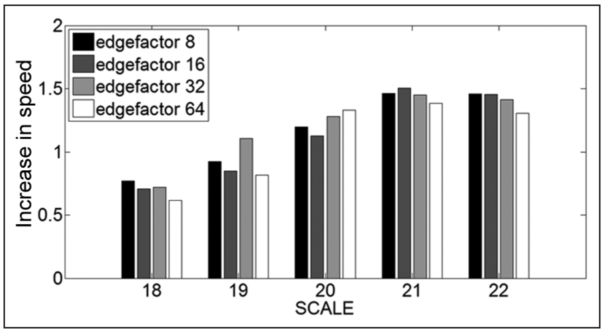

Figure5. The expanded vertex count in each level.

(2) the corresponding *in* values are gathered to test if neighbors are in the frontier. The process repeats until all vertices are visited or all neighbors have been scanned. To gain speed, it requires two conditions for a level: (1) many vertices have not been visited and (2) most vertices are found to be visited in only a few iterations. According to the analysis above, the middle levels satisfy the two conditions. However, some exceptions need careful consideration to avoid extra overhead.

Obviously, there are two exception conditions: (1) for some ranges, most vertices may have been visited and (2) some vertices may iterate far more times than the average value. Both these conditions reduce the SIMD usage rate. To solve these problems, some thresholds can be set. The SIMD instructions are used only when the unvisited vertex count exceeds a threshold. The iteration exits if the end condition is satisfied or if the iteration times exceed another threshold, with the remaining computing handled without SIMD.

4. Heterogeneous BFS algorithm

In this section, we discuss the BFS algorithm for the heterogeneous node that contains both CPUs and accelerators (our research is based on a MIC, but this algorithm is not limited to a MIC).

4.1. Motivation

Figure 5 shows the expanded vertex count in each level for graphs with 1M vertices and 16M undirected edges (note that the y-axis is logarithmic). From Figure 5, we

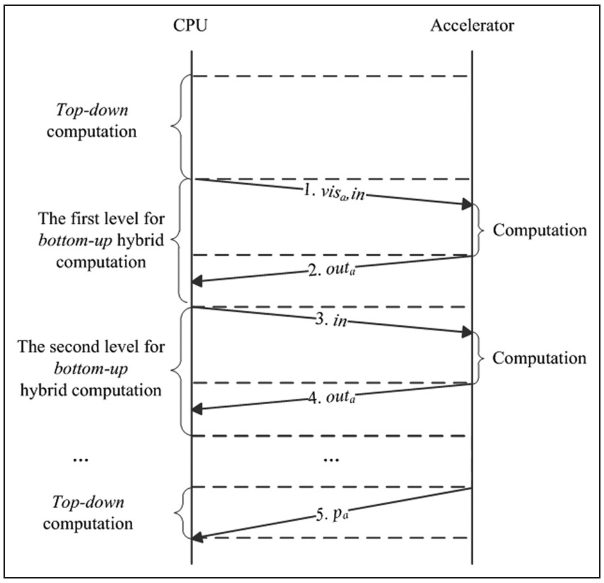

Figure 6. The time chart for the CPU and accelerator.

know that most vertices are expanded in the middle levels, which means the middle levels consume most of the computational time. As a result, accelerators can be used to accelerate the middle levels.

In the *hybrid* BFS algorithm, the *bottom-up* strategy can be applied to the middle levels. The CPU and accelerator can simply be assigned to handle different vertices for cooperative computing.

4.2. Algorithm outline

The state machine of the heterogeneous algorithm has the same framework with the *top-down* and *bottom-up* hybrid method as shown in Figure 1, except that both the CPU and accelerator participate in the *bottom-up* computation.

The CPU and accelerator can be assigned to handle different vertices in the *bottom-up* method and update different ranges for *out, vis* and *p* in algorithm 3. out_c, vis_c and p_c represent the part in charge by the CPU; out_a, vis_a and p_a denote the part handled by the accelerator. The accelerator only stores part of the graph denoted by $G(V_a, E_a)$.

The time chart for the CPU and accelerator is shown in Figure 6. When converting from the *top-down* method to the *bottom-up* method, vis_a is transferred to the accelerator. In each cooperative computing level, the CPU sends *in* and receives out_a. When converting from the *bottom-up* method to the *top-down* method, p_a is transferred back. All data transmission is asynchronous. Our design goal is to keep the CPU as busy as possible. The computation of the CPU is overlapped with the communication; however, the accelerator must wait for the data before computing.

The algorithm for the CPU is shown in algorithm 6. At first, the *top-down* strategy is applied (lines 1–8).

Algorithm 6: Heterogeneous BFS Algorithm (CPU)

```
Input : $G(V,E)$, $r$ // source vertex
Output: $p$ // predecessor map
1  $in \leftarrow \{r\}$
2  $vis \leftarrow \{r\}$
3  $p[r] \leftarrow r$
4  repeat
5      out ← Ø
6      top-down-step ($in, out, vis, p$)
7      $in \leftarrow out$
8  until $|in| > (|V| - |vis|) / \alpha$
9  start-send-transfer ($vis_a$)
10 repeat
11     start-send-transfer ($in$)
12     send-start-message ()
13     out ← Ø
14     bottom-up-step ($in, out_c, vis_c, p_c$)
15     recv-end-message ()
16     wait-transfer-end ($out_a$)
17     $in \leftarrow out$
18     $vis \leftarrow vis \cup out$
19 until $|in| < |V| / \beta$
20 start-recv-transfer ($p_a$)
21 repeat
22     $out \leftarrow \emptyset$
23     top-down-step ($in, out, vis, p$)
24     $in \leftarrow out$
25 until $|in| = \emptyset$;
26 wait-transfer-end ($p_a$)
```

Before converting to the *bottom-up* method, it starts the asynchronous transmission of vis_a to the accelerator (line 9). In each bottom-up level, the asynchronous transmission of *in* is started first (line 11). Then a blocked *start* message is sent to notify the accelerator that *in* has begun to transfer (line 12). After that, the CPU handles its own task (lines 13 and 14). When the *end* message (line 15) is received, the CPU waits until the transmission of out_a ends (line 16). To start a new level, *in* and *vis* are updated according to *out* (lines 17–18). vis_a is just sent once, since the accelerator can keep it up-to-date in its memory. Before switching to the *top-down* method again, the asynchronous transmission of p_a is started (line 20). At the end, we should wait until the transmission of p_a ends (line 26).

The algorithm for the accelerator is shown in algorithm 7. When the *start* message (line 2) is received, the accelerator waits until the transmission of *in* ends (line 3). Then the *bottom-up* method is performed to travel the graph $G(V_a,E_a)$ (line 4). After finishing the computation, it starts asynchronous transmission of out_a (line 5) and then sends the *end* message to notify the CPU that the computation has finished (line 6).

Algorithm 7: Heterogeneous BFS Algorithm (Accelerator)

```
Input : $G(Va, Ea)$
1 repeat
2     recv-start-message ()
3     wait-transfer-end ($in$)
4     bottom-up-step ($in, out_a, vis_a, p_a$)
5     start-send-transfer ($out_a$)
6     send-end-message ()
7 until
```

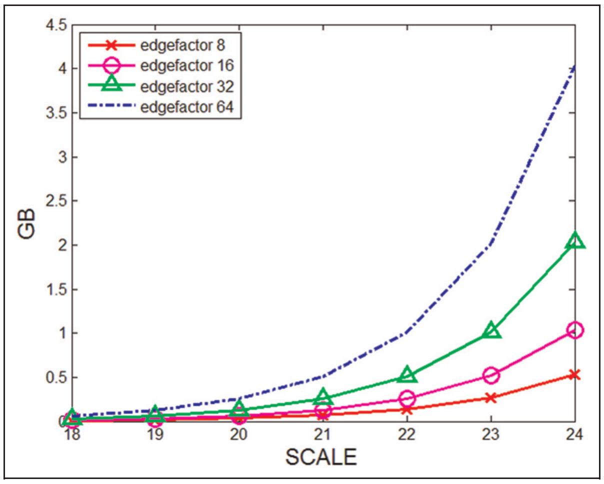

Figure 7. The memory size of graphs on a MIC (task partition ratio 1:1).

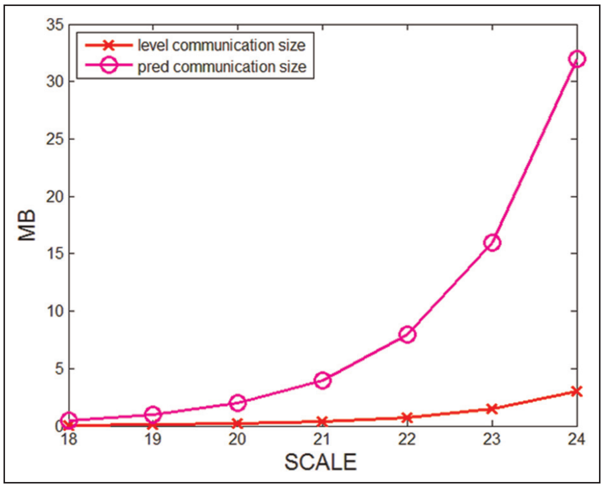

Figure 8. The communication traffic size (task partition ratio 1:1).

Converting the above algorithm to support multi-accelerators is simple. We can simply assign the CPU and accelerators to handle different vertices. For example, if we have three accelerators, the vertices are divided into four parts. The CPU is in charge for one part, and each accelerator is responsible for a part. The CPU should transfer corresponding input data to all the accelerators and receive output data from all the accelerators.

4.3. Algorithm analyses

SCALE and *edgefactor* are represented by s and f, respectively. Then the vertex number n is 2^s and the average degree d is $2f$. Assume the vertices are represented by 32-bits integers. For simplicity, suppose k and l are two parameters for task partition. We assign $k / 2^l$ vertices to an accelerator. That is to say, an accelerator is in charge of $k \bullet 2^{s-l}$ vertices. We denote the task partition ratio as the vertex number handled by the CPU divided by the vertex number handled by the accelerator. For example, if $k = l = 1$, the task partition ratio is 1:1.

Using the CSC storage format, the memory size of the graph in an accelerator is approximately $4 \bullet (k \bullet 2^{s-l} + 1) + 4 \bullet 2f \bullet k \bullet 2^{s-l}$ bytes. Figure 7 illustrates the total memory size of the graph on a MIC with a 1:1 task partition ratio ($k = l = 1$). Compared with the CPU, a MIC has a relatively small memory size. The memory size determines the maximum scale of the graphs that a MIC can handle. The MIC we used has about 8 GB of memory. If the *edgefactor* is 64, it can handle graphs with 2^{24} vertices at most, as the total memory size will be more than 8 GB for graphs with 2^{25} vertices.

Let us calculate the communication traffic now. vis_a and out_a are $k \bullet 2^{s-l-3}$ bytes and in is 2^{s-3} bytes. p_a is $4k \bullet 2^{s-l}$ bytes. In a travel, we need to transfer vis_a and p_a one time and transfer in and out_a in each cooperative computing level. For each heterogeneous level, the total communication size is $2^{s-3} + k \bullet 2^{s-l-3}$ bytes (the first cooperative level has $k \bullet 2^{s-l-3}$ bytes extra communication size). At last, a $4k \bullet 2^{s-l}$ byte predecessor map should be transferred back. Figure 8 illustrates the communication size with a 1:1 task partition ratio ($k = l = 1$). For graphs with 2^{24} vertices, the communication size for each level is approximately 3 MB and the predecessor map is approximately 32 MB. PCI Express 2.0 provides bi-directional bandwidth of about 6.6 GB/s. The overhead for the communication of the predecessor map is about 4.73 ms. This overhead can be hidden with the computation of the last several levels. In each cooperative computing level, the communication overhead is small, approximately 0.44 ms.

In fact, for the last several levels of *bottom-up* computing, the communication overhead is too large when compared with the small amount of computation. As a result, we can change the above algorithm a little. At first, the *bottom-up* levels are calculated by the CPU and the accelerator. When the frontier size is smaller than $|V| / \gamma$ (γ is another parameter), only the CPU participates in the bottom-up computing to avoid the overhead of the heterogeneous computing level.

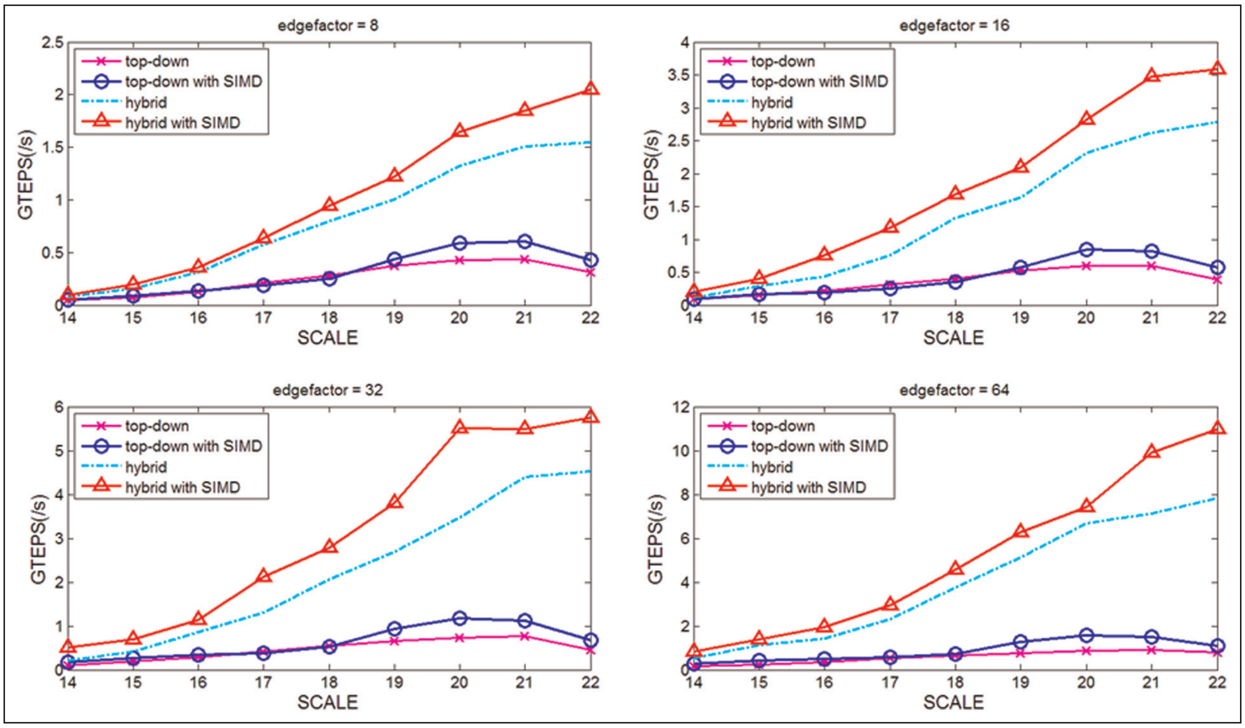

Figure 9. Native BFS algorithm optimization result. (GTEPS: Giga TEPS)

5. Experimental results

In this section, we present experimental results and analyses. The CPU we used is an Intel Xeon E5-2692 and its clock rate is 2.20 GHz. Each CPU has 12 cores and each core has a 32 KB L1 cache and a 256 KB L2 cache. The 30 MB L3 cache is shared. The MIC we used has 57 cores running at 1.1 GHz. Each core can run four hardware threads. There are 32 KB L1 cache and 512 KB L2 cache per core.

The Graph 500 graph generator is used to create the graphs and R-MAT probabilities A, B and C are set as 0.57, 0.19 and 0.19, respectively. The time for constructing the CSR or CSC is not included in the result and we pre-sort the vertices of adjacency lists. As undirected graphs are used, the CSR and CSC have the same storage format, so we do not distinguish between them. All tests are run 64 times from random chosen source vertices similar to the Graph 500 benchmark. We run one thread per core on the CPU and four threads per core on the MIC. The KMP_AFFINITY environment variable is used to pin threads to particular cores.

5.1. Native BFS algorithm experiments

To evaluate the native BFS algorithm, we implement four versions on the MIC. The *top-down* implementation is based on the atomic method as shown in algorithm 4. The *top-down with SIMD* implementation uses the SIMD to accelerate the *top-down* algorithm. The *hybrid* is the implementation of the top-down and bottom-up hybrid algorithm. And *the hybrid with SIMD* implementation uses the SIMD to accelerate both the top-down and the bottom-up levels of the hybrid algorithm. In the *hybrid* algorithm, the parameter α and β are set to 1024 and 64 respectively, since they provide the best tuned results. The experimental results are shown in Figure 9. The increase in speed of the *top-down with SIMD*, the *hybrid* and the *hybrid with SIMD* method compared with the *top-down* method is shown in Figure 10.

The *top-down with SIMD* is obviously faster than the *top-down*, when the vertex number reaches 1M. For small-scale graphs, there is not an obvious gain in speed, as there is not enough work to accelerate. For a graph with 1M vertices, the SIMD algorithm increases speed by 1.38 times, 1.43 times, 1.60 times and 1.85 times with the *edgefactor* being 8, 16, 32 and 64, respectively. When the *edgefactor* is higher, the increase in speed that we can gain is larger. This is due to the fact that when the *edgefactor* is higher, more neighbors can be scanned by the SIMD instructions.

The performance for the *hybrid* method is much better than the *top-down* method, since it can combine the advantages of the *top-down* and *bottom-up* methods.

It is clear that the *hybrid with SIMD* method compared with the *hybrid* method can gain an obvious increase in speed, when the graph scale is large. It proves that the SIMD instructions can accelerate the graph traversal.

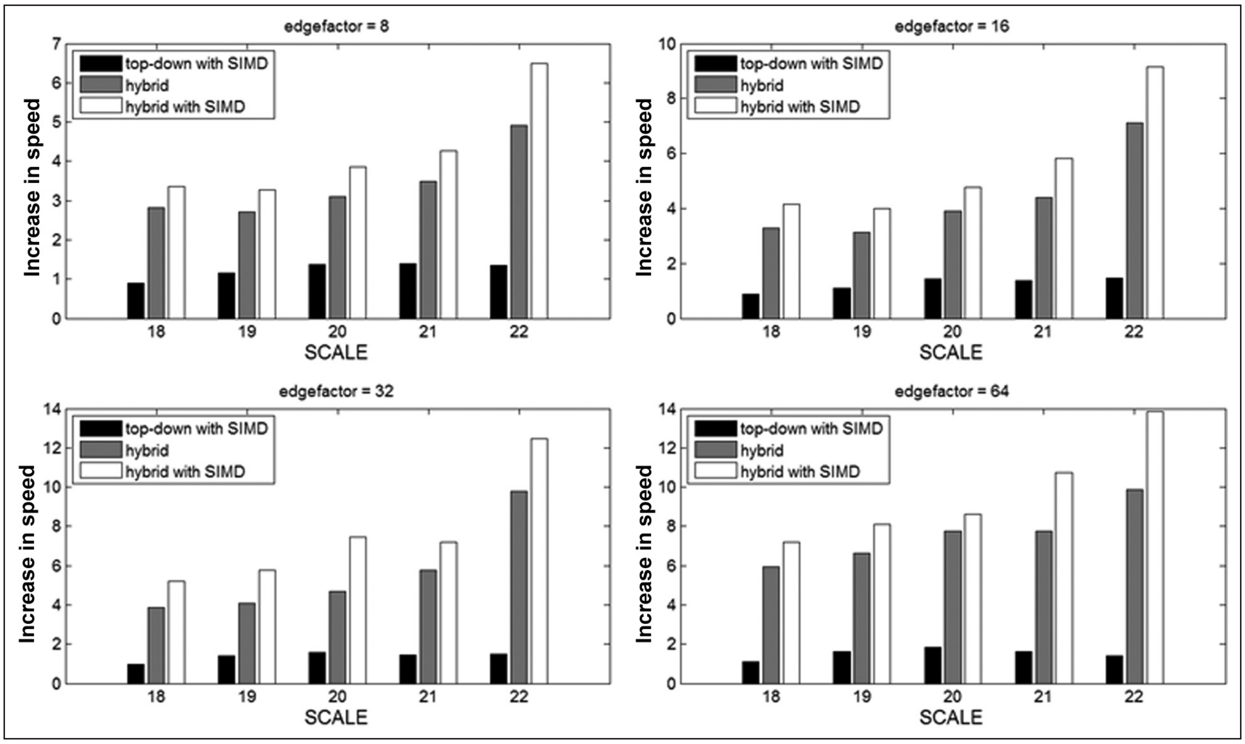

Figure 10. Native BFS algorithm optimization increase in speed.

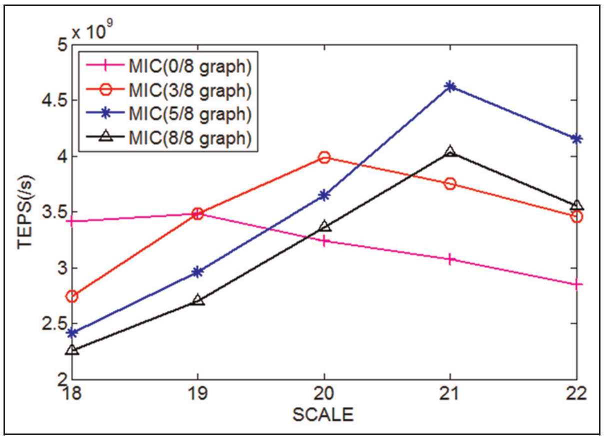

Figure 11. The result for different task partition ratios.

As we know, the highest published performance for a single GPU is 1.25 GTEPS (Giga TEPS) for kroon_g500_logn20 graph (Merrill et al., 2012) (their result should be halved as we calculate the TEPS based on undirected edges). Our *top-down with SIMD* algorithm can reach 1.42 GTEPS and the *hybrid with SIMD* algorithm can gain 6.64 GTEPS for the same graph. The increase in speed is approximately 1.14 times and 5.31 times, respectively.

5.2. Heterogeneous BFS algorithm experiment

We tune the best task partition ratio first. The tuned result is shown in Figure 11. α, β and γ are set to 1024, 256 and 4, respectively. The *edgefactor* is set to 16. When the MIC handles 0/8 of the graph, it means all the work is done by the CPU. When the MIC handles 8/8 of the graph, it means all work is done by the MIC for the cooperative computing level. When the graph scale is small, the MIC cannot accelerate the computing, since the overhead is relatively large. For large-scale graphs, when the MIC handles 5/8 of the graph, it gains the best performance. As a result, we partition 5/8 of the graph to the MIC.

To evaluate the heterogeneous algorithm, we compare the result of the CPU and MIC cooperative computing with the result of the CPU only, as shown in Figure 12. The graph is transferred to the MIC before computing and the result does not contain the transmission time. We partition 5/8 of the tasks to the MIC, which is the best tuned value.

From Figure 12, we can see clearly that the heterogeneous algorithm can gain speed for large-scale graphs. The increase in speed compared with the results of the CPU is shown in Figure 13. For graphs with 2M vertices, we can increase the speed by 1.46, 1.45, 1.41 and 1.30 times with the *edgefactor* being 8, 16, 32 and 64, respectively. For small-scale graphs, there is not an obvious increase in speed as not enough work can be accelerated by the MIC.

6. Related work

Buluç and Madduri (2011) provide a thorough taxonomy of related work in parallel BFSs. Beamer,

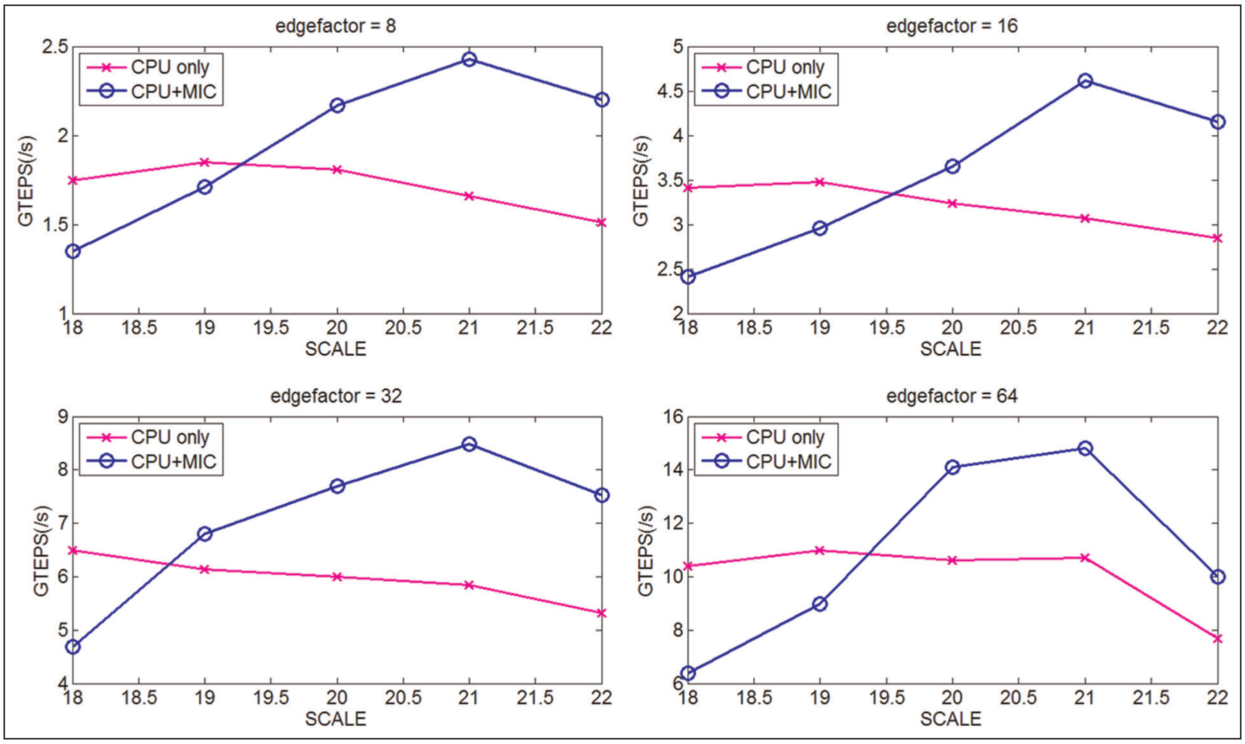

Figure 12. The heterogeneous BFS result (GTEPS: Giga TEPS).

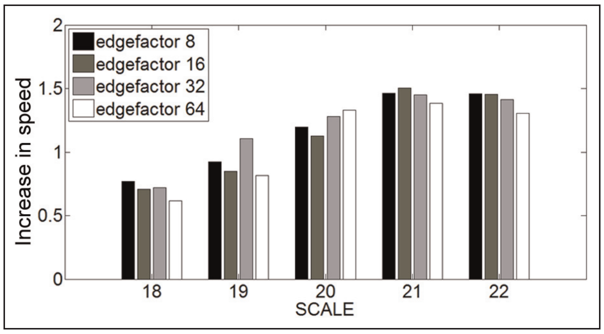

Figure 13. The heterogeneous BFS algorithm increase in speed.

Asanovic and Patterson (2012) summarize the work on shared-memory systems.

Saule and Catalyurek (2012) did an early evaluation of the scalability of graph algorithms on the Intel MIC architecture. They simply transplant the general multi-core algorithm to the MIC architecture and do not discuss the usage of the 512-bits SIMD instructions. Instead, we mainly discuss how to use the SIMD to gain a higher increase in the speed of performance, as transporting the general multi-core algorithm to the MIC is straightforward.

Hong, Oguntebi and Olukotun (2011) propose a hybrid algorithm with the CPU and GPU. The first few levels are executed on the CPU. If there is an exponential growth in the number of nodes in each level, the GPU is initiated. The execution returns to the CPU until finished. In their algorithm, either the CPU or GPU participates in the computation. However, our heterogeneous algorithm can combine the computation of the CPU and accelerators.

Our prior paper (Tao et al., 2013) reports the early results for using a MIC to accelerate the BFS. The prior paper only involves the top-down BFS optimizations; instead, we discuss both the top-down and bottom-up BFS in this paper. While the heterogeneous BFS algorithm in the prior paper is based on the top-down strategies only, we present a more efficient heterogeneous algorithm based both on the top-down and bottom-up strategies in this paper.

Other work involving a MIC is often about accelerating compute-intensive applications (Yang et al., 2013) and the related work about data-intensive application is seldom seen.

7. Conclusion and future work

We have discussed using a MIC to accelerate graph traversal. Both the native BFS algorithm and heterogeneous BFS algorithm are discussed. For the native BFS algorithm, optimizations are proposed for both the *top-down* and the *bottom-up* method. We mainly discuss how to use the SIMD instructions. The experiment proves that these optimizations can improve the performance. Our native BFS performance is 5.31 times that of the highest published performance for the GPU. A heterogeneous BFS algorithm is also proposed, which combines the computation of the CPU and accelerators. Our experiments with a MIC prove that the increase in speed that we can gain is about 1.4 times compared with the CPU.

Our work is valuable when using a MIC to accelerate the BFS. It is also a general guidance for using a MIC to accelerate data-intensive applications.

In the future, we will extend the algorithm to multi-nodes. Many aspects, such as graph representation, task partitioning and communication optimization, need further research.

Funding

This work is supported in part by the National Natural Science Foundation of China under grant 61120106005 and the National High Technology Research and Development 863 Program of China under grant 2012AA01A301.

References

Agarwal V, Petrini F, Pasetto D, et al. (2010) Scalable graph exploration on multicore processors. In: 2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis (SC10), New Orleans, LA, US, 13–19 November 2010, pp.1–11.

Albert R, Jeong H and Barabási AL (1999) Internet: diameter of the world-wide web. *Nature* 401(6749): 130–131.

Bader DA and Madduri K (2006) Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. In: International conference on parallel processin, Columbus, OH, US, 14–18 August 2006, pp. 523–530.

Barabási AL and Albert R (1999) Emergence of scaling in random networks. *Science* 286(5439): 509.

Beamer S, Asanovic K and Patterson D (2012) Direction-optimizing breadth-first search. In: 2012 ACM/IEEE international conference for high performance computing, networking, storage and analysis (SC12), Salt Lake City, UT, US, 10–16 November 2012, pp 1–10.

Brette R, Rudolph M, Carnevale T, et al. (2007) Simulation of networks of spiking neurons: a review of tools and strategies. *Journal of Computational Neuroscience* 23(3): 349–398.

Buluç A and Madduri K (2011) Parallel breadth-first search on distributed memory systems. In: 2011 ACM/IEEE international conference for high performance computing, networking, storage and analysis (SC11), Seattle, WA, US, 12–18 November 2011, pp. 65–76.

Chakrabarti D, Zhan Y and Faloutsos C (2004) R-MAT: a recursive model for graph mining. In: 4th international conference on data mining, Lake Buena Vista, FL, US, 22–24 April 2004, pp 442–446.

Checconi F, Petrini F, Willcock J, et al. (2012) Breaking the speed and scalability barriers for graph exploration on distributed-memory machines. In: 2012 ACM/IEEE international conference for high performance computing, networking, storage and analysis (SC12). Salt Lake City, UT, US, 10–16 November 2012, pp. 1–12.

Chhugani J, Satish N, Kim C, et al. (2012) Fast and efficient graph traversal algorithm for CPUs: maximizing single-node efficiency. In: 2012 IEEE 26th international parallel and distributed processing symposium (IPDPS12). Shanghai, China, 21–25 May 2012, pp. 378–389.

Dodds PS, Muhamad R and Watts DJ (2003) An experimental study of search in global social networks. *Science* 301(5634): 827–829.

Duran A and Klemm M (2011) The Intel Many Integrated Core architecture, Intel Corporation.

Hong S, Oguntebi T and Olukotun K (2011) Efficient parallel graph exploration on multi-core CPU and GPU. In: Parallel architectures and compilation techniques (PACT). Galveston Island, TX, US, 10–14 October 2011, pp. 78–88.

Leskovec J, Chakrabarti D, Kleinberg J, et al. (2010) Kronecker graphs: an approach to modeling networks. *Journal of Machine Learning Research* 11: 985–1042.

Lumsdaine A, Gregor D, Hendrickson B, et al. (2007) Challenges in parallel graph processing. *Parallel Processing Letters* 17(1): 5–20.

Manyika J, Chui M, Brown B, et al. (2011) Big data: the next frontier for innovation, competition, and productivity, McKinsey Global Institute.

Merrill D, Garland M and Grimshaw A (2012) Scalable GPU graph traversal. In: 17th ACM SIGPLAN symposium on principles and practice of parallel programming (PPoPP12). New Orleans, LA, US, 25–29 February 2012, pp. 117–128.

Murphy RC, Wheeler KB, Barrett BW, et al. (2010) Introducing the Graph 500, Cray User's Group (CUG).

Satish N, Kim C, Chhugani J, et al. (2012) Large-scale energy-effcient graph traversal: a path to efficient data-intensive supercomputing. In: 2012 ACM/IEEE international conference for high performance computing, networking, storage and analysis (SC12), Salt Lake City, UT, US, 10–16 November 2012, pp. 14.

Saule E and Catalyurek UV (2012) An early evaluation of the scalability of graph algorithms on the intel mic architecture. In: 2012 IEEE 26th international parallel and distributed processing symposium workshop (IPDPSW), Shanghai, China, 21–25 May 2012, pp. 1629–1639.

Tao G, Yutong L and Guang S (2013) Using MIC to accelerate a typical data-intensive application: the breadth-first search. In: 2013 IEEE 27th international parallel and distributed processing symposium workshop and PhD forum (IPDPSW), Boston, MA, US, 20–24 May 2013, pp. 1117–1125.

Watts JD and Strogatz HS (1998) Collective dynamics of 'small-world' networks. *Nature* 393(6684): 440–442.

Yang X, Liao X, Xu W, et al. (2010) Th-1: China's first petaflop supercomputer. *Frontiers of Computer Science in China* 4(4): 445–455.

Yang CQ, Wu Q, Chen C, et al. (2013) Accelerating PQMRCGSTAB Algorithm on Xeon Phi. *Advanced Materials Research* 709: 555–562.

Yoo A, Chow E, Henderson K, et al. (2005) A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In: 2005 ACM/IEEE international conference for high performance computing, networking, storage and analysis (SC05), Seatle, WA, US, 12–18 November 2005, pp. 25–34.

Author biographies

Tao Gao is pursuing a PhD degree at the School of Computer Science at the National University of Defense Technology (NUDT). He mainly focuses his attention on the research of parallel computing.

Yutong Lu is a professor, PhD, at the State Key Laboratory of High Performance Computing, School of Computer Science at NUDT. She is involved in the research and implementation of system software for high-performance computing. Her research interests include operating systems, communication systems, resource management systems, parallel file systems and MPI.

Baida Zhang obtained his PhD from NUDT in 2012. He is an assistant professor at the School of Computer Science at NUDT. He mainly focuses his attention on the research of parallel computing.

Guang Suo obtained his PhD from NUDT in 2007. He is an assistant professor at the School of Computer Science at NUDT. He mainly focuses his attention on the research of parallel computing.