Parallel Subgraph Isomorphism

Aaron Blankstein
E-mail: blanks@mit.edu

Matthew Goldstein
E-mail: austein@mit.edu

***Abstract*—The subgraph isomorphism problem deals with determining whether a given graph H is isomorphic to some subgraph of another graph G. In this paper we attempt to parallelize a fast serial subgraph isomorphism library, VFLib, which uses backtracking search to find a solution. Our parallel solution runs on `Cilk++` for efficient execution on multicore machines. In our work we examine the benefits and drawbacks of several data structures when used during this backtracking search. We also examine several heuristics for spawning threads. Finally, we use conditional copying to achieve near-linear speedup with the number of CPU cores on random graphs and reasonable performance on parasitic inputs.**

I. Introduction

The subgraph isomorphism problem asks the following question. *Given two graphs G, H is H isomorphic to any subgraph of G?* Beyond just structural isomorphism, the vertices of the graphs may have additional attributes such as labels which add an additional requirement to the isomorphism. These labels can be repeated throughout the target subgraph and are often repeated many times, leading to partial subgraph matches. This problem is provably NP-complete [1].

The fastest general algorithm for finding subgraph isomorphisms is a matching algorithm developed by Ullmann [2]. Ullmann's algorithm returns a *match-set*, which is the set of vertex pairs in the subgraph isomorphism. The algorithm finds the match set by translating the problem into a search. Potential vertex pairs are incrementally added to a match set until either a subgraph isomorphism is discovered or no more potential pairs exist. Because of the potential for failed paths in the search tree, this search requires use of backtracking.

Because Ullmann's algorithm uses backtracking search, these partial matches can cause searching to be computationally expensive. This also leaves many opportunities for parallelism, as any branch is the search could be handled as a new thread of computation.

The subgraph isomorphism problem has a number of applications. In chemical engineering, it is used to find particular chemical structures in large molecules. Subgraph isomorphism has also been used in automated circuit layout and design algorithms.

In this paper, we perform a simple parallelization of a subgraph isomorphism library, VFLib, which has near-linear speedup when working with random graphs. We then move to 2d meshes and parasitic hard cases where the parallelism is reduced. Several methods to improve parallelism are examined, from both data structure and thread spawning heuristic standpoints. One method, Conditional Copying, is found to have near-linear speedup even in the parasitic bad cases.

In the remainder of this section, we detail the VFLib library and describe the Cilk++ language. In Section 2, we present initial parallelization results using a naive approach and the problems this approach created. In Section 3, we outline our attempts at fixing some of these problems using alternative data structures, spawning heuristics, and conditional copying. In that section we also present the results of those attempts. In Section 4, we speculate on future work and in Section 5, we review our contributions.

A. VFLib

We began our work with an open source subgraph isomorphism library, VFLib. VFLib uses an optimized serial version of Ullmann's algorithm. The algorithm proceeds by creating and modifying a match state. The match state contains a *matched-set*, which is a set of vertex pairs that match between the two graphs. If the *matched-set* contains all of the query graph, H, then the algorithm is successful and returns. Otherwise, the algorithm attempts to add a new pair. It does this by tracking the *in-set* and *out-set* of each graph, which are the sets of vertices immediately adjacent to the *matched-set*. These two sets define the potential vertices that can be added to a given state. The only pairs that can be added are either in the *in-set* of both graphs or the *out-set* of both graphs. The algorithm uses backtracking search to find either a successful match state, or return a failure.

B. Cilk++ Language

To efficiently execute our application on multicore machines, we are using the Cilk++ language. In this language, applications run in the Cilk++ runtime, which manages parallel execution using computation workers. These workers run on separate Operating System threads and there is one worker per CPU core.

The Cilk++ language is C++ with some additional language features. The most important feature for our work is the `cilk_spawn` keyword. This keyword marks a function call as a candidate for parallel work. When the application is executed, the Cilk++ runtime decides whether or not the function call is executed on a different computational worker. This way, a developer can add many `cilk_spawn` calls to their code and if the algorithm is sufficiently parallel, the Cilk++ runtime will prevent too many OS threads from spawning.

When dealing with parallel code, two measurements are extremely important. The first is total work performed by the application. The second is the amount of work performed

```
function match(state0){                                    1
  for (p in state->nextPair()){
    if (spawningAtThisLevel){                              3
      nextState = cleanCopy->deepClone()
      nextState -> addPair()                               5
    }else{
      nextState = state                                    7
      nextState -> addPair()
    }                                                      9
    cilk_spawn match(nextState)
  }                                                        11
}
```

Fig. 1: Pseudocode for the match function with `cilk_spawn`.

along the shortest parallel execution path. This latter measure, the span, is the bottleneck on parallel speedup. The theoretical maximum speedup is the total work divided by the span. For our project, we collected measurements of work and span using the cilkview program, which calculates the theoretical and burdened parallelisms by counting instructions executed.

See work by Leiserson [4] for a more complete discussion of Cilk++.

II. Simple Parallel Solution

VFLib's subgraph isomorphism algorithm does a backtracking search, so the simplest way to parallelize the library is to allow an execution in parallel with each recursive call of the search. In order to allow each worker to operate without conflict in parallel, we clone the match state (*matched-set*, *in-set*s, and *out-set*s) at each level. We then use `cilk_spawn` on every recursive call in the search allowing the called function to execute in parallel on another worker. Figure 1 contains pseudocode for this approach.

This strategy has a number of drawbacks. For inputs graphs with 1,000 vertices, the match state contains 5,000 integers. On random graphs of that size, the clone operation is performed 10,000 times and it significantly slows down the performance of the library. This simple parallel solution runs two to three orders of magnitudes slower than the serial version on the problems tested.

Our first solution, what we will call our naive solution, used coarsening to solve this problem. We called `cilk_spawn` only at the 3-4 highest levels of the search, stopping spawning at a point when many parallel executions had been spawned. On random input graphs, this results in abundant parallelism and linear speedup over the serial algorithm with the number of CPU cores.

A. Random Graph Results

Our intial implementation achieved near-linear speedup with the number of processors on random graphs.

Results for the naive solution were collected from the `Cilkview` application. `Cilkview` measures the total work performed and the span of a cilk application by counting instructions executed. This way, the parallel speedup can be measured accurately based on the actual work done. Otherwise, inflated speedup numbers can be achieved for inputs where a parallel algorithm will find a solution quickly by luck.

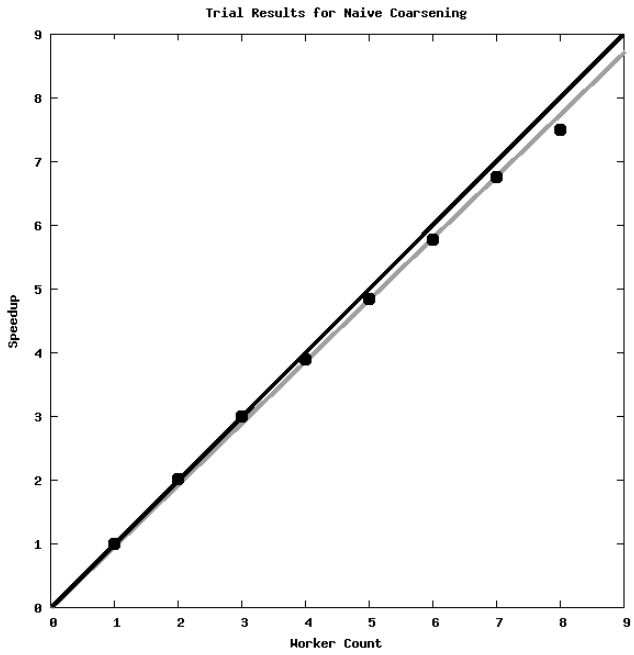


Fig. 2: Cilkview output for our naive parallelization implementation working on random graphs. This graph plots the theoretical speedup in as a black line, the burdened parallelism as a gray line, and the actual speedup as black points.

Our results are shown in Figure 2. Even with the scheduling overhead of running multiple workers, the burden on parallelism was only 312.66/314.71. We consider these results to be a success for the general problem of subgraph isomorphism.

B. Parasitic Cases and 2D Meshes

Unfortunately, it is not hard to imagine parasitic inputs for our naive solution. For example, consider if the input graphs begin extremely linearly and then fan out after 5-6 levels. Our naive solution will have no spawns to perform in the first few levels, where our spawns usually occur. Then, in the part of the graph that contains nearly all of the work, our algorithm will have only one running worker and spawn no others, giving no parallelism over the entire graph.

Indeed, in our benchmark suite, we found a series of graphs for which the naive approach had a parallelism of about 1. These graphs are two dimensional meshes. The problem with these meshes is that while the branching factor is high, many edges introduce vertices with only one new neighbor. This creates a situation where the search has very low potential for parallelism. Because our naive strategy relies on abundant parallelism, especially early in the inputs, we failed to achieve any speedup. In the next section, we will present our attempts to remedy this problem.

III. Alternative Data Structures

Our first attempt at solving the 2D Mesh problem is to use `cilk_spawn` at every recursive call. This approach however is extremely slow. This is not because of scheduling overhead but because cloning the match state at every level takes a long time. While other backtracking search problems avoid this overhead by keeping small data structures, the nature of subgraph isomorphism is that a potentially large subgraph must be matched, so data on the current matching needs to be kept. Further, the heuristics used by Ullmann's algorithm and VFLib also require large datastructures. We hoped that by finding more efficient ways to store this data, it may be possible to make a copy and spawn at every recursive call, while only slightly slowing down other operations.

In the original implementation, the one that our naive solution is based on, all of the data sets are stored using integer arrays. The arrays are indexed by graph vertex identifier and contain a zero when that particular vertex is not in the set. Because the array stores integers, the original implementation stores numbers in the array which can be used for logging information required by the backtracking operation.

Our first alternative data structure used map sets to store data rather than plain arrays. We measured the expected size of these sets during a typical search and found that as little as $1 - 2\%$ of the graph is being considered at a time, yet the arrays allocate space for every element. While an array should provide faster access than a mapset, we hoped the saved cost from doing a deep copy would offset that slowdown.

Our second alternative used bitsets rather than arrays to store the matching. While an integer array allows a log of Match additions to be tracked, it also takes more memory and may take longer to access when the log functionality is not needed. Because this logging information needed to be tracked somehow, we required additional memory structures for this information.

A. Results

To test our data structures, we measured the runtime of the various operations and counted their usage. To do this, we ran our implementation in serial on a random graph input with 1000 vertexes. Additionally, we counted a clone operation at every recursive call, rather than using our coarsening technique.

Our results from data structure testing are summarized in Figure 3. This figure shows the total runtime per required clone operation. In general, `getNextPair` is called 20 times for every clone operation and the backtrack operation occurs 0.6 times as often as clone. This makes `getNextPair` an extremely significant operation. However, if improvements to the clone operation do not affect `getNextPair` too much, we can experience speedups in parallel execution.

Switching the data structures to Mapsets did not have the desired effect. Searching for next pairs in a mapset was about 10 times slower than the original implementation. Similarly, backtracking took much longer. Surprisingly, the actual time spent on copying the data structure also took longer despite being a supposedly smaller structure. We suspect this is because cloning a mapset requires multiple link dereferences while an array clone can use `memcpy`. These results prevent us from using mapsets in any way.

Using Bitsets also failed to show much overall improvement. Both generating next pairs and identifying backtracking steps took more than twice as long with this data structure as with plain arrays. The actual memory copying process had a small speed-up when using Bitsets, but this did not offset the other increased costs.

IV. Spawning Heuristics

As attempting to decrease the memory overhead such that it was practical to perform a deep copy at every recursive call failed, our alternative was to develop a more intelligent heuristic for performing `cilk_spawn`s. The naive implementation does spawning only in the first few levels of the graph. This leaves potentially lost parallelization when several of the original branches die out and there is only one running worker and there are potential branches where that worker is running but their depth is too deep for spawning. So, we consider three different heuristics that allow spawning at these lower levels.

The first method is simply to use a non-linear cutoff for the spawn depth. To do this, we tried a logarithmic heuristic such that some later depths would use spawns again. This heuristic spawned only if there was no remainder after dividing the logarithm of the depth by some value.

A second, similar, approach simply repeated the series of levels of spawns again at deeper depths after the initial cutoff, allowing spawns to happen again should there be the opportunity.

These methods, while allowing for deeper spawning, do not take into account for the number of possible recursions from a

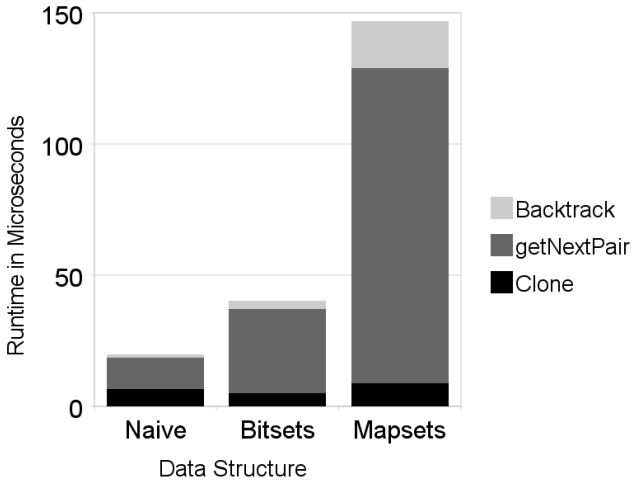


Fig. 3: This figure shows the total runtime per clone operation in μs for the three most common operations. This data was collected using the serial version of the algorithm on a 1000 vertex random graph input.

given state. In cases where there are not many possible recursive calls, there should be no reason to copy the entire match state and trigger a spawn. Instead, the same worker could continue its work in serial. Furthermore, in cases where there are many potential branches from a given state, attempting to make clones and spawn for all recursive calls is likely not the best use of resources. This is because the number of workers limits the amount of data that can be worked on in parallel.

Based on these insights, we attempted to apply one more spawning heuresting. In this heuristic, our implementation copies and spawns only for the first branch from any given state. Doing so eliminates the possible overuse of memory when there are many children.

A. Results

The non-linear cutoff heuristic did not work well. In some cases, it attempted spawns deep in the search where there were not multiple neighbors for spawning to create parallelism from. In other cases, it created a large memory overhead when copying the data structures many times at levels with too many vertexes.

Our second heuristic which spawned in the middle of the tree yielded similar results to the first non-linear spawning heuristic. Again, we achieved no greater speedup.

To see why we failed to achieve this speedup, we added structures to the search process which showed us how many nodes were added at each level in the search. We were examining these numbers to see whether the graph search ever fanned out significantly after a fan in. Even in the examples that most displayed this behavior, once the search has passed the first significant fan in, the number of vertexes added to the search after that point were an order of magnitude less than the number of vertexes that came before. This suggests that there is very little parallelism to be gained by spawning heuristics that spawn at those low levels.

Our third heuristic, of spawning only the first child, seems like a better approach, but also suffers some drawbacks. In a random graph, with the amount of branching varying by level, a spawn at the level with the greatest amount of branching will only happen once. This relies on other branching opportunities at lower levels to create sustained parallelism.

For 2D meshes in particular, this style of spawning is unlikely to be particularly beneficial. Any newly spawned branch will only be effective if it continues for several levels and does not just end. Particularly on the outside of a 2D mesh, recursive calls are not likely to have many new children. Because of this, we expected this heuristic may also fail to achieve much speedup. Testing confirmed this, with parallelism numbers near 1.

For our tested graphs, searches were fairly shallow – going only about 13 levels. This shallowness means that any good spawning heuristic would need to make some informed decisions about the structure of the graph and its branching properties. These decisions could conceivably make use of max-flow analysis, but this would likely take too much time to compute. A nearly optimal solution could be written that is aware of how many workers are currently active, spawning only when workers are waiting. Our last solution, conditional copying, is in a similar vein.

```
function match(state0, modifyingParentState,
needsToCleanParent){                                    2
  bool isModifyingState0 = false
  snapshot = state->deepClone()                         4
  for (p in state->nextPair()){
    needsToCleanUp = ! isModifyingState0                6
    if (isModifyingState0){
      nextState = cleanCopy->deepClone()                8
      nextState -> addPair()
    }else{                                              10
      isModifyingState0 = true
      nextState = state                                 12
      nextState -> addPair()
    }                                                   14
    cilk_spawn match(nextState, &isModifyingState0,
    needsToCleanUp)                                     16
  }
                                                        18
  if (needsToCleanParent){
    nextState -> backTrack()                            20
    *modifyingParentState = false
  }                                                     22
}
```

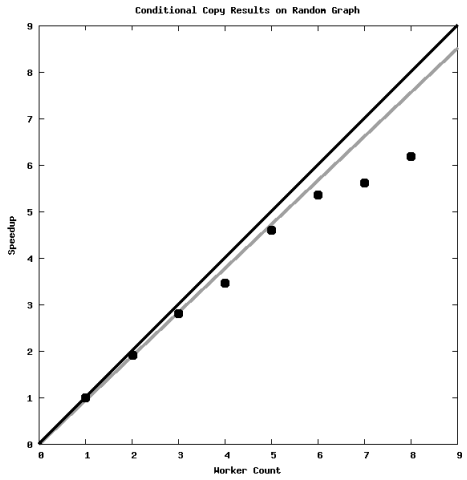
Fig. 4: Pseudocode for the match function with conditional copying.

V. Conditional Copy

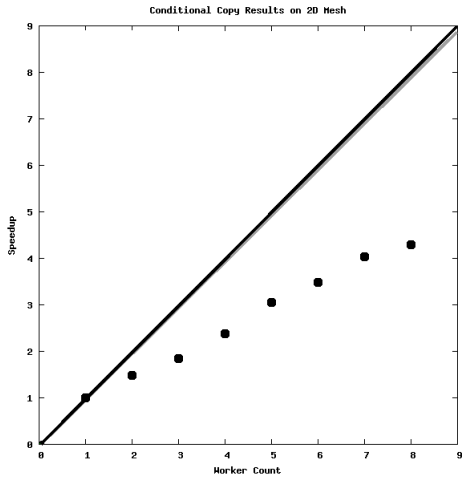
Our attempts to find good spawn heuristics or alternative data structures failed, so we attempted to enable spawns at every level while only performing clones when absolutely required. Unfortunately, `Cilk++` does not provide a mechanism for determining if there are idle workers, so we were left to devise a method of our own. Our goal was to copy our data structure only when a spawn actually caused work to move to another computation worker (this event is called a steal in `Cilk++`.)

To do this, we clone once to create a snapshot at every step along the backtracking search except for the dead ends. Pseudocode for the conditional copying match function is provided in Figure 4. If the search continues in a different worker, we create a clone from the initial snapshot. Otherwise, we just continue modifying the state as in the purely serial version. To detect when recursive calls begin executing in a different worker, we use a boolean flag. This flag is true while any worker is modifying the input state. Modification is finished only when the nested function call returns. Because this call is spawned, only the nested function call knows when it is returning. Therefore, we pass this flag by reference to the nested call so that it can unset the flag when it has finished modifying the state.

Though this version incurs clone costs at every step except for dead ends, it still performs well and successfully parallelizes the 2D Mesh cases. The results from this version are presented in the next section.



(a)



(b)

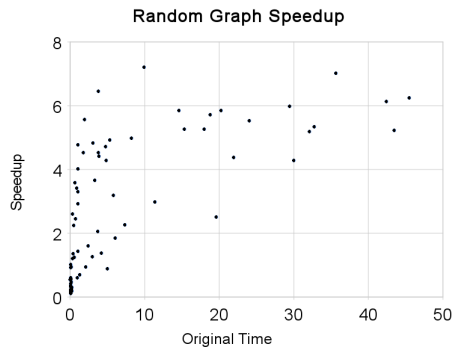
Fig. 5: Cilkview results for conditional copy implementation on (a) random graph input and (b) 2D mesh graph input

A. Results

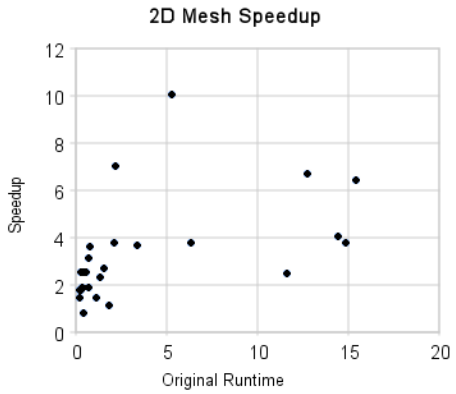
Our primary concern with this implementation was that we still achieved good performance on random graph inputs. Results obtained through cilkview are shown in Figure 5a. This implementation has higher burden than the naive implementation and therefore doesd not achieve as much speedup. However, the algorithm continues to obtain good speedup as the number of CPU cores increases.

Because our goal was to achieve some parallelism on 2D Meshes as well, we also wanted good cilkview results from those inputs. These results are shown in Figure 5b. As you can see, there is less parallelism than in the random case. However, the implementation does achieve some amount of speedup.

We then compared the runtime of these implementations with the original library's runtime on two sets of inputs. The first set asks for matches between two random graph inputs with 1000 and 600 vertices. The second set asks for matches between two adjusted 2D Mesh graphs with 1000 and 600



(a)



(b)

Fig. 6: Cilkview results for conditional copy implementation on (a) random graph input and (b) 2D mesh graph input

vertices. We plots these results in Figure 6. This figure shows the speedup on 8 CPU cores against the original runtime of the input problem. Since we expect faster inputs to experience less speedup, we focus our attention on the measured speedup for problems which originally took longer than about 2 seconds. With these results we found that random graph speedup is about 6x faster than the serial version. This is to be expected as the high burden prevents us from achieving 8x speedup on 8 cores.

VI. Future Work and Possible Language Features

The first line of additional work for this project involves removing the need for a snapshot copy in our conditional copy implementation. We could do this by keeping one clean copy at the initial state and then maintaining a log of operations as we perform our search. Then, whenever a clone is required, we would replay the log on a clone of the initial copy. This would increase the cost of a clone, but remove the need for snapshotting at each level.

Our work also suggests some language features for `Cilk++`. It would be extremely useful to have a hyperobject that would abstract the conditional copying code. This could be a splitter object. This object would "magically" determine when it was appropriate to clone the underlying data structure

and abstract away the actual operation. This way, the `Cilk++` language could handle the process of snapshotting and logging.

VII. Contributions

Our project has contributed the following:

1) We compared performance of several data structures for subgraph isomorphism
2) We implemented a faster than current state-of-the-art subgraph isomorphism match algorithm.
3) We detailed a general approach to dealing with large data structure copying for spawns in cilk platform
4) We speculated on useful language features to enable conditional copying in cilk

Acknowledgment

The authors would like to thank Charles E. Leiserson, Bradley C. Kuszmaul, and Jim Sukha for their guidance throughout our project and the 6.884 class.

References

[1] S. A. Cook, *The complexity of theorem-proving procedures*, in STOC '71: Proceedings of the third annual ACM symposium on Theory of computing. New York, NY, USA: ACM Press, 1971, pp. 151-158.

[2] J. R. Ullmann, *An algorithm for subgraph isomorphism*, J. ACM, vol. 23, no. 1, pp. 31-42, January 1976.

[3] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, *An improved algorithm for matching large graphs*, in In: 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, Cuen, 2001, pp. 149-159.

[4] C. E. Leiserson *The Cilk++ concurrency platform*, in DAC '09: Proceedings of the 46th Annual Design Automation Conference. New York, NY, USA: ACM, 2009, pp. 522-527.