PARALLEL IMPLEMENTATION OF BREADTH-FIRST SEARCH

Yauhen Klimiankou, Lukas Strebel, Stephanie Christ

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

1. INTRODUCTION

Graph is one of the most powerful and widely used abstract data type, because it is convenient for representation of wide range of real-world objects in computer applications. Most of the graph applications and appropriate algorithms involve graph traversals, during which knowledge's about graph is updated as each node and vertex become visited. Depth-first search (DFS) and Breadth-first search (BFS) are two basic strategies for graph traversal and searching in graph. When the DFS starts graph traversal at the root and explores as far as possible along each branch before backtracking, BFS in contrast inspects all the neighboring nodes starting from the root and then for each of those neighbor nodes in turn, it inspects their neighbor nodes which were unvisited, and so on level by level. The set of nodes with the same distance from the root node is called a level in terms of BFS.

BFS is one of the most basic graph algorithms and a foundation for wide range of more specific algorithms used for such tasks like finding of all nodes within one connected component of the graph, collection copying in garbage collection algorithms, finding of the shortest path between two specified nodes, testing a graph for bipartiteness, mesh numbering, computation of the maximum flow in a flow network etc. BFS is a first class algorithm commonly used for solving of different real-life and scientific problems on the systems represented by networks. Most notable examples of its industrial applications are navigation systems for finding of the shortest path between two specified destination points on the network of roads, finding of the shortest route to the specified host in the computer network or shortest route to the host with specified properties, web indexing performed by web crawlers used by web search engines to maintain they search database in the actual state, social interconnections investigation on the base of social networks.

At this time the information technology industry experiencing a great shift introduced by mass migration to multicore processors and emergence of many-core computer systems (up to 120/240 physical/logical cores in theory and up to 60/120 physical/logical cores in practice on Intel platform, and up to 64 physical cores on AMD platform) and coprocessors (computation accelerators) like Intel Xeon Phi (up to 61 physical cores). All this wide-spreading and emerging computer systems are examples of shared memory architectures (SMA). Wide dissemination of computer systems with shared memory architecture and trends indicating that the development of such systems is the main direction of the further performance improvement of computer systems on the one hand, and the widespread use of BFS for a wide range of applications makes actual the question about the most efficient and scalable variant of parallel version of BFS in the environment of such kind of systems.

In this paper we describe an experimental study of different variants and approaches of parallel BFS based on the OpenMP for the SMA computer systems and design, implementation and evaluation of the variant to which we finally came up as to the most promising. We refer to this variant as optimistic parallel BFS. According to our experimental results it significantly outperforms all other approaches in the environment of AMD SMA platform and Intel Xeon Phi accelerator. Our experiments and analysis of its results highlighted that the efficiency of BFS algorithm variants heavily depends on the properties of underlying environment (hardware platform, operating system, compiler) as well as on the properties of the graph to which it applied. We conclude also that the cost of synchronization which is usually used for preserving consistency can be too high, but in some cases can be eliminated.

It is important to note that the optimistic parallel BFS can't be considered as a universal optimal variant of BFS for the all kinds of the SMA computer systems. Instead it must be considered as a source of more general approach to the implementation of BFS for the environment of particular kind of SMA system. To achieve most optimal approach for particular computer system investigation of different variants of parallel BFS and most suitable optimizations must be done for the environment of system of interest, because our results tell us that the efficiency of BFS itself and optimizations used in it heavily depends on the target environment.

Related work. In this section we will give a brief overview of related work. In the first part we will discuss the work covering the top-down-bottom-up hybrid approach and in the second part an approach that focuses on avoiding atomic operations.

Beamer et al.[**?**] report a different algorithm to deal with the performance issues encountered when designing a BFS algorithm. The proposed hybrid algorithm combines the usual top down approach with a new bottom up part. In the bottom up part a level is processed by searching a parent for all unvisited vertices where a parent is only valid if it is a neighbour of the unvisited vertex. This is advantageous for small-world graphs because it saves accesses and data processing when a large fraction of the vertices are in the frontier. To get optimal results a hybrid algorithm is proposed where a heuristic switching criteria controls the use of top down or bottom up step depending on the size of the frontier and a predicted size of the next frontier. Yasui et al.[**?**] describe an implementation of such a hybrid algorithm for kronecker and R-MAT graphs as well as a detailed description of the heuristic switching parameters.

Berrendorf[**?**] describes a technique to avoid atomic operations in a generalized scenario. The scenario is given as an if-statement followed by some operations that change a state, where multiple threads might execute the predicate and execute the operations afterwards. The operations need to change the state to the same value if executed multiple times otherwise there exists a race condition, i.e. the change of the distance of a visited vertex to the value of the level or the addition of a vertex to the next frontier. The trade-off is that doing a BFS this way can result in additional work, since any unvisited vertex may get added multiple times.

2. BACKGROUND: BREADTH-FIRST SEARCH

In this section we will give a brief overview over the idea of breadth-first search and its sequentiell asymptotic runtime cost and a short discussion of different graphs and graph properties.

Breadth-first search.

Breadth-first search (BFS) is a graph traversal algorithm which starts at a source and either travels until it finds a specific vertice or until it has explored all connected vertices. In the first step all vertices adjacent to the source are explored and stored in some data structure (called frontier or next) as well as marked as visited. In the second step the newly visited vertices become the new sources (called neighbours or current) from which the search continues by repeating this step. Doing a traversal in this way assures that all the nodes at the same distance to the source are explored on the same level before any vertices with greater distance can be explored.

The desired output of a BFS can differ depending on where it is applied. For example with minimal modifications BFS can deliever a predecessor map, where every vertex points to only one parent, or a distance map, where the distance to the source for every vertex is saved. Since the predecessor map is not necessarily unique, we choose to return a distance map as our output to make verification of correctness simpler.

The sequential version of a BFS can be implemented using a single queue and has a theoretical asymptotic runtime of $\mathcal{O}(|V|+|E|)$ where $|V|$ is the number of vertices and $|E|$ is the number of edges of the connected graph.

Graphs.

3. YOUR PROPOSED METHOD

We implemented many different algorithms and multiple variants for most of them. They return a distance map from one source vertex to all reachable vertices in the graph.

All our implementations are based on OpenMP for synchronization.

Topdown. A lot of our approaches are based on a simple topdown algorithm. The idea is to do a level-synchronous traversal of the graph by keeping record of the vertices in the current as well as those in the next level ("frontier" and "neighbour") in two data structures of the same type and setting the frontier to the current neighbours after each level. This leads to an implicit barrier, as the work done by the different threads has to be synchronized. We experimented with different data structures and synchronization methods.

The naive version of the topdown algorighm has a global standard vector for the frontier to allow easy dynamic splitting between the threads in each level. This balances the load between treads, however it has a significant overhead because it relies on a critical section (`OMP critical`) for checking whether the vertices were already visited and inserting them into the neighbour data structure.

To improve the naive implementation, we used atomics, the built-in `__sync_val_compare_and_swap` (CAS) to atomically check whether a vertex was visited and set the correct distance. It also uses a local neighbourhood data structure (standard vector) to prevent needing a critical section. Only at the end of each level, a lock (`omp_lock_t`) is used to combine the local neighbourhoods to a global one, which can then be distributed between the threads for the next level. This is an idea adapted from Berrendorf [**?**].

An extended version of the algorithm before first uses a non-atomic check if a vertex visited before each CAS.

Also inspired by Berrendorf [**?**] is the idea to remove atomics alltogether. This results in a race condition where additional work for the next level is created, as some vertices might be added to the neighbours multiple times. However, expensive atomics are omitted, which results in a tradeoff between the additional time from the added work and

the faster runtime by leaving out the CAS. This algorithm still uses a local neighbourhood and a lock when combining them.

Instead of using a standard vector for the frontier and the neighbour, one of our implementation relies on a bool array for all data structures (visited vertices, frontier, neighbour). This makes it possible to remove all critical sections, as the insertion of an element into a vector was what made them necessary in the first place. Similarly to the algorithm before, there might be additional work due to avoiding atomics. The downside of this implementation is that in each level, you have to loop through all the vertices, not just the current frontier, to be able to explore from there. Depending on the structure of the graph, this can be much more work.

4. EXPERIMENTAL RESULTS

Here you evaluate your work using experiments. You start again with a very short summary of the section. The typical structure follows.

Experimental setup. We run experiments on three different platforms.

The first platform is the EULER cluster which is operated by the HPC Group of ETH. We had access to one node with a 12-core Intel Xeon E5-2697v2 processors (2.7 GHz nominal, 3.0-3.5 GHz peak). It supports hyper-threading, so we ran our algorithms with up to 24 threads. On this platform, we used the gcc compiler with the -O2 flag.

The next platform we ran our algorithms on is the Xeon Phi provided through the class.

Lastly, we also used an AMD FX-8350 (4GHz x8, 8Gb, W2k3).

Results. Next divide the experiments into classes, one paragraph for each. In each class of experiments you typically pursue one questions that then is answered by a suitable plot or plots. For example, first you may want to investigate the performance behavior with changing input size, then how your code compares to external benchmarks.

For some tips on benchmarking including how to create a decent viewgraph see pages 22–27 in

Comments:

- Create very readable, attractive plots (do 1 column, not 2 column plots for this report) with readable font size. However, the font size should also not be too large; typically it is smaller than the text font size. An example is in Fig. 1 (of course you can have a different style).
- Every plot answers a question. You state this question and extract the answer from the plot in its discussion.
- Every plot should be referenced and discussed.

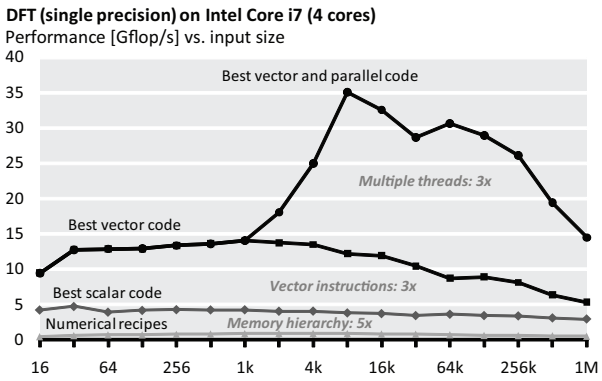

Fig. 1. Performance of four single precision implementations of the discrete Fourier transform. The operations count is roughly the same. The labels in this plot are maybe a little bit too small.

5. CONCLUSIONS

Here you need to summarize what you did and why this is important. *Do not take the abstract* and put it in the past tense. Remember, now the reader has (hopefully) read the report, so it is a very different situation from the abstract. Try to highlight important results and say the things you really want to get across such as high-level statements (e.g., we believe that .... is the right approach to .... Even though we only considered x, the .... technique should be applicable ....) You can also formulate next steps if you want. Be brief. After the conclusions there are only the references.

6. FURTHER COMMENTS

Here we provide some further tips.

Further general guidelines.

- For short papers, to save space, I use paragraph titles instead of subsections, as shown in the introduction.
- It is generally a good idea to break sections into such smaller units for readability and since it helps you to (visually) structure the story.
- The above section titles should be adapted to more precisely reflect what you do.
- Each section should be started with a very short summary of what the reader can expect in this section. Nothing more awkward as when the story starts and one does not know what the direction is or the goal.
- Make sure you define every acronym you use, no matter how convinced you are the reader knows it.

- Always spell-check before you submit (to us in this case).
- Be picky. When writing a paper you should always strive for very high quality. Many people may read it and the quality makes a big difference. In this class, the quality is part of the grade.
- Books helping you to write better:
- Conversion to pdf (latex users only):

  dvips -o conference.ps -t letter -Ppdf -G0 conference.dvi

  and then

  ps2pdf conference.ps

Graphics. For plots that are not images *never* generate the bitmap formats jpeg, gif, bmp, tif. Use eps, which means encapsulate postscript. It is scalable since it is a vector graphic description of your graph. E.g., from Matlab, you can export to eps.

The format pdf is also fine for plots (you need pdflatex then), but only if the plot was never before in the format jpeg, gif, bmp, tif.