The Phoenix-based Parallel Algorithm for Constructing Extremal Graphs

Ruijun Zheng,Yongqi Sun[a], Yali Wu and Rui Zhang

School of Computer and Information Technology, Beijing Jiaotong University,

Beijing, 100044, P. R. China

[a]yqsun@bjtu.edu.cn,

Keywords: Parallel Computing, MapReduce, Phoenix, Extremal Graphs, OpenMP

Abstract. Phoenix is an implementation of MapReduce on shared memory, aiming at supporting parallel computing based on multi-core/multi-processor efficiently. The extremal graph is a graph with the maximum number of edges without some given subgraphs. In this paper, by allocating the data of tasks appropriately and setting identifiers to distinguish different tasks, a parallel algorithm is proposed and used to construct the extremal graphs without hexagon. The experimental results show that the average speedup is 7.0432 on 8-core CPU and the average efficiency is 88.04% for constructing the extremal graphs of order no more than 28. Finally, three extremal graphs of order 29 without hexagon are obtained by employing the algorithm.

Introduction

We consider only finite undirected graphs without loops or multiple edges. For a graph $G = (V(G), E(G))$ of order $|V(G)|$ and size $|E(G)|$, $V(G)$ denotes the set of vertices and $E(G)$ denotes the set of edges of G respectively. K_m is the complete graph of order m, C_m is the cycle of length m. The extremal graph is a graph with the maximum number of edges containing no given subgraphs. The extremal graphs with small order can be constructed by an algorithm efficiently. However, the execution time of the algorithm grows exponentially with the increase of the orders of extremal graphs. So it seems to be difficult to solve this problem by a single-core computer, thus parallel algorithm for constructing extremal graphs based on multi-core chips will be a new trend in the future.

MapReduce model, which is introduced by Google Inc., was originally used in the environments of large-scale clusters of nodes and data center, and its high-level API is also being used to develop the data-intensive applications [1]. Phoenix is an implementation of MapReduce on shared memory, aiming at supporting computations in parallel based on multi-core/multi-processor efficiently. It contains a set of C and C++ programming API and an efficient runtime system, and it is implemented on top of P-threads [2]. In this paper, we will design a parallel algorithm based on Phoenix system to construct extremal graphs without containing hexagons.

Related Works

In this section, we will introduce the definitions and results of extremal graphs, Parallel programming, MapReduce model and Phoenix system.

Extremal graphs**.** We use $ex(H, n)$ to denote the maximum size of a H-free graph of order n. The graph of size $ex(H, n)$ is called an *extremal graph*, and let $EX(H, n)$ denote the set of all corresponding extremal graphs. Yang and Rowlinson [3] also gave the exact values of $ex(C_6, n)$ and corresponding extremal graphs for $n \le 21$. Sun, Zhao and Zhang [4] determined the exact values of $ex(C_6, n)$ and the graphs in $EX(C_6, n)$ for $22 \le n \le 26$.

Parallel programming and evaluation. The shared-memory programming is one of the most common parallel program methods. The advantages of shared-memory programming are that the global address space provides a user-friendly programming perspective to memory, and data sharing between tasks is fast and uniform. The Phoenix, which is used in this paper, is a toolkit of parallel program development based on shared memory. It handles the complicated concurrency and locality that make parallel programming difficult [2].

For parallel algorithms on multiple processors, how to measure their performance is an important issue in parallel computing. The speedup S_p and parallel efficiency E_p defined in following formulas are two common evaluating standards. Hence, we will use them to evaluate our parallel algorithm.

$$S_p = T_1 / T_p \tag{1}$$

$$E_p = S_p / p \tag{2}$$

In the formulas, p is the number of processors, T_1 is the execution time of the sequential algorithm, and T_p the execution time of the parallel algorithm with p processors.

MapReduce**.** The main operations of MapReduce are Map and Reduce methods. The Map takes as input a single <key, value> pair, and produces as output any number of new <key, value> pairs. It is crucial that the map operation is stateless, that is, it operates on one pair at a time. The Reduce takes all of the values associated with a single key, and outputs a multiset of <key, value> pairs with the same key [5]. The distributed system Hadoop is developed by Apache software foundation [6], which is a common open source implementation of MapReduce.

Phoenix system**.** Phoenix is an implementation of MapReduce for shared memory. The data structure used to communicate basic function information and buffer allocation between the user code and runtime is of type *scheduler_args_t*, whose fields are summarized in Table 1. The basic fields provide pointers to input/output data buffers and the user-provided functions. The remaining fields are optional, used to control scheduling decisions by the runtime.

Table 1 The *scheduler_args_t* data structure type

Basic Fields	
Splitter	Pointer to Splitter function
Map	Pointer to Map function
Reduce	Pointer to Reduce function
Optional Fields for Performance Tuning	
Unit_size	Pairs processed per Map/Reduce task
L1_cache_size	*L*1 data cache size in bytes

The runtime system is controlled by the scheduler, which is initiated by user code. The scheduler creates and manages the threads that run all Map and Reduce tasks. It also manages the buffers used for task communication. The programmer provides the scheduler with all the required data and function pointers through the *scheduler_args_t* structure. After initialization, the scheduler determines the number of cores used for this computation. Each core spawns a worker thread that is dynamically assigned some numbers of Map and Reduce tasks [2].

Algorithms

In this section we will describe the algorithm for constructing extremal graphs without hexagons and designing a parallel algorithm based on the MapReduce model.

Parallel Algorithm FCG_Phoenix**.** Let $S_n(C_6)$ denote the set of C_6–free graphs of order n. For a graph G, if $C_6 \nsubseteq G$, $C_6 \subseteq G + e$ for any $e \notin E(G)$, we call G a *Critical Graph*. Let $S^*_n(C_6)$ denote the set of critical graphs with order n not containing C_6, and $S^*_n(C_6, l_n)$ denote the subset of $S^*_n(C_6)$ in which the size of each graph is no less than l_n. In [4], a sequential algorithm FCG for constructing the graphs in $S^*_n(C_6, l_n)$ was descripted, we do not repeat it due to lack of space.

The parallel algorithm called *FCG_Phoenix* is shown in Fig. 1, where the value of l_N needs to be set in advance, N is the maximum order of extremal graphs for constructing. The other values of l_n($6 \le n < N$) would be calculated by the Eq. 3, the proof of Eq. 3 is in [4]. In addition, the isomorphism graphs would be removed by the algorithm employed in [4] during the construction.

$$l_n = l_{n+1} - \lfloor 2\times l_{n+1}/(n+1) \rfloor \tag{3}$$

```
1.  $S_5^*(C_6, l_5) = \{K_5\}$; $n = 6$;
2.  while ($S^*_5(C_6, l_5) \neq \emptyset$) and ($n \leq N$) do
3.        $S^*_n(C_6, l_n) = \emptyset$, $S'_n(C_6, l_n) = \emptyset$;
4.        for every $F \in S^*_{n-1}(C_6, l_{n-1})$ do
5.            $S'_n(C_6, l_n) = S'_n(C_6, l_n) \cup \{F': F' = F \cup \{v_0\}\}$;
6.        endfor
7.        Initialize task_data; unit_size; cache_size;
8.        divide $S'_n(C_6, l_n)$ into $k$ subsets $S'_{n,i}(C_6, l_n)$ ($1 \leq i \leq k$)
9.        worker begin
10.        Map task $i$ begin ($1 \leq i \leq k$):
11.          read $gn_i$, mapid;
12.          while $S'_{n,i}(C_6, l_n) \neq \emptyset$ do
13.             for every $G \in S'_{n,i}(C_6, l_n)$ do
14.                if ($C_6 \subseteq G$)
15.                    for every $v_iv_j \in C_6$ ($1 \leq i < j \leq n-1$) do
16.                        $G^* = G - v_iv_j$; //delete edge
17.                    endfor
18.                else
19.                    if $G$ is a critical graph and $|E(G)| >= l_n$
20.                        $S^*_{n,i}(C_6, l_n) = S^*_{n,i}(C_6, l_n) \cup \{G^*\}$;
21.                    endif
22.                    for every $v_0v_i \in E(\overline{G})$ ($1 \leq i \leq n-1$) do
23.                        $G^* = G + v_0v_i$; //add edge
24.                    endfor
25.                endif
26.             endfor
27.          endwhile
28.        Map task $i$ end
29.        Reduce task begin:
30.           $S^*_n(C_6, l_n) = S^*_{n,1}(C_6, l_n) \cup \ldots \cup S^*_{n,k}(C_6, l_n)$;
31.           write $gn_i$, mapid;
32.        Reduce task end
33.        worker end
34.        $n$++;
35.  endwhile
```

Fig. 1 The algorithm FCG_Phoenix

In the algorithm FCG_Phoenix based on MapReduce, a worker of Phoenix contains k Map tasks and one reduce task. As described in [4], the graphs set $S^*_n(C_6, l_n)$ are constructed from the graphs in $S^*_{n-1}(C_6, l_{n-1})$ by adding a vertex v_0 to them. We describe the parallel algorithm in detail as follows. Initially, $n = 6$ and $S^*_5(C_6, l_5) = \{K_5\}$. First, a set of some graphs is initialized, and the variables task_data, unit_size, and cache_size are initialized by Phoenix runtime system properly. Let gn_i denote the graph number processed by Map i, $1 \leq i \leq k$. Then the graphs of $S^*_{n-1}(C_6, l_{n-1})$ are divided into k parts such that $gn_1 = gn_2 = \ldots = gn_{k-1} = \lceil |S^*_{n-1}(C_6, l_{n-1})|/k \rceil$ and $gn_k = \lceil |S^*_{n-1}(C_6, l_{n-1})|/k \rceil - k \times gn_1$. At line 11, each Map gets the graph number gn_i according to its mapid, and then constructs the critical graphs by the processing called *deleting edges* and *adding edges* as shown between line 12 and line 27, which are similar to FCG. The subset of critical graphs is generated by Map i after executing the above while-loop, we denote it by $S^*_{n,i}(C_6, l_n)$. Note that the subset of graphs is divided into two parts and stored in the memory and storage respectively if the capacity of the required memory is exceeded. The Reduce merges the subsets $S^*_{n,1}(C_6, l_n)$ $S^*_{n,2}(C_6, l_n)$… $S^*_{n,k}(C_6, l_n)$ into $S^*_n(C_6, l_n)$ (line 30), in which the isomorphism graphs are removed, and thus the graphs with the maximal size of S^*_n (C_6, l_n) are the ones in $EX(C_6, n)$. In addition, Reduce needs to write mapid and gn_i to a file for constructing $S^*_{n+1}(C_6, l_n)$.

In order to make use of the MapReduce model and enable the parallel algorithm to run efficiently, we take some measures to realize FCG_Phoenix as follows.

(1) Design of the key-value pairs. MapReduce is a parallel programming model for a vast amount of data based on key-value pairs. Therefore, it is critical to properly design key-value pairs in an algorithm. Similar to [4], each key of key-value pairs is given a same specific key which does not have a real meaning, and each value is a decimal number transformed from the adjacency matrix of a graph. As an improvement, the argument that Map passes to Reduce is a pointer instead of value in our algorithm. The pointer points to a two-dimensional array which the first dimension is mapid, and the second dimension is the key of key-value pairs. Because the isomorphism graphs are removed in

each Map, we only need to remove the isomorphism graphs between different Maps in Reduce. Hence the above improvement can increase the efficiency of Reduce.

(2) The improvement of data structure *map_args_t*. In FCG_Phoenix, we need to distinguish different Maps for the data distribution and computation. However, the Phoenix system has no information of identifier of Map task. By the commutation with Yoo, one of the authors of Phoenix [7], an identifier of each Map task is added to the fields of *map_args_t*. Moreover, since the graphs of $S^*_{n+1}(C_6, l_n)$ are constructed from $S^*_n(C_6, l_n)$ in FCG_Phoenix, we need to get the exact value of $|S^*_n(C_6, l_n)|$ at the beginning. Hence, the number of graphs gn_i is added to the fields of *map_args_t*, where i is the identifier of Map.

(3) Writing files and data allocation. For each Map, the results need to be written to storage if the capacity of required memory is exceeded. However, there will exist write conflicts when the data is simultaneously written to a same file by more than one Map tasks. To solve this problem, each Map writes its result to a file named with its mapid, say Map i write data to file f_i. The main process of Reduce task is to remove the isomorphic graphs as follows. $f_1, f_2, \ldots, f_k$ are merged to one file, say $f_{storage}$ using an external sorting algorithm. Then the data in memory is merged with $f_{storage}$ to a new file f_{all}, which is the input file for the next while-loop (between line 2 and line 35).

In order to ensure load balance, we use the static allocation as the data allocating strategy, which is similar to OpenMP directive statement "#pragma omp parallel for schedule (static, size)". Assume the critical graphs in f_{all} are arranged in descending order of the size of graphs, and let each Map get the continuous data from f_{all} according to the value of *cache_size*, and call it method A. Note that the larger the edge number of a graph is, the longer the execution time is. So, as an alternative, we can assign the graphs in f_{all} to each Map as even as possible according to the size of graphs. We call it method B, and the file f_{all} is pre-processed before used in this method. In order to compare the two methods, we do an experiment for constructing extremal graphs of order no more than 26 without hexagons, with $l_{26} = 64$. Note that the total execution time is mainly used to construct the graphs of $|S^*_n(C_6, l_n)|$ for $15 \leq n \leq 24$, we show the execution time for each n in Fig. 2, respectively. The experimental results show that the algorithm with method B is faster than that with A. Hence we use the method B for data allocation in our algorithm.

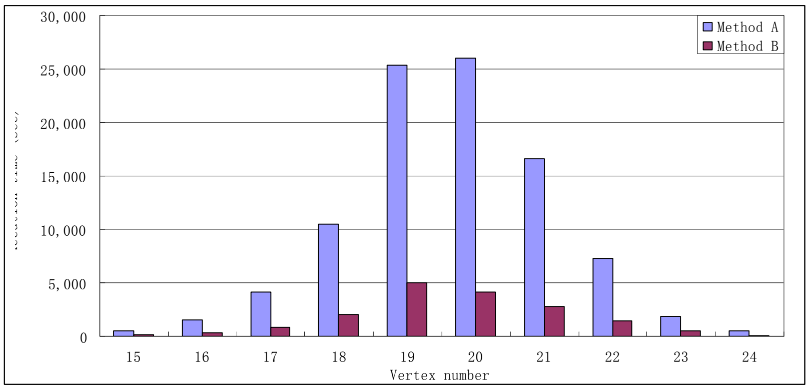

Fig. 2 Comparison of two methods for data allocation

Experiments

Environments. The hardware is a server with an eight-core 2.4 GHz, 12GB memory and 1TB disk. The software are Ubuntu 10.04 lts(x86_64), Phoenix-2.0.0, GCC 4.4.3, CodeBlocks 8.02.

The evaluation of FCG_Phoenix**.** Note that FCG_Phoenix is executed on the server with 8-core CPU, we set $p = 8$ in Eq. 1 and Eq. 2. The comparison between the results of algorithms FCG and FCG_Phoenix is shown in Table 2, where $l_{28} = 70$ and the other l_n ($6 \leq n \leq 27$) are calculated by the Eq. 3. When vertex numbers are no more than 11, both T_1 and T_p are less than one second, thus we do not give them in Table 2. The speedup and efficiency are enhanced when n is greater than 11. They are reached the peak 7.1137 and 88.92% respectively when n is equal to 21. The total execution time of FCG is 1,074,240 seconds while the one of FCG-Phoenix is 152,523 seconds. Thus, the average speedup is 7.0432, and the average efficiency is 88.04%.

Table 2 Comparison the results between two algorithms

n	l_n	$\lvert S^*_n(C_6, l_n)\rvert$	T_1	T_p	S_p	E_p
12	16	1,789	9	2	5.4000	67.50%
13	18	6,320	43	6	6.7895	84.87%
14	20	23,433	222	34	6.5294	81.62%
15	23	74,223	841	124	6.7823	84.78%
16	26	183,816	2,649	385	6.8746	85.93%
17	29	408,573	7,209	1,049	6.8701	85.88%
18	32	877,570	19,221	2,760	6.9633	87.04%
19	35	1,939,261	49,867	7,196	6.9295	86.62%
20	38	3,887,788	124,307	17,530	7.0911	88.64%
21	41	7,856,799	315,705	44,380	7.1137	88.92%
22	45	5,332,338	269,920	38,182	7.0693	88.37%
23	49	2,368,863	171,051	24,256	7.0520	88.15%
24	53	779,193	78,538	11,379	6.9020	86.28%
25	57	156,395	26,422	3,905	6.7668	84.58%
26	61	18,508	7,205	1,162	6.2005	77.51%
27	65	1,411	998	165	6.0363	75.45%
28	70	1	31	6	5.1667	64.58%

The comparison of FCG-MR and FCG_Phoenix**.** In order to test the efficiency of FCG_Phoenix, we compare its efficiency with the distributed algorithm FCG-MR in [4], which constructs extremal graphs of order no more than 26 without hexagons. Similarly, since the execution times are short when the orders are small, we just compare their efficiency for $13 \le n \le 26$ shown in Fig. 3. The experimental results show that the average efficiency of FCG-MR is 85.14% and the maximum efficiency is 88.86% while the average efficiency of FCG_Phoenix is 90.89% and the maximum efficiency is 92.40%.

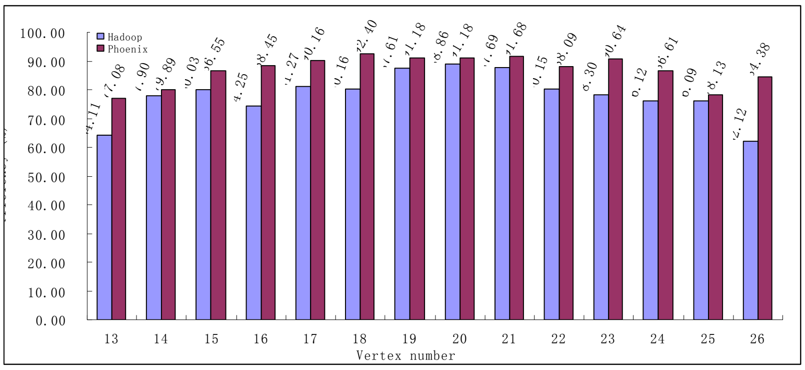

Fig. 3 Comparison of the efficiencies of two algorithms

The construction for the graphs in $S^*_n(C_6, l_n)$ for $n \le 29$. We use FCG_Phoenix to construct the graphs in $S^*_n(C_6, l_n)$ for $6 \le n \le 29$, the values of $ex(C_6, n)$ and $|EX(C_6, n)|$ are shown in Table 3. The three extremal graphs of $EX(C_6, 29)$ are shown in Fig. 4.

Table 3 The values of $ex(C_6, n)$ and $|EX(C_6, n)|$ $(6 \le n \le 29)$

n	6	7	8	9	10	11	12	13	14	15	16	17
$ex(C_6, n)$	11	13	16	20	21	23	26	30	31	33	37	40
$EX(C_6, n)$	1	1	1	1	3	3	3	2	7	11	1	4
n	18	19	20	21	22	23	24	25	26	27	28	29
$ex(C_6, n)$	41	44	48	50	52	56	59	61	64	67	70	72
$EX(C_6, n)$	32	8	2	12	16	2	1	1	3	9	1	3

Conclusion

The muti-core algorithm for constructing extremal graphs based on Phoenix is studied in this paper. Besides mapping the key-value pairs properly, we take some methods to enhance the efficiency of the

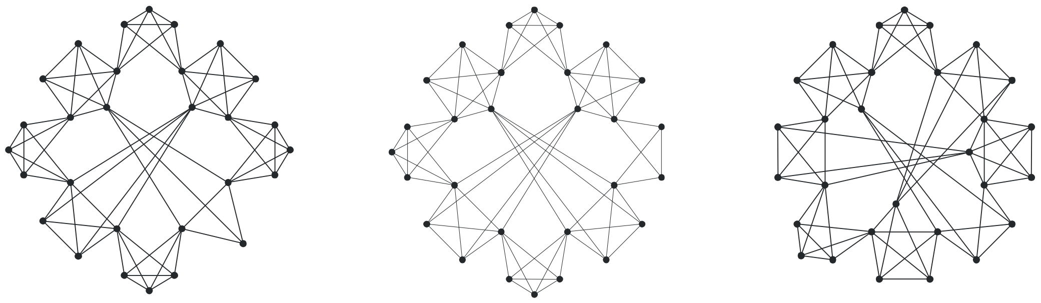

Fig. 4 The three graphs of $EX(C_6, 29)$

muti-core algorithm, including data structure improvement and data pre-processing before allocation. Next, some experiments are taken on 8-core CPU to evaluate our algorithm. The experimental results show that the average speedup and efficiency of the algorithm are 7.0432 and 88.04%, and their peaks are 7.1137 and 88.92% when the vertex number is equal to 21. Moreover, it also shows that the efficiency of our algorithm is higher than FCG-MR, a distributed algorithm based on Hadoop for constructing the graphs in $S^*_n(C_6, l_n)$ for $n \leq 26$ and $l_{26} = 64$. Finally, we use the muti-core algorithm to construct the graphs of $EX(C_6, n)$ for $6 \leq n \leq 29$.

References

[1] C. Chu, S. K. Kim, Y. Lin, Y. Yu, G. Bradski, A. Y. Ng and K. Olukotun, Map-Reduce for Machine Learning on Multicore, In Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems (NIPS), MIT Press, (2006) 281-288.

[2] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski and C. Kozyrakis, Evaluating MapReduce for Multi-core and Multiprocessor Systems, In Proceedings of Thirteenth High Performance Computer Architecture, HPCA, (2007) 13-24.

[3] Y. Yang, P. Rowlinson, On Graphs without 6-Cycles and Related Ramsey Number, Utilites Mathematica, 44 (1993) 192-196.

[4] Y. Sun, N. Zhao and R. Zhang, The Algorithm for Constructing Extremal Graphs Based on MapReduce, In Proceedings of the International Conference on Networking and Distributed Computing, ICNDC, (2012) 54-58.

[5] H. Karloff, S. Suri and S. Vassilvitskii, A Model of Computation for MapReduce, In Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA, (2010) 938-948.

[6] Apache Software Foundation, Hadoop Software, http://hadoop.apache.org, (2012).

[7] R. M. Yoo, A. Romano and C. Kozyrakis, Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System, IEEE International Symposium on Workload Characterization, (2009) 198-207.