Semestral project NI-PDP 2021/2022

Parallel algorithm for a maximum bipartite subgraph problem

Anton Bushuiev

Master's program, FIT CTU, Thákurova 9, 160 00 Praha 6

May 5, 2022

1 Problem definition and sequential algorithm

Given an undiricted connected graph $G = (V, E)$ with non-negative edge weights $w : E \to \mathbb{N}^+$, the problem of a *maximum bipartite subgraph* is to find each connected bipartite subgraph $G^* = (V^*, E^*)$ with the maximum weight among all connected bipartite subgraphs, i.e.

$$E^* \in \arg\max_{B \subseteq E} \sum_{e \in B} w(e). \tag{1}$$

The considered problem is NP-hard [1] and the *sequential solver* implemented in this project is a branch and bound algorithm. It is based on the fact that a graph is bipartite if and only if it can be colored with two colors. Thus, for a candidate solution B the algorithm recursively tries all the possible options for each edge $u, v \in E$: (i) add edge to B and color u, v with colors $0, 1$ respectively, (ii) add edge to B and color u, v with colors $1, 0$ respectively and (iii) ignore edge. While building B, the coloring consistency is checked and when all the edges are processed, the candidate is ready. This exhaustive search leads to $\mathcal{O}(3^{|E|})$ time complexity. To mitigate it, the implemented solver contains a simple pruning based on the sum of the weights of all yet unprocessed edges. Having such a pruning, it's rational to assume that sorting edges by their weights in descending order may reduce the algorithm's running time. Therefore, the implementation also provides this as an option. For details of the algorithm please see Algorithm 1. The solver is implemented in C++ and can be found in `solvers/sequential.cpp`.

2 Task-parallel algorithm and its implementation in OpenMP

Recursive steps of the sequential algorithm generate a ternary tree. Since the branches of a tree generated by recursion are independent, the described sequential algorithm can be easily parallelized. Thus, the idea behind the *task-parallel* algorithm is that each recursive step can be treated as a separate independent task. Each thread takes one job from the common pool, performs it, and inserts up to three new ones. This process repeats for all threads until no more jobs are left and the initial computation is started by a single one. During a computation, when only a small number of edges (equal to the `TASK_THRESHOLD` constant) are left unprocessed, a thread finishes them sequentially to avoid unnecessary granulation. The details of the algorithm can be found in Algorithm 2.

Using OpenMP, the implementation of the task-parallel algorithm is simple. The initial computation is started using the directives `omp parallel` and `omp single`. Jobs are added to the task pool with `omp task` and the atomicity of solution update is ensured with `omp critical`. In the experiments described below, the threshold number of edges `TASK_THRESHOLD` is set to 4. `solvers/task-parallel.cpp` source file contains an implementation of this algorithm.

3 Data-parallel algorithm and its implementation in OpenMP

The other approach to parallelizing the sequential algorithm is to directly divide the initial tree into disjoint subtrees and process them one by one by multiple threads. The roots of subtrees can be obtained by running the breadth-first search (BFS) algorithm for some iterations and taking the content of an underlying BFS-queue. The desired number of subtrees is set as a simple function of input size: the number of edges multiplied by the constant `N_JOBS_PER_THREAD`. Note, however, that stopping the BFS when queue size is equal to the desired number may not be possible since each search iteration adds 1 to 3 new nodes. That is why the present implementation generates at most the desired number of subtrees. Algorithm 3 provides a pseudo-code for the data-parallel solver.

The OpenMP-based implementation of the data-parallel algorithm is straightforward. It requires only two modifications of the sequential code. Firstly, the algorithm generates subtrees, and then they are processed in a for-loop with the directive `omp parallel for schedule(dynamic)`. Dynamic scheduling means that threads process subtrees one by one taking a new job only after finishing a previous one. For the experiments, `N_JOBS_PER_THREAD` is set to 100. The `solvers/data-parallel.cpp` source file contains a complete implementation.

4 Distributed algorithm and its implementation in Open MPI

The *distributed* algorithm adapts the sequential solver to run across multiple processes and threads. It is based on a master-slave approach: firstly, the master process generates jobs with the BFS algorithm, in the same way as discussed in Section 3. Then, it sends them one by one to free slave processes and waits for responses. In its turn, a slave process solves an assigned job with the data-parallel algorithm, sends the solution, and waits for another one until the terminal signal is not received. Again, the number of jobs constructed by the BFS depends on the constant `N_JOBS_PER_PROCESS`. See Algorithm 4 for the details.

The implementation of this algorithm can be found in `solvers/distributed.cpp`. It utilizes an Open MPI framework for communication between processes. Serialized messages are sent and received via blocking functions `MPI_Send` and `MPI_Recv` through the buffer of constant size. The serialization is implemented straightforwardly using a `vector<int>` container. For all the experiments, `N_JOBS_PER_PROCESS` for a master process and `N_JOBS_PER_THREADS` for slave processes are both set to 10.

5 Experiments

Framework

All the following experiments were conducted on STAR cluster (`star.fit.cvut.cz`) on NUMA nodes with 2 CPUs and 10 cores each with no hyperthreading. Therefore, the implementations of task-parallel and data-parallel algorithms were tested for 2, 4, 8, 16, and 20 threads. The implementation of a distributed MPI program was evaluated on 2, 3, and 4 nodes with 10 and 20 threads each.

The Bash script `submit-job.sh` serves for automated testing. It generates a Sun Grid Engine (SGE) script and submits it to the cluster's queue. The script has parameters to configure a required number of processes and threads, and a type of the solver and its parameters. Executions of the script generate output files with measured jobs' running times which can be then composed into a single `.csv` file using the `compose-results.sh` script.

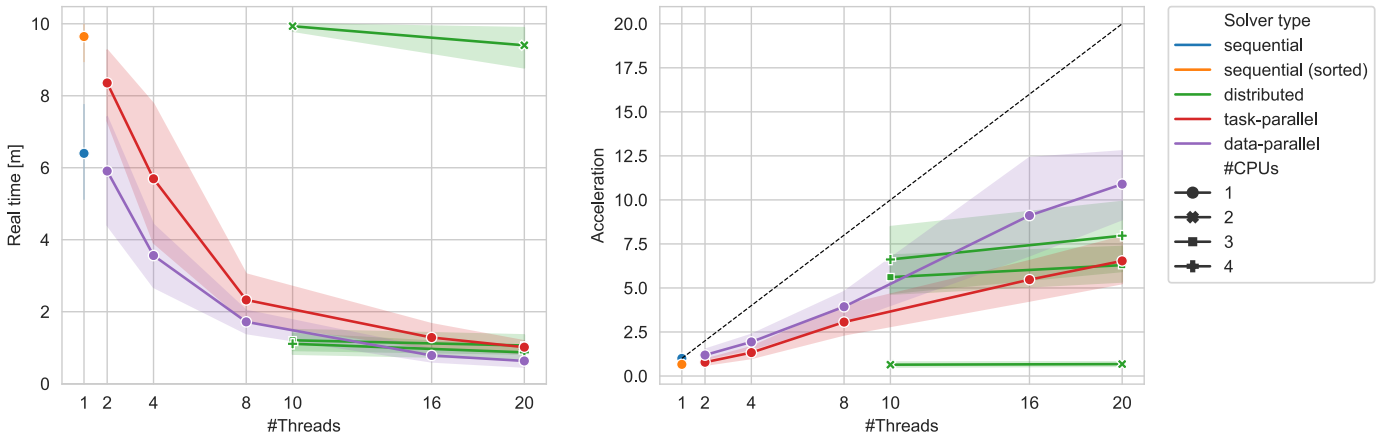
For the final test, 8 input graphs were generated such that the sequential algorithm solves them in 4 to 10 minutes. Namely, these are graphs of 15 to 20 nodes with either an average degree of 5 to 7 or an equal degree in the same range for each node. For the details please see `data/graphs_final`.

Results

The experimental results are visualized in Figure 1. First of all, quite surprisingly, Figure 1a demonstrates that sorting edges by their weights, as discussed in Section 1, has a negative influence on running time. Then, we can observe that the task-parallel and data-parallel solvers show an expected performance: as the number of allocated threads increases, the running time decreases. Figure 1b suggests that both algorithms have near-linear acceleration and the data-parallel approach consistently outperforms the task-parallel one. Moreover, the plot suggests the average lower bound on running time – about 1 minute.

Regarding the MPI solution, the communication overhead is quite significant. Indeed, on two cores the distributed solver just "simulates" the data-parallel approach. However, the running time of such a simulation is the highest among all the solvers. On the other hand, the solver's performance on more than two CPUs is impressive: on three cores it already hits the estimated lower bound, and allocation of four processes leads only to a slight improvement.

Note that tuning of the constants was taken out of the project's scope despite a proper setup of their values may be crucial. Moreover, for the data-parallel and distributed algorithms, the number of subtrees is determined by a simple function dependent only on the number of available threads. The incorporation of input size may lead to better performance. Besides that, the distributed algorithm's implementation could be improved by more advanced serialization and synchronization of the best-found solutions for more efficient pruning.



(a) Solvers' running time.

(b) Solvers' acceleration

Figure 1: Experimental results for different types of solvers. The values are averages over 8 generated inputs, and margins depict standard deviation.

6 Conclusion

In this project, I have realized the solver for a *maximum bipartite subgraph* problem in C++. Then, I implemented its three parallelizable modifications with OpenMP and Open MPI frameworks. Running on multiple threads and CPUs, they confidently and consistently outperform the original solver on randomly generated data.

References

[1] Gutin, Gregory and Yeo, Anders, 2021, Lower Bounds for Maximum Weighted Cut, 10.48550/ARXIV.2104.05536.

Algorithm 1 Sequential solver

```
SOLUTION ← ∅

function BB-DFS-Sequential(edge, state)
    score ← sum of G edge weights by state.edge_mask

    // Analyze state if terminal
    if edge ≥ |E(G)| then
        if G.not_connected() then
            return
        end if
        if SOLUTION.score ≤ score then
            update SOLUTION with state
        end if
    end if

    // Prune
    score_future ← sum of all G weights for edges > edge
    if score + score_future < SOLUTION.score then
        return
    end if

    // "Ignore edge" case
    state.edge_mask[edge] ← 0
    BB-DFS-Sequential(edge+1, state)

    // "Add edge" cases
    state.edge_mask[edge] ← 1
    u, v ← G.edges[edge]
    for colors in {(0,1),(1,0)} do
        if state.colors[u] ≠ colors[0]
        or state.colors[v] ≠ colors[1] then
            continue
        end if
        state.colors[u] ← colors[0]
        state.colors[v] ← colors[1]
        if coloring in state is consistent then
            BB-DFS-Sequential(edge+1, state)
        end if
    end for
end function

function Main(infile, sort)
    G ← parse infile
    if G is bipartite then
        SOLUTION ← {G}
        print SOLUTION and exit
    end if
    if sort is True then
        Sort edges of G by weights
    end if
    BB-DFS-Sequential(0, ∅)
    print SOLUTION and exit
end function
```

Algorithm 2 Task-parallel solver

```
SOLUTION ← ∅

function BB-DFS-Task(edge, state)
    score ← sum of G edge weights by state.edge_mask

    // Analyze state if terminal
    if edge ≥ |E(G)| then
        if G.not_connected() then
            return
        end if
        Critical section {
        if SOLUTION.score ≤ score then
            update SOLUTION with state
        end if
        }
    end if

    // Prune
    score_future ← sum of all G weights for edges > edge
    if score + score_future < SOLUTION.score then
        return
    end if

    // "Ignore edge" case
    state.edge_mask[edge] ← 0
    if coloring in state is consistent then
        if |E(G)| − 1 − edge > TASK_THRESHOLD then
            Add task "BB-DFS-Task(edge+1,state)"
        else
            BB-DFS-Task(edge+1, state)
        end if
    end if

    // "Add edge" cases
    state.edge_mask[edge] ← 1
    u, v ← G.edges[edge]
    for colors in {(0,1),(1,0)} do
        if state.colors[u] ≠ colors[0]
        or state.colors[v] ≠ colors[1] then
            continue
        end if
        state.colors[u] ← colors[0]
        state.colors[v] ← colors[1]
        if coloring in state is consistent then
            if |E(G)| − 1 − edge > TASK_THRESHOLD then
                Add task "BB-DFS-Task(edge+1,state)"
            else
                BB-DFS-Task(edge+1, state)
            end if
        end if
    end for
end function

function Main(infile, sort)
    G ← parse infile
    if G is bipartite then
        SOLUTION ← {G}
        print SOLUTION and exit
    end if
    if sort is True then
        Sort edges of G by weights
    end if
    Run by single thread {
        BB-DFS-Task(0, ∅)
    }
    print SOLUTION and exit
end function
```

Algorithm 3 Data-parallel solver

```
SOLUTION ← ∅

function BB-DFS-DATA(edge, state)
    score ← sum of G edge weights by state.edge_mask

    // Analyze state if terminal
    if edge ≥ |E(G)| then
        if G.not_connected() then
            return
        end if
        Critical section {
        if SOLUTION.score ≤ score then
            update SOLUTION with state
        end if
        }
    end if

    // Prune
    score_future ← sum of all G weights for edges > edge
    if score + score_future < SOLUTION.score then
        return
    end if

    // "Ignore edge" case
    state.edge_mask[edge] ← 0
    BB-DFS-DATA(edge+1, state)

    // "Add edge" cases
    state.edge_mask[edge] ← 1
    u, v ← G.edges[edge]
    for colors in {(0, 1), (1, 0)} do
        if state.colors[u] ≠ colors[0]
        or state.colors[v] ≠ colors[1] then
            continue
        end if
        state.colors[u] ← colors[0]
        state.colors[v] ← colors[1]
        if coloring in state is consistent then
            BB-DFS-DATA(edge+1, state)
        end if
    end for
end function

function MAIN(infile, sort)
    G ← parse infile
    if G is bipartite then
        SOLUTION ← {G}
        print SOLUTION and exit
    end if
    if sort is True then
        Sort edges of G by weights
    end if
    n_jobs ← N_THREADS * N_JOBS_PER_THREAD
    jobs ← generate n_jobs jobs with BFS
    Run for in parallel with dynamic scheduler {
    for job in jobs do
            BB-DFS-DATA(job.edge, job.state)
    end for
    }
    print SOLUTION and exit
end function
```

Algorithm 4 Distributed solver

```
SOLUTION ← ∅

function MAIN(infile, sort)
    G ← parse infile
    if Master process and G is bipartite then
        SOLUTION ← {G}
        Send terminate signal to all other processes
        print SOLUTION and exit
    end if
    if sort is True then
        Sort edges of G by weights
    end if

    if Master process then
        n_jobs ← N_PROCESSES * N_JOBS_PER_PROCESS
        jobs ← generate n_jobs jobs with BFS
        n_workers ← N_PROCESSES
        Send one job to all other processes
        while n_workers > 0 do
            solution ← receive from some other proc
            if jobs is not empty then
                Send next job to proc
            else
                Send terminate signal to proc
                n_workers − −
            end if
            SOLUTION.update(solution)
        end while
        print SOLUTION and exit

    else
        while True do
            msg ← receive from master process
            if msg is job then
                Solve job with data-parallel algorithm
                Send SOLUTION to master process
            else
                Break
            end if
        end while
    end if
end function
```