TOPOLOGICALLY ADAPTIVE PARALLEL BREADTH-FIRST SEARCH ON MULTICORE PROCESSORS

Yinglong Xia
Computer Science Department
University of Southern California
Los Angeles, CA 90007, U.S.A.
email: yinglonx@usc.edu

Viktor K. Prasanna
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90007, U.S.A.
email: prasanna@usc.edu

ABSTRACT
Breadth-first Search (BFS) is a fundamental graph theory algorithm that is extensively used to abstract various challenging computational problems. Due to the fine-grained irregular memory accesses, parallelization of BFS can exhibit limited performance on cache-based systems. In this paper, we study the relationship between the topology of input graphs and the performance of BFS on multicore systems. We propose a model to estimate the scalability of BFS with respect to a given graph. Using this model, we propose a topologically adaptive parallel BFS algorithm on multicore systems. The proposed algorithm estimates scalability of each iteration of BFS with respect to the input graph at runtime. An adaptive barrier is developed for this algorithm, which dynamically adjusts the number of threads participating in the BFS according to the estimated scalability. In this way, we reduce the synchronization overhead. We evaluate the proposed algorithm using various graphs on state-of-the-art multicore systems. The proposed method exhibits improved performance compared with traditional parallel BFS algorithms for which the number of threads is fixed.

KEY WORDS
Breadth-first search, multicore processor, adaptive barrier.

1 Introduction

Almost all recent processors are dedicated to process simultaneous threads in order to achieve higher performance than single core processors. Typical examples of multicore processors available today include the AMD Opteron and Intel Xeon. While chip multiprocessing has been devised to deliver increased performance, exploiting the parallelism within the architecture remains an important challenge.

Some recent research provides insight into algorithm design for graph problems on multicore processors [1, 3]. Among various graph problems, Breadth-first Search (BFS) is a fundamental operation widely used for finding all the connected components in a graph, finding the shortest path between two nodes in an unweighted graph, or testing a graph for bipartiteness, etc [7]. BFS also builds the computation basis for many other scientific applications such as belief propagation in statistical inference [6].

Given an arbitrary undirected graph, Breadth-first Search starts from a designated root and constructs a breadth-first tree by traversing every node in the graph. A first-in-first-out queue can be used to manage the traversal process. In the course of scanning the neighbors of an already discovered node u, a new node v will be added to the tree if v has never been visited before. The algorithm thus visits all nodes at a certain distance from the root before visiting any nodes at further distance.

It is challenging to parallelize BFS on multicore systems. The memory accesses of BFS are highly irregular, which can lead to poor performance on cache-based systems [2]. Due to the memory latencies and synchronization costs of parallel shared memory systems, the performance of BFS is largely dependent on the topology of the input graph and the scalability is limited in most cases.

In this paper, we make the following contributions: (1) We develop a model to analyze the scalability of BFS on general-purpose multicore platforms. (2) We show the dramatic inconsistency of the performance of parallel BFS with respect to the topology of input graphs, and analyze the impact of topology on performance. (3) We propose a novel parallel BFS algorithm, which estimate the scalability of each iteration of BFS according to the graph topology. (4) We develop an adaptive barrier that dynamically adjusts the number of active threads. (5) We conduct extensive experiments to evaluate the proposed method. (6) We experimentally show that scalability can be achieved for BFS in real applications, where a certain amount of computation is associated with each node.

The rest of the paper is organized as follows: In Section 2, we briefly review related research work. In Section 3, we propose a model to analyze a parallel BFS algorithm. We propose a new parallel BFS algorithm in Section 4. Experimental results are presented in Section 5. Conclusions and discussions are presented in Section 6.

2 Background and Related Work

Breadth-first Search is a graph traversal algorithm on an undirected graph. Let $G = (V, E)$ denote the input undirected graph, where $V = \{0, 1, \cdots, N-1\}$ is the *node set* and E is the *edge set*. We assume that G is connected. Given G and a root node $s \in V$, BFS begins at s and ex-

plores *all* the neighbor nodes. Then, for each of the neighbor nodes, BFS visits the unexplored neighbor nodes, and so on, until it traverses all the nodes. During this procedure, all the nodes at the same distance to s, i.e., the nodes at the same level, must be explored before their neighbor nodes in the next level can be explored.

In this paper, the input graph G is represented as an *adjacency array*, a widely used data structure for graphs [4]. Assuming there are N nodes in G, the adjacency array consists of a *node array* and upto N *neighbor arrays*. The node array has N elements, each storing the basic information of a node in G, such as the ID of the node, the flag showing if the node has been visited, the owner of the node, the link to the corresponding *neighbor array* and the size of the neighbor array. The neighbor array of a node v consists of the IDs of nodes adjacent to v.

BFS has been extensively studied for decades. The published BFS solutions are either based on commodity processors [2, 9, 10] or dedicated hardware [5]. For example, in [10], the authors studied BFS on a cluster with distributed memory. This is different from multicore processor systems, where the same memory is shared by all the cores. In [2], the authors explored BFS on Cray MTA-2, a multiprocessor system with very different architecture from the general-purpose multicore processors. An implementation of BFS on the Cell BE processor has been proposed in [8], where the authors control the memory access using the special features of the processor. However, due to the architectural differences, the techniques provided in [8] can not be used for state-of-the-art multicore processors. Since general-purpose multicore processors, e.g., Intel Xeon and AMD Opteron, are widely used, performance analysis of BFS on such processors remains interesting. However, to the best of our knowledge, the performance of BFS on such multicore processors has not been studied.

3 Scalability of Parallel BFS

3.1 Typical Strategy for Parallel BFS

Many parallel BFS algorithms adopt a similar strategy to explore the concurrency of input graphs [8, 10]. Note that control dependencies exist during a BFS traversal: all nodes at the same level must be explored prior to any node in the next level. Thus, many parallel BFS algorithms explore concurrency within the same level, and synchronize the traversal across different levels to meet the dependency requirements. This strategy is shown in Figure 1 (a).

Before we perform BFS on a system with P threads, we assume that the vertices in the input graph $G(V, E)$ is *statically* partitioned into P subsets denoted $V_0, V_1, ..., V_{P-1}$, where $V = V_0 \cup V_1 \cup \cdots \cup V_{P-1}$ and $V_i \cap V_j = \emptyset$ for any $i, j = 0, 1, \cdots, P-1$ and $i \neq j$. Thread i, $0 \leq i < P$, explores the nodes in V_i only and thus is called the *owner* of the nodes in V_i.

The execution of each thread consists of *three* computation steps and *two* barriers. Each computation step of a thread corresponds to a round-corner block in Figure 1 (a). For Thread i, $i = 0, 1, ..., P-1$, the first computation step initializes a local queue Q_i given the root r. In the second step, all the neighbors of the nodes in Q_i are dispatched into P queues denoted by $Q_{i,0}, Q_{i,1}, \cdots, Q_{i,P-1}$, so that all nodes in $Q_{i,j}$, $j = 0, 1, \cdots, P-1$, are owned by Thread j only. In the third step, Thread i unites $Q_{j,i}$ for all j and marks the nodes at current level. The newly marked nodes are stored into Q_i for the next iteration.

The barrier steps corresponds to the bars across threads in Figure 1 (a). The first barrier guarantees that no thread will access any incompletely updated $Q_{i,j}$ in the third computation step. The second barrier reflects the dependency between the levels of nodes in BFS. The first barrier, however, can be eliminated using some implementation optimizations such as fine-grained locks or double buffering. This can reduce the synchronization cost on multicore systems. We assume hereinafter that only the second barrier exists in each iteration.

3.2 Inconsistency in Performance

Through extensive experiments, we observe that the performance of the parallel BFS in Figure 1 (a) varies dramatically with respect to input graphs on multicore systems. We show the the experiments in Section 5. Here, we simply demonstrate the variation in throughput in Table 1, where each graph has 10,000 nodes and the node degree is 16. The construction of the three graphs and implementation details are shown in Section 5. The performance is measured as the number of million edges that can be processed per second (ME/s) [8].

	Serial BFS	1 Thread	4 Threads
Good_1 graph	**193**	144	64
Good_2 graph	**177**	150	115
Rand graph	145	106	**225**

Table 1. Inconsistencies in performance (ME/s).

Table 1 shows the performance of a serial BFS and parallel BFS on a multicore system with 1 and 4 available concurrent threads, respectively. According to the results, the parallel BFS fails to show scalability for the first two graphs. However, for the last graph, we observed scalability. Considering all the graphs have the same number of nodes and node degrees, the variety of performance implies the potential relationship between the graph topology and the scalability of the parallel BFS.

3.3 Scalability and Topology

We divide each iteration into four stages shown in Figure 1 (b). The four stages are: (1) `Barrier-Computation` T_{BC}^k: This stage is defined as the time difference between threads from the first time a

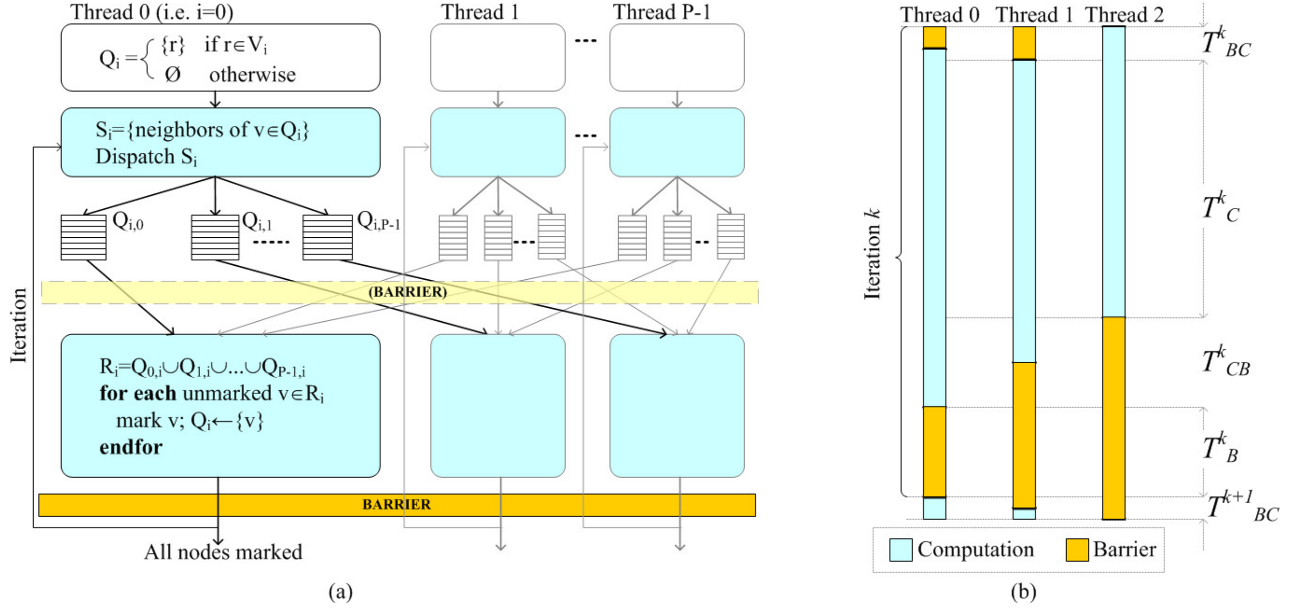


Figure 1. (a) Illustration of a typical parallel BFS algorithm. All the blocks at the same level have identical routines. (b) Each iteration of the BFS can be divided into several stages. For the sake of simplicity, we assume there are three threads only.

thread begins the computation step of Iteration k to the last time a thread begins the computation step. (2) `Pure-Computation` T_C^k: This is the duration when all the threads are in the computation step of Iteration k. (3) `Computation-Barrier` T_{CB}^k: This is the time difference between threads from the first time a thread reaches the barrier in Iteration k to the last time a thread reaches the barrier. (4) `Pure-Barrier` T_B^k: This stage is the duration when all the threads stay in the barrier.

Using the above notations, the parallel execution time of Iteration k, denoted T_P^k, is given by $T_P^k = (T_{BC}^k + T_C^k + T_{CB}^k) + T_B^k$. The term $(T_{BC}^k + T_C^k + T_{CB}^k)$ is the duration between the time the first thread begins the computation step and the time the last thread completes the computation step. Ideally, we have $T_{BC}^k = T_{CB}^k = 0$. In such a case, given the total workload of the iteration denoted T_S^k, i.e. the serial execution time of Iteration k, we can rewrite the parallel execution time as $T_P^k = T_S^k/P + T_B^k$, where P is the number of threads. However, due to load imbalance and latency of memory access in real systems, the values of T_{BC}^k and T_{CB}^k are always positive. For this reason, we introduce a factor $\alpha \in [1/P, 1]$ and let $T_P^k = T_S^k/(\alpha \cdot P) + T_B^k$. The lower bound of α is achieved when the entire workload is assigned to a single thread; the upper bound is achieved when the workload is evenly distributed across the P threads. For the sake of simplicity, we assume α is a constant for a given graph. The last term in T_P^k is T_B^k. This term depends on the implementation of the barrier and the number of threads P. More number of threads participating in the barrier leads to a higher cost of T_B^k. Since the barrier implementation is independent of the BFS, we let $T_B = T_B^k$ for any $0 \leq k < l$, where l is the total number of BFS iterations (levels).

Let $T_S = \sum_{k=0}^{l-1} T_S^k$ denote the overall serial BFS execution time on a given graph. Therefore, the overall parallel execution time T_P of BFS for the same input graph can be represented by:

$$T_P = \sum_{k=0}^{l-1} T_P^k = \sum_{k=0}^{l-1} \left((T_{BC}^k + T_C^k + T_{CB}^k) + T_B^k\right)$$
$$= \sum_{k=0}^{l-1} \left(\frac{T_S^k}{\alpha \cdot P} + T_B\right) = \frac{T_S}{\alpha \cdot P} + l \cdot T_B \quad (1)$$

To achieve scalability, we must have $T_P < T_S$. Substituting Eq. (1) for T_P in the inequation, we have

$$T_S/l > \frac{T_B}{1 - 1/(\alpha \cdot P)} \quad (2)$$

According to Eq. (2), the scalability of parallel BFS can be impacted by the following three factors:

- Average workload per level, i.e. T_S/l. This workload depends on the number of nodes per BFS level and the node degrees, which is determined by the topology of the input graph.
- The cost of the barrier T_B. This cost depends on both the barrier implementation and number of threads involved in the barrier. Using lightweight barrier and reducing the number of threads can reduce such a cost.
- Load balance and system latency reflected by α. The load balance of the parallel BFS depends on how the owners of nodes are assigned across the threads, which is often fixed in parallel BFS algorithms.

Assuming the graph partition is fixed, we focus on the first two factors. We have observed that T_S/l varies largely for different graph topologies. Let us take the three graphs

in Section 3.2 as an example. Since all the graphs have an identical node degree, T_S/l is the average number of nodes per level, which is 16, 30, 2000 for $Good_1$, $Good_2$ and $Rand$ graphs, respectively. If the topology of the input graph provides a sufficient number of nodes per level, e.g., the $Rand$ graph, then the left-hand side of Eq.(2) is likely to be greater than the right-hand side. Thus, scalability can be achieved. Otherwise, if we do not have sufficient number of nodes per level, e.g. the $Good_1$ and $Good_2$ graphs, we must reduce the value of T_B to achieve scalability. This explains the observations in Section 3.2. An approach to reduce T_B is to decrease the number of threads involved in the barrier.

4 Topologically Adaptive BFS

4.1 Complete Algorithm

We propose a parallel BFS algorithm shown in Algorithm 1. This algorithm dynamically estimates the workload in the next level and adjusts the number of threads accordingly. If there is sufficient workload in a level, more threads are invoked to traverse the nodes in parallel; otherwise, one or more threads are put to idle to reduce the synchronization cost. The threads that are not idling are called *active* threads.

The input to Algorithm 1 is a graph $G(V, E)$ with a given root r, where V and E are the node and edge sets, respectively. The node set V is statically partitioned into P disjoint subsets, where P is the number of available threads. The nodes in subset V_i are owned by Thread i only. The output is the BFS level of each node.

We briefly explain the proposed algorithm. Line 1 declares global variables. Variables $count$ and $flag$ are used by the barriers. λ denotes the number of current active threads and $workload$ is used to adjust λ dynamically. Line 2 launches BFS for all the threads. Line 3 initializes local variables: $level$ records the current BFS level; Q_i is a queue for storing nodes belonging to Thread i at current level. $flag_i$ is a binary variable used for the adaptive barrier. The rest of the algorithm can be divided into *three* parts: Lines 4-8 is the *first* part where we explore the neighbors of the root. The neighbors stored in S_i are dispatched into P queues $Q_{i,j}$, $0 \leq j < P$ according to the owners. The *second* part is Lines 10-20 where we explore the neighbors of the nodes at current level. Lines 10 and 11 identify the work for Thread i. Although Thread k is supposed to explore the nodes in $Q_{j,k}$, $\forall j$. It is possible that Thread k has been put to idle. In such a case, the nodes for Thread k will be explored by an active thread determined by Line 11. In Line 12, we define $\cup_j Q_{j,k} = Q_{0,k} \cup Q_{1,k} \cup \cdots \cup Q_{P-1,k}$, where the nodes for current thread is collected. Lines 13-15 visit the unexplored nodes and assign the level flags to them. The neighbors of the newly marked nodes are placed in S_k and partitioned into P subsets dispatched in Lines 16-17 for the next iteration. Lines 21-34 and Line 36 are the *third* part, where an adaptive barrier is proposed for dynamically adjusting the number of active threads λ. The details of the barrier are discussed in Section 4.2.

Note that the node set V is partitioned statically into P subsets with equal workload, and each subset is processed by a separate thread. When a certain thread is deactivated, it is possible that the workload is not balanced across the active threads, according to Lines 10 and 11. To avoid such a situation, we let the number of active threads $\lambda \in \{1, 2, 4, \cdots, 2^t, \cdots\}$, where t is a nonnegative integer number. This ensures that each active thread processes an equal number of subsets and therefore the workload is balanced.

4.2 Adaptive Barrier

The adaptive barrier in Lines 21-34 is the key part of Algorithm 1. Unlike ordinary barriers, the adaptive barrier dynamically activates/deactivates certain threads. The **boxes** in Algorithm 1 show the difference between the proposed barrier and an ordinary one. in Line 21, we flip a local binary variable $flag_i$. This variable is used in Line 31 to keep Thread i waiting until the barrier can be passed through. A lock is used in Line 22 to form a critical section, where we modify the value of global variables $count$, $workload$ and $flag$. The variable $count$ is increased in Line 23 once a thread reaches the barrier. In Line 24, the overall workload at current level is computed. $workload_i$ is obtained in Line 14, where $|Q_k|$ is defined as the total number of edges connecting to the nodes in Q_k. That is, the workload to process Q_k. $|Q_k|$ is computed while nodes are inserted into Q_k. Line 24 calculates the workload of all the active threads. When the number of active threads reaches a threshold, the barrier is reset in Line 27. The threshold is actually λ, the number of active threads. λ is updated in Line 26 by mapping $workload$ to the number of active threads. The update is based on the workload estimate of all threads (Line 24). If the ID of a thread is greater than λ, the thread is put to idle (Line 33). We simply let an idling thread perform an dummy loop. Line 34 is used since $flag_i$ of a thread that was just invoked can be stale. Line 36 is executed when a thread exits the while loop (Lines 9-35), which implies that all the nodes have been traversed. In this case, any thread stuck in Line 33 can be released.

The statements in the boxes distinguish the barrier from the baseline version. Note that two of these statements (Lines 24 and 26) are located in the critical section, i.e. between the lock and unlock statements. If the two statements are computationally expensive, then the barrier cost T_B increases and therefore the scalability can be adversely affected according to Eq.(2). The cost of Line 24 is very low, since there is only an integer addition. The cost of Line 26 depends on the implementation of the mapping $H(\cdot)$. A precise estimate based on Eq.(2) can lead to high cost. Note that $\lambda \in \{1, 2, 4, \cdots, 2^t, \cdots\}$. Thus, given the platform, we approximate H by looking up a sorted list: We initialize H as a list with $(\log P + 1)$ elements, where P is the number of available threads. The value of the j-th

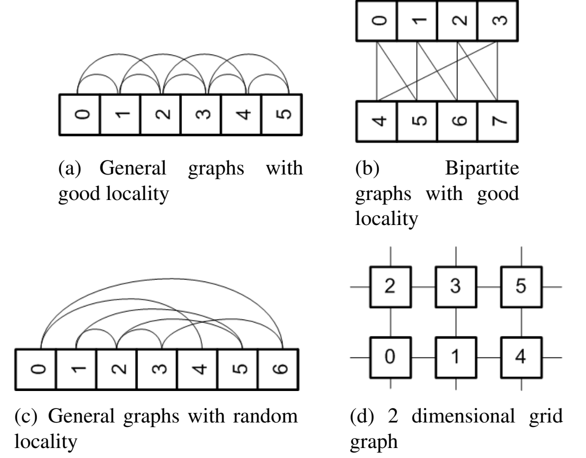
Algorithm 1 Topologically adaptive parallel BFS

Input: graph $G(V, E)$, root r, number of threads P, node set partition $V_0, V_1, \cdots, V_{P-1}$

Output: BFS level for each node: $l_v,\ \forall v \in V$

1: Initialize global variables: $count = 0$, $flag = 0$, number of active threads $\lambda = P$, $workload = 0$
2: **for** Thread i $(i = 0, 1, \cdots, P-1)$ **pardo**
3: Initialize local variables: $level = 0$, $Q_i = \emptyset$, $flag_i = 0, workload_i = 0$
4: **if** $r \in V$ **then**
5: $Q_i \leftarrow \{r\}; l_r = level; level = level + 1$
6: S_i ={neighbors of v, $\forall v \in Q_i$}
7: $Q_{i,j} = S_i \cap V_j,\ j = 0, 1, \cdots, P-1$
8: **end if**
9: **while** $\cup_i Q_i \neq \emptyset,\ i = 0, 1, \cdots, P-1$ **do**
10: **for** $k = 1$ to P **do**
11: **if** $k\%\lambda = i$ **then**
12: $Q_k = \emptyset,\ R_k = \cup_j Q_{j,k},\ j = 0, 1, \cdots, P-1$
13: **for all** $v \in R_k$ and v is unmarked **do**
14: mark v; $l_v = level$; $Q_k \leftarrow \{v\}$; $workload_i = workload_i + |Q_k|$
15: **end for**
16: S_k ={neighbors of v, $\forall v \in Q_k$ }
17: $Q_{k,j} = S_k \cap V_j,\ j = 0, 1, \cdots, P-1$
18: **end if**
19: **end for**
20: $level = level + 1$
——— adaptive barrier begin———
21: $flag_i = 1 - flag_i$
22: **lock**
23: $count = count + 1$
24: $\boxed{workload = workload + workload_i}$
25: **if** count = λ **then**
26: $\boxed{\lambda = H(workload, P);\ workload = 0}$
27: $count = 0, flag = flag_i, workload_j = 0$
28: **unlock**
29: **else**
30: **unlock**
31: **while** $(flag \neq flag_i)$; **endwhile**
32: **end if**
33: $\boxed{\textbf{while } (i > \lambda - 1); \textbf{ endwhile}}$
34: $flag_i = flag$
——— end ———
35: **end while**
36: $\lambda = \infty$ {release threads stuck at the barrier, if any}
37: **end for**

element is the workload enough for 2^j active threads, $0 \leq j < P$. Then, by comparing $workload$ with the values in the list, we map $workload$ to λ. Using binary search, the mapping needs $O(\log(\log P))$ comparisons. That is, for a system with 8 threads, there are no more than 2 comparisons. Therefore, the statements in the boxes have almost no impact on the execution time of the adaptive barrier. On the other hand, because of the statement in boxes,



(a) General graphs with good locality

(b) Bipartite graphs with good locality

(c) General graphs with random locality

(d) 2 dimensional grid graph

Figure 2. Illustration of the topologies of input graphs.

the adaptive barrier reduces the cost of data reloading in the critical section. The baseline barrier needs to reload the cache line containing *count* for $(P-1)$ times, since every time a thread updates *count* the cache line becomes invalid for other threads. In contrast, if we organize *count* and $workload$ to be in the same cache line, we just need to reload the cache line $(\lambda - 1)$ times. In addition, load balancing becomes more challenging if more threads are involved. That is, T^k_{BC} and T^k_{CB} in Figure 1 tend to be shorter for the proposed method compared with the baseline barrier, since $\lambda \leq P$.

5 Experiments

The experiments were conducted on the following two state-of-the-art multicore systems: (1) A dual Intel Xeon 5580 (Nehalem) quadcore system. Intel Xeon 5580 (Nehalem) is a 64-bit quadcore processors, running at 3.2 GHz with 8 MB shared L3 cache. The system had 16 GB DDR3 memory. The operating system was Red Hat Enterprise Linux WS Release 4 (Nahant Update 7). (2) A dual AMD Opteron 2350 (Barcelona) quadcore platform, running at 2.0 GHz with 2 MB shared L3 cache. The system had 16 GB DDR2 memory, shared by all the cores, and the operating system was CentOS version 5 Linux. For both systems, *gcc* 4.1.2 was used with flags "-O2 -pthread" to generate the executables. We were the only user on both the systems when conducting the experiments. We had consistent results on both systems. Due to space limitation, we show the results on the AMD system only.

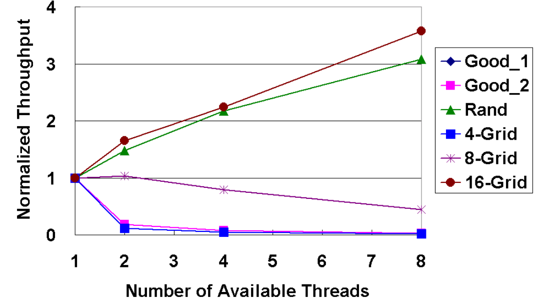
We evaluated the performance of parallel BFS using various graph topologies including: (1) General graphs with good data locality, denoted as *Good_1*. The nodes were arranged as shown in Figure 2(a). Each node, marked by the numbers in the squares, had edges connecting to its left and right neighbors. (2) Bipartite graphs with good data locality, denoted as *Good_2*. The nodes were arranged in two sets as shown in Figure 2(b), where each node had

edges connecting to a given number of nodes with adjacent IDs in the other set. (3) Random graphs, denoted as $Rand$. Each node had edges connecting to a random subset of the nodes. Figure 2(c) illustrates one such graph. (4) k dimensional grid graph, denoted as k-$Grid$. The nodes are laid out as hyper grid. Each node connects to its neighbors in k dimensional grid. Figure 2(d) shows a 2 dimensional grid graph (2-$Grid$). Note that a node in k-$Grid$ graph has $2k$ neighbors. For each of the above topologies, we generated a set of graphs with various number of nodes N, including 10000, 100000 and 1000000. We also varied the node degree d from 4 to 32. The graph sizes varied from 0.8 MB to 80 MB.

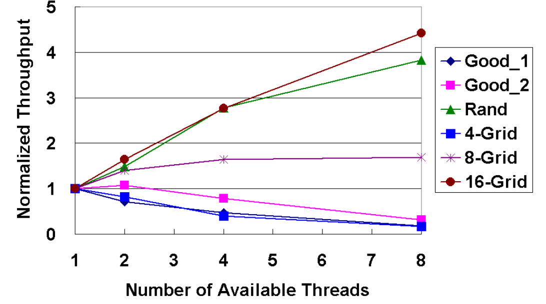
We stored the graphs according to the following data layout. The nodes of a graph were stored from 0 to $(N-1)$ contiguously in the memory, where each node had a pointer to an array storing its neighbors. For $Good_1$, if nodes i and j were neighbors, then we have $i - d/2 < j \leq i + d/2$ where $j \neq i$ and $0 \leq j < N$. For $Good_2$, if nodes i and j were neighbors, then we have $C \leq j < C + d$ and $i \neq j$, where $0 \leq C < N - d$. For both $Good_1$ and $Good_2$, the neighbors of a node were therefore stored close to each other in the memory. For $Rand$, we still stored the nodes in the increasing order of their IDs, but the neighbors of node i were random nodes among $0 \cdots (N-1)$. For k-$Grid$, nodes i and j were connected if and only if they were neighbors in the k dimensional grid.

In our experiments, we used three baseline methods for comparison: (1) `Serial BFS` was an implementation of a straightforward sequential BFS algorithm. (2) `Pthread BFS` was the parallel BFS shown in Figure 1(a), where we utilized the ordinary Pthreads barriers. (3) `Spinlock BFS` was also the parallel BFS shown in Figure 1(a), but we implemented the barriers using spinlocks. Spinlocks avoid overhead due to operating system rescheduling or context switching.

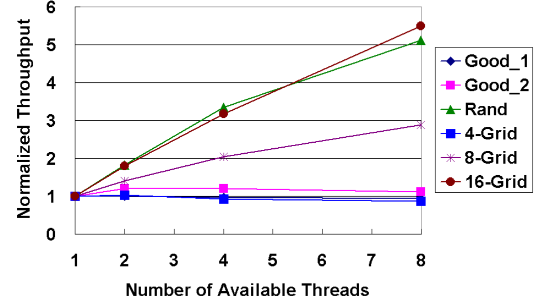
In Figure 3, we show the significant inconsistencies of parallel BFS performance with respect to various input graphs. Each graph had 1,000,000 nodes. The node degree was 16 for $Good_1$, $Good_2$ and $Rand$, and $2k$ for k-$Grid$ graphs. The Y-axis was the normalized throughput: We measured the number of million edges processed per second (ME/s) by a BFS implementation with respect to various number of available threads, where each thread was bound to a separate core. For the sake of comparison, we normalized the throughputs for each graph. Figure 3(a) and 3(b) illustrate the normalized throughputs of the Pthread BFS and Spinlock BFS, respectively. Figure 3(c) demonstrates the results of the proposed method i.e. Algorithm 1. As we can see from Figure 3, the throughput varied dramatically for different graphs. The two baseline methods led to much worse performance for graphs such as $Good_1$, $Good_2$ and 4-$Grid$, when more cores (threads) were used. In contrast, the proposed method was more robust for various graph topologies. When the number of cores increases, it did *not* degrade performance for any input graph considered by us.



(a) Throughput of the Pthread BFS baseline method.



(b) Throughput of the Spinlock BFS baseline method.



(c) Throughput of topologically adaptive parallel BFS.

Figure 3. Stark contrast of throughputs achieved by parallel BFS with respect to various graph topologies.

In Table 2, we compared the throughput of our method, the Pthread BFS and the Spinlock BFS, when all eight cores were available. The throughput is defined as the number of million edges processed per second (ME/s). The input graphs were the same as those used in Figure 3. As we can see, the proposed method consistently showed superior performance compared with the two baseline methods. Note that, although some graphs such as $Good_2$ exhibited little scalability in Figure 3, it had higher throughput since such graph topologies ensure better data locality.

In Figure 4 and Figure 5, we show the impact of the node degree and graph size on the performance of the proposed method with respect to various graph topologies. Since the degree of graph k-$Grid$ is fixed by the parameter k, we changed the node degree of graphs $Good_1$, $Good_2$ and $Rand$ only. The Y-axis of Figure 4 is defined as the

	Pthread BFS	Spinlock BFS	Our method
$Good_1$	2.20	28.84	**167.19**
$Good_2$	4.29	57.03	**201.02**
$Rand$	63.42	77.14	**104.67**
4-$Grid$	1.21	15.89	**88.95**
8-$Grid$	51.34	202.46	**219.07**
16-$Grid$	219.91	273.22	**311.93**

Table 2. Throughput (ME/s) versus input graph topologies.

execution time of the proposed method with 8 threads divided by the execution time of a serial BFS. In Figure 5(a) and (b), we generated the same type of graphs with 100,000 and 10,000 nodes, respectively. In this figure, we show the speedup of the proposed method with respect to the serial BFS, using 1 and 8 threads. The speedups of the parallel BFS with 1 thread were very close to 1. This implied that the overhead of the parallel BFS was very low, if there is no synchronization across the threads. The results with 8 threads were consistent with our observations in Figure 3(c). Thus, the proposed method is robust for various node degrees and graph sizes.

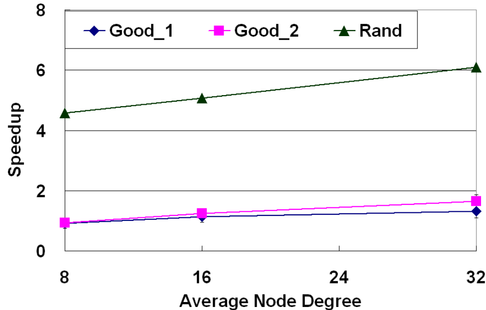
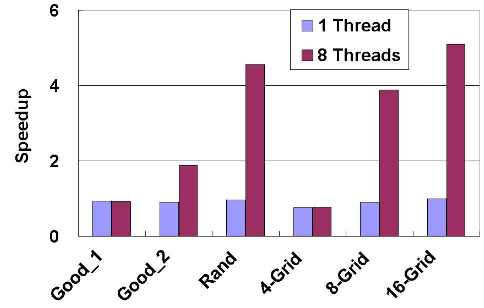
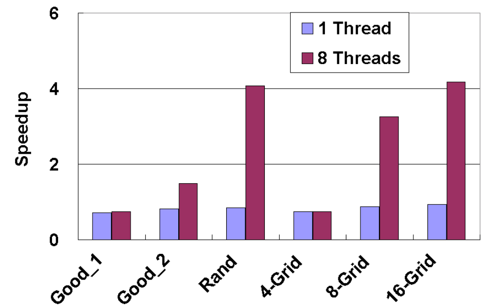


Figure 4. Impact of the node degree on speedup of topologically adaptive parallel BFS.

Figure 6 illustrates the impact of each stage of the parallel BFS on performance. The stages are defined in Section 3.3. The upper part shows the results of Spinlock BFS and the lower part shows that of our proposed method. The box includes graphs that exhibited scalability in Figure 3(b). A common characteristic of these graphs was that T_C was relatively high and T_B was very low. For the graphs which failed to achieve scalability, T_B was very high. This explained the observed relationship between graph topologies and parallel performance. For graphs with many BFS levels (i.e. large T_B) but a few nodes/edges between levels (i.e. small T_C), the synchronization overhead dominated performance. Thus, the performance degraded for those graphs using the baseline methods. Our proposed algorithm, however, dynamically adjusted the number of threads to control the synchronization overhead and therefore exhibited better performance. Comparing the subfigures in Figure 6, our proposed method required less threads



(a) Speedup of topologically adaptive parallel BFS with medium size graphs.



(b) Speedup of topologically adaptive parallel BFS with small size graphs.

Figure 5. Impact of the graph size on speedup of topologically adaptive parallel BFS.

to process $Good_1$, $Good_2$ and 4-$Grid$. Note that T_{CB} is a noticeable part of the execution time. This implies that load balancing can be further improved. T_{CB} was decided by the fixed partitioning of the node set.

Although the proposed method showed better performance compared with the baseline methods, for certain graphs such as $Good_1$, we had little improved scalability as shown in Figure 3. The observation in Figure 6 implies that poor scalability resulted from insufficient computational workload between BFS levels. Using more threads for such graphs can only make the overhead to dominate and therefore degrades the throughput.

To confirm the above analysis, we considered a real problem in belief propagation, where we traversed a graph with node tasks [6]. Once a node was visited for the first time, we executed the corresponding task. The task execution increased the computation workload between BFS levels. In Figure 7, we performed a dense operation, e.g. matrix multiplication, at each node during BFS traversal. We conducted the experiment with various node data sizes: 10×10, 30×30, and 50×50. According to Figure 7, when the node task is computationally expensive, we had good speedup for all the graphs, which shows the practical benefit of using parallel BFS for real applications on general-purpose multicore systems.

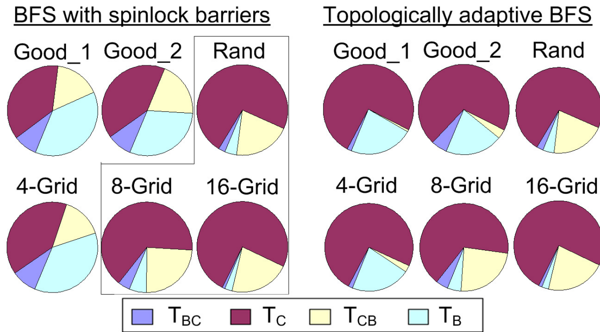


Figure 6. Percentage of execution time for each stage of parallel BFS with respect to various graph topologies.

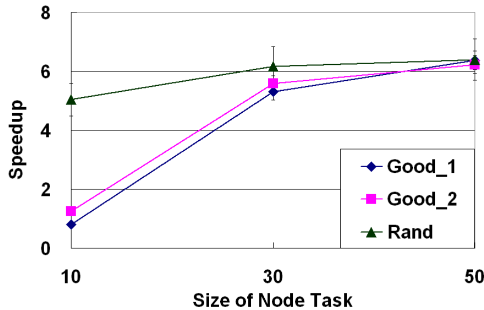


Figure 7. Impact of the size of node tasks on speedup of topologically adaptive parallel BFS.

6 Conclusions

In this paper, we analyzed the scalability of a traditional parallel BFS algorithm on general-purpose multicore systems. Based on the analysis, we pointed out that, for graphs with limited workload between BFS levels, the synchronization overhead dramatically degrades the performance. To reduce the synchronization overhead, we proposed a topologically adaptive parallel BFS algorithm. The algorithm estimates the scalability of each iteration of BFS with respect to a given graph. We developed an adaptive barrier to dynamically adjust the number of threads according to the estimated scalability. We experimentally showed that the proposed method reduced the synchronization overhead across the threads and exhibits improved performance. In addition to the parallel BFS, the contributions in this paper can also be used to improve the performance of other graph problems on multicore systems.

- **Acknowledgements:** This research was partially supported by the National Science Foundation under grant number CNS-0613376. NSF equipment grant CNS-0454407 is gratefully acknowledged. We would also like to thank Professor Bo Hong and Edward Yang for helpful discussion.

References

[1] A. Ali, L. Johnsson, and J. Subhlok. Scheduling fft computation on smp and multicore systems. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 293–301, New York, NY, USA, 2007. ACM.

[2] D. A. Bader and K. Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2. In *ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing*, pages 523–530, Washington, DC, USA, 2006. IEEE Computer Society.

[3] D. A. Bader and K. Madduri. A graph-theoretic analysis of the human protein-interaction network using multicore parallel algorithms. *Parallel Computing*, 34(11):627–639, 2008.

[4] D. K. Blandford, G. E. Blelloch, and I. A. Kash. An experimental analysis of a compact graph representation. In *Proceedings of the Sixth Workshop on Algorithm Engineering and Experiments*, pages 49–61, 2004.

[5] M. deLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. Uribe, T. Knight, and A. DeHon. Graphstep: A system architecture for sparse-graph algorithms. In *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, pages 143–151, 2006.

[6] J. Gonzalez, Y. Low, and C. Guestrin. Residual splash for optimally parallelizing belief propagation. In *Proceedings of AI & Statistics*, pages 1–8, 2009.

[7] J. Kleinberg and E. Tardos. *Algorithm Design*, pages 79–82. Prentice Hall Inc.

[8] F. Petrini, O. Villa, and D. Scarpazza. Efficient breadth-first search on the Cell BE processor. *IEEE Transactions on Parallel and Distributed Systems*, (10):1381 – 1395, 2007.

[9] V. Subramaniam and P.-H. Cheng. A fast graph search multiprocessor algorithm. In *Proc. of the Aerospace and Electronics Conf*, pages 1–8, 1997.

[10] 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 *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, pages 25–32, Washington, DC, USA, 2005. IEEE Computer Society.