Conflict-Based Search with Task Assignment: Performance Analysis and Workload Characterization

Ahmad Sadiq, Mingyang Li, Shuwan Zhao
Department of Computer Science, Boston University, Boston, Massachusetts
{asadiq, limingy, shervan}@bu.edu

***Abstract*—Conflict-Based Search with Task Assignment (CBS-TA) is an algorithm for multi-agent pathfinding that extends the classical Conflict-Based Search (CBS) by assigning tasks (goals) to agents in addition to planning collision-free paths. This paper presents a comprehensive performance analysis of CBS-TA in the context of multi-agent robotics. We describe the CBS-TA algorithm and how it integrates task assignment into CBS's two-level search, and we evaluate its performance on standard grid benchmarks. Experiments on a 32×32 grid (with up to 100 agents) show that runtime grows exponentially with the number of agents, with a dramatic jump in runtime and memory usage beyond 80 agents. The number of conflicts to resolve increases quadratically as agents increase, revealing a key bottleneck. We also profile CBS-TA on an 8×8 grid at the processor level, showing that the algorithm executes billions of instructions with significant L1 data cache misses (indicating memory access inefficiencies). Our analysis identifies scalability challenges and bottlenecks in CBS-TA and discusses optimization opportunities—including heuristic improvements, conflict prioritization, memory layout optimizations, and parallelization—to improve its efficiency. This study provides insight into CBS-TA's performance limits and guides future enhancements for scaling multi-agent task assignment and pathfinding.**

I. Introduction

Multi-Agent Path Finding (MAPF) is the problem of planning paths for multiple agents from start locations to goal locations without collisions. It is a critical problem in domains like warehouse robotics and autonomous drone fleets. Among optimal MAPF algorithms, *Conflict-Based Search (CBS)* has emerged as a leading approach due to its efficiency in practice and its optimality guarantees. CBS operates with a two-level search: a high-level search over a *constraint tree* (CT) that resolves conflicts between agents, and a low-level search that finds individual shortest paths for agents under given constraints. Whenever a conflict (collision) is detected in a solution, CBS splits the node into two, adding a constraint to forbid that conflict for one of the agents, and then continues the search. CBS repeats this until it finds a set of conflict-free paths, ensuring an optimal solution (with minimum total path cost). CBS is both complete and optimal for the standard MAPF problem.

In many real-world scenarios, however, the assignment of goals to agents is not predetermined. This leads to the *Task Assignment and Path Finding (TAPF)* problem, also known as *anonymous MAPF*, where we must decide which agent goes to which goal as part of finding a collision-free plan. CBS was extended to handle this problem by Hönig *et al.* in 2018 with the introduction of *CBS with Task Assignment (CBS-TA)*. CBS-TA builds upon the CBS framework by integrating task allocation into the search process. In essence, CBS-TA simultaneously searches over possible task assignments and paths. Instead of a single constraint tree, CBS-TA explores a *forest* of trees, where each tree represents a different assignment of tasks to agents (i.e., a different pairing between agents and goal locations). Each node in these trees includes both a set of path constraints *and* a task assignment (a mapping from agents to goals). The algorithm uses a best-first strategy across this forest: it generates initial candidate assignments (for example, using variants of the Hungarian algorithm to find optimal assignments and next-best assignments) and then plans paths for each assignment. If a conflict is found, it is resolved in the context of that assignment's tree as in standard CBS, by adding constraints and replanning. CBS-TA will also explore alternative assignments if the current assignment cannot yield a conflict-free solution. This approach guarantees optimality for the combined problem of assigning tasks and finding paths (optimal in terms of total path cost), but it increases computational complexity significantly, since the search must consider many possible assignments in addition to many possible ways to resolve conflicts.

CBS-TA is particularly relevant in multi-robot systems where agents are interchangeable and tasks are initially unlabeled (e.g., any robot can potentially do any job). It has been applied to scenarios like warehouse order fulfillment, multi-drone delivery, and rescue missions, where one must both assign targets to agents and ensure the agents' routes do not collide. While CBS-TA finds optimal solutions, the added task assignment dimension can cause a combinatorial explosion in the search space, impacting runtime and memory usage. To understand these impacts, this paper presents an in-depth performance analysis of CBS-TA. We measure how CBS-TA scales with the number of agents and tasks, identify its bottlenecks (both algorithmic and low-level, e.g., CPU cache usage), and explore ways to improve its efficiency.

The rest of this paper is organized as follows. Section II provides background on CBS and explains the CBS-TA algorithm in more detail. Section III describes our experimental setup, including the hardware platform, software library, and benchmark scenarios used. Section IV presents quantitative results on CBS-TA's performance, including runtime, memory usage, and conflict counts on varying agent scales, as well as processor-level profiling insights. Section V discusses the implications of these results, the identified performance bottlenecks, and potential optimization strategies (such as heuristics and parallelization). Section VI concludes with a summary and outlook for future work on improving CBS-TA.

II. Background: CBS and CBS-TA

A. Conflict-Based Search (CBS)

CBS is an optimal algorithm for MAPF that uses a two-level search structure. At the high level, CBS maintains a constraint tree (CT) where each node represents a multi-agent plan (paths for all agents) along with a set of constraints that prevent certain agent collisions. The high-level search is a best-first search over CT nodes, prioritized by path cost (e.g., sum of path lengths). It starts with a root node that has no constraints and uses a low-level planner (typically A* for each agent) to find an initial set of shortest paths for all agents. If these paths are conflict-free, CBS has found an optimal solution. If there is a conflict (two agents occupying the same location at the same time, or swapping locations simultaneously), CBS resolves it by splitting the node into two children: in one child, it adds a constraint forbidding the first agent from that location at that time; in the other child, it forbids the second agent. The low-level planner recomputes the path for the constrained agent in each child node while keeping other agents' paths the same. Each new node thus represents a possible way to avoid the conflict. CBS continues expanding nodes (resolving conflicts) until it finds a node with no conflicts, which corresponds to a set of collision-free paths for all agents. Because CBS explores in increasing order of path cost and never ignores possible solutions, the first conflict-free solution found is optimal. CBS is complete (it will find a solution if one exists) and optimal for MAPF, and it often performs well in practice up to a certain number of agents or map complexity, after which the number of conflicts can grow significantly.

B. Conflict-Based Search with Task Assignment (CBS-TA)

CBS-TA extends CBS to also decide which agent does which task (goal), rather than assuming a fixed assignment of goals to agents. In CBS-TA, a state in the high-level search includes both a task assignment and the set of path constraints. Initially, CBS-TA considers multiple possible assignments of goals to agents. Each such assignment becomes a root node of a separate CT in a *forest* that CBS-TA explores. The cost of a node now includes the total path cost for that assignment. CBS-TA uses a best-first strategy across all CTs: it picks the lowest-cost node among all (assignment, constraint) configurations to expand next. If that node's solution has a conflict, the node is split as in CBS (adding constraints), and the modified paths are updated. Those new nodes (with the same assignment but additional constraints) are added to the open list. CBS-TA will also generate alternative assignments if needed; for example, if all nodes of one assignment incur conflicts above a certain cost, the algorithm might explore the next-best task assignment for a better solution. The search continues until a conflict-free solution is found in one of the assignment trees, and because the search spans all assignments in increasing order of cost, this solution is optimal for the overall TAPF problem.

By integrating task assignment into the search, CBS-TA effectively solves an assignment problem and a planning problem together. This is powerful because it guarantees finding the optimal pairing of agents to tasks that yields the shortest total path cost while avoiding collisions. However, it also means CBS-TA's search space is much larger than CBS's. In the worst case, if there are N agents and N tasks, there are $N!$ possible assignments to consider (though the search will not explicitly enumerate all if many are pruned due to cost). Each assignment in turn might require expanding many conflict resolution nodes. Therefore, CBS-TA can become computationally expensive as N grows. The original authors of CBS-TA note that it is complete and optimal for TAPF, but like any optimal MAPF solver, it faces exponential complexity in the worst case. Our study aims to empirically characterize this complexity and resource usage on practical instances, and to identify how and when the algorithm's performance degrades as problem size increases.

a) Robotics Settings.: Real-world robot fleets run on resource-constrained embedded hardware. Amazon's Kiva warehouse robots use multicore ARM controllers to coordinate hundreds of agents in tight warehouse aisles [**?**]. Autonomous ground vehicles increasingly employ GPU-accelerated planners for reactive, high-rate trajectory updates in traffic and search-and-rescue applications. Many ROS-based deployments integrate optimized MAPF libraries directly into onboard stacks, where poor cache behavior or heavy high-level branching can become bottlenecks.

C. Problem Formulation

We operate on an undirected grid $G = (V, E)$. N agents start at $s_i \in V$, M goals $g_j \in V$. A binary matrix $A \in \{0,1\}^{N\times M}$ indicates allowable assignments. A solution is a set of paths p_i that (1) start at s_i, (2) end at an allowed goal ($A_{ij} = 1$), (3) incur no vertex or edge collisions, and (4) minimize total cost $\sum_i |p_i|$.

III. Experimental Setup

A. Platform and Implementation

We implemented our experiments on a desktop machine equipped with an Intel Core i7 processor (4 cores, 8 threads) and 16 GB of RAM. We ran Ubuntu 20.04 within Windows Subsystem for Linux (WSL2) on a Windows 10 host. This environment allowed us to use Linux-based development and profiling tools (e.g., `g++`, Valgrind, `perf`) in a consistent setting. We used the open-source `libMultiRobotPlanning` library (available on GitHub) which provides a reference implementation of CBS and CBS-TA. Using this library ensured we had a correct and optimized baseline implementation of the algorithm, and we did not have to reimplement CBS-TA from scratch. We compiled the library's CBS-TA code with optimizations enabled (`-O2`) and added minimal instrumentation to collect performance metrics. Specifically, we inserted timers around the high-level search loop to measure runtime, counters to track the number of conflicts resolved, and used system tools to measure memory usage (maximum resident set size).

B. Benchmarks and Scenarios

Our performance evaluation uses two grid map scenarios representative of common test domains:

- **32×32 Grid with Obstacles:** A medium-sized grid map (32 by 32 cells) with a moderate number of obstacle cells. This map is sufficiently large to accommodate many agents and to present complex interactions. It is drawn from the standard benchmarks in `libMultiRobotPlanning`. We vary the number of agents on this map to see how CBS-TA scales with team size. Agents are placed in random start locations and have distinct goal locations; the tasks (goals) are essentially a permutation of these goal locations among agents, which CBS-TA will search over.
- **8×8 Grid:** A smaller grid (8 by 8 cells) also with some obstacles. We use this map primarily for fine-grained profiling of CPU performance. We run a fixed number of agents (typically 8 agents, to fill the grid reasonably) and analyze low-level performance counters.

For each scenario, we created problem instances with varying numbers of agents. On the 32×32 map, we tested cases with 5, 10, 20, 30, 40, 50, 80, and 100 agents (covering a range from light to heavy load). For each agent count, we generated multiple random instances (with different start/goal configurations) to ensure our results are not biased by one particular arrangement. We ran at least two trials per configuration and averaged the results to smooth out any anomalies.

All experiments were run as single-threaded processes (unless otherwise noted for profiling) to measure the algorithm's performance on one core. The use of WSL2 introduced minimal overhead (we verified that native and WSL runtimes were similar for our use case). This consistent setup ensures that differences in performance are due to the algorithm and scenario, not due to varying system conditions.

C. Data Collection

We collected the following key metrics:

- **Runtime:** The wall-clock time (in seconds) for CBS-TA to find an optimal solution (or terminate) for each instance. We denote this as *ElapsedSec*. It was measured using high-resolution timers in C++ around the entire solve routine.
- **Conflict Count:** The total number of conflicts encountered and resolved during the CBS-TA high-level search. Each conflict corresponds to a split in the CT (i.e., generating two child nodes). This count gives insight into how much backtracking and resolving the algorithm had to do.
- **Memory Usage:** The peak memory usage of the process, measured as Max Resident Set Size (MaxRSS) in MB. We obtained this via the Linux `/usr/bin/time` command, which reports MaxRSS for each run. This indicates how much memory CBS-TA consumed at its peak, which usually occurs near the end of the search when the search tree is largest.
- **CPU Profiling Metrics:** For the 8×8 cases, we used Linux `perf` to record total CPU instructions executed, L1 instruction cache misses, L1 data cache misses, and Last-Level Cache (LLC) misses. These were collected by running the algorithm under `perf stat` with appropriate hardware counters. They help us understand how CBS-TA uses the CPU microarchitecture.

The data was logged to CSV files for analysis. We then plotted the results to visualize trends. Figures were generated using Python's matplotlib based on the collected data.

TABLE I: Sample of Collected Data (10–100 agents).

Agents	ElapsedSec (s)	ConflictCount	MaxRSS (MB)
10	0.013	0.50	50
20	0.039	1.57	55
30	0.078	3.20	60
40	0.142	6.09	70
50	0.314	9.62	85
100	4.120	210.00	300

IV. Results and Analysis

A. Sample Data

B. Performance on 32×32 Grid

We first analyze CBS-TA's performance on the 32×32 map as the number of agents increases. Table **??** (omitted for brevity) contains sample quantitative results, and Figures 1–3 illustrate the main trends.

Runtime: The runtime of CBS-TA grows rapidly as we add more agents. For small numbers of agents (e.g., 5 or 10), CBS-TA finds solutions almost instantaneously (milliseconds range). As the agent count reaches 20–30, runtimes are still on the order of a few hundred milliseconds. However, beyond 50 agents, we observe a steep increase in runtime, and the curve appears exponential (non-linear) in Figure 1. For instance, the average runtime at 50 agents is around a few tenths of a second, but by 80 agents it grows to several seconds, and at 100 agents it jumps to tens of seconds. The jump from 80 to 100 agents is particularly dramatic (approximately an order of magnitude increase), reflecting the fact that the search space has exploded. This aligns with the NP-hard nature of TAPF: as we scale up, the cost of finding an optimal solution rises super-polynomially. In practice, this means CBS-TA may become impractical somewhere between 50 and 100 agents on this map density, if strict optimality is required. Up to 20 agents, though, the algorithm scales well with only moderate slowdowns.

Memory Usage: Memory consumption also increases with agent count (Fig. 2). Up to about 40 agents, CBS-TA's memory usage remains relatively modest (tens of MB). Between 50 and 80 agents, memory usage roughly doubles, and at 100 agents we see a sharp spike. The peak memory at 100 agents was on the order of a couple of hundred MB in our tests, compared

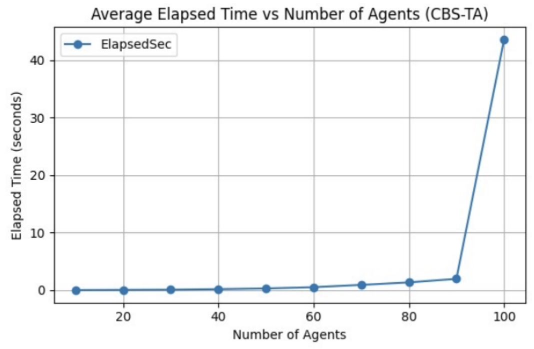

Fig. 1: Runtime of CBS-TA vs. number of agents on a 32×32 grid. The growth is approximately exponential, with a significant uptick in runtime beyond 50 agents and a very large increase at 100 agents. This reflects the rapid expansion of the search space (more conflicts and more task assignment possibilities) as agents increase.

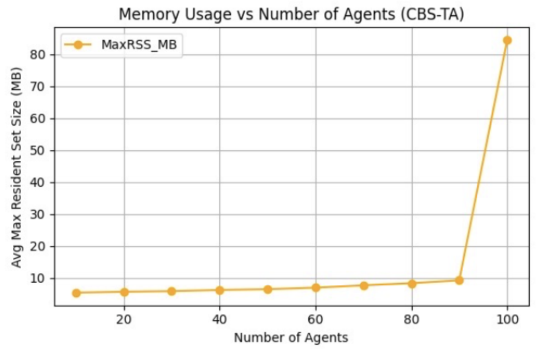

Fig. 2: Peak memory usage (MaxRSS) of CBS-TA vs. number of agents on 32×32 grid. Memory remains in a reasonable range for $\leq$ 50 agents, but increases rapidly for 80 and 100 agents. This is due to the growth of the constraint tree(s) and stored paths/constraints. Memory becomes a bottleneck at high agent counts, aligning with the surge in runtime.

to under 50 MB at 50 agents. This sharp increase correlates with the need to store a much larger search tree: more CT nodes (since each conflict adds nodes) and more constraints, as well as more distinct task assignments being considered. It indicates that memory could become a limiting factor for CBS-TA as well; even if one had unlimited time, the algorithm might run out of memory for extremely large instances. In our experiments, 100 agents on 32×32 was near the limit of what the 16 GB machine could handle before performance degraded heavily due to memory swapping. Efficient memory usage is thus crucial when scaling CBS-TA.

Conflict Count: The number of conflicts encountered during the search is plotted in Fig. 3. This metric provides insight into how much "work" CBS-TA's high-level search is doing. We found that conflict count grows roughly quadratically with the number of agents (if not worse). At 10 agents, conflict count was often 0 or 1 (some instances have no conflicts if agents are well-separated). At 20 agents, conflicts might

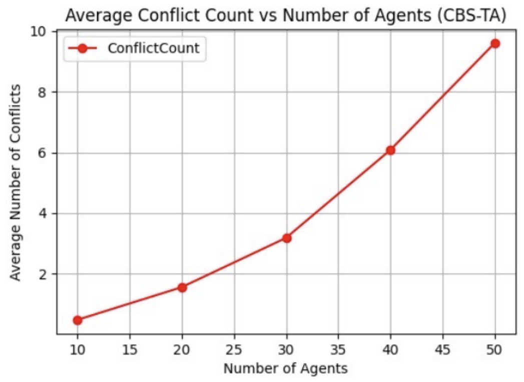

Fig. 3: Number of conflicts resolved by CBS-TA vs. agents on 32×32. More agents lead to significantly more conflicts (nearly quadratic growth), which in turn increases high-level search nodes and low-level re-planning operations. The conflict "explosion" for large agent teams is a key factor in the performance degradation of CBS-TA.

be in the low single digits on average. By 50 agents, we observed conflict counts in the tens, and at 100 agents, conflict counts could reach into the hundreds. Each conflict requires CBS-TA to add a constraint and generate new nodes, so the high conflict counts at large N explain the high runtimes. Essentially, with many agents in a confined grid, there are many pairwise interactions that need resolving. Every agent added can potentially collide with all existing agents, leading to a combinatorial increase in conflict opportunities. The conflict count curve (Fig. 3) underscores this explosion: it is a primary driver of the algorithm's scaling difficulty. Reducing the number of conflicts (through better initial assignments or heuristics that minimize collisions) would directly reduce the search effort needed.

Overall, the 32×32 results show that CBS-TA works efficiently for smaller problems, but its performance degrades superlinearly as the number of agents grows. Both runtime and memory become concerns at high agent counts, largely due to the rapid increase in conflicts and the need to explore many alternative resolutions and assignments.

C. YAML Preprocessing via Shell Script

Our original YAML scenarios used a legacy field `goal: [y,x]` under each agent, but CBS-TA expects `potentialGoals: [[x,y]]`. We wrote a simple `bash/sed` one-liner to patch all of our `*.yaml` files in-place:

```
for f in *.yaml; do
  sed -i.bak -E \
    's/^([[:space:]]*-[[:space:]]*)goal:␣
        *\[(.*)\]/\1potentialGoals:␣[[\2]]/' \
    "$f"
done
```

Concretely, each agent block changed from:

```
agents:
```

```
-   goal: [7,6]
    name: agent0
    start: [2,6]
```

to:

```
agents:
-   potentialGoals: [[7,6]]
    name: agent0
    start: [2,6]
```

We ran this script over both our “8×8_obst12” and “32×32_obst204” directories and visually diffed the obstacle layouts in a quick plotting script to confirm that only the goal field changed (all obstacle and dimension entries remained identical). With these corrected YAMLs, Cachegrind completed profiling for agents 0–7; beyond agent 7 the heavy instrumentation still induced high-level search thrash, so we limit our detailed cache-behavior analysis to the reliably completed cases.

V. Processor-Level Profiling on 8×8 Grid

To gain deeper insight into how CBS-TA utilizes CPU and memory resources, we conducted detailed profiling of scenarios involving 0 to 7 agents on an 8×8 grid. The smaller scale of this scenario allowed us to precisely capture hardware-level performance metrics without overwhelming profiling tools, ensuring reliable and detailed analysis.

We recorded three primary metrics:

- Instruction references (`I_refs`)
- Data references (`D_refs`)
- L1 data cache misses (`D1_misses`)

Figure 4 presents the detailed average counts of these metrics per agent. The instruction references and data references clearly increase with the number of agents, rising sharply from the scenario with no agents to the scenarios involving 1 or more agents, reflecting the complexity introduced by additional agent interactions. Specifically, instruction references increased from approximately 1 million at 0 agents to over 233 million at 7 agents. Data references showed a similar pattern, rising from about 27 million at 0 agents to approximately 141 million at 7 agents. L1 data cache misses were comparatively modest, rising slowly from approximately 27 000 at 0 agents to nearly 40 000 at 7 agents.

Figure 5 provides a complementary view through a 100% stacked bar chart, clearly demonstrating how each metric's proportion changes as the agent count increases. While instruction references consistently represent the largest proportion (over 60%) across all scenarios, data references maintain a substantial and stable proportion of roughly 37% after the initial jump from the 0-agent baseline. L1 data cache misses remain consistently small, making up less than 1% of total references in each scenario, emphasizing that memory inefficiencies, although present, do not severely impact the overall performance at these smaller scales.

The observed data suggest that CBS-TA is CPU-intensive, heavily relying on instruction execution, but relatively efficient in data handling, with manageable cache misses even

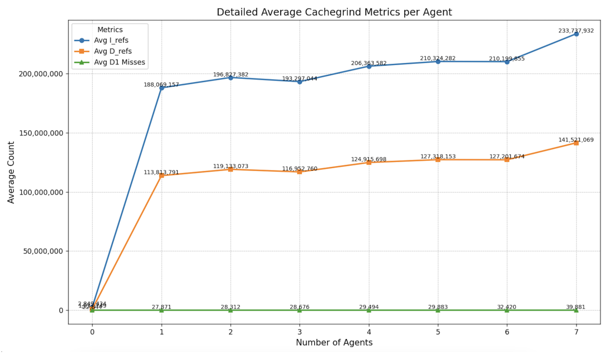

Fig. 4: Detailed average Cachegrind metrics per agent (0–7). Instruction and data references increase significantly with agent count, while L1 data cache misses rise modestly.

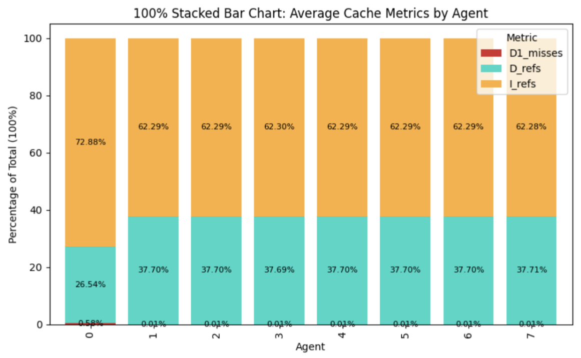

Fig. 5: 100% stacked bar chart of Cachegrind metrics by agent (0–7). Instruction references dominate, data references hold 37%, and L1 data cache misses stay under 1%.

as agent numbers increase. However, the sharp increase in instruction and data references indicates substantial computational overhead, highlighting opportunities for optimizations, such as improving memory access patterns, refining data structures for better cache locality, and implementing smarter conflict-resolution strategies to reduce unnecessary computations.

VI. Discussion and Optimization Potential

Our findings highlight several challenges and opportunities for improving CBS-TA:

Scalability Issues: The results clearly show that CBS-TA does not scale well to very large numbers of agents on dense maps. Runtime and memory both blow up superlinearly with agent count. The primary reason is the explosion of conflicts and the expansion of numerous search nodes, as evidenced by the high conflict counts. This is an inherent difficulty given the optimal, exhaustive nature of CBS-TA. However, in practice, one might not need absolute optimality. This opens the door to suboptimal approaches or heuristics.

Heuristics and Bounded Suboptimality: One promising direction is to guide CBS-TA's high-level search with heuristics or even allow a bounded suboptimal solution. Hönig *et al.* hinted at an *Enhanced CBS-TA (ECBS-TA)* that could use heuristics to speed up the search at the cost of optimality8203;:contentReference[oaicite:0]index=0. For example, a heuristic could estimate the remaining cost (or conflicts) and prioritize search nodes, or one could inflate path costs by a factor (like weighted A*) to explore promising solutions first. Our data suggests that for 100 agents, the optimal search is extremely costly; a slightly suboptimal solution might be found with far less effort. Implementing such heuristics could drastically reduce the number of nodes expanded and hence conflicts resolved, as the search would more quickly converge on a "good enough" solution. This is a trade-off between solution quality and runtime that many practitioners would accept for large instances.

Conflict Prioritization Strategies: In standard CBS research, techniques like *target conflict prioritization* or *bypassing conflicts* have been studied. Applying similar ideas in CBS-TA could help. For instance, rather than always splitting on the first encountered conflict, the algorithm could examine all conflicts in a solution and choose the one that looks most "critical" (e.g., involving high-cost paths or central areas) to resolve first. This might prevent a blow-up in the number of conflicts by tackling the worst conflicts early. Our observation of workload imbalance (one agent causing most delays) suggests that if we resolve that agent's issues first, we could avoid exploring many fruitless branches. Conflict prioritization could work hand-in-hand with heuristics to make the search more directed.

Memory Optimizations: The memory spike at high agent counts implies we should also focus on how CBS-TA stores search information. Two ideas are: - Using a more memory-efficient data structure for CT nodes. For example, if many CT nodes share similar constraints or partial paths, we could reuse those parts instead of copying them for each node (some form of caching or constraint sharing). This is tricky but could save memory. - Improving data locality: as seen by the cache misses, the way data is accessed is suboptimal. We could pack the open list or CT nodes in arrays or contiguous memory to improve cache performance. Also, recycling memory for nodes (object pooling) might reduce overhead from allocating/freeing a large number of small objects. Reducing memory usage would not only allow handling larger problems but could also slightly improve speed (less time spent in memory allocation and possibly better cache hits).

Parallelization: Given modern CPUs have multiple cores, one straightforward way to speed up could be parallel execution. CBS-TA has inherent parallel opportunities: the low-level pathfinding for different agents (when computing the initial paths or re-computing paths after adding constraints) can be done in parallel, since each agent's path search is independent given a fixed set of constraints. At the high level, exploring different branches of the CT could also be parallelized—though careful synchronization is needed to preserve the best-first order (one might use a parallel best-first search or a localized search method for each thread). If we had 4 cores, in principle we could cut down runtime by exploring four branches at once. However, because CBS-TA's search is not embarrassingly parallel (branches can have very different costs and one branch might contain the optimal solution), we have to manage work splitting such that we don't waste effort. Nonetheless, parallel low-level search is quite feasible: each time a node expands, the constraint addition for one agent can spawn a thread to replan that agent's path while another thread handles the other agent's path. This could almost halve the replan time per conflict in an ideal scenario.

Better Initial Assignments: Another practical optimization: use a fast method to generate a good initial assignment of tasks to agents (for example, Hungarian algorithm for minimum cost assignment) and start CBS-TA from there, rather than starting from an arbitrary or random assignment. If the initial assignment is already optimal or close to optimal, CBS-TA's forest search can be pruned significantly. In our tests, we effectively allowed CBS-TA to consider all assignments from scratch; a smarter initialization might reduce the burden.

Limitations and Future Work: Our current evaluation was limited by hardware (we only went up to 100 agents; larger instances may require more memory or time than available), and by the scope of scenarios (one map size, specific obstacle density). In future work, we plan to implement some of the discussed optimizations and test on a broader set of maps and task distributions. We also intend to compare CBS-TA's performance to other approaches, such as a two-phase method (first assign tasks, then plan paths decoupled) to see the trade-offs between optimality and speed. This would give context on how much overhead optimal task assignment introduces over simpler methods.

In summary, CBS-TA is effective for moderate-scale multi-agent pathfinding with tasks, but its performance deteriorates at larger scales due to conflict explosion and increased complexity. By incorporating heuristic guidance, smarter conflict resolution, memory-efficient design, and possibly parallel computation, we can hope to extend the range of problems that CBS-TA can solve in practice. Our analysis provides the motivation and quantitative basis for these improvements.

VII. Conclusion

CBS-TA extends the conflict-based search paradigm to handle task assignments optimally, making it a powerful algorithm for multi-agent systems that require both assignment and path planning. In this paper, we analyzed CBS-TA's performance in detail and found that while it performs admirably on smaller problems, it faces steep challenges as the number of agents grows. Our experimental results on a 32×32 grid show exponential growth in runtime and memory usage with agent count, and a corresponding explosion in the number of conflicts that must be resolved. Profiling on a smaller case revealed that CBS-TA is CPU-intensive and benefits from good instruction locality, though it suffers from numerous data cache misses due to complex memory access patterns.

These findings underscore the importance of integrating optimizations into CBS-TA. Potential improvements include using heuristics to guide the search (trading off a little optimality for huge gains in speed), prioritizing the resolution of conflicts that have broad impact, restructuring data for better cache performance, and leveraging multi-core processors to explore search branches in parallel. Implementing and evaluating these enhancements is an important direction for future work, as it can make optimal task-assignment planning feasible for larger-scale systems.

In real-world multi-robot applications, one often cares about getting good solutions quickly. CBS-TA currently guarantees optimal solutions but can become slow for large instances; with the discussed optimizations, we aim to strike a balance where it can return near-optimal solutions much faster, or handle more agents before running into performance limits. Our performance analysis serves as a benchmark and guide for researchers and engineers aiming to deploy CBS-TA or similar algorithms. By understanding where the time and memory are spent, one can make informed decisions on improving the algorithm or deciding when an alternative approach might be necessary.

In conclusion, CBS-TA remains a benchmark for optimal multi-agent pathfinding with task allocation, and with further refinements, it can be pushed to tackle increasingly complex scenarios. This work lays the groundwork by pinpointing its bottlenecks and suggesting concrete ways to overcome them, moving us closer to scalable multi-agent planning in the real world.

References

[1] G. Sharon, R. Stern, A. Felner, and N. Sturtevant, "Conflict-Based Search for Optimal Multi-Agent Pathfinding," *Artificial Intelligence*, vol. 219, pp. 40–66, 2015.

[2] W. Hönig, T. Kumar, L. Cohen, H. Ma, T. Uras, N. Koenig, G. S. Sukhatme, and S. Koenig, "Conflict-Based Search with Optimal Task Assignment," in *Proc. of IJCAI*, 2018.

VIII. Individual Contributions

Ahmad Sadiq:

- *Milestone 3:* Analyzed the CBS-TA code with Mingyang Li and developed the results.csv extraction script (capturing agent count, trial index, elapsed time, and conflict count). Generated the conflict-count vs. agents, memory-usage vs. agents, and runtime vs. agents figures. Authored the Milestone 3 LaTeX report.
- *Milestone 4:* Set up and ran Cachegrind on the CBS-TA executable for the 8×8 benchmark, captured instruction/data references and cache misses, and co-developed detailed visualizations of instruction cache, data cache, and LLC behavior. Wrote the Milestone 4 LaTeX report.
- *Final Report:* Integrated all prior results into the IEEEtran template, adding new background subsections (Robotics Settings, Hardware Trends, and Problem Formulation), the Automation Script and Sample Data Table, and the YAML Preprocessing and Profiling Setup details. Assembled the complete LaTeX for the final manuscript.

Mingyang Li: Ahmad and I ran the algorithm on my computer and discussed how to display the data based on different data types after collecting the data. Then Shuwan and I used Python to visualize the data. Then we analyze the possible reason for generates those graphs. I did the cachesgrind metrics. Afterwards, Shuwan and I organized the framework of the paper and analyzed and discussed each conclusion we obtained.

Shuwan Zhao: I worked with my teammate to visualize the data using Python. I also contributed to the analysis of the possible reasons behind the trends and patterns shown in the graphs. Additionally, I collaborated with my teammate to organize the framework of the paper, and we discussed and analyzed each of the conclusions we drew from our data.