Department of Computer and Electrical Engineering and Applied Maths

UNIVERSITÀ DEGLI STUDI DI SALERNO

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Implementation of a matching algorithm on large graphs in multicore and GPU architectures

Supervisor:

Ch.mo Prof. Pierluigi Ritrovato

Co-Supervisor:

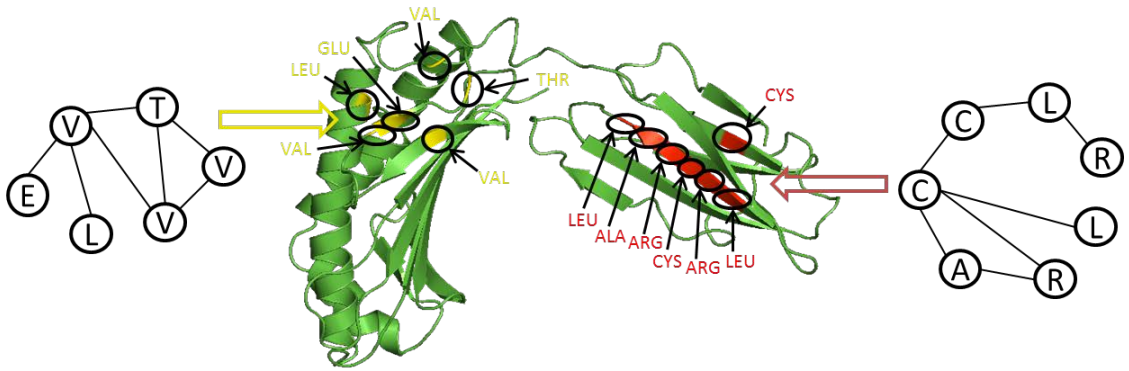
Ing. Vincenzo Carletti

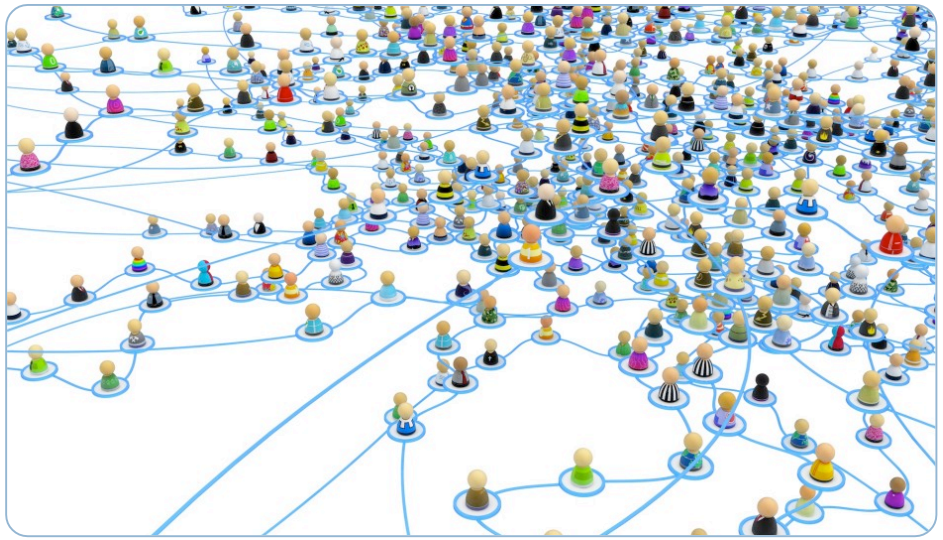
Author:

Christian Guarini

Student ID: 0622700389

Academic Year 2017/2018





How to represent information?

- Graphs data structures can describe a set of objects with their relationships

- Domain of application:
  - Biology : structural representation of proteins
  - Social networks : relationships between two or more entities
  - Pattern Recognition
  - …

Problem Statement

Analysing these data often requires measuring the similarities between two graphs

This kind of problems is known as Graph Matching

Worst-Case Complexity exponential due to the combinatorial nature of the problem

Need for a methodology to reduce the matching time

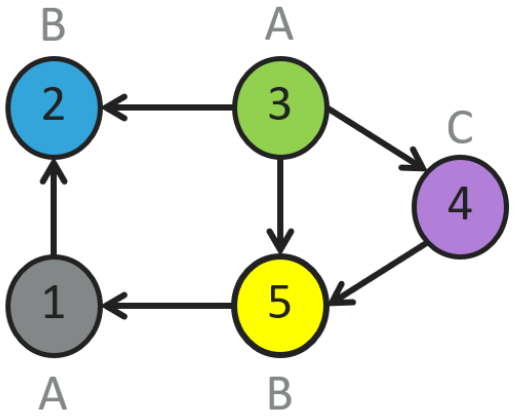
Outline

- Graph Matching problem and parallel computing paradigm
- Briefly description about VF3
- Parallel implementation of VF3
- Discussion of the obtained results
- Conclusions

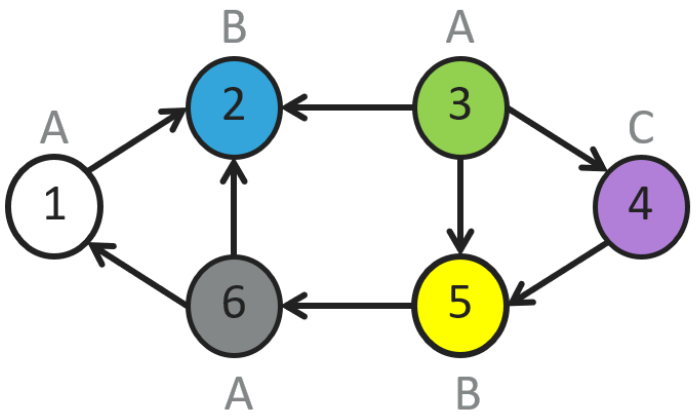
Graph Matching Problem

- Graph Matching problem: compare two graphs to find a correspondence between their nodes and edges

- Subgraph Isomorphism problem between two graphs
  - Pattern graph (the smaller one)
  - Target graph (the bigger one)



Pattern graph



Target graph

Subgraph Isomorphism computational complexity

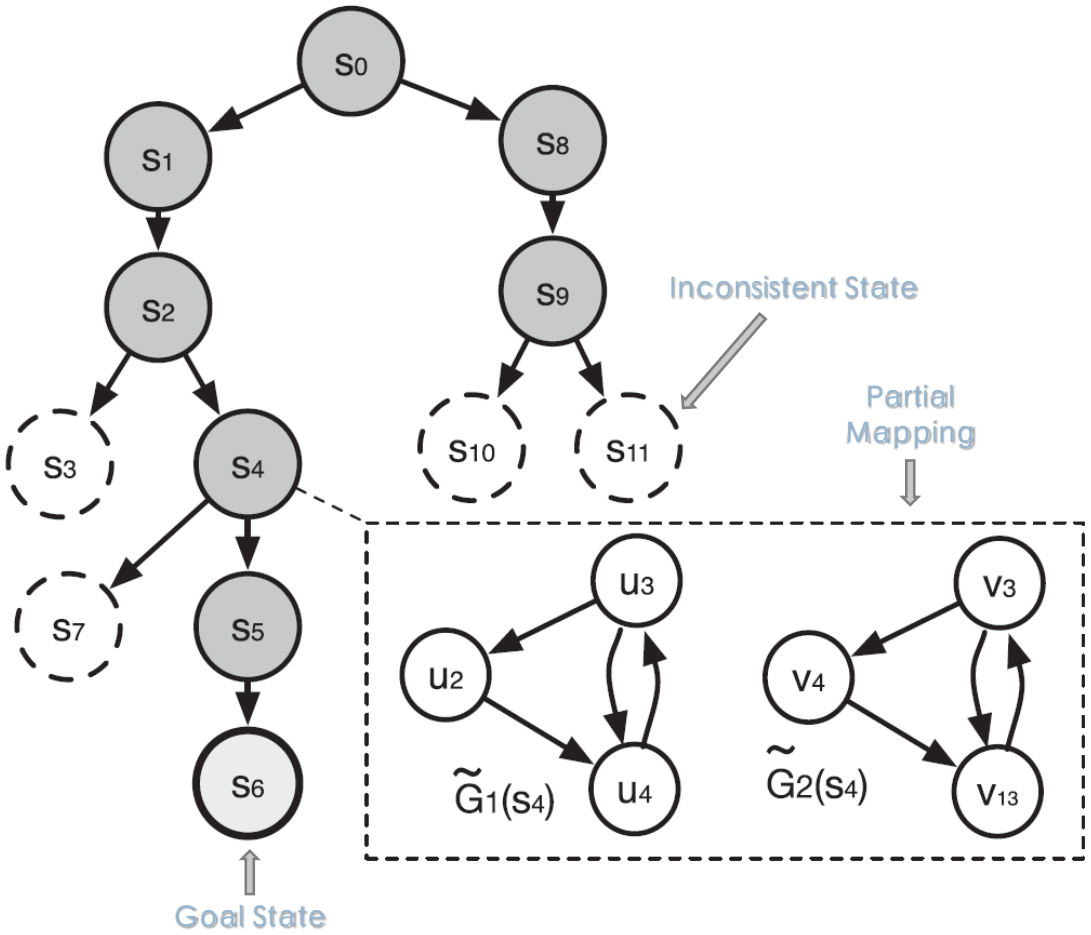
- Worst case computational complexity: exponential
  - Need of a methodology to reduce the execution time of graph matching process
- *Parallel methods* for resolving graph matching problems

Why use parallel computing?

- Save time and money
  - employing more resources to complete a task will shorten its execution time
  - parallel computers can be built from cheap components
- Solve larger and more complex problems
  - many problems are so large and complex that it is impractical to solve them on a single computer, especially given limited computer memory
- Provide concurrency
  - a single resource can only do one thing at a time, multiple resources can do many things simultaneously
- Make better use of underlying parallel hardware
  - in most cases, serial programs that run on modern computers waste potential computational power

VF3: main features

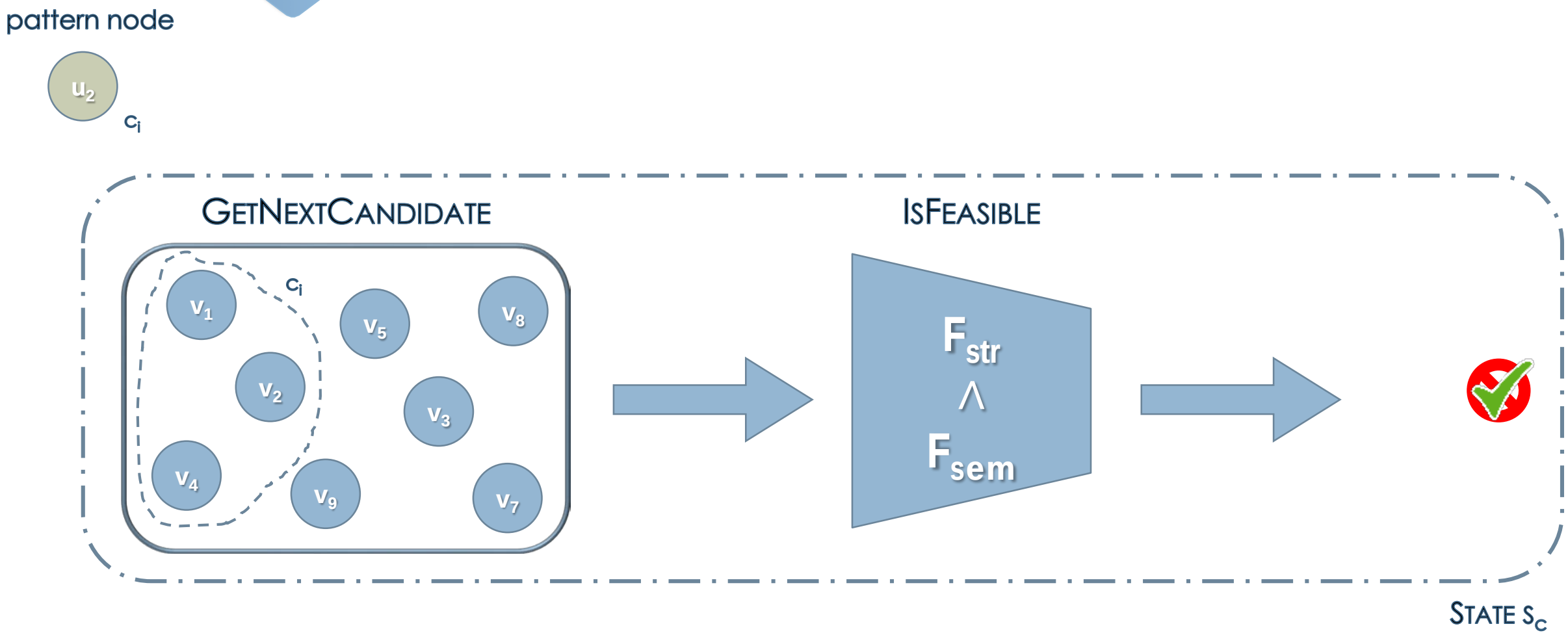
- State Space Representation (SSR)
  - Each state represents a *partial mapping* between the two given graphs
  - A *goal state* represents a consistent complete solution
- Depth-First Strategy with backtracking
  - State space as a tree with a total order relationship
- Heuristics to reduce the number of visited states
  - Introduction of some feasibility rules to explore the space
  - Implementation of a node classification function ψ



VF3: Main functions

- PreprocessPatternGraph
  - Iteratively calculates the feasibility of the pattern graph that are equal for all the states belonging to the same level of the search tree
  - Assigns to each node a parent node that is successively used by the GetNextCandidate procedure

VF3: Main functions (2)



Parallel VF3: assumptions

What parts of the algorithm are suitable for the parallelization?

Two main elaboration steps:

- pre-processing phase
- matching phase

Pre-processing phase

Algorithm 5 Procedure to preprocess the first graph G_1 given an exploration node sequence N_{G_1}. It returns the associative map *Parent*, giving the parent for each node in the sequence.

1: **function** PARPREPROCESSPATTERNGRAPH(G_1, N_{G_1})
2: $i = 0$
3: **for all** $u \in N_{G_1}$ **do**
4: **for all** $u' \in \mathcal{P}_1(u) \cup \mathcal{S}_1(u)$ **do in parallel**
5: $c_i = \leftarrow(u')$
6: **if** $u' \in \mathcal{P}_1(u) \wedge u' \notin \widetilde{\mathcal{P}}_1^{c_i}$ **then**
7: Put u' in $\widetilde{\mathcal{P}}_1$ at level i
8: $Parent(u') = u$
9: **if** $u' \in \mathcal{S}_1(u) \wedge u' \notin \widetilde{\mathcal{S}}_1^{c_i}$ **then**
10: Put u' in $\widetilde{\mathcal{S}}_1$ at level i
11: $Parent(u') = u$
12: $i = i + 1$
13: **return** *Parent*

- Operations take a time at most linear with the size of the two input graphs
- PREPROCESSPATTERNGRAPH presents some indipendent operations in the innermost loop at line 4, where the sets $\mathcal{P}_1$, $\mathcal{S}_1$ and $Parent$ are calculated
- The outermost loop cannot be parallelized, because each iteration requires the results of the previous ones

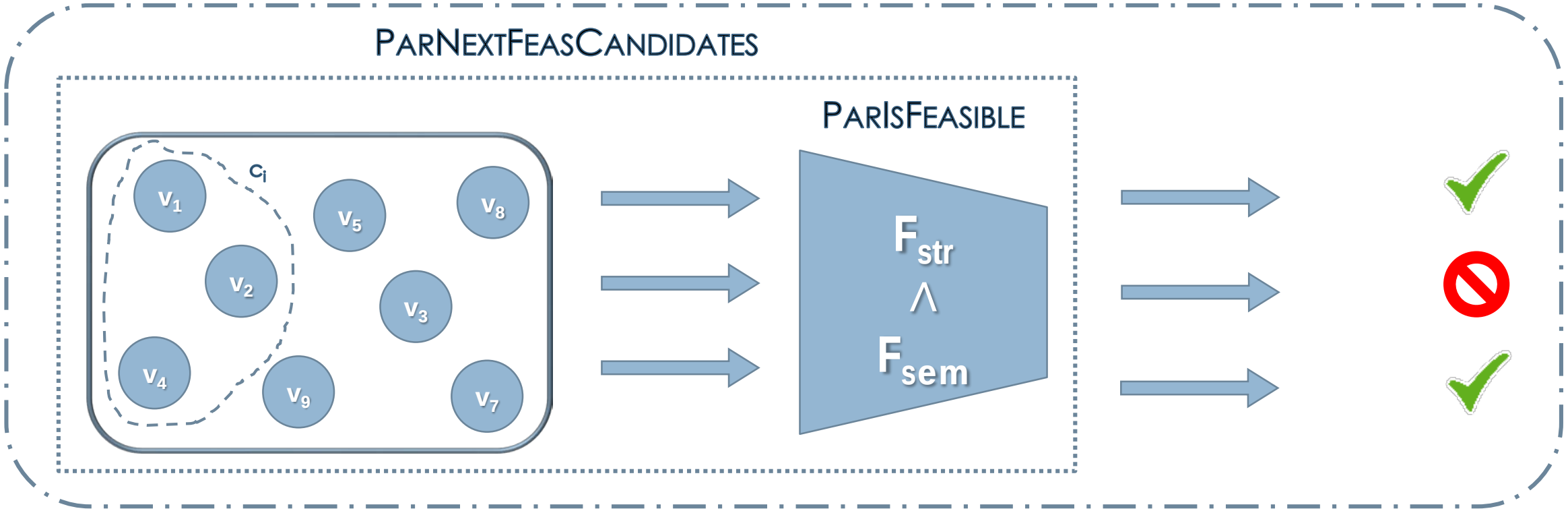
Matching Phase

Algorithm 6 The parallel matching procedure used by Parallel VF3. The inputs are the initial state s_0, the graphs G_1 and G_2 and the two structures, namely N_{G_1} and *Parent*, used during the candidate selection. The procedure also takes as input the set *Solutions* where to store the found solutions. The procedure returns *True* if the solution exists, *False* otherwise.

```
function ParMatch(s_0, G_1, G_2, N_{G_1}, Parent, Solutions)
    Set (u_c, v_c) = (ε, ε)
    Let S be an Empty Stack
    (u_n, v_n) = ParNextFeasCandidates(s_0, N_{G_1}, Parent, G_1, G_2)
    for each (u_{n_i}, v_{n_i}) ∈ (u_n, v_n) do
        s_{n_i} = s_c ∪ (u_{n_i}, v_{n_i})
        Push (S, s_{n_i})
    while S is not empty do
        s_c ← Pop(S)
        if IsGoal(s_c) then
            Append(M̃(s_c), Solutions)
            return True
        else if IsDead(s_c) then
            return False
        else
            (u_n, v_n) = ParNextFeasCandidates(s_c, N_{G_1}, Parent, G_1, G_2)
            for each (u_{n_i}, v_{n_i}) ∈ (u_n, v_n) do
                s_{n_i} = s_c ∪ (u_{n_i}, v_{n_i})
                Push (S, s_{n_i})
    return False
```

- No recursive procedure
- Stack data structure stores all the feasible states, in order to be processed later
- The GetNextCandidate and IsFeasible have been merged into the ParNextFeasCandidates procedure

Matching phase (2)



State s_c

OpenACC

- The standard chosen to parallelize VF3 is OpenACC because it is designed to simplify parallel programming of heterogeneous CPU/GPU systems

- The programmer can annotate C, C++ or Fortran source code to identify the areas that should be accelerated using compiler directives and additional functions

- This gain in portability across different platforms
  - results in a loss of performance
  - programmers are faced with more options and ways to code memory allocations, data transfers, etc.

OpenACC
More Science, Less Programming

Environmental Setup: Dataset

- Dataset of unlabeled and labeled graphs
  - Labeled graphs having eight different numeric labels attached to the nodes
  - Labels assigned using two probability distributions: one is uniform, the other is non-uniform
  - *Unlabeled* target graphs size range: [300 - 7500]
  - *Labeled* target graphs size range: [300 - 10000]

- Density parameter η
  - the larger the value of η is, the more difficult the matching problem becomes
  - η value used for the experimentations: 0,2

- Pattern graph size 20% of size of the target graph

Environmental Setup: Test Machines

CPU: Intel® Xeon® E5-1650 3.2 GHz processor with 6 cores (12 threads) and 12MB cache

RAM: 32GB

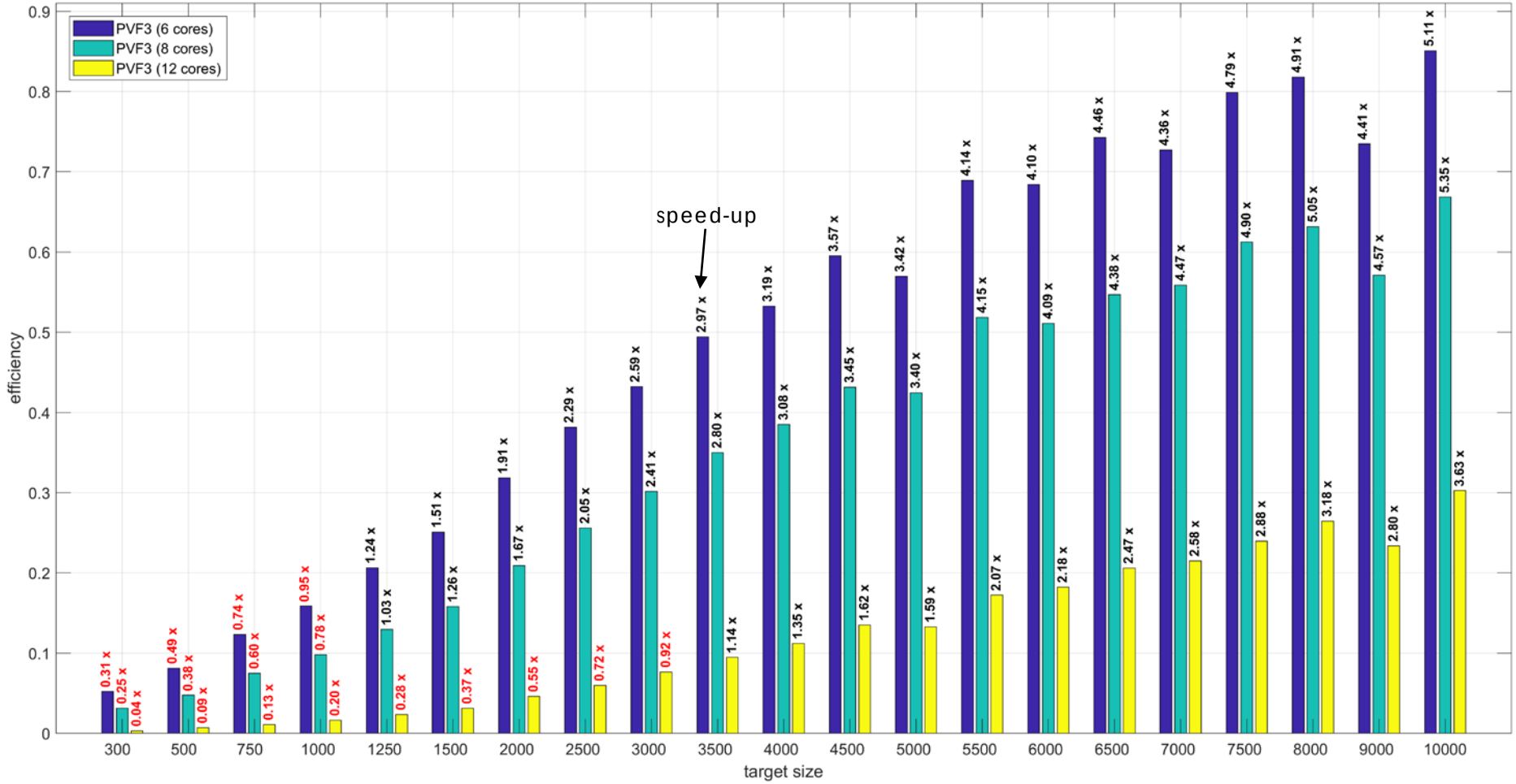
CPU: Intel® Core™i7-6700 HQ processor with 4 cores (8 threads) and 6MB cache

RAM: 16GB

GPU: NVIDIA® GeForce® GTX 960M with 2GB GDDR5

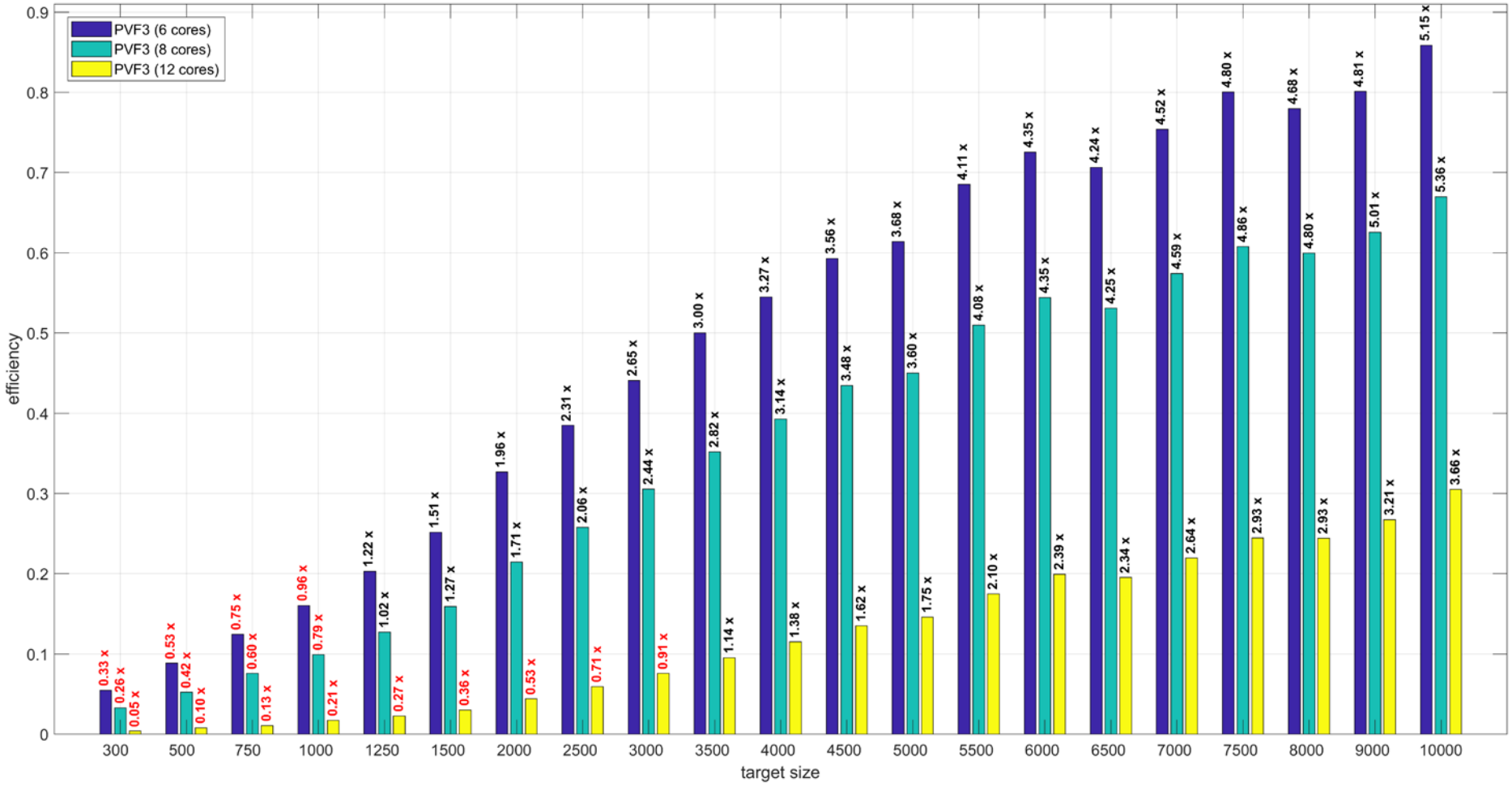
CPU Tests: Efficiency and Speed-Up for uniform labeled graphs

- It is desirable to use the sequential implementation when size < 1250
- Good efficiency when using 6 cores
- For size ∈ [1250 - 6500] 6 cores better than 8 cores
- For size ∈ [7000 - 10000] 8 cores better than 6 cores



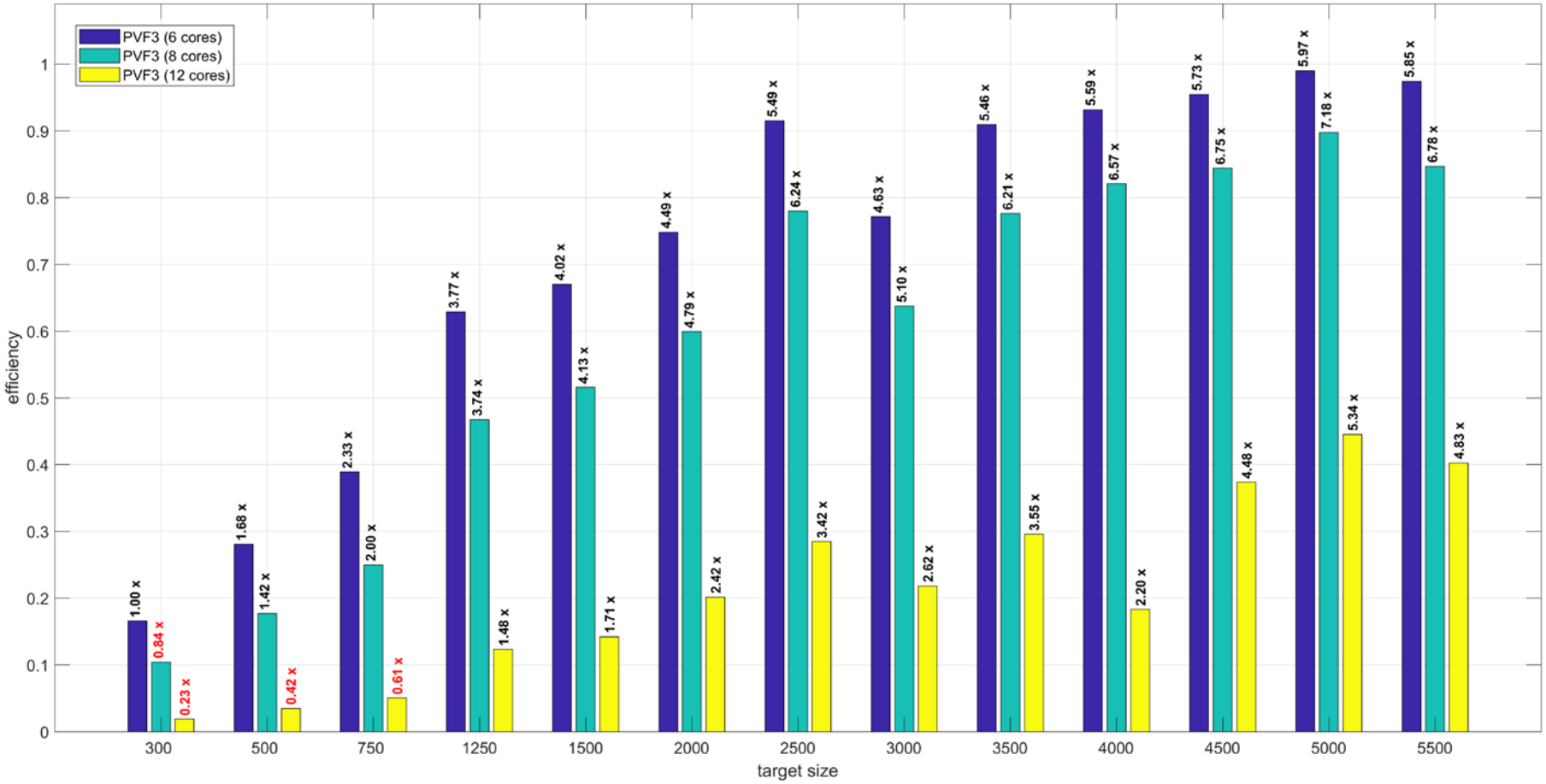
CPU Tests: Efficiency and Speed-Up for non uniform labeled graphs

- It is desirable to use the sequential implementation when size < 1250
- Good efficiency when using 6 cores
- For size ∈ [1250 - 5500] 6 cores better than 8 cores
- For size ∈ [6500 - 10000] 8 cores better than 6 cores



CPU Tests: Speed-Up and Efficiency for unlabeled graphs

- PVF3 performs better than the sequential version
- Good efficiency when using 6 cores
- When size > 1250 good speed-up using 8 cores

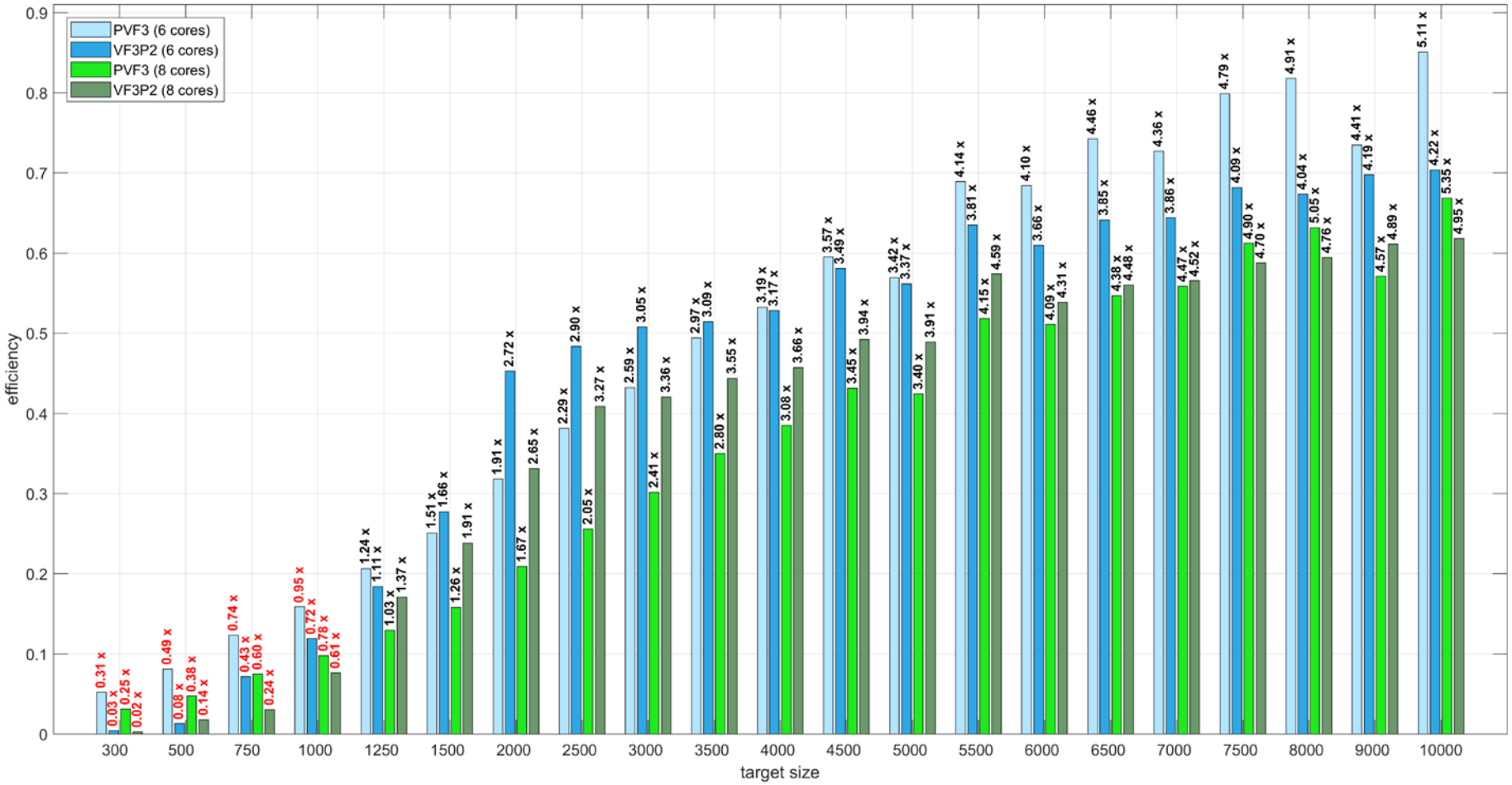


VF3P2

- A multithread release of VF3, implemented by Mivia Lab
  - A shared state stack to store all the feasible states to be processed later
  - Each thread explores more than a state and then stores the feasible states in a local stack, until it completes at least one path of the DFS, from a root state to a leaf
  - When the local stack becomes empty, the thread is allowed to access the global stack to get the next state to explore
  - The algorithm stops when the shared stack and the local stacks become empty and all the threads have finished to explore their remaining states
  - More memory consumption in comparison to the sequential VF3 because of the stacks of unexplored state

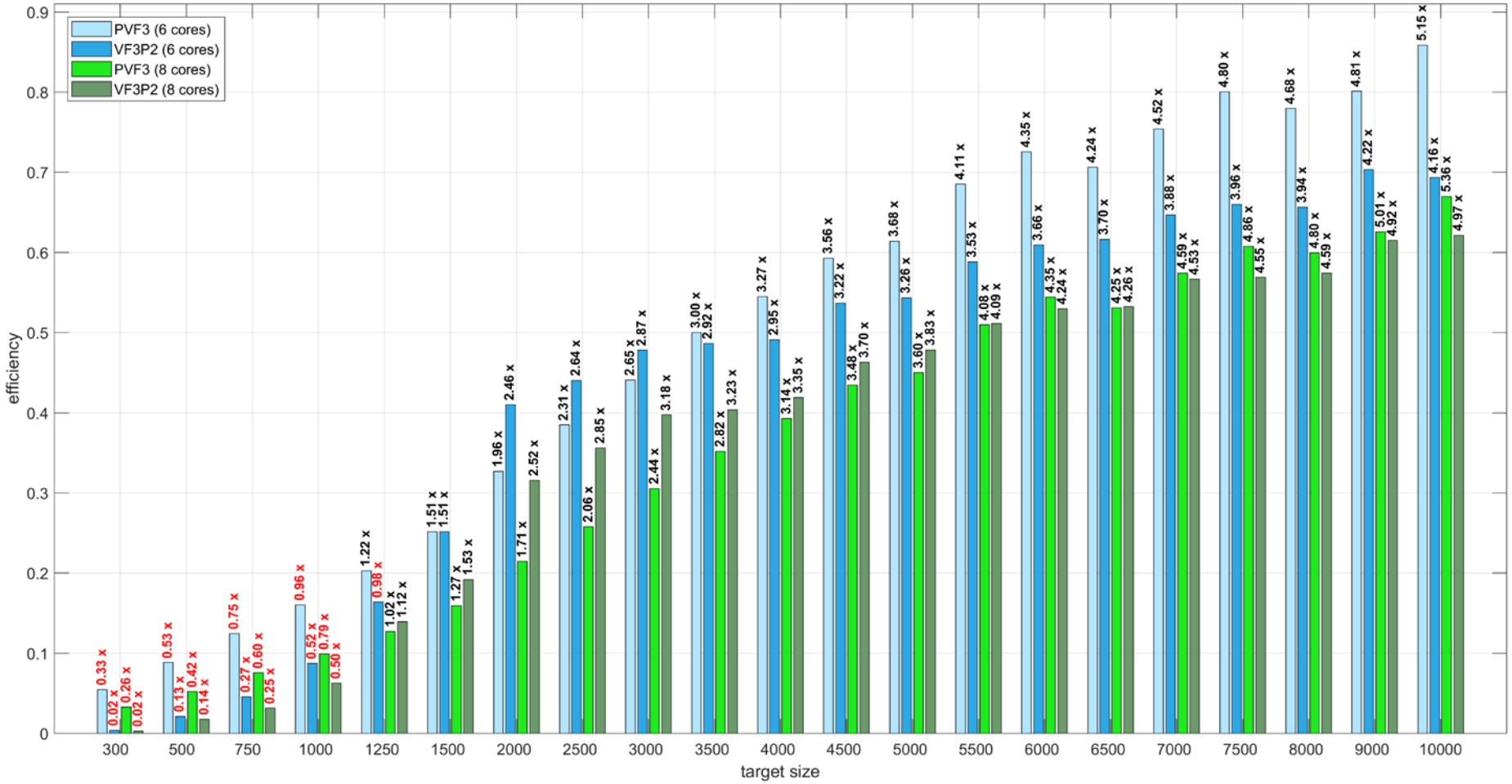
PVF3 vs VF3P2
Uniform labeled graphs

- It is desirable to use the sequential implementation when size < 1250
- 6 cores
  - Size ∈ [1500-3500] VF3P2 better than PVF3
  - Size ∈ [4000-10000] PVF3 better than VF3P2
- 8 cores
  - Size ∈ [1500-7000] VF3P2 better than PVF3
  - Size ∈ [7500-10000] PVF3 better than VF3P2



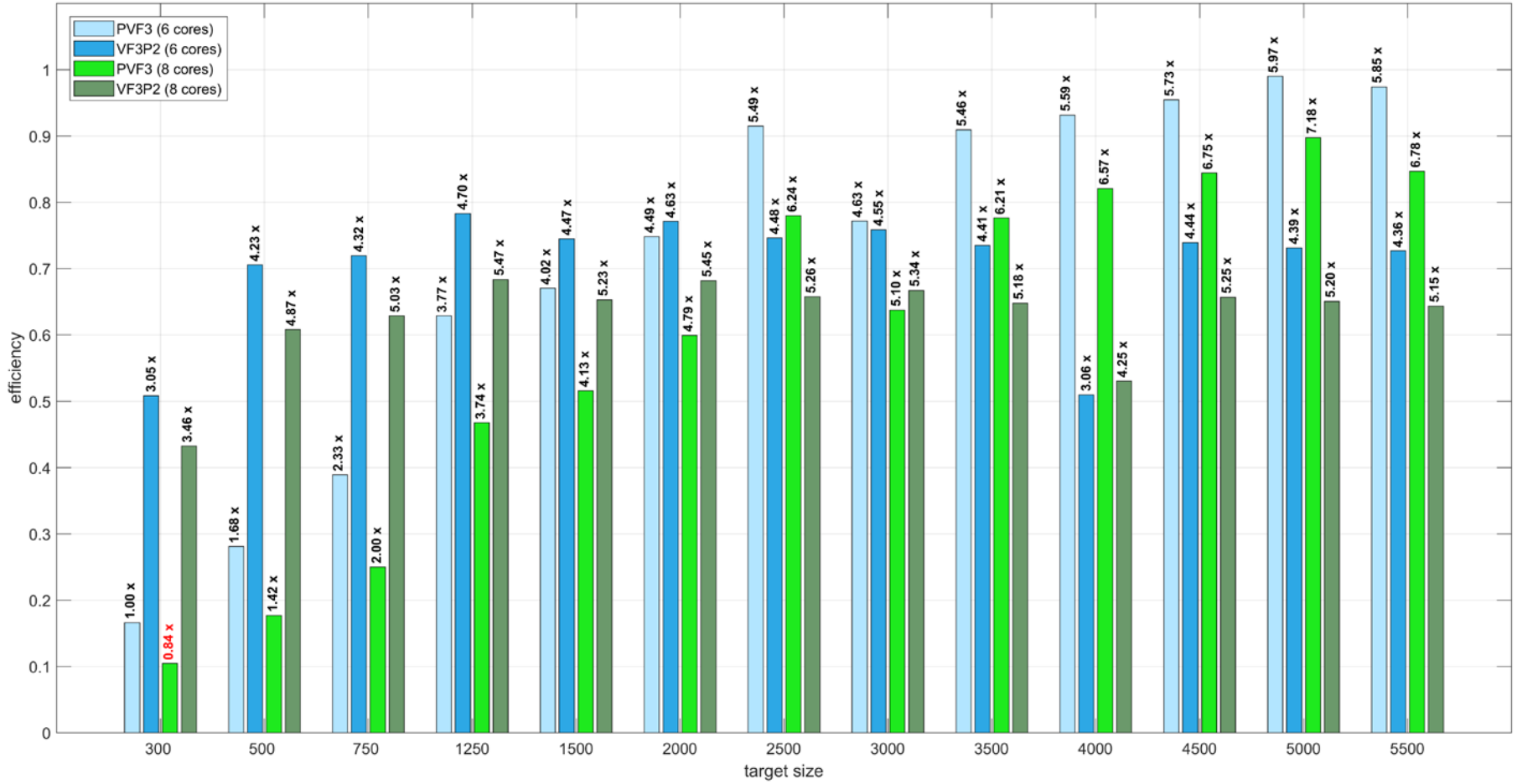
PVF3 vs VF3P2
Non uniform labeled graphs

- It is desirable to use the sequential implementation when size < 1000
- 6 cores
  - Size ∈ [1500-3000] VF3P2 better than PVF3
  - Size ∈ [3500-10000] PVF3 better than VF3P2
- 8 cores
  - Size ∈ [1250-5500] VF3P2 better than PVF3
  - Size ∈ [6000-10000] PVF3 better than VF3P2

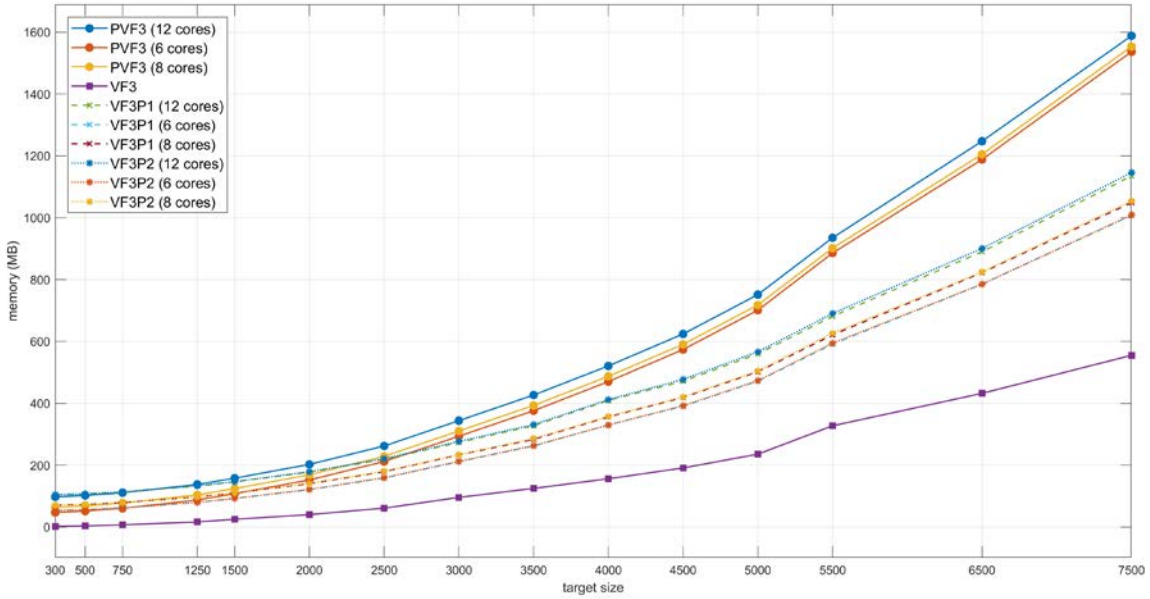


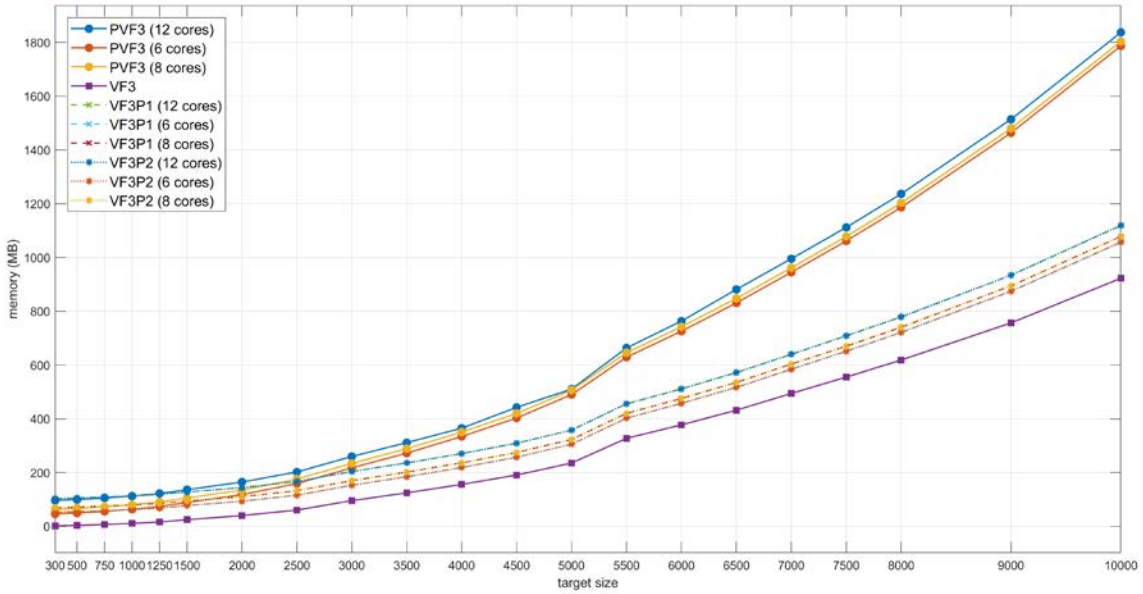
PVF3 vs VF3P2
Unlabeled graphs

- Size ∈ [300 - 2000]
  - VF3P2 better than PVF3
- Size ∈ [2500 - 5500]
  - VF3P2 better than PVF3



CPU Tests: Memory Consumption





PVF3 always requires more memory than the other competitors, because it has to pre-allocate more memory than needed in order to reorganize data structures

GPU Tests: Time Tables

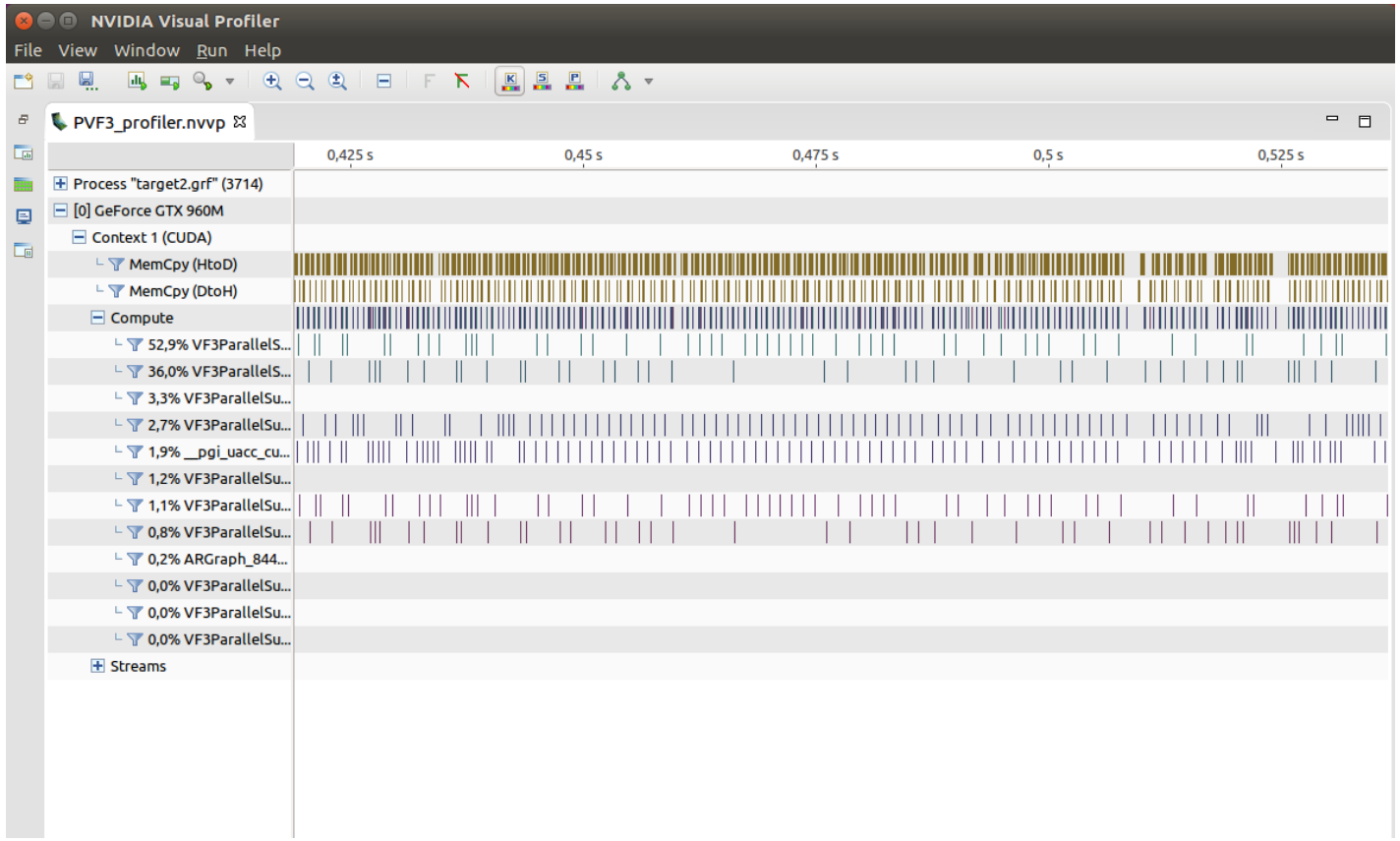
	Labeled Graphs	
Target size	**PVF3**	**VF3**
500	0.336 s	0.001 s
1000	1.116 s	0.007 s
1500	4.369 s	0.043 s
2000	6.179 s	0.074 s

	Unlabeled Graphs	
Target size	**PVF3**	**VF3**
500	79.5 s	1.021 s
1000	804.9 s	16.313 s
1500	1455.4 s	43.69 s
2000	2765.8 s	95.574 s

GPU Tests: considerations

- Considerations about the bad performances of PVF3:
  - OpenACC APIs do not provide to full exploit some architectural optimizations
  - Limited GPU memory and the current compiler restrictions introduce some limitations to the exploration of the state space.
    The problem resolution has inherently sequential characteristics and it is impossible to estimate how many memory requires to complete the tasks
- No full parallelization of the matching phase on GPU: data be copied over the PCI-Express bus from the CPU to the GPU and the results have to be copied back to CPU continuously

GPU Tests: Visual Profiler result



Number of data transfers between CPU and GPU

Computation Time

Conclusions

- Parallel implementation of VF3 subgraph isomorphism using C++ and OpenACC
- PVF3 is the most memory consuming algorithm in comparison to the competitors
  - Suggestion: employment of fixed data strucuters
- GPU tests show bad performances
  - To gain in portability and code maintenance we have to accept these bottlenecks
- Possible GPU implementation using CUDA
  - Available only for NVIDIA hardware (code less portable)
  - Hard to maintain
  - More hardware exploitation

Thanks for the attention