TCS-TR-A-12-57

TCS Technical Report

Shared-Memory Parallel Algorithms for Frontier-Based Search

by

Shogo Takeuchi, Jun Kawahara, Akihiro Kishimoto and Shin-ichi Minato

Division of Computer Science

Report Series A

April 26, 2012

Hokkaido University
Graduate School of
Information Science and Technology

Email: minato@ist.hokudai.ac.jp

Phone: +81-011-706-7682
Fax: +81-011-706-7682

Shared-Memory Parallel Algorithms for Frontier-Based Search

Shogo Takeuchi
JST ERATO Minato Project
Graduate School of Info. Sci. and Tech.
Hokkaido University
Sapporo 060-0814, Japan

Jun Kawahara
JST ERATO Minato Project
Graduate School of Info. Sci. and Tech.
Hokkaido University
Sapporo 060-0814, Japan

Akihiro Kishimoto
Dept. of Math. and Computing Sciences
Graduate School of Info. Sci. and Eng.
Tokyo Institute of Technology,
Tokyo 152-8550, Japan

Shin-ichi Minato*
Division of Computer Science
Graduate School of Info. Sci. and Tech.
Hokkaido University
Sapporo 060-0814, Japan

April 26, 2012

(Abstract) Knuth's Simpath algorithm is an efficient algorithm enumerating all paths between two locations. This paper presents three approaches to parallelizing frontier-based search in Simpath in shared-memory environments: *node-based*, *range-based* and *edge-based* approaches. Our results on solving grid graphs show that the lock-free edge-based approach performs best and achieves seven-fold speedup with 32 CPU cores, while the others suffer from severe synchronization overhead due to locks, resulting in performance saturation with more than 12 cores.

1 Introduction

Enumerating all solutions efficiently has been a subject of algorithm research for decades. In particular, computing all the paths between two locations has been a fundamental research topic due to many real-world applications such as network reliability analysis [4], solving and generating puzzle instances [16] and finding configurations minimizing the loss of energy in the electric power network [5].

Given two vertices s and t in graph G, Knuth's Simpath algorithm presented in the latest volume of "The Art of Computer Programming" (exercise 225 in Section 7.1.4, [8]) computes all the loop-free paths from s to t with a compact representation of Zero-suppressed Binary Decision Diagram (ZDD) [10], a variant of Binary Decision Diagram (BDD) [2].

*He also works for JST ERATO Minato Project.

Simpath performs breadth-first search called *frontier-based search (FBS)* to build a binary decision graph by marking edge e_i in G as selected or unselected and by checking whether selecting/not selecting e_i leads to a dead-end or an actual s-t path. When FBS generates two nodes n_1 and n_2 with the identical set of *frontier vertices* S used to enumerate all possible connections among S and t, n_1 and n_2 are merged into one node to avoid duplicate search effort. FBS continues this procedure until considering all the combinations of edges. Simpath then reduces the binary decision graph to a ZDD. To our best knowledge, Simpath is so far the most efficient algorithm that is difficult to achieve further performance improvement. However, enumerating paths is still a computationally intensive, difficult task, because its computational complexity is #P-complete.

Parallel computing is one way for achieving speedups and has become important due to the wider availability of multi-core CPUs. Moreover, since the speed of the individual CPU core has been less rapidly improved recently, parallelizing algorithms will become the only way to obtain benefits from the hardware soon.

Efficient parallelization of Simpath is a non-trivial issue. For example, serial FBS uses the hash table to check if two nodes are merged. In shared-memory parallel FBS, if duplicate nodes are allocated to various threads, the hash table must be shared among the threads for duplicate detection. That is, parallel Simpath may incur the synchronization overhead (idle time) caused by mutual exclusion (mutex) lock on shared data, which never arises in serial FBS. Not merging duplicates is not a choice of parallel Simpath, since it would result in an exponential increase of searching extra nodes explored only by parallel search.

This paper presents the first attempt to parallelize FBS of Simpath. The advantage of our work is that the speed of FBS is improved for free, once parallel FBS is implemented and when new techniques for increasing the number of cores are developed from the hardware perspective. We develop three shared-memory parallel algorithms: *node-based*, *range-based* and *edge-based* approaches. These approaches guarantee that two nodes with the same set of frontier vertices are always detected as a duplicate. The node-based approach uses a shared task queue among threads. However, non-negligible overhead caused by mutex lock operations is incurred for managing the task queue. Although the range-based approach alleviates the overhead of the node-based approach by exploiting locality of task allocation, it still suffers from the overhead regarding locks. In contrast, the edge-based approach uses a different work distribution strategy based on the levelized structure of the binary decision graph. This approach can access the hash tables and task queues with no locks. Although one drawback is that the edge-based approach requires larger hash tables, they fit into memory of modern PCs in our experiments. Although Simpath has a procedure of reducing nodes to build a ZDD, parallelizing this procedure currently remains future work.

We ran experiments to measure the performance of the above algorithms using up to 32 CPU cores on grid graphs, a representative domain used to investigate ideas for many real-world applications including geographical information process-

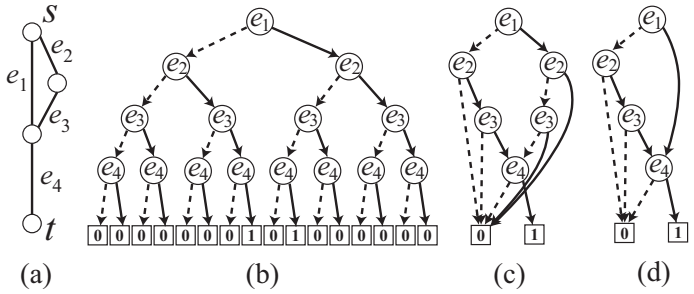


Figure 1: Binary decision tree, Simpath's DAG and ZDD

ing and network reliability analysis. Our results show that the edge-based approach performed best and yielded about seven-fold speedups on 32 cores.

2 Sequential Simpath

This section first deals with a naive approach to path enumeration and briefly introduces ZDD. It then describes the sequential Simpath algorithm.

2.1 Naive Approach and ZDD

Let $\{e_1, e_2, \cdots, e_m\}$ be a set of edges and $\{s, v_1, v_2, \cdots, v_\ell, t\}$ be a set of vertices of graph G. Vertices s and t are respectively the source and destination. Edges $e_1, \cdots, e_m$ are ordered in a breadth-first manner starting at s. The task of Simpath is to calculate all the paths from s to t without forming any cycles.

A naive approach to solve this problem is first to assign either 0 or 1 to each variable e_i which indicates that edge e_i is respectively unselected or selected and then to check if the set of selected edges constructs an s-t path. This can be seen as a process of building a binary decision tree that represents a Boolean function. Figure 1 (b) illustrates an example of the binary decision tree representing all the s-t paths of a graph shown in Figure 1 (a). The outgoing dotted and solid lines (called *0-arc* and *1-arc*, respectively[1]) from circle e_i (called a *node*) indicate respectively values 0 and 1 are set to e_i. A Boolean value inside a square (called a *terminal node*) indicates whether or not an s-t path is formed with a full assignment to $e_1, e_2, \cdots, e_m$. For example, since selecting edges e_1 and e_4 forms an s-t path, the outcome for assignment $\{e_1 = 1, e_2 = 0, e_3 = 0, e_4 = 1\}$ is 1.

The binary decision tree requires $2^{m+1} - 1$ nodes to represent all the paths for the graph with m edges. In contrast, Simpath leverages ZDD that compresses the set of paths as a directed acyclic graph (DAG) by removing all nodes without

[1] Although they are usually called "edges" in the BDD research community, we call them arcs to avoid the confusion with edges in the graph.

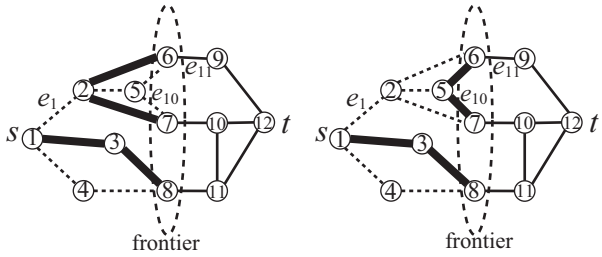


Figure 2: An Example of nodes considered to be identical

Algorithm 1 Frontier-Based Search

```
1: N_1 ← {n_root} // n_root is the root node labeled by e_1.
2: N_i ← ∅ for i = 2, ..., m + 1.
3: for i = 1 to m do
4:     for each n ∈ N_i do
5:         for each x ∈ {0,1} do // Assign e_i = x.
6:             n' ← CheckTerminal(n, i, x) // Returns 0-terminal, 1-terminal or nil.
7:             if n' = nil then // n' is not 0/1-terminal.
8:                 Create a new node n' and set it to n'
9:                 if there exists n'' ∈ N_i s.t. n'' is equivalent to n' then
10:                    n' ← n''
11:                else
12:                    N_{i+1} ← N_{i+1} ∪ {n'}
13:                end if
14:            end if
15:            Create x-arc to connect n and n'.
16:        end for
17:    end for
18: end for
```

which the equivalent set of paths can be represented. In building a ZDD, if node n whose 1-arc directly points the terminal node with the value of 0 (called the *0-terminal node*), n is removed from the current DAG and the subgraph pointed by n's 0-arc is directly connected to n's parents p with the arc that used to connect n and p. A unique form of ZDD is obtained until no node can be removed with this reduction rule (see [10] for details). Figure 1 (d) illustrates the ZDD with an equivalent representation to Figure 1 (b). In case of $e_1 = e_2 = 0$, no s-t path can be generated irrespective of the value assignment of e_3 and e_4. The nodes with assigning values to e_3 and e_4 do not therefore exist in the ZDD.

2.2 The Simpath Algorithm

As in [4, 13], instead of first building a binary decision tree and then transforming it to its corresponding ZDD, Simpath directly constructs a DAG that is later efficiently reduced to the ZDD for the sake of time and space efficiency. For example, Simpath directly builds the DAG shown in Figure 1 (c) for the graph in Figure 1

(a) and then transforms it to ZDD in Figure 1 (d). An explanation of the algorithm reducing the constructed DAG to ZDD is omitted here (see [8, pp 216–218]), since parallel reduction is currently beyond the scope of the paper.

To build a DAG, Simpath performs breadth-first search in a top-down manner from s, which we call *frontier-based search (FBS)*. During the breadth-first search, FBS prunes out combinations of edge selections that generate no s-t path. For example, if node n with the partial assignment of $\{e_1 = val_1, \cdots, e_k = val_k\}$ forms either a cyclic path or a spanning branch connected to s or t, no s-t path can be formed irrespective of the remaining value assignment. Therefore, n is connected to the 0-terminal node with the arc of value val_k.

The heart of the enhancement to FBS is to merge nodes that always return the identical binary outcomes with assignments for the remaining variables. Figure 2 illustrates two nodes that are merged into one. Assume that the binary values are already assigned to variables $e_1, e_2, \cdots, e_{10}$ and that FBS is about to assign either 0 or 1 to variable e_{11}. In both diagrams vertices 1, 6, 7 and 8 are the tip vertices of two partial paths. This indicates that vertices 6, 7 and 8 called *frontier vertices* must be passed through in building an s-t path with the remaining assignment. That is, irrespective of $e_1, e_2, \cdots, e_{10}$, if a partial assignment A for the remaining variables contributes to generating an s-t path in the left diagram, it also holds for the right diagram. Analogously, if A forms no s-t path in the left diagram, it does not yield a path in the right diagram either. As a result, FBS safely regards these nodes to be the "same" to process the unprocessed edges. The "mate" structure is used to efficiently detect frontier vertices and cyclic paths (see [8] for details). The hash table is used to check whether two nodes are equivalent or not in terms of their frontier vertices. The hash keys are computed based on the mate information of the nodes.

Algorithm 1 shows the pseudo-code of FBS. CheckTerminal checks if a node n with an assignment of $e_i = x$ generates the terminal node (i.e. an s-t path or a dead-end). Duplicate node detection is performed in lines 9-13.

3 Parallel Frontier-Based Algorithms

This section presents our new shared-memory parallel frontier-based algorithms.

3.1 Node-Based Approach

Let a node of variable e_i be a node with *level* i and N_i be a set of nodes with level i. The node-based approach (NBA) starts processing the nodes in N_{i+1} after all the nodes in N_i are expanded and all the possible nodes generated from the nodes in N_i are saved in N_{i+1}. This indicates that the synchronization overhead is incurred whenever NBA switches to processing nodes in the next level.

Assume that NBA is about to process N_i to generate the successor nodes that are saved in N_{i+1}. NBA uses two shared first-in-first-out (FIFO) task queues to represent N_i and N_{i+1}, respectively, and one shared hash table for duplicate node detection. In this paper we assume that the hash table is implemented as chaining. Each thread dequeues one node n from N_i with a mutex lock operation. The lock is released when the thread obtains n from N_i. The thread then generates two nodes n_1 and n_2 by respectively assigning variable e_i to 0 and 1 and checks the shared hash table to check whether n_1 and n_2 are duplicates or not. If n_1 (or n_2) is a new node, it is saved in the shared hash table with a lock operation[2]. The thread next stores n_1 (or n_2) in N_{i+1} with a lock operation.

NBA is the simplest strategy that tries to evenly distribute work to threads. However, since N_i and N_{i+1} are accessed very frequently by all the threads, the synchronization overhead caused by the contention for N_i and N_{i+1} limits the performance improvement especially with a large number of threads.

3.2 Range-Based Approach

The range-based approach (RBA) is an improvement to NBA. When threads deal with nodes in N_i, RBA alleviates the synchronization overhead by eliminating the mutex lock operations on N_i in NBA. More specifically, when RBA starts to proceed to level i, N_i is partitioned into disjoint sets so that each thread can locally deal with the nodes with no locks. The FIFO queue for N_{i+1} is shared among the threads as in NBA. The lock operation is therefore required to consistently manage N_{i+1}, which still incurs the synchronization overhead. Additionally, locks are required for accesses to the shared hash table.

3.3 Edge-Based Approach

The edge-based approach (EBA) is a completely lock-free approach by modifying the way of managing nodes. Let k be the number of threads and j be $i \bmod k$ ($1 \leq i \leq m$) where m is the number of edges. In EBA, thread j must hold N_i. Each of N_i is represented as a FIFO queue. When thread j dequeues node n in N_i and generates n_1 and n_2 by assigning variable e_i to 0 and 1, respectively, n_1 and n_2 are moved to thread l where l is $(i+1) \bmod k$. The duplication check of n_1 and n_2 is performed by thread l by using thread l's local hash table that requires no lock operations to read/write information there.

At a first sight, locks are apparently required in case of enqueueing from and dequeueing to N_i a node, because N_i is accessed by two threads. However, at the price of occasionally increasing the synchronization overhead, N_i can be represented as a lock-free structure by leveraging the property that there is only one

[2] If read operations to the hash table are lock-free, some other threads may have saved n_1 or n_2 in the hash table after the first duplicate check. Therefore, the duplicate check procedure must be performed again.

writer thread and one reader thread to access N_i. More specifically, assume that the FIFO queue is implemented as an array and two indexes are prepared: one pointing a node to dequeue and the other indicating the place to enqueue a node. Then, the writer increments the index of N_i after it stores a node in N_i. When the writer's index is not incremented yet but the node is already in N_i, the reader might consider that no node exists in N_i. In this case, the reader tries to find work in another N_n or waits until the writer increments its index for N_i.

In a way, EBA behaves like a depth-first search except the fact that when successor nodes are generated, they must be moved to a different thread. When each thread has enough amount of work, EBA starts initiating parallelism.

EBA assumes that m is much larger than k. This assumption holds for many practical domains including [4, 16, 5]. In our test domain explained in Section 4, m ranges between 456 and 480 and k is at most 32.

Unlike NBA and RBA, EBA processes nodes with different levels in parallel. This indicates that EBA must preserve nodes previously stored in the hash table during FBS. In contrast, as described in [8], if NBA and RBA start processing N_i, the information on N_{i-1} can be safely discarded from the hash table by outputting N_{i-1} to a file that is later used to reduce the nodes to build a ZDD. This indicates that the amount of memory required for EBA is larger than the others. However, Knuth's node reduction approach using the file is a sequential algorithm. If the node reduction approach is parallelized, the nodes removed from memory must be most likely to be preserved in memory, which implies that both NBA and RBA would require the same amount of hash tables as EBA.

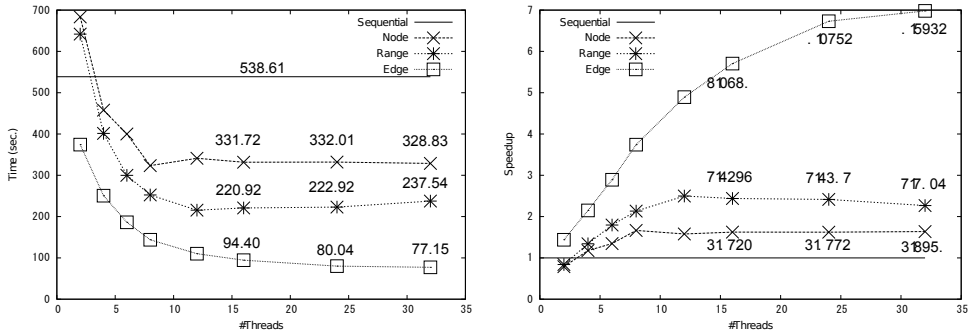
Because the order of node expansion in each level in EBA is identical to that in sequential FBS, both algorithms construct the same DAG. In contrast, NBA and RBA might construct a different DAG with the same size. However, the node reduction algorithm reduces these different DAGs to the same ZDD.

4 Experiments

4.1 Experimental Setting

We performed experiments on a machine that consists of eight quad-core 3.1 GHz AMD Opteron 8393 SE processors (i.e, 32 CPU cores in total) and 512 GB DDR2 memory shared among cores. Each implementation used at most 10 GB memory that fits into the memory size of modern PCs. All the algorithms were implemented in C++ with Boost C++ (version 1.46.1) and Standard Template Libraries and were compiled with the version 4.1.2 of the GNU C/C++ compiler. To avoid high overhead for locks, we used not only the atomic function offered by GNU C++ but also a highly efficient spin lock implementation in the Open Shogi Library[3], which uses the "xchgl" assembly operation.

[3] `http://gps.tanaka.ecc.u-tokyo.ac.jp/gpsshogi/pukiwiki.php?OpenShogiLib`



(a) Execution time (b) Speedup

Figure 3: Performance graphs on solving the 16×16 complete grid graph. We measured the cases of using 2, 4, 6, 8, 12, 16, 24 and 32 cores.

Due to inconsistent executions in parallel algorithms, we ran each method three times in solving problems and calculated the average runtime.

4.2 Experimental Results Using a Complete Grid Graph

We prepared the 16×16 complete grid graph consisting of 256 vertices and 480 edges to perform thorough empirical analysis of each algorithm. Since there are many paths leading to the same vertex, grid graphs are a difficult domain to enumerate paths, often resulting in an astronomically large number of paths. Knuth also used this domain to test Simpath.

The original search space of this problem is 3.12×10^{144} and the number of solution is 2,266,745,568,862,672,746,374,567,396,713,098,934,866,324,885,408,319,028. Such solutions are represented as a ZDD with 464,004,180 nodes and sequential FBS' DAG contains 883,640,712 nodes.

Figure 3 shows the execution time and speedups of NBA, RBA and EBA when the number of CPU cores (shown as "#Threads") is varied. The horizontal solid line indicates sequential baseline FBS.

Although all the approaches with more than four cores solved the problem more quickly than sequential FBS, our results show that EBA clearly outperformed NBA and RBA. RBA performed better than NBA. However, if the number of cores is increased to more than 12, the speedups of both NBA and RBA saturated and resulted in only 1.59 and 2.27 fold with 32 cores, respectively. In contrast, EBA scaled well up to 24 cores. Although the speedup improvement between 24 and 32 cores was small, which might imply the performance saturation with additional cores, EBA achieved a 6.98-fold speedup with 32 cores, indicating the importance of the lock-free approach. With 32 cores, the execution time was reduced from 539 seconds to 77 seconds.

In EBA, since each thread used about 300 MB memory for its own hash table to detect duplicate nodes, it used about a total of 10 GB memory with 32 cores.

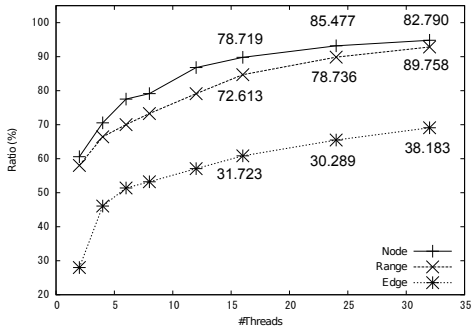


Figure 4: Synchronization overhead on the complete grid graph

In contrast, the other approaches used only 300 MB memory for the hash table shared among cores. Although the additional memory consumption is a drawback of EBA, the size of 10 GB memory fits into the physical memory of recent standard PCs. Additionally, as discussed in Section 3, when parallel algorithms to reduce nodes are developed in the future, this memory overhead might also become a price to pay for NBA and RBA.

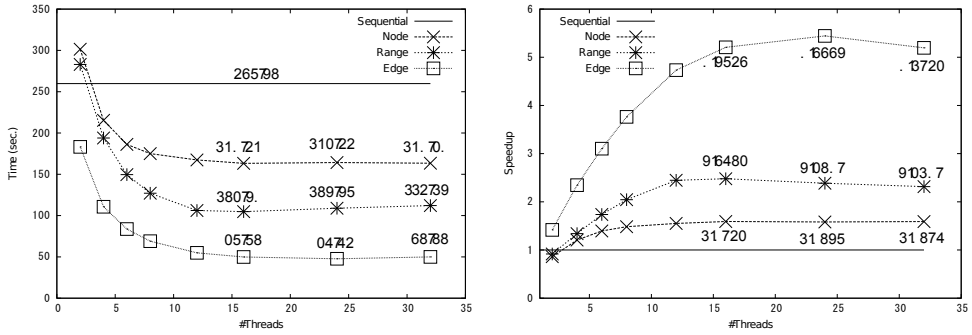
There are several reasons parallel efficiency was degraded in each approach. The *start-up overhead* indicates the overhead incurred to start up a parallel algorithm such as thread creation and allocation of the additional structures used only by the parallel algorithm. The *synchronization overhead* refers to the overhead of idle time starving for work and accessing shared data by using locks. The *termination overhead* is the overhead incurred when the threads are terminated and their additional data structures are de-allocated.

We easily measured the start-up and termination overheads. However, we could not measure the synchronization overhead because lock acquisitions/releases occurred very frequently and the duration regarding each lock operation was very short. We therefore approximated the synchronization overhead as follows:

$$Synchronization\ overhead = \frac{(T_{par} - T_{start} - T_{term}) - \frac{T_{seq}}{k}}{T_{par}} \times 100\ (\%)$$

where T_{par} and T_{seq} are respectively the execution time of parallel and sequential algorithms, T_{start} and T_{term} are respectively the runtimes related to the start-up and termination overheads, and k is the number of cores.

Figure 4 shows the synchronization overhead (SO). As the number of cores increases, SO of all the approaches increased. In particular, although RBA slightly reduced SO, both NBA and RBA had significantly high SO. Even in RBA, 92.8 % of the execution time with 32 cores was related to SO, which was a major reason why it achieved only a 2.27-fold speedup. In contrast, EBA's lock-free approach reduced SO to 69 %, resulting in a better speedup value than NBA and RBA. However, because EBA still suffered from non-negligible SO, reducing this overhead remains a challenge to maximize the efficiency of parallel algorithms.



(a) Execution time (b) Speedup

Figure 5: Performance graphs on 16×16 incomplete grid graphs

Compared to T_{par}, $T_{start} + T_{term}$ for NBA and RBA was very small (the ratio of 0.05 % to T_{par} with 32 cores). However, this number was larger for EBA (the ratio of 8.82 % to 32 cores). This might be related to the fact that EBA had to allocate/de-allocate larger hash tables than NBA and RBA.

Load balance refers to how evenly the work is distributed among the threads and is defined as the ratio of the maximal number of nodes allocated to a core to the average number of nodes allocated to each core. With 32 cores, the load balance of all approaches ranged 1.02–1.05, which was reasonably effective.

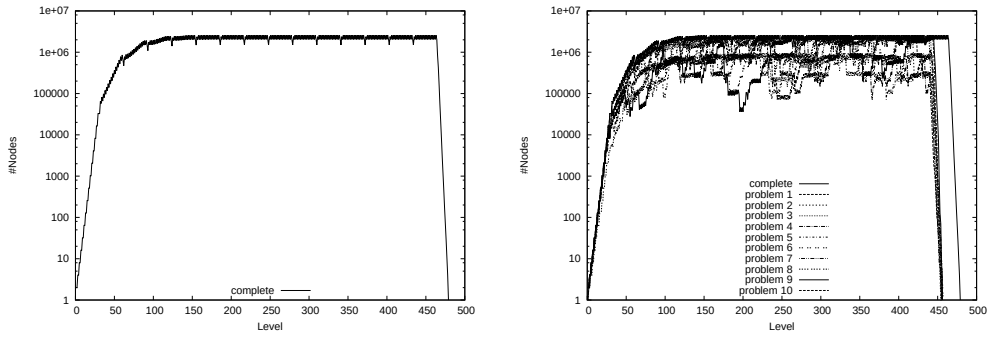
4.3 Experimental Results Using Incomplete Grid Graphs

We prepared ten 16×16 grid graphs with some edges removed as a more practical test suite (called *incomplete grid graphs*) to calculate the average runtime per problem. In constructing each grid graph, we randomly removed 24 edges from the complete grid graph. Each algorithm performed FBS to enumerate all the paths from the top-left corner to the bottom-right corner in the graphs.

Figure 5 shows the execution time and speedups of all the approaches. Again, EBA significantly outperformed NBA and RBA. However, EBA's speedup value was lower than that of the complete grid graph. EBA suffered from performance degradation when the number of cores was increased from 24 to 32. This is not surprising, not only because sequential FBS solved each incomplete grid graph more quickly compared to the case of the complete grid graph, but also because the number of nodes in each level varied significantly in incomplete grid graphs (see Figure 6), which made EBA harder to evenly distribute work.

5 Related Work

No prior work exists on parallelizing Simpath. We therefore review the related literature on parallelizing other algorithms.



(a) Complete grid graph (b) Incomplete grid graphs

Figure 6: The number of nodes generated at each level

Kumar et al. presented two parallel A* search algorithms [9]. In their centralized strategy, the open list, which corresponds to the task queue of N_i in FBS, is shared among processors. In this sense, their approach is similar to NBA. However, while NBA's task queue is FIFO, their open list is a priority queue that incurs a higher operational overhead. Additionally, while their parallel A* must place generated successors back to the shared open list, NBA places them to N_{i+1}, which is different than N_i. Their parallel A* would therefore suffer from severer synchronization overhead than NBA. In contrast, in their distributed strategy, the open list is partitioned over processors as in RBA. While this approach exchanges nodes in the local open lists among processors to achieve load balancing, RBA does so with no communications. The most essential difference between our work and Kumar et al.'s is that their search space is not a DAG but a tree. That is, because of no requirement for detecting duplicates, their closed list can be easily implemented. In contrast, our approaches must consider effective duplication detection techniques using the hash tables.

Despite a number of existing parallel BDD construction algorithms, many of them intend to work well on specific hardware such as vector machines where data structures must be vectorized [11] and distributed-memory machines where identical BDD nodes must be detected efficiently in the presence of much higher communication overhead than shared-memory machines (e.g., [1, 14]). Chen and Banerjee's shared-memory parallel algorithms use depth-first search with very different work distribution schemes that are not easily applicable to FBS [3].

In Ranjan et al.'s breadth-first BDD construction on a PC cluster, each PC manages several levels of BDD nodes to use a vast amount of memory [12], while EBA splits nodes per level. However, their algorithm was not parallelized.

Yang and O'Hallaron's parallel BDD construction algorithm [15] performs node expansions in both breadth-first and depth-first manners. Both EBA and their approach manage nodes in a breadth-first manner but they are often expanded in a depth-first manner. However, while their approach controls the amount of

depth-first node expansions based on the threshold, EBA automatically controls the right balance between depth-first and breadth-first search.

Karp and Zhang presented a random work allocation strategy in which generated successors are sent to randomly selected processors [6, 7]. While this approach achieves load balancing, their search space is not a DAG but a tree. Their algorithms must be combined with an effective duplication detection scheme, which remains a challenge as future work. In contrast, EBA not only balances the workload effectively but also efficiently detects duplicates with no locks.

6 Conclusions and Future Work

We have developed three shared-memory parallel FBS algorithms: NBA, RBA, and EBA. By eliminating the overhead on locks, EBA outperformed RBA and NBA in solving grid graphs which are an abstract domain for real-world geographical information processing and network reliability analysis. As a result, EBA yielded a 7-fold speedup using 32 cores in the best case. Considering the fact that this paper is the first attempt to parallelize FBS, this is an encouraging result. Additionally, once the presented approaches are implemented, the speed of FBS will be automatically improved with advances in multi-core CPUs.

There are a number of possible extensions to our work. First of all, we are currently trying to parallelize the node reduction algorithm to develop a completely parallel Simpath algorithm in shared-memory environments. Next, investigating ideas to improve the performance of presented approaches is of importance. Particularly, reducing the synchronization overhead is a key feature to obtain better speedups. Finally, developing parallel Simpath in distributed-memory environments will be challenging since it is notoriously hard to achieve satisfactory speedups there due to the communication overhead.

References

[1] F. Bianchi, F. Corno, M. Rebaudengo, M. S. Reorda, and R. Ansaloni. Boolean function manipulation on a parallel system using BDDs. In *High-Performance Computing and Networking*, volume 1225 of *Lecture Notes in Computer Science*, pages 916–928, 1997.

[2] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[3] J.-S. Chen and P. Banerjee. Parallel constuction algorithms for BDDs. In *Proceedings of the 1999 IEEE International Symposium on Circuits and Systems (ISCAS '99)*, volume 1, pages 318–322, 1999.

[4] G. Hardy, C. Lucet, and N. Limnios. K-terminal network reliability measures with binary decision diagrams. *IEEE Transactions on Reliability*, 56(3):506–515, September 2007.

[5] T. Inoue, K. Takano, T. Watanabe, J. Kawahara, R. Yoshinaka, A. Kishimoto, K. Tsuda, S. Minato, and Y. Hayashi. Finding all configurations satisfying operational constraints in delivery networks by ZDDs (in Japanese). In *Proceedings of the Institute of Electrical Engineers of Japan National Conference*, 2012.

[6] R. Karp and Y. Zhang. A randomized parallel branch-and-bound procedure. In *Proceedings of the 20th ACM Symposium on Theory of Computing (STOC)*, pages 290–300, 1988.

[7] R. Karp and Y. Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *Journal of the Association for Computing Machinery*, 40(3):765–789, 1993.

[8] D. E. Knuth. *The Art of Computer Programming, Volume 4A, Combinatorial Algorithms, Part 1.* Addison-Wesley Professional, 1st edition, March 2011.

[9] V. Kumar, K. Ramesh, and V. N. Rao. Parallel best-first search of state-space graphs: A summary of results. In *Proceedings of th 10th National Conference Artificial Intelligence (AAAI)*, pages 122–127. Press, 1988.

[10] S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the 30th ACM/IEEE Design Automation Conference*, pages 272–277, 1993.

[11] H. Ochi, K. Yasuoka, and S. Yajima. Breadth-first manipulation of SBDD of boolean functions for vector processing. In *Procedings of the 28th ACM/IEEE Design Automation Conference*, pages 413–416, 1991.

[12] R. K. Ranjan, J. V. Sanghavi, R. K. Brayton, and A. Sangiovanni-Vincentelli. Binary decision diagrams on network of workstations. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD '96)*, pages 358–364, 1996.

[13] K. Sekine, H. Imai, and S. Tani. Computing the Tutte polynomial of a graph of moderate size. In *Proceedings of the 6th International Symposium on Algorithms and Computation (ISAAC)*, pages 224–233, 1995.

[14] T. Stornetta and F. Brewer. Implementation of an efficient parallel bdd package. In *Proceedings of the 33rd annual Design Automation Conference*, DAC '96, pages 641–644, New York, NY, USA, 1996. ACM.

[15] B. Yang and D. R. O'Hallaron. Parallel breadth-first BDD construction. In *In Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 145–156. ACM Press, 1997.

[16] R. Yoshinaka, T. Saitoh, J. Kawahara, K. Tsuruma, H. Iwashita, and S. Minato. Finding all solutions and instances of numberlink and slitherlink by ZDDs. *Algorithms*, 5(2):176–213, 2012.