2013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum

Using MIC to accelerate a typical data-intensive application: the Breadth-first Search

Gao Tao, Lu Yutong and Suo Guang
School of Computer
National University of Defense Technology
Changsha, Hunan, China, 410073
Email: {gaotao, ytlu, suoguang}@nudt.edu.cn

***Abstract*—Data-intensive applications draw more and more attentions in the last few years. The breadth-first search (BFS), a typical data-intensive application, is so widely used that the Graph 500 benchmark uses it to rank supercomputers' performance. The Intel MIC (Many Integrated Core), which is designed for highly parallel computing, hasn't been fully evaluated for data-intensive applications. In this paper, we discuss how to use MIC to accelerate the BFS. Optimizations both for native mode and for offload mode are discussed. About native mode, we propose optimizations for thread-level and data-level parallelism. We exploit the thread-level parallelism by relaxing inter-thread dependence. The optimized algorithm is proved to be more scalable. Data-level parallelism is exploited by 512-bits single instruction multiple data (SIMD) instructions. The maximum speedup we further gain is up to 3.4 times. About offload mode, we present an offload algorithm. By careful task partition and communication optimizations, it can gain speedup for large graphs which can't run natively on MIC as the limited memory size. We believe that the work is valuable for using MIC to accelerate the BFS. Meanwhile, it's a general evaluation of the MIC for data-intensive applications.**

***Keywords*-BFS; MIC; data-intensive application; hybrid computing**

I. Introduction

Data-intensive applications draw more and more attentions in recent years, due to the demand for exploring large scale data. Graph search, a challenging problem [1], is an important data-intensive application, as the relationship between objects often can be represented in the form of graphs. Among various graph search algorithms, breadth-first search (BFS), which is widely used in numerous applications (genomics, astrophysics, artificial intelligence, information analytics, national security and data mining), is particularly important [2, 3]. Recently, the BFS is used as the kernel component of the Graph 500 benchmark, which is used to rank computer's performance of data-intensive applications [4]. In the last few years, many researches about the BFS have been conducted on distributed memory systems [2, 5–7], shared memory systems [3, 8–13] and graphic processing units (GPU) [12, 14].

The Intel many integrated core (MIC) is designed for highly parallel computing. For instance, the MIC we used has more than 50 physical cores with 4 way simultaneous multi-threading (SMT) and each core has 512-bits vector processing units (VPU). As mainly targets for compute-intensive applications, the MIC hasn't been fully evaluated for data-intensive applications. Recently, study in [15] does an early evaluation of graph algorithms on MIC.

In this study, we concentrate on using MIC to accelerate the BFS. We propose some optimizations for both *native mode* using MIC only and *offload mode* using MIC as an accelerator. By doing this work, we present some optimizations for using MIC to accelerate the BFS, meanwhile it's a general evaluation of the MIC for data-intensive applications.

The specific contributions of this paper are:

- We present some native optimizations of the BFS on MIC, which has more cores and wider vector processing units. By relaxing the inter-thread dependence, we present an optimized BFS algorithm, which is scalable even running with more than 50 cores. Meanwhile, we discuss how to make use of 512-bits single instruction multiple data (SIMD) instructions to gain more performance improvement on MIC.
- We propose and implement an offload algorithm for the BFS. By careful task partition and communication optimizations, it can gain speedup for large scale graphs, which can't run natively on MIC, as the limited memory size.
- Our work is a general evaluation of MIC for data-intensive applications. It's valuable for those who want to use MIC to accelerate their data-intensive applications.

This paper is organized as follows. Section II introduces the background. Section III describes optimizations of using MIC to accelerate the BFS. Section IV shows and analyses the experimental results. The related work is discussed in section V. Our conclusion and future work are discussed in section VI.

978-0-7695-4979-8/13 $26.00 © 2013 IEEE
DOI 10.1109/IPDPSW.2013.197

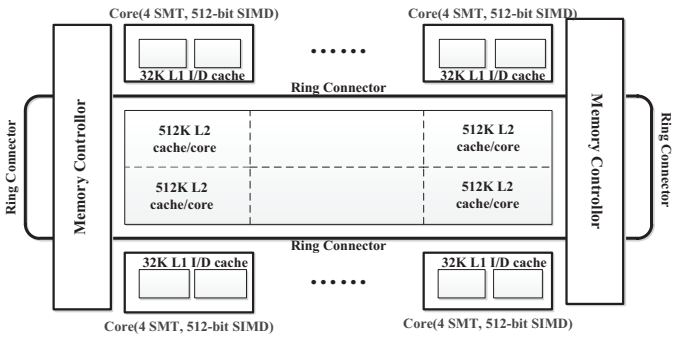


Figure 1. The Basic Architecture of MIC

II. Background

A. The MIC

Figure 1 shows the basic architecture of the MIC. The MIC co-processor we used contains 57 Intel architecture cores running at 1100MHz. They are connected by a high-performance on-die bi-directional interconnect. Every core has four-way hardware multi-threads and 512-bits vector processing units. There are 32KB data cache and 32KB instruction cache for each core. The L2 cache with 512KB per core is shared. In addition to that, there are 8 memory controllers supporting 16 GDDR5 channels at most. The card can communicate with the host and other card over the PCI express bus.

MIC supports the general parallel programming model, such as MPI [16] and OpenMP [17]. Intel provides an implementation of MPI based on MPICH2 [18]. Both intra-device and inter-device communications are supported. OpenMP is supported by the Intel compiler.

B. Parallel BFS

Breadth-first search is the algorithm of exploring all vertexes in a given graph from a root. We usually use *G(V, E)* to denote a graph, in which *V* is the vertex set and *E* is the edge set. A vertex can use an integer number v_0 to represent and (v_0, v_1) represents an edge (undirected edge) between v_0 and v_1. If vertex v_1 is expanded by vertex v_0, we call v_0 is the predecessor of v_1. The spanning tree is represented by a predecessor map.

In the last several decades, there are many researches conducted on the BFS. In recent years, the parallel BFS based on shared memory systems is widely researched [3, 8–12] as the multi-core and many-core processors are widely used. Most of work is based on level synchronous BFS.

In level synchronous BFS algorithm, there are a current queue and a next queue. The current queue holds vertexes will be explored in this level and the next queue holds vertexes should be explored in the next level. Within each level, the algorithm scans vertexes in the current queue. Neighbors, which haven't been explored, are inserted into the next queue. At end of a level, the next queue becomes the current queue. The process repeats until all the vertexes have been explored.

The level synchronous BFS algorithm can be paralleled as its parallelism in nature. At least, two level parallelisms can be exploited [8]. Both scanning the current queue and exploring vertex's neighbors can be done concurrently. But multi-threads implementations should handle the access to the shared next queue and predecessor map carefully.

One of the efficient implementations uses bitmaps to represent the current queue and the next queue. To reduce the working set size, a bitmap for the predecessor map is also used. To make sure the correctness, multi-threads use the *__sync_or_and_fetch* atomic operation to test and set the bitmaps concurrently [3].

C. Graph 500

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 [19, 20]. The *SCALE* and *edgefactor* parameters can be assigned. The generated graphs (undirected graph) have 2^{SCALE} vertexes and *edgefactor**2^{SCALE} edges. It uses the traversed edges per second (TEPS) to measure the performance.

The Graph 500 benchmark version 2.1.4 contains some BFS implementations [4]. A bitmap based implementation called replicated csc (compressed sparse column) is efficient in our test. We reference this algorithm as *graph500-replicated-csc*. It's a MPI and OpenMP hybrid program. The adjacency matrix is divided by processes in a one-dimension manner. By this way, the neighbors of a vertex is distributed among all processes and every process is responsible for computing part of the next queue. Within each process, threads are used to scan the current queue simultaneously. As bitmaps are used to represent the current queue and next queue, the *__sync_or_and_fetch* atomic operation is used to make sure the correctness when multi-threads update the next queue.

III. Optimizations of accelerating BFS With MIC

In this section, we describe methods of using MIC to accelerate the BFS. The MIC can be used for both native mode and offload mode, so we discuss native optimizations and offload optimizations respectively. About native mode, both thread-level and data-level parallelisms are discussed. We enhance the thread-level parallelism by relaxing inter-thread dependence and exploit data-level parallelism with SIMD support. About offload mode, we gain speedup for large scale graphs by careful task partition and communication optimizations. All optimizations are based on the *graph500-replicated-csc* algorithm in Graph 500.

A. The Native Optimizations

The native optimizations focus on exploiting thread-level and data-level parallelism. Using MIC natively is similar

to using general multi-core CPU. But when optimizing an algorithm on MIC, two key differences should be considered: (1) the MIC has more cores. (2) the MIC core has wider vector processing units. As a result, it's important to make full use of many cores and the SIMD instructions. The optimized algorithms are based on *graph500-replicated-csc* and optimizations are applied to each MPI process.

Algorithm 1: The Relaxed BFS Algorithm

```
   Data: G(V,E), r // r is the root vertex
   Result: P[1...n] // Predecessor Map
1  in_queue, out_queue, visited ← 0
2  for v ∈ V do
3      P[v] ← ∞
4  P[r] ← r
5  SetBitmap(in_queue, r)
6  while in_queue isn't empty do
7      out_queue ← 0
8      for v0 ∈ in_queue parallel do
9          for v1 adjacent to v0 do
10             if Test(visited, v1)=0 ∩ P[v1]=∞ then
11                 P[v1] ← v0 − n
12                 SetBitmap(out_queue,v1)
13     barrier
14     for i = 0 to bitmap size parallel do
15         if out_queue[i] ≠ 0 then
16             for v1 whose bit ∈ out_queue[i] do
17                 if P[v1] < 0 then
18                     P[v1] ← P[v1] + n
19                     SetBitmap(out_queue,v1)
20                     SetBitmap(visited,v1)
21     Swap(in_queue, out_queue)
```

1) The Thread-Level Parallelism Optimization: The thread-level parallelism optimization focuses on relaxing the inter-thread dependence. Within one process, the original *graph500-replicated-csc* algorithm exploits thread-level parallelism by scanning vertexes in the current queue concurrently. To ensure correctness, the *__sync_or_and_fetch* atomic operation is used. It ensures that only one thread wins the race when multi-threads try expanding the same vertex. However, the atomic operation results in inter-thread dependence, which is inefficient. We use a method similar to that in [11] to relax the dependence.

The relaxed algorithm is shown in Algorithm 1. The current queue and the next queue are represented by *in_queue* and *out_queue* bitmap array respectively. The predecessor map is denoted by *P* and the *visited* is bitmap array for *P*. The ***SetBitmap****(bitmap, index)* sets bit for *index* in *bitmap* array and ***Test****(bitmap, index)* returns bit value for *index* in *bitmap* array. The *in_queue*, *out_queue* and *visited* are initialized at first (line 1). The *P* is initialized in lines 2-3. According to the Graph 500 specification, the predecessor of root vertex is defined as itself. This assignment is performed in line 4. Firstly, the root vertex is inserted into the current queue in line 5. Then the exploring process is repeated until the current queue becomes empty (lines 6-21). In each level, the next queue is emptied at first (line 7). Multi-threads scan vertexes in the current queue concurrently (line 8). For a vertex v_0 in the current queue, all neighbors are searched (line 9). For a neighbor v_1, the *visited* and *P* array are tested to determine if it has been visited (line 10). If not, the *P* and *out_queue* are set for v_1 (lines 11-12). To begin a new level, the *in_queue* and *out_queue* are swapped (line 21).

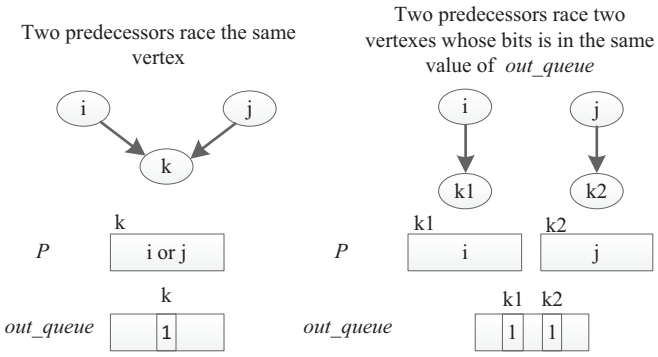


Figure 2. Two Data Race Conditions

The correctness of this algorithm is ensured by hardware and restoring. The race of setting *P* and *out_queue* in lines 11-12 needs further analysis. Multi-threads exploring the same vertex v_1 may pass the test in line 10 at the same time. There exits two data race conditons as shown in Figure 2 : (1) vertex *i* and *j* expand the same vertex *k* (2) vertex *i* and *j* expand vertex *k1* and *k2* respectively, but the bits for vertex *k1* and *k2* in *out_queue* are in the same value. In the former case, if 8/16/32/64-bits are used to represent the predecessor values, the update to $P[v_1]$ (line 11) is always consistent [11, 21]. Although the predecessor for vertex *k* may be set to vertex *i* or *j*. It's always a valid expanding tree. As both vertex *i* and *j* set the same bit in *out_queue*, the resulted *out_queue* is always correct. In the other case, the predecessor value is determined and the resulted *out_queue* may lose some information. However, it's sure that at least one bit for *k1* or *k2* is set correct. We need restore the *out_queue* before a new level. After a barrier (line 13), the restoring is applied as shown in lines 14-20. To distinguish the vertexes in the next queue, the predecessor value is taken away the vertexes' amount *n* (line 11). In the restoring process, the values in *out_queue* are scanned. If a nonzero value *out_queue[i]* is found (line 15), the *P* array is tested to determine whether the corresponding vertexes should be in the next queue (lines 16-17). If so, the *P*, *out_queue*

and *visited* are restored (lines 18-20). Multi-threads can do the restoring concurrently without data racing, as different threads scan the different part of *out_queue*. By this way, the correctness of this algorithm is ensured.

This algorithm is efficient. Because it relaxes inter-thread dependence and the restoring process accesses memory in order. Although some extra work is done in lines 10-12 and in lines 14-20, experimental results show that this algorithm is scalable. Most importantly, the algorithm can be further accelerated by SIMD instructions. We reference this algorithm as *native-relaxed*.

Algorithm 2: The SIMD BFS Algorithm

```
   Data: G(V,E), r // r is the root vertex
   Result: P[1...n] // Predecessor Map
1  in_queue, out_queue, visited ← 0
2  for v ∈ V do
3      P[v] ← ∞
4  P[r] ← r
5  SetBitmap(in_queue, r)
6  while in_queue isn't empty do
7      out_queue ← 0
8      for v_0 ∈ in_queue parallel do
9          for all vertexes adjacent to v_0 do
10             v_1 ← VecLoadNeighbors(v_0)
11             vec_visited ← VecGather(visited, v_1)
12             mask ← VecTest(vec_visited, v_1)
13             if mask = 0 then
14                 continue
15             VecStore(child, v_1, mask)
16             for i ← 0 to 16 do
17                 if mask[ith] = 1∩P[child[i]]=∞ then
18                     P[child[i]] ← v_0 − n
19                     VecSetBitmap(out_queue, child[i])
20     barrier
21     for i = 0 to bitmap size parallel do
22         if out_queue[i] ≠ 0 then
23             for vertexes whose bits ∈ out_queue[i] do
24                 v_0 ← VecLoad(out_queue[i])
25                 mask ← VecCompareLessThan(v_0, 0)
26                 v_0 ← VecAdd(v_0, n, mask)
27                 VecStore(v_0, mask)
28                 VecSetBitmap(out_queue, mask)
29                 VecSetBitmap(visited, mask)
30     Swap(in_queue, out_queue)
```

2) The Data-Level Parallelism Optimization: The relaxed algorithm can be further accelerated by SIMD instructions. If 32-bits integers are used to represent vertexes and bitmaps, 512-bits processing vector units can calculate 16 operands at the same time. The most time consuming part of Algorithm 1 is neighbors scanning shown in lines 9-12 and restoring process shown in lines 14-20. Both of them can be accelerated by vector processing units.

The SIMD algorithm is shown in Algorithm 2. Scanning the neighbors is shown lines 9-19. Firstly, neighbors are loaded (line 10) and *visited* values are gathered (line 11). Then a vector mask is calculated (line 12). If a vertex hasn't been visited, the mask bit for this vertex is set to 1. If the mask equals to 0, no further work is needed (lines 13-14). Otherwise, neighbors, which haven't been visited, are stored to a temporary array called *child* (line 15). Lastly, by scanning the *child* array and combining the mask, the *P* and *out_queue* are updated (lines 16-19) like *native-relaxed* algorithm.

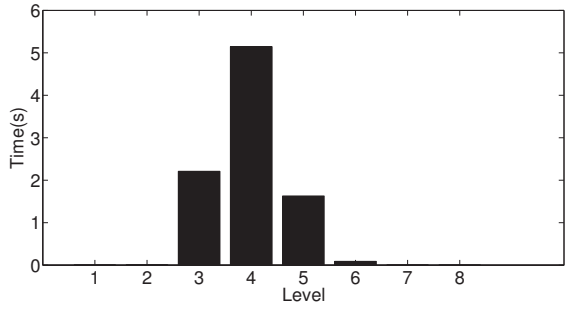
After a barrier (line 20), the restoring starts as shown in lines 21-29. When a nonzero value *out_queue[i]* is found, vector operation is used to scan all corresponding vertexes. Firstly, predecessors are loaded (line 24) and tested (line 25). The result is a vector mask. If a vertex should be in the next queue, its mask bit is set to 1. According to the mask, the *P*, *out_queue* and *visited* are restored (lines 26-29). The restoring operations are all 512-bits SIMD instructions.

This optimization can gain speedup as we can make full use of the 512-bits vector processing units on MIC. We reference this algorithm as *native-SIMD*.

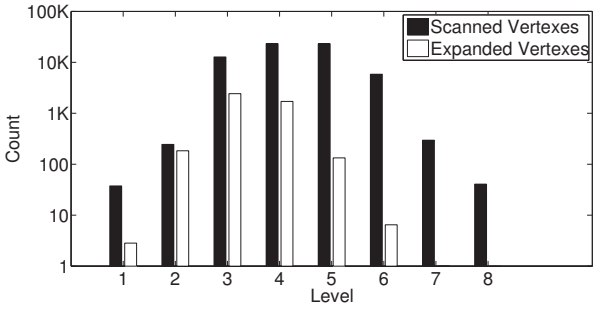
B. The Offload Optimizations

Offload optimizations focus on designing an efficient algorithm using MIC as an accelerator. Using MIC as an accelerator of the CPU is a general type of hybrid computing. It's a challenging task to design an efficient offload algorithm for data-intensive applications because of the large scale data volume and few float operations. Based on the *graph500-replicated-csc* algorithm, we propose an offload algorithm. By careful task partition and communication optimizations, it can gain speedup for large scale graphs without especial native optimizations.

1) Motivation: The BFS algorithm has potential to be accelerated by MIC. Let's look at the statistical data of *graph500-replicated-csc* algorithm first. Figure 3a shows the time of each level. From this figure, we can know that several middle levels consume most of the time. It implies that the middle levels have potential to be accelerated. For offload algorithm, vertexes in the current queue and all their neighbors should be transferred to MIC. For large scale graphs, the transferred data volume may be very large. Figure 3b shows the scanned and expanded vertexes in every level (note that the Y axis is logarithmic). From this figure, we can know that although many scanned vertexes should be transferred to MIC, much less expanded vertexes have to be transferred back. Based on these observations, we can design an offload algorithm.



(a) Time in Every Level



(b) Scanned and Expanded Vertexes in Every Level

Figure 3. The Statistical Data of the *graph500-replicated-csc* Algorithm, 2^{24} vertexes , 2^{28} undirected edges

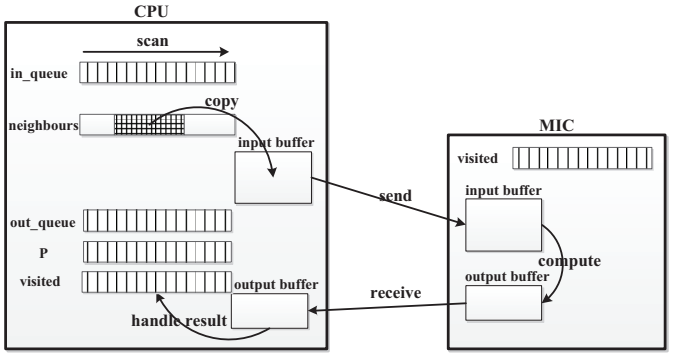


Figure 4. The Offload Algorithm

Algorithm 3: Neighbors Scanning on MIC

```
   Data: visited, CQ // the current queue
   Result: output // the output buffer
1  local_queue ← ∅
2  for v0 ∈ CQ parallel do
3      for v1 adjacent to v0 do
4          if Test(visited, v1) then
5              if IsFull(local_queue) then
6                  InsertQueue(ouput, local_queue)
7                  local_queue ← ∅
8                  Insert(local_queue, (v1, v0))
9              else
10                 Insert(local_queue, (v1, v0))
11 InsertQueue(ouput, local_queue)
```

2) Algorithm Outline: The algorithm outline is shown in Figure 4. There exits input and output buffer both on MICs and CPUs. When scanning vertexes in the current queue *in_queue*, the vertex and its neighbors are copied to the input buffer. When the input buffer is full, the data is sent to MIC. Scanning neighbors is done by MIC, and the results are stored in output buffer. The results are transferred back to CPU to update the next queue *out_queue*, the predecessor map *P* and *visited* bitmap array.

Scanning neighbors on MIC is shown in Algorithm 3. The *visited* bitmap array is sent to MIC at start of each level. Every thread has a *local_queue* to avoid frequently updating *output* buffer (line 1). The thread scans vertexes v_0 in the current queue concurrently (line 2). For a neighbor v_1 of v_0 (line 3), the *visited* is used to test if it has been visited (line 4). If not, the (v_1, v_0), which means v_0 is the predecessor of v_1, is inserted to the *local_queue* (line 8,10). When the *local_queue* is full, it is copied to the *output* buffer (lines 5-6). CPU updates *out_queue*, *P* and *visited* according to these vertex pairs.

3) Task Partition: Task partition contains two aspects: (1) partition the computation of different levels between CPU and MIC. (2) partition the computation of different parts within a level between CPU and MIC. Figure 3a shows that different levels take dramatically different amounts of time. Several middle levels account for most of the total time, while others take much less time. To address the inefficient processing of non-critical levels, a hybrid scheme is proposed. The idea is quite simple: (1) if the vertex amount in the current queue exceeds a baseline, MIC is used to accelerate (2) otherwise, only CPU is used. When using MIC to accelerate a level, the MIC only in charge of neighbors scanning. All other work is done by CPU, like sending the input buffer, receiving the output buffer and updating *out_queue*, *P* and *visited*. By this way, we keep the CPU busy when MIC is busy.

4) Communication Optimizations: Communication optimizations mainly contain buffer pipeline and overlapping communication with computation. There are an *input* buffer queue and an *output* buffer queue both on CPU endpoint and on MIC endpoint. We use the *input* buffer queue in a pipeline way. The computation on MIC, the computation on CPU and the communication on DMA device should be overlapped to gain better performance. After starting an asynchronous data

Table I
THE DEVICE INFORMATION

	CPU	MIC
name	Intel(R) Xeon(R) E5-2670	Knight Corner
clock rate	2.60GHz	1.1GHz
sockets	2	1
cores(per socket)	8	57
threads(per core)	2	4
L1 cache(per core)	32KB	32KB
L2 cache(per core)	256KB	512KB
L3 cache(per socket)	20MB	-

transmission, the CPU gets and handles the output data. At the same time, the MIC may be scanning neighbors which transferred before. By these communication optimizations, overheads are reduced.

IV. EXPERIMENTAL RESULTS

In this section, we present experimental results and analyses. All experiments are conducted on a single platform with two CPUs and two MICs. The detail device information is shown in Table I. The CPU is Intel Xeon E5-2670 and the clock rate is 2.60GHz. Each CPU has 8 cores and each core with 32KB L1 cache and 256KB L2 cache can run two hardware threads. The 20MB L3 cache is shared. The 57 MIC cores run at the 1.1GHz. Each core can run 4 hardware threads. There are 32KB L1 cache and 512KB L2 cache per core.

We use the Intel C compiler and Intel MPI. The Graph 500 graph generator is used to create the graphs and all parameters are default. When running, the *edgefactor* is assigned as the default value 16. All tests run 64 times from random chosen root just like the Graph 500 benchmark.

All implementations are based on *graph500-replicated-csc* in Graph 500, so they are MPI and OpenMP hybrid programs. The I_MPI_PIN and KMP_AFFINITY environment variables are used to pin processes and threads to physical cores.

A. Native Optimization Experiments

1) Relaxed Optimization Result: To evaluate the relaxed optimization, we implement a *native-relaxed* algorithm based on *graph500-replicated-csc* implementation in Graph 500. The experiments are conducted on both MIC and CPU.

Figure 5 shows the result of relaxed algorithm compared with *graph500-replicated-csc* algorithm. All experiments run with one process and multi-threads are used within a process. Different threads are pinned to different physical cores. The relaxed algorithm is scalable on CPU up to 16 threads as shown in Figure 5a. The performance of *graph500-replicated-csc* algorithm drops when the thread amount reaches 9, because threads distribute on two Sockets. Figure 5b shows experimental results on MIC. Even the thread amount reaches 57, the relaxed algorithm gains speedup. These experiments prove that the relaxed optimization is valuable.

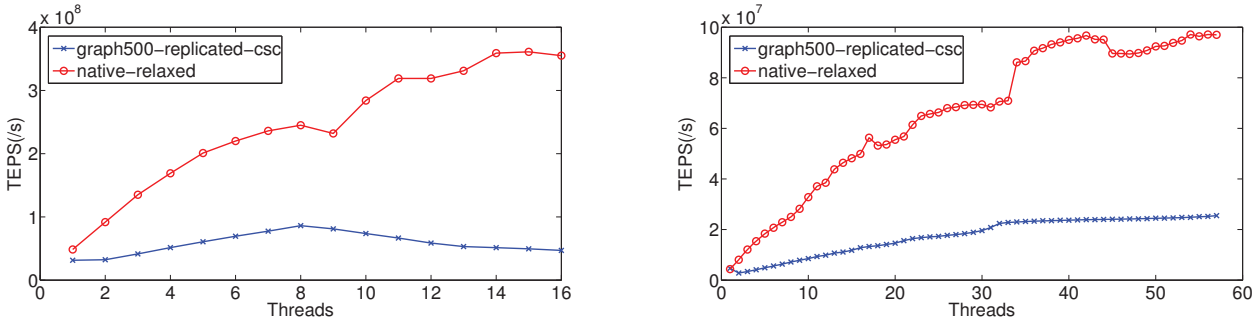
2) SIMD Optimization Result: To evaluate the SIMD optimization, we implement *native-relaxed* algorithm and *native-SIMD* algorithm based on *graph500-replicated-csc* implementation in Graph 500. Both implementations are MPI and OpenMP hybrid programs just like *graph500-replicated-csc*. We evaluate the speedup for MPI processes and OpenMP threads respectively. Instead of using 64-bits integers in original algorithm, we use 32-bits integers to represent vertexes and bitmaps. It ensures that the 512-bits vector processing units can calculate 16 operands at the same time.

Figure 6 shows the speedup of the SIMD optimization, compared with the relaxed algorithm. All experiments are conducted on MIC. Figure 6a shows the speedup with different process amount. Different processes are bound to different physical cores. The speedup reaches 3.4 with 1 process running graphs with 2^{14} vertexes. The speedup drops a little as the *SCALE* increases, but it's still close to 2 even with 16 processes running graphs with 2^{19} vertexes. When *SCALE* is the same, the speedup drops as the process amount increases. The reason is that the more processes the less continuous neighbors in one process can by be accelerated. Figure 6b shows the speedup with one process and different thread amount. Four hardware threads can run on the same physical core on MIC. This experiment is used to evaluate the SIMD optimization on 4-way SMT on MIC, so threads are bound to one physical core. We can gain more than 2 times speedup for both 2 threads and 4 threads, as shown in Figure 6b. These experiments prove SIMD optimization can further accelerate relaxed algorithm.

B. Offload Algorithm Experiment Results

We choose the symmetric communications interface [22] to implement the offload algorithm. Figure 7 shows the experimental result of the offload algorithm. One process is used for running graphs with 2^{24} vertexes; two processes are used for running graphs with 2^{25} vertexes; four processes are used for running graphs with 2^{26} vertexes. Suppose the *SCALE* is 24, there are about 2^{28} edges. If 64-bits integers are used to represent vertexes, then at least 2^4 GB is needed to store the edge information. As the limited memory size, the BFS can't run natively with so large graphs on MIC.

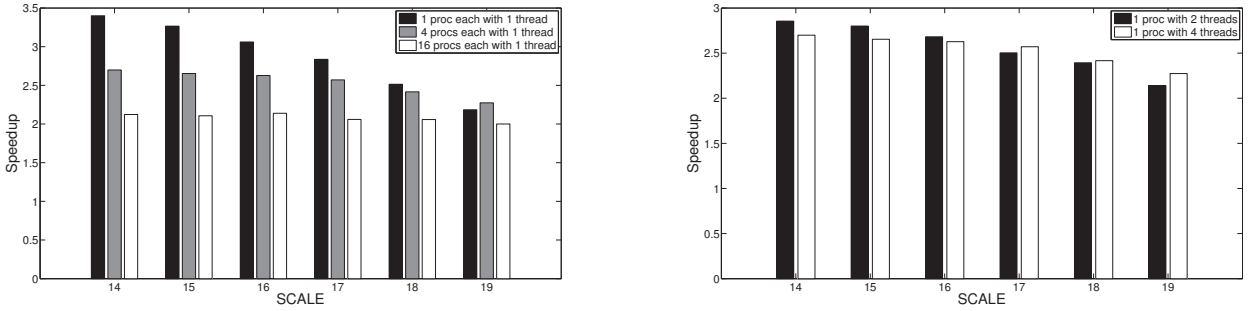
Figure 7a compares the time consumed in every level. It's clear that the offload algorithm reduces the time in several middle levels and doesn't have high overheads in other levels. Figure 7b shows the speedup, compared with the original implementation in Graph 500. One process with *SCALE* being 24 can gain 1.67 times speedup, while two processes with *SCALE* being 25, can gain 1.52 times speedup. Notice, four processes with *SCALE* being 26 are



(a) The Experimental Results on CPU

(b) The Experimental Results on MIC

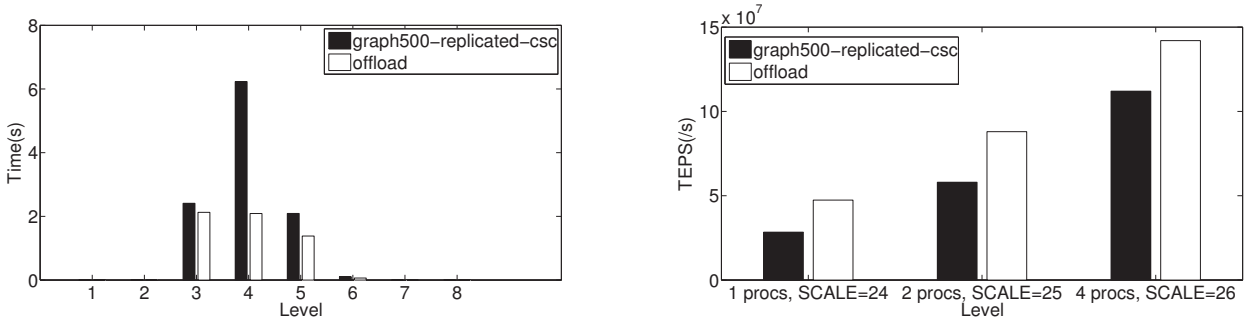
Figure 5. The Experimental Results of Relaxed Optimization compared with *graph500-replicated-csc* Algorithm



(a) Speedup of Different Process Amount

(b) Speedup of One Process with Different Thread Amount

Figure 6. Speedup of SIMD Optimization compared with Relaxed Optimization



(a) The Time in Every Level

(b) The TEPS of Offload Algorithm

Figure 7. The Experimental Results of Offload Algorithm

accelerated by two MICs and every MIC accelerates two processes. We also get 1.27 times speedup.

In fact, the speedup is far away from the ideal value. There exits some reasons. Firstly, the communication traffic is too large. For example, when the *SCALE* is 24, dozens of GB data are transferred. Secondly, the offload algorithm isn't applied native optimizations, such as relaxed optimization and SIMD optimization.

V. Related Work

Buluc and Madduri [7] provide a thorough taxonomy of related work in parallel breath-first searches. The work of shared-memory implementations are summarized in [9]. We focus on the work relevant to this study, such as relaxed optimizations and hybrid computing.

The study in [10] relaxes addition to queue. In [11], an atomic-free algorithm is firstly proposed. Our relaxed algorithm is similar to these ideas. However, our algorithm totally uses bitmaps to represent the current queue and

the next queue, which is proved to be efficient. And we further discuss how to use SIMD instructions to accelerate it. We haven't seen related work using the 512-bits SIMD instructions to accelerate the BFS.

The work in [12] proposes a hybrid algorithm with CPU and GPU. However, in their study, the graph data is loaded to GPU at first, so the graph scale is limited by GPU memory size. Instead, our offload algorithm dynamically transfers data to MIC. It is an early work to use MIC as an accelerator of CPU for the BFS.

The evaluation in [15] only contains native mode without SIMD support. Instead, our work contains both offload mode and SIMD optimization.

VI. Conclusion and Future Work

We have evaluated using MIC to accelerate the BFS based on Graph 500 benchmark. About native mode, we propose relaxed optimization and SIMD optimization. By relaxing inter-thread dependence and using SIMD instructions, we make full use of both many cores and wide vector processing units on MIC. The relaxed optimization is proved to be scalable with more than 50 cores; the SIMD optimization can further gain speedup about 2-3 times. About offload mode, by careful task partition and communication optimization, we gain speedup for large scale graphs, which can't run natively as the limited memory size on MIC.

Our work is valuable for using MIC to accelerate the BFS. Meanwhile, it's a general evaluation of the MIC for data-intensive applications.

In the future, we plan to combine all optimizations discussed in this paper to design an efficient BFS algorithm for one node with CPUs and MICs. Then, we'll extend the algorithm to multi-nodes. Many aspects, like graph represent, task partition and communication optimizations, need further researches.

Acknowledgment

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

References

[1] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 1, pp. 5–20, 2007.

[2] 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 *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2005.

[3] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable graph exploration on multicore processors," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[4] Graph 500 benchmark. [Online]. Available: http://www.graph500.org

[5] F. Checconi, F. Petrini, J. Willcock, A. Lumsdaine, A. R. Choudhury, and Y. Sabharwal, "Breaking the speed and scalability barriers for graph exploration on distributed-memory machines," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[6] N. Satish, C. Kim, J. Chhugani, and P. Dubey, "Large-scale energy-efficient graph traversal: A path to efficient data-intensive supercomputing," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[7] A. Buluc and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[8] D. A. Bader and K. Madduri, "Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2," in *International Conference on Parallel Processing(ICPP)*, 2006.

[9] S. Beamer, K. Asanovic, and D. A. Patterson, "Direction-optimizing breadth-first search," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[10] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm(or how to cope with the nondeterminism of reducers)," in *Symposium on Parallel Architectures and Algorithms(SPAA)*, 2010.

[11] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey, "Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency," in *International Parallel and Distributed Processing Symposium(IPDPS)*, 2012.

[12] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core cpu and gpu," in *Parallel Architectures and Compilation Techniques(PACT)*, 2011.

[13] D. A. Bader and K. Madduri, "Snap, small-world network analysis and partitioning: an open-source parallel graph framework for the exploration of large-scale networks," in *International Parallel and Distributed Processing Symposium(IPDPS)*, 2008.

[14] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," in *Principles and Practice of Parallel Programming(PPoPP)*, 2012.

[15] E. Saule and U. V. Catalyurek, "An early evaluation of the scalability of graph algorithms on the intel mic architecture," in *International Parallel and Distributed*

Processing Symposium Workshop(IPDPSW), 2012.

[16] *MPI: A Message-Passing Interface Standard*, Message Passing Interface Forum Std. 2.2. [Online]. Available: http://www.mpi-forum.org

[17] *OpenMP Application Program Interface*, The OpenMP ARB Std. 3.1. [Online]. Available: http://www.openmp.org

[18] Mpich2. Argonne Natinal Laboratory. [Online]. Available: http://wwww.mcs.anl.gov/mpi/mpich2

[19] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," in *Proceedings of 4th International Conference on Data Mining(SDM)*, 2004.

[20] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *Journal of Machine Learning Research*, vol. 11, pp. 985–1042, Feb. 2007.

[21] *Intel 64 and IA-32 Architectures Software Developers Manual, System Programming Guide, Part 1*, Intel Press.

[22] *Intel Many Integrated Core (Intel MIC) Symmetric Communications Interface (SCIF) User Guide*, Intel Std., Rev. 0.8, June 2012.