MPMOS: Massively Parallel Multi-Objective Shortest Paths

Leo Gold
University of Connecticut
Storrs, CT USA
leo.gold@uconn.edu

David Sidoti
US Naval Research Laboratory
Monterey, CA USA
david.m.sidoti.civ@us.navy.mil

Krishna Pattipati
University of Connecticut
Storrs, CT USA
krishna.pattipati@uconn.edu

Omer Khan
University of Connecticut
Storrs, CT USA
khan@uconn.edu

Abstract

The Multi-Objective Shortest Path (MOS) problem finds a set of unique Pareto-optimal paths from a source to a destination in a multi-attribute graph. This NP-hard problem exhibits exponential complexity growth with increasing objectives, rendering exact solutions computationally intractable. This paper presents MPMOS, the first GPU-accelerated architecture for exact MOS computation, addressing three key challenges: irregular label processing, unpredictable memory requirements that exceed static limits, and maintaining correctness under massive parallelism. Our contributions include: (1) concurrent extraction and processing of lexicographically ordered labels, (2) work-aware label distribution, (3) label-level vectorization with objective-aware dominance and pruning checks, (4) smart compaction to eliminate fragmentation in data structures, and (5) dynamic memory allocation enabling scalability to an increasing number of objectives. Experimental evaluation on NVIDIA GH200 across real-world maritime vessel routing and road network graphs shows MPMOS achieves an order of magnitude geometric mean speedup over a CPU parallel MOS, and two orders of magnitude over a state-of-the-art sequential MOS algorithm with exact Pareto-optimality maintained. The dynamic memory allocator enables GPU acceleration for exact multi-objective optimization at unprecedented computational scales.

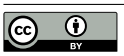
CCS Concepts

• **Computing methodologies** → **Massively parallel algorithms**;
• **Computer systems organization** → *Parallel architectures.*

Keywords

Multi-objective Optimization, Graph Processing, Shortest Path, Parallel Algorithm, GPU, Multicore, High-Performance Computing

ACM Reference Format:
Leo Gold, David Sidoti, Krishna Pattipati, and Omer Khan. 2026. MPMOS: Massively Parallel Multi-Objective Shortest Paths. In *2026 International Conference on Supercomputing (ICS '26), July 06–09, 2026, Belfast, United Kingdom.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3797905.3800547



ICS '26, Belfast, United Kingdom

ACM ISBN 979-8-4007-2522-7/2026/07
https://doi.org/10.1145/3797905.3800547

1 Introduction

Many real-world applications consider numerous conflicting and non-convex inputs to make optimal decisions. Multi-objective optimization considers multiple, often competing objectives simultaneously. However, the multi-objective shortest-path (MOS) problem is NP-hard, unlike its single-objective variant, which has polynomial time algorithms [28]. In a MOS problem, each graph edge is assigned a vector of non-negative costs such that each element corresponds to an objective. With competing objectives, no single path can optimize all objectives simultaneously. This leads to a rapid growth in complexity with an increasing number of objectives [4, 7, 10, 21, 28].

In MOS, unlike single-objective shortest-path, multiple Pareto-optimal (non-dominated) paths, also known as labels, may exist from the start node to any node in the graph. A label is not dominated if no single objective of the label can be improved without causing at least one of the other objectives to deteriorate in quality. When a new candidate label is discovered, it is compared against all other non-dominated labels at that node to check for *dominance*. If no such label is found that is better across all objectives (i.e., not dominated), the label is inserted into the node's frontier set. The frontier set is searched to check if the new label dominates existing labels, which are then removed. To enable efficient *dominance and pruning checks*, the frontier set is organized at the node granularity. There is no known static way to determine the number of Pareto-optimal labels that exist for each node in the graph. This leads to complex runtime-dependent operators and data structures in MOS. As the number of objectives increases, so does the complexity and variability in the number of dominance checks on MOS data structures. These operators and data structures have been a major focus of research in the MOS literature [12, 23–25]. Solving for the exact Pareto-optimal solutions is challenging for real-world graphs, and there is an urgent need to address the complexity [26].

Modern graph algorithms exploit hardware parallelism to accelerate performance [11, 22]. In 2025, OPMOS proposed to harness hardware parallelism in MOS using an ordered parallel and asynchronous execution model. However, even with multiple orders of magnitude performance gains, it is limited by the hardware-level parallelism in modern multi-core CPUs. OPMOS lacks support for accelerators (such as GPUs) that offer massive hardware parallelism. However, because hardware support for dynamic memory and coherence/consistency is limited, runtime-dependent, memory-bound operators and data structures are difficult to use efficiently. The

irregular data structures that support dominance/pruning operators require novel GPU methods to efficiently manage label processing.

GPU frameworks (such as Gunrock [34]) expose generic operators and data structures that apply to many existing graph problems. However, the irregular data structures and operators in MOS do not inherently map to the existing graph frameworks. There are three major challenges for an efficient GPU MOS implementation: ordering of rapidly increasing labels during runtime, high variability in runtime computation and data access requirements for dominance and pruning checks, and unpredictable storage requirements to maintain the frontier set. Ordering labels requires a priority queue (PQ) that is hard to realize on GPU systems. Graph frameworks opt for approximate PQ implementations to harness parallelism in GPUs. The design space ranges from an unordered queue to an exact PQ with tradeoffs between implementation complexity, parallelism, and work efficiency. The proposed massively parallel MPMOS uses a lexicographically sorted parallel-PQ to efficiently order labels in the frontier set. A system parameter allows concurrent extraction of many high-priority labels to saturate the parallel hardware of the GPU.

The work done per label processed is highly irregular, primarily due to the node-level variability in the number of labels checked during the dominance and pruning operations. This creates computational complexity and load imbalance problems that lead to serialization when multiple labels are processed in parallel. Even within a single dominance comparison between two labels, it is possible that not all objectives need to be compared to determine dominance. Beyond computation, random-access in-place deletes from the data structures are expensive. An invalidation-based approach does not require expensive data movements, but leaves behind "holes" in the data structures. These holes still need to be checked during dominance/pruning operations, and they continue to consume GPU memory. The variability in the number of labels per node creates challenges with the static mapping of memory on the GPU. Without dynamic memory support, large MOS problems cannot be allocated on even the largest GPUs. Fundamentally, tackling these sources of variability is a key challenge for mapping MOS to the GPU.

To address the variability in label processing, a work-stealing label distribution, as well as an objective-aware mapping of dominance and pruning operators, is proposed. A novel lightweight method of compacting valid entries removes holes within the per-node data structures. This adds computation overhead, but leads to avoiding unnecessary checks in the dominance and pruning operators. The compaction operator enables an additional feature where memory is dynamically allocated, enabling larger instances of MOS to execute to completion. This is crucial for the scalability of MOS on GPU systems. MPMOS combines the variability-aware label processing innovations to enable massively parallel MOS.

MPMOS is evaluated on NVIDIA GH200 using real-world maritime vessel routing and road-network graphs. It unlocks massive GPU parallelism and achieves 356× geometric mean speedup over sequential, and 37× speedup over the state-of-the-art CPU-parallel baseline. The proposed label ordered and asynchronous-parallel approach balances work efficiency and parallelism tradeoffs to produce the exact solutions that match sequential execution. The impressive performance contributions are largely attributed to the work-efficient label distributions in the GPU, objective-aware processing of dominance and pruning operators, and smart compaction of data structures, leading to an 11.3× speedup compared to when these optimizations are all turned off in MPMOS. Finally, MPMOS is shown to scale to an increasing number of objectives, where both computational and memory resources of the GPU are stressed. Using the proposed dynamic memory allocator for the underlying data structures, MPMOS is shown to achieve unprecedented scalability across multi-attribute graphs.

2 Related Work and Motivation

Computing the exact Pareto-optimal solution set for the MOS problem is computationally hard [4, 7, 10, 21, 28]. MOS literature addresses the computational complexity and proposes novel data structures and algorithmic optimizations based on the multiobjective extension of the A* search algorithm [18, 26, 31, 33]. A New Approach to Multi-Objective A* (NAMOA*) [18] and its variants EMOA* [25], NAMOA*-dr [23], LTMOA*, and LazyLTMOA* [12] ensure that an exact set of solution labels is computed from a source to a goal node. However, despite all algorithmic innovations, the core problem remains: the complexity of these *exact Pareto* approaches rises exponentially with the number of objectives.

To deal with this complexity, the MOS literature has explored a variety of approximate approaches. Genetic and evolutionary algorithms [2, 5, 14, 15, 36, 38] suffer from low explainability and accuracy of solutions with minimal computational efficiency gains. Approximate Pareto approaches reduce computational complexity through runtime state-space reductions. For example, an ϵ dominance operator [35] allows pruning of candidate solutions within an ϵ-bounded range. This introduces a tradeoff between solution accuracy and runtime efficiency [37]. As the ϵ-bound is increased, more candidate labels are pruned, and the runtime efficiency improves. However, Pareto-optimal labels are dropped from consideration, and the approximation method degrades solution quality and falters with explainability to the decision makers. In mission-critical environments, maintaining high trust and explainability is necessary, pointing back towards the computationally hard exact Pareto approaches.

Extracting parallelism through ordered graph processing is the cornerstone of accelerating modern graph algorithms [11, 22]. Researchers have explored parallelization of MOS, such as [29] that computes a single-objective shortest path for each objective and combines them for a single solution. Such inexact MOS problems are out of scope for MPMOS, which aims for high explainability through the computation of an exact Pareto-optimal solution set. Recently, limited parallelization has also been explored in the literature for the exact MOS problem at the objective [1] and operator [13] level. Sanders and Mandow [27] present a parallel bi-objective algorithm, relying on identifying multiple globally Pareto-optimal labels for extraction on each iteration, a task the authors assert is not possible for more than two objectives. OPMOS [9] introduces the first parallel single-instance exact Pareto algorithm, presenting a scalable label parallel approach that exploits hardware CPU parallelism to accelerate the MOS execution time. OPMOS identifies high variability in label processing as a key challenge to unlocking

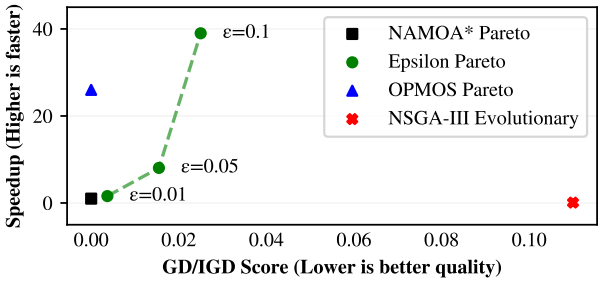


Figure 1: MOS Performance vs. Solution Quality Tradeoff. The solution quality score is measured using an average of the Generational Distance (GD) and the Inverse Generational Distance (IGD) [3]. GD measures the distance from each solution in the approximate Pareto front to the closest solution on the exact Pareto front, whereas IGD measures the distance from each solution in the exact Pareto front to the closest solution on the approximate Pareto front. Distance is the average normalized distance for each objective.

MOS parallelism, in addition to massive compute and memory scaling with increasing objectives. Despite these challenges, OPMOS shows high potential for label-level parallelism and highlights a lack of hardware parallelism on the CPU as the key performance limitation for further scaling. This suggests the hypothesis that exploiting the massive parallelism available on modern GPUs can deliver even further performance gains. However, there is no prior work on exploiting GPU parallelism in MOS.

The CPU parallel OPMOS bridges the complexity gap between exact Pareto approaches and approximate solutions. Figure 1 shows the time-to-solution and solution-accuracy tradeoff of various MOS algorithms, normalized to the sequential NAMOA* exact Pareto algorithm. The state-of-the-art NSGA-III [5, 15] evolutionary algorithm exhibits slow time-to-solutions while delivering the lowest quality solutions. The exact Pareto approach delivers exact solutions, but is still constrained by runtime complexity. The epsilon-based approximate Pareto approach trades performance efficiency with degraded solution quality, reducing explainability. The CPU-parallel acceleration in OPMOS delivers high efficiency with no compromise in solution quality. However, the runtime of OPMOS remains intractable in many practical settings, pointing towards massively parallel systems as the next logical step towards tackling the high complexity in MOS.

Parallel graph processing on modern GPUs has been extensively studied [34], and NVIDIA supports the cuGraph library for graph analytics [16]. However, no known work has extended support for the MOS problem on GPUs. With unbounded time and space complexity, it is challenging to map MOS to the existing graph frameworks. MOS requires unique operators and data structures to handle its high variability and dynamic runtime complexity. Once these operators and structures are discovered for GPU-parallel MOS, novel graph frameworks can be devised. However, the goal of this paper is to formulate the data structures and operators that efficiently map MOS onto the GPU systems. Furthermore, prior work does not explicitly identify the operators needed to address the challenges with multiple potentially non-optimal label extractions. MPMOS proposes the precise set of additional dominance and pruning checks to ensure the parallel algorithm converges on exact solutions, and aims to set the standard for parallel exact Pareto algorithms.

3 MOS Background and Challenges

Multiobjective optimization aims to simultaneously minimize (or maximize) a vector of objective functions. Generally, there is not a single solution that minimizes all objectives, especially when dealing with objective functions that lead to non-convex optimization. Instead, multiobjective optimization finds the non-dominated, or *Pareto-optimal* solutions, such that no solution improves one objective without worsening at least one other [7]. Formally, this can be stated as follows for d objectives, where $X \subseteq \mathbb{R}^n$ represents the feasible solution space, and $f_i(x)$ represents the objective function for the i-th objective.

$$\min_{x \in X} [f_1(x), \cdots, f_d(x)] \tag{1}$$

Given a weighted graph $G = (V, E, c)$ with a set of nodes V, edges E, and non-negative edge weights c, the shortest path problem computes the minimum-cost path from a source node v_s to a goal node v_g in the graph [32]. In a multi-objective setting, each edge $e \in E$ is given a non-negative cost vector $c(e)$ of length d objectives (constant length for each edge in a graph), with each element corresponding to an objective. When these objectives conflict, no single path can optimize all of them simultaneously. This requires the use of labels to track the candidate Pareto-optimal solutions, where a label $l = (v, \hat{g})$ represents a path from v_s to a specific node v with a given feasible solution cost vector $\hat{g}$ of length d [19, 25, 27]. The vector $\hat{g}$ denotes the path cost from v_s to v, calculated as the sum of the cost vectors $c(e)$ for all edges present on the path. For simplicity, we denote $v(l)$ to be the vertex and $\hat{g}(l)$ to be the cost vector contained in l. The number of Pareto-optimal solutions is *unbounded*, and each potential new path to a node introduces a new feasible solution. This results in a rapid, unpredictable growth in Pareto-optimal solution labels with the number of nodes and objectives in the graph. The NP-hard formulation of MOS uniquely differentiates it from traditional graph algorithms that guarantee polynomial bounds on their time and space complexity.

Since multiple objectives may compete, MOS introduces a dominance check such that given two labels l and l^* with d objectives, l dominates l^* (denoted $l \succeq l^*$) if and only if they share the same node ($v(l) = v(l^*)$) and $\hat{g}(l) \succeq \hat{g}(l^*)$ (i.e., $\hat{g}(l)[i] \leq \hat{g}(l^*)[i], \forall i \in 1, 2, ..., d$, and $\hat{g}(l)[i] < \hat{g}(l^*)[i], \exists i \in 1, 2, ..., d$). All non-dominated labels from v_s to the goal node v_g constitute the *Pareto-optimal* solution set. MOS aims to find a *cost-unique* Pareto-optimal solution set where no two paths in the subset have the same cost vector. As the number of objectives and nodes increases, so does the computational complexity and the number of Pareto-optimal paths [25].

To efficiently compute the Pareto-frontier, A*-style algorithms (like NAMOA* and its variants) compute *all* Pareto-optimal labels to the goal node, unlike the single-objective A*, which terminates once the first solution is found. These algorithms require the use of a **consistent heuristic** for the A* search to ensure that an exact

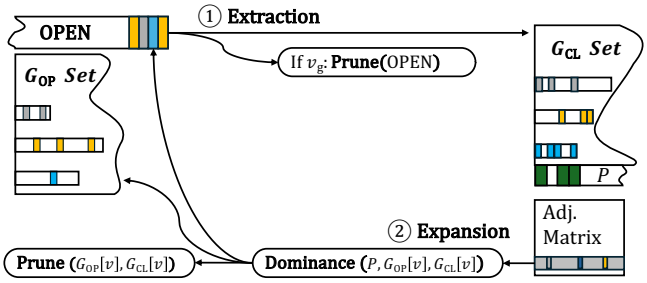


Figure 2: MOS algorithm dataflow with operators and data structures.

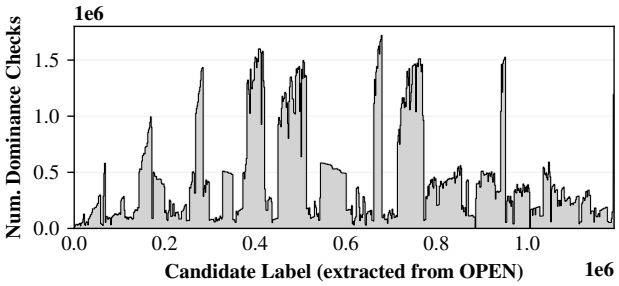


Figure 3: Variability in dominance checks per label processed for Graph TMP1 at 12 objectives.

set of solution labels is computed to the goal node. A heuristic vector $\hat{h}(v)$ is an *admissible* heuristic such that it dominates all Pareto-optimal solution labels from node v to the goal node [18]. One method to derive this heuristic is to compute the single-source shortest path (SSSP) from the goal node for each objective. The resulting shortest paths to all nodes in the graph represent the lower bound path costs, defined as $\hat{h}(v(l))$. A vector $\hat{f}(l)$ denotes the estimated lower-bound of the path cost from the start node to the goal node for a given label, calculated as $\hat{f}(l) = \hat{g}(l)+\hat{h}(v(l))$. To support the computations for a *cost-unique* Pareto-optimal solution set, MOS algorithms center around a set of key data structures and operators.

Frontier Sets: MOS operates on open (G_{OP}) and closed (G_{CL}) sets. The G_{CL} set contains all non-dominated Pareto-optimal labels organized on a per-node granularity. Each label in $G_{CL}(u)$ is a partial solution path from v_s to u. For example, $G_{CL}(v_g)$, denoted P, maintains the Pareto-optimal solutions to the goal node, v_g. The output of the MOS is the exact set of Pareto-optimal solutions in P. G_{OP} contains all labels that need to be settled into the Pareto-frontier. It is also organized at the node granularity for efficient processing.

OPEN mirrors all unsettled labels in G_{OP}, and is organized as a queue of labels prioritized by $\hat{f}(l)$ in increasing lexicographic order of objectives. This guarantees that a globally Pareto-optimal label is extracted and processed in each iteration (or frontier in traditional graph terminology). A priority queue (PQ) data structure is used to ensure that a label with the highest probability of remaining in the final solution is processed. This enables the optimal number of labels to be processed for algorithmic convergence.

NotDominated $(l, S, cost)$ operator compares the label l with labels in a given set S to verify if a label exists in S that dominates l. The *cost* specifies whether the path-cost ($\hat{g}$) or the heuristic-cost ($\hat{f}$) is used for the dominance checks. It returns *false* if l is dominated by a label in S, and returns *true* otherwise.

Prune $(S, l, cost)$ operator searches through all labels in a given set S, and removes all labels in S that are dominated by l using the path-cost ($\hat{g}$) or the heuristic-cost ($\hat{f}$).

Using these data structures and operators, MOS algorithms compute an exact Pareto-optimal solution set. In a single-objective shortest path, there is only one minimum solution label for each node in the graph, resulting in static storage requirements and constant-time comparisons. However, MOS does not guarantee a single solution path, as multiple non-dominated labels can exist from the start node to any other node in the graph. This places MOS algorithms into a unique class of graph processing as they execute an unpredictable number of labels for each node in the graph, and each label performs an unstructured and variable number of computations.

This paper uses OPMOS [9] as the baseline, which parallelizes the sequential NAMOA* algorithm. Figure 2 shows a high-level overview of the MOS operator and data flow in NAMOA*. In the extraction phase, a goal node label is extracted from the OPEN queue and inserted into $G_{CL}(u)$ and P. A goal node label also opportunistically prunes dominated labels from OPEN. The expansion phase generates candidate labels for the neighboring nodes. Each expanded label must not be dominated by an existing label in G_{OP} and G_{CL} for that node, and the solution set P, to be classified as a unique candidate Pareto-optimal label. This unique label is then inserted into OPEN. It must also perform a pruning check against all existing labels for that node in G_{OP} and G_{CL} to ensure labels dominated by this expanded label are removed. These phases are repeated (termed as iterations) until no labels exist in OPEN. At the end, P contains the final labels representing the Pareto-optimal solution paths (also referred to as exact solutions).

3.1 Challenges in MOS Unpredictability

Traditional graph problems have complexity measured in the number of nodes and edges. In MOS, a third dimension is added: the number of objectives. The runtime of MOS grows exponentially with objectives, characterized by a rapid increase in the number of labels processed. The computational complexity challenge exacerbates due to the high unpredictability in label processing. There are three key sources of unpredictability: the difference in execution flow between the expansion of goal and intermediate node labels, the difference in the number of neighbors per node (out-degree) during expansion, and the dominance and pruning operations. The dominance/pruning checks dominate the runtime, requiring tens to millions of comparisons to complete. These checks may exit early if a label is found that causes the checks to fail, introducing further variability in required computations.

OPMOS [9] quantified the MOS unpredictability and showed orders of magnitude differences in label latency, with the gap widening with increasing numbers of objectives. Figure 3 shows six orders

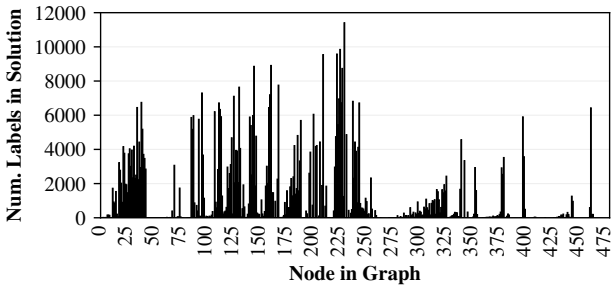


Figure 4: Variability in the number of labels per node for Graph TMP1 at 12 objectives.

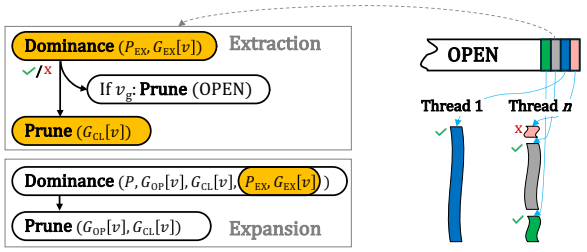


Figure 5: Parallel label dataflow and processing in MPMOS.

of magnitude variability in the number of dominance checks performed per label for a representative graph (TMP1 in Section 5 with 12 objectives). Beyond label variability, each node also maintains an unpredictable number of candidate labels. Figure 4 shows the number of labels per node generated at the end of NAMOA* for the representative graph. With high variability at the node level, each dominance and pruning check has a highly variable number of comparisons that must be performed. Adding together all of the sources of variability in MOS, there is no known way to statically predict the complexity of each label, potentially making efficient and balanced parallel distribution difficult. Storing these labels also leads to significant memory management challenges, especially on GPU systems that do not explicitly support dynamic memory management.

4 Massively Parallel MOS

This paper introduces the first massively parallel algorithm to address the unique multiobjective graph processing challenges and accelerate MOS problems on modern parallel throughput hardware. Extracting sufficient parallelism to utilize the underlying hardware is a key requirement for efficient parallel mapping. Recent CPU-parallel OPMOS [9] suggests that multiple labels can be extracted from the OPEN queue for parallel processing, where the extracted labels encompass an iteration of the algorithm. To expose the maximum available parallelism, the entire OPEN may be extracted on every iteration. However, the number of labels in OPEN is unpredictable and grows rapidly. Processing too many labels per iteration may create a significant amount of redundant labels that do not settle in the final frontier set. The resulting decrease in work efficiency is detrimental to runtime despite the increase in available parallelism. Thus, a system parameter, NUM_EXTRACT, is introduced to allow a controlled number of label extractions from the OPEN queue on every iteration. This enables a user-defined control over the expansion of work as the algorithm progresses, allowing sufficient parallelism with a bound on work efficiency by maintaining an order to label processing. However, not only must the right amount of labels be extracted, but also the right labels.

Sanders and Mayers [27] presented the first parallel bi-objective shortest-path algorithm using an exact globally Pareto-optimal priority queue. However, beyond two objectives, there is no known global priority order, and lexicographical ordering introduces redundant label computations. These non-Pareto-optimal labels must be removed through additional dominance and pruning checks to ensure the algorithm still converges on exact solutions. Optimal ordering for MOS remains an open research problem [26]. Following MOS literature, OPEN is implemented as a lexicographically ordered queue, while ensuring efficient algorithmic convergence.

Figure 5 shows the additional dominance and prune operators (highlighted boxes) that must be performed when multiple labels are concurrently extracted on an iteration. A label may dominate another extracted label. Left unchecked, this introduces duplicate and non-Pareto-optimal labels into G_{CL}. To address this, all labels being extracted are checked for dominance against one another to ensure that the label qualifies to be inserted into G_{CL} and expanded. These checks are performed not only against labels at the label's node, but also against labels at the goal node that are being considered for insertion into P in a given iteration. To accomplish the parallel dominance checks, the extracted labels from OPEN are inserted into additional per-node data structures, G_{EX} for the regular and P_{EX} for the goal node labels. The labels that pass this first check continue processing. An extracted label does not guarantee that it is Pareto-optimal for that node due to the lexicographical ordering used in parallel extractions. Therefore, when the Pareto-optimal label is extracted in a future iteration, it must perform a prune operator to make sure any labels in the G_{CL} set (at its node) are pruned if they are dominated by the extracted label. A new candidate label generated in the expansion phase may be dominated by the parallel extracted labels (in the G_{EX} and P_{EX} sets). Thus, dominance checks are added in the expansion phase to ensure that a newly generated non-Pareto-optimal label is not inserted in OPEN and the G_{OP} set. These additional checks ensure exact Pareto-optimal solutions, but add computational overhead that must be amortized by exploiting parallelism.

Due to high variability in label processing, statically distributing the extracted labels among parallel threads leads to load imbalance, where an extracted label may fail the dominance check on G_{EX}, while another candidate label performs multiple dominance and pruning operations. As shown in Figure 5, a work-stealing approach is proposed where each thread fetches a label on demand from OPEN. This allows threads that finish their work early to continue while long-running threads complete their assigned computations.

Algorithm 1 MPMOS

Input: Edge costs C, heuristics H, start v_s and goal v_g nodes

```
1: while OPEN not empty do
2:     Launch Extract Kernel
3:     Launch Expand Kernel
4:     P.end ← P.exp_end ; P_EX.end ← 0
5:     for all nodes i do G_OP(i).end ← G_OP(i).exp_end ; G_EX(i).end ← 0
6:     Launch Sort Kernel
7:     Launch Compact Kernel

8: function Extract Kernel
9:     OPEN.ext_end ← MIN (NUM_EXTRACT, OPEN.end)
10:    for i =WARP_ID to OPEN.ext_end by NUM_WARPS do
11:        l ← OPEN[i]
12:        if l.isInvalid then continue
13:        G_CL(v(l)).insAt(l, atomicAdd(end, 1))
14:        G_OP(v(l)).invalidate(l)
15:        if v(l) = v_g then P_EX.ins(l, atomicAdd(end, 1))
16:        else G_EX(v(l)).ins(l, atomicAdd(end, 1))

17: function Expand Kernel
18:    PruneOPEN(OPEN, P_EX, f̂)
19:    for i =WARP_ID to OPEN.ext_end by AtomicFetchNextOPEN( ) do
20:        l ← OPEN[i]
21:        if l.isInvalid then continue
22:        if NotDominated(l, P_EX, f̂) and NotDominated(l, G_EX(v(l)), ĝ) then
23:            if v(l) = v_g then
24:                Prune(P, l, f̂)
25:                P.insAt(l, atomicAdd(exp_end, 1)))
26:            else
27:                Prune(G_CL(v(l)), l, ĝ)
28:                for all v' ∈ GetNeighbors(v(l)) do
29:                    l' ← (v', ĝ(l) + ĉ(v(l), v')); parent(l' ← l)
30:                    f̂(l') ← ĝ(l') + ĥ(v(l'))
31:                    if NotDominated(l', P, f̂) and
32:                    NotDominated(l', G_CL(v(l')), ĝ) and
33:                    NotDominated(l', G_OP(v(l')), ĝ) and
34:                    NotDominated(l', G_EX(v(l')), ĝ) and
35:                    NotDominated(l', P_EX, f̂)) then
36:                        Prune(G_CL(v(l')), l', ĝ)
37:                        Prune(G_OP(v(l')), l', ĝ)
38:                        OPEN.insAt(l', atomicAdd(exp_end, 1))
39:                        G_OP(v(l')).insAt(l',atomicAdd(exp_end, 1))
40:        else
41:            G_CL(v(l)).invalidate(l)
```

4.1 Algorithmic Dataflow

The proposed massively parallel GPU algorithm for MOS is introduced using the NVIDIA terminology. However, the proposed algorithm can be deployed to other vectorized parallel hardware systems. The GPU-parallel pseudocode is described at the warp granularity in Algorithm 1. Each data structure maintains an *end* pointer, such that the valid region of labels lies between 0 and *end*. Efficient dynamic memory management is a challenge for MPMOS data structures, and the proposed methods are discussed later.

At the top of an iteration, it is assumed that the OPEN queue is sorted in priority order using the Sort Kernel that is discussed later. Within the Extract Kernel, up to NUM_EXTRACT labels are extracted from OPEN. The cutoff point for extraction is set using a pointer, OPEN.*ext_end* (line 9), and the extracted labels are distributed in a round-robin fashion, with each label being moved from G_{OP} to G_{CL} in parallel. Since multiple extracted labels may be concurrently inserted into G_{CL} at the same node, an atomic increment on the *end* pointer is used to allocate a new entry, and the label is subsequently inserted at that position (line 13). To avoid data duplication and maintain consistency among tracked open labels, a pointer association between OPEN and G_{OP} is maintained and kept consistent throughout the label processing. The association between OPEN and G_{OP} leads to entries needing to be removed from random locations in G_{OP} (line 14) as labels are extracted from OPEN. These deletions leave behind "holes" with random invalid entries spread throughout the G_{OP} arrays. The impact of these holes, and the proposed compaction strategy (Compact Kernel), is discussed later in the section. The extracted labels from OPEN are inserted into G_{EX} for the regular and P_{EX} for the goal node labels (lines 15-16). The Extract Kernel implicitly propagates the data consistently among all warps, and the Expand Kernel is invoked to process the extracted labels.

The expansion phase commences in each warp by first checking if the label being processed is dominated by other extracted labels in P_{EX} or G_{EX} (line 22). If the label is not dominated and at the goal node (line 23), then it first checks if it can prune out any current non-dominated labels in P (line 24), and is inserted into P (line 25). The NAMOA* algorithm performs an additional operator to aggressively prune labels in OPEN that are dominated by goal node labels. Since OPEN grows rapidly, the computational complexity of the PruneOPEN operator grows disproportionally large relative to the other operators. Therefore, MPMOS proposes to exploit the massive parallelism in OPEN to accelerate the PruneOPEN operator independent of the warp-level processing of the extracted labels (line 18). Parallel label extractions in MPMOS allow non-globally Pareto optimal labels to make it into G_{CL}. Before commencing the expansion phase for a regular label, G_{CL} is pruned (line 27). During regular label expansion, each neighbor is visited, and a new candidate label is generated (lines 28-30). If any label in the open set (G_{OP}) or the closed set (G_{CL} and P) dominates this candidate label, it is discarded (lines 31-33). Additionally, the extraction sets (G_{EX} and P_{EX}) are checked for dominance (lines 34-35). If not dominated, the label is inserted into OPEN and G_{OP} (lines 38-39), and is used to prune out labels it may dominate in G_{CL} and G_{OP} (lines 36-37). In the Expand Kernel, the dominance and pruning operators incur a varying number of label checks with complex control flow paths. Therefore, efficient parallelization of these operators is discussed later in this section.

During the expansion phase, newly generated labels are inserted into the open set (OPEN and G_{OP} at lines 36-37), and extracted goal node labels are settled and inserted into P (line 25). The *end* pointer is not modified to enable the dominance and pruning operators to execute concurrently with label insertions. However, a separate *exp_end* pointer on each structure is progressed to allocate new insertions. The Expand Kernel implicitly propagates data consistently among all warps, and all data structure pointers are reset before the next iteration (lines 4-5).

4.2 Label Ordering

The MOS literature implements consistent heuristic-driven algorithms that generate an exact set of Pareto-optimal solutions to a goal node. Prior works [26, 27] show that lexicographical ordering of objectives impacts the labels processed (work efficiency), implying there is not one exact optimal priority order. Objective ordering remains an open research problem in MOS literature. MPMOS implements a lexicographically ordered queue to enable parallel

Algorithm 2 Sort Kernel

1: $\text{OPEN}_{\text{next}}.end \leftarrow \text{OPEN}.exp\_end - \text{OPEN}.ext\_end$
2: $\text{OPEN}_{\text{next}}.exp\_end \leftarrow \text{OPEN}_{\text{next}}.end$
3: **if** $(\text{OPEN}.exp\_end - \text{OPEN}.ext\_end) >$ NUM_EXTRACT **then**
4: SORT(OPEN[end **to** exp_end])
5: $\text{OPEN}_{\text{next}} \leftarrow$ MERGE(OPEN[ext_end **to** end], OPEN[end **to** exp_end])
6: **else**
7: $\text{OPEN}_{\text{next}} \leftarrow$ OPEN[ext_end **to** exp_end]
8: SWAP($\text{OPEN}_{\text{next}}$, OPEN)

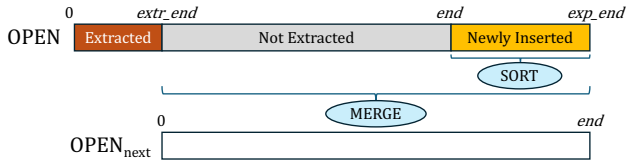


Figure 6: Double-buffered OPEN queue operator.

extraction of high-priority labels from OPEN, while ensuring the algorithm converges to produce exact solutions with high efficiency.

To prepare OPEN to extract high-priority labels for the next iteration, the non-extracted and newly-inserted labels are lexicographically sorted. This configuration is termed "global sorting" and implemented using state-of-the-art GPU-parallel sorting algorithms, such as Merge Sort using the NVIDIA CUB library. However, global sorting incurs runtime overhead that must be mitigated. We observe that non-extracted labels sorted in the previous iteration do not need to be sorted again. As described in Algorithm 2 and illustrated in Figure 6, a double-buffered strategy for OPEN is proposed in which the newly inserted labels are sorted and merged with the non-extracted labels. If all labels that are scheduled for sorting are extracted on the next iteration, then sorting is skipped for that iteration. The double buffering of OPEN ensures data consistency across iterations. A key advantage of the proposed approach is that it allows the system to dynamically size OPEN at an iteration granularity.

Despite the proposed runtime optimizations, sorting introduces overhead. However, label processing in MOS results in complex dominance and pruning operations that amortize this cost. What is more crucial is that high-priority labels are extracted first to keep the algorithm from processing labels that can be avoided. An important parameter for keeping labels ordered as close to high-priority as possible is NUM_EXTRACT, which must be set to trade off parallelism and work efficiency. At an extreme, when NUM_EXTRACT approaches infinity, all labels are extracted from OPEN for processing, and the OPEN operates in an unordered manner. This parameter space is empirically explored in the evaluation section.

4.3 Dominance and Pruning Operators

Label processing constitutes dominance and pruning checks against the open and closed sets that grow unpredictably and consume most of the runtime. Therefore, it is desirable not only to exploit maximum GPU parallelism at the vector hardware (warp) granularity, but also to make these checks efficient so they do not perform unnecessary computations. Algorithm 3 outlines the sequential

Algorithm 3 Dominance and Prune Operators

1: **function** NOTDOMINATED-SEQUENTIAL(l, $S[0, end]$, $cost$)
2: **for** $i = 0$ **to** S.end **do**
3: Dominated $\leftarrow$ `true`
4: **for** $d = 0$ **to** $\#Obj$ **do**
5: **if** $cost(l)[d] < cost(S[i])[d]$ **then**
6: Dominated $\leftarrow$ `false`
7: **break**
8: **if** Dominated **then return** `false`
9: **return** `true`

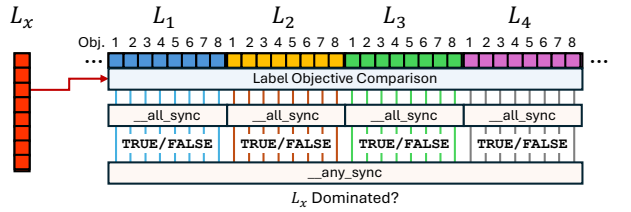
10: **function** NOTDOMINATED-OBJECTIVEVECTORIZED(l, $S[0, end]$, $cost$)
11: SW_ID $\leftarrow$ LANE_ID/# Lanes per subwarp
12: SW_LID $\leftarrow$ LANE_ID % # Lanes per subwarp
13: **for** i = SW_ID **to** S.end **by** NUM_SUBWARPS **do**
14: Dominated $\leftarrow$ `false`
15: **if** `__all_sync`(SW_MASK, $\hat{g}(l)$[SW_LID] $\geq \hat{g}(S[i])$[SW_LID]) **then**
16: Dominated $\leftarrow$ `true`
17: **if** `__any_sync`(Dominated) **then return** `false`
18: **return** `true`

19: **function** NOTDOMINATED-LABELVECTORIZED(l, $S[0, end]$, $cost$)
20: **for** i = LANE_ID **to** S.end **by** LANES_PER_WARP **do**
21: Dominated $\leftarrow$ `true`
22: **for** $d = 0$ **to** $\#Obj$ **do**
23: **if** $\hat{g}(l)[d] < \hat{g}(S[i])[d]$ **then**
24: Dominated $\leftarrow$ `false`
25: **break**
26: **if** `__any_sync`(Dominated) **then return** `false`
27: **return** `true`

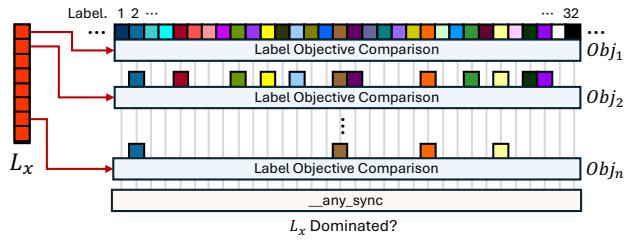
28: **function** PRUNE-LABELVECTORIZED($S[0, end]$, l, $cost$)
29: **for** i = LANE_ID **to** S.end **by** LANES_PER_WARP **do**
30: Dominated $\leftarrow$ `true`
31: **for** $d = 0$ **to** $\#Obj$ **do**
32: **if** $\hat{g}(S[i])[d] < \hat{g}(l)[d]$ **then**
33: Dominated $\leftarrow$ `false`
34: **break**
35: **if** Dominated **then** $S[i]$.isInvalid $\leftarrow$ `true`

NotDominated operator, where a label l is compared against labels in a set S. This function returns `false` if the label is dominated by any label in S for all of its objective costs. If no label dominates l, the operator returns `true`.

A natural way to parallelize is at the objective level, where each objective in the dominance comparison is compared on adjacent lanes of the warp. Furthermore, multiple labels from the set S are processed in parallel to achieve high lane utilization in a warp. The NOTDOMINATED-OBJECTIVEVECTORIZED function outlines the objective parallel NotDominated operator (lines 10-18). Multiple subwarps process independent labels from the set S in parallel. Each lane performs a dominance comparison to determine if the label l at a given objective is dominated by the label in set S. If all lanes of the subwarp determine (using the `__all_sync` synchronization primitive) that the label l is dominated by all objectives of that label (line 15), then it is concluded that the label in set S dominates label l (line 16). Since multiple labels from the set S are compared in parallel across the subwarps, if any of these labels (using the `__any_sync` synchronization primitive) dominate the label l, the NotDominated operator returns `false` (line 17). If no labels dominate l, the operator returns `true` (line 18). Figure 7a shows an example where a label L_x is compared against four labels in S with 8 objectives each to pack the 32 lanes of the warp. It determines if any of the four labels dominates L_x. This implementation utilizes vector-level parallelism



(a) Objective Vectorized



(b) Label Vectorized

Figure 7: Objective-level vectorization (a) performs dominance checks against all objectives, whereas label-level (b) may complete checks with fewer objectives.

in the GPU. Moreover, when the per-node data structures organize all objectives per label along the consecutive fastest-moving indices, the GPU maximizes coalesced memory accesses to global memory. However, once one objective of a given label is determined not to dominate L_x, the subsequent objective comparisons lead to unnecessary computations and memory accesses.

In the sequential setting, an early termination prevents unnecessary computations when an objective determines that a label will not be dominated (lines 6-7 of Alg. 3). To take advantage of work-efficient dominance checks and reduce the unnecessary memory operations, a label-level data structure and vectorization approach is proposed in the NotDominated-LabelVectorized function (lines 19-27) and illustrated with an example in Figure 7b. Each lane of a warp checks a label in S for the first objective to determine if L_x is dominated by that label's objective. If the objective is dominated, then the next objective is checked, and so on until either an objective does not dominate L_x, or L_x is determined to be dominated. When performing the dominance checks against a given objective n, all labels may determine that L_x is not dominated, and the remaining objective comparisons are skipped. If any lane determines a label to be dominated (using an __any_sync synchronization primitive), then the dominance check exits early. Otherwise, the next set of labels is mapped to the warp lanes, and the process repeats.

Naturally storing labels with all objectives for each label in contiguous memory negates some of the advantages of this method, leading to uncoalesced memory accesses. Instead, the data is organized such that all labels per objective are organized along the consecutive fastest-moving indices. This ensures the memory accesses are coalesced in the GPU memory system. A key advantage is that if a warp lane does not need to check more than the first few objectives, the remaining objectives do not need to be loaded from memory. This reduces stress on the memory bandwidth requirements to improve performance.

So far, this discussion has focused on the dominance operator. However, a similar label-level vectorization approach is proposed for the Prune operator (lines 28-35 of Alg. 3), inverting the comparison operands (line 32), and invalidating the label from S if it is dominated (line 35).

Algorithm 4 PruneOPEN Operator

```
1: function PruneOPEN-LabelVectorized(OPEN, P_EX, f̂)
2:     for i =WARP_ID + OPEN.ext_end to OPEN.end by NUM_WARPS do
3:         l ← OPEN[i]
4:         if l.isInvalid then continue
5:         if not NotDominated(l, P_EX, f̂) then
6:             l.isInvalid = true
```

4.3.1 PruneOPEN Operator. A key operator to maintain high work efficiency by curtailing label expansion in MOS is PruneOPEN. When a new goal-node label is extracted from OPEN, it likely dominates existing labels in OPEN. Pruning these labels prevents their future expansion. However, OPEN contains labels for all nodes and grows rapidly with increasingly large graphs and the number of objectives. To address this disproportionately high complexity, the PruneOPEN operator is parallelized across all warps in the Expand Kernel (line 18 of Alg. 1). The proposed PruneOPEN operator is shown in Algorithm 4. The labels in P_{EX} are mapped across the lanes in a warp, and a label in OPEN performs a NotDominated check to determine if it is dominated by a label in P_{EX}, and can be pruned. The labels in OPEN are distributed across warps to fully exploit GPU parallelism. More importantly, a label in OPEN performs a dominance check against P_{EX} labels one objective at a time. If any of these checks determine that a label in OPEN is dominated, then the remaining P_{EX} checks are avoided to save unnecessary computation and memory accesses. This is visualized by considering the example in Figure 7b, where the L_x is a label from OPEN, and the labels being compared against are P_{EX} labels.

4.4 Data Structure Compaction

During MOS execution, there are multiple points at which labels must be removed from data structures, such as during pruning and deletion operations on the open and closed sets (G_{OP}, G_{CL}, and P). To realize these deletions, there are two options: in-place removal, which requires fine-grain synchronizations, or label invalidation. MPMOS deletes labels via invalidation (lines 14 and 41 of Alg. 1), and in a later iteration, when these labels are extracted from OPEN, they are dropped (lines 12 and 21 of Alg. 1). These invalidated labels leave "holes" in the data structures that are skipped during dominance and pruning checks, but still require computations and memory. We propose a compaction operator that removes these holes to enable efficient data structures.

Compaction can be performed on every delete, which requires a low-level synchronization primitive for each label access. To avoid the additional synchronization latency, we propose to perform compaction for all data structures at the iteration granularity. This results in a highly parallel compact kernel, where parallelism is exploited across different data structures as well as their nodes. Due to its large relative size, P is assigned to a thread block, and each

Algorithm 5 Compact Kernel

```
1:  function Compact Kernel
2:      Compact(P)
3:      for nid = WARP_ID to #Nodes by NUM_WARPS do
4:          Compact(G_CL(nid))
5:          Compact(G_OP(nid))
6:  function Compact(S[0 to end])
7:      S.holes ← 0
8:      for i = THREAD_ID to S.end by NUM_THREADS do
9:          if S[i].isInvalid then atomicAdd(S.holes, 1)
10:     if WARP then __syncwarp() else __syncthreads()
11:     NumValidLabels ← S.end − S.holes
12:     for i = THREAD_ID to NumValidLabels by NUM_THREADS do
13:         while S[i].isInvalid do
14:             LastElement = atomicAdd(S.end, −1)
15:             if not S[LastElement].isInvalid then
16:                 S[i] ← S[LastElement]
17:     S.end ← NumValidLabels
```

per-node G_{OP} and G_{CL} data structure is assigned to a warp. The proposed Compact operator efficiently reads from global memory twice, and only writes to each hole once ($2N$ plus the number of *holes* global memory operations).

The novel compaction strategy is shown in Algorithm 5. First, a set S is traversed in parallel to count the number of holes in that set (lines 8-9). Using the number of holes and the size of the array, the number of valid labels are computed (line 11). The valid labels are distributed among the threads in a round-robin fashion (line 12). In cases where a thread does not point to a hole, the thread does nothing and moves on to the next valid label to process. Once a thread identifies a hole to fill, the *end* pointer is decremented until a valid entry is found. The thread moves the valid label to fill the hole (lines 13-16). The thread then moves on to the next valid label to process. Once all threads complete processing, all valid labels are contiguously stored in the front of the set S, and the end pointer is updated accordingly (line 17).

4.5 Dynamic Memory Management

Beyond computation, the unpredictable number of labels processed per node makes it difficult to statically allocate memory for the underlying data structures in MOS. Therefore, static memory allocation must size the label sets for each node to hold the maximum possible number of labels. On larger graphs with a high number of nodes and objectives, even modern GPUs with close to a hundred gigabytes run out of memory.

MPMOS proposes an efficient dynamic memory allocator for all label sets. A bucket is defined to be a unit of memory holding a user-defined number of labels. Small bucket sizes allow fine-grain memory management, whereas large sizes improve runtime efficiency by avoiding bucket management overhead. A bucket size of 1024 is quantitatively determined to balance the memory and runtime tradeoff. Figure 8 shows an illustration of one of the label sets during an iteration of MPMOS. Each per-node set is assigned a single bucket to start, with each bucket holding a pointer to the next bucket. In Figure 8-①, bucket A is attached to bucket B for a given node's label set. A pointer pointing to the next available bucket C is maintained to allow for efficient allocation. Each set's *end* pointer allows for efficient reads in dominance/pruning checks.

When a new label is inserted into a set, the end pointer is incremented to allocate a new entry. If this is the first label in a bucket,

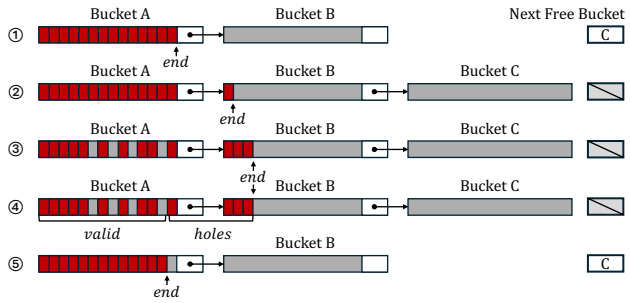


Figure 8: Life cycle of dynamic memory management in MPMOS.

then a new bucket is allocated (②). By allocating one bucket ahead, the chance that subsequent inserts by other threads need to wait for the first label to complete bucket allocation is reduced. To allocate a bucket, the free list is consulted to reserve a new bucket (bucket C). A reference from the previous bucket to the new bucket is made to allow easy traversal, similar to a linked list. Note that in the example, there are no more buckets available for allocation, so the next free bucket pointer is empty. For dominance and pruning operations, buckets are traversed until the end pointer, jumping to the next bucket by reference when needed.

At the end of an iteration, labels may have been inserted and removed from the buckets. This creates "holes" in the data structures (③). The proposed parallel compaction operator is used to remove any holes and ensure all valid data is stored contiguously in memory (④-⑤). If, after compaction, there is more than one empty bucket at the end of the set, these unused buckets are de-allocated and released back to the free-list (⑤). Once this process completes for all data structures, there are no holes in the system, allowing for efficient dominance checks and freeing up memory for the next iteration. With the proposed dynamic memory allocator, MOS problems that stress the GPU memory scale.

5 Methods

The NVIDIA GH200 [8] Superchip is used for evaluation. It integrates the Grace CPU with 72 Arm cores operating at 3.1 GHz, which are used for the evaluation of the sequential NAMOA* and the CPU-parallel OPMOS baselines. The GPU-parallel evaluation of MPMOS is done using the Hopper H100 GPU that supports 96 GB HBM3 memory, 4 TB/sec memory bandwidth, and 132 streaming multiprocessors (SMs) operating at 2 GHz frequency.

The NAMOA* [18] algorithm serves as the baseline for MOS. Both NAMOA* and the CPU-parallel OPMOS [9] are replicated, to the best of our ability, using C++ (compiled using GCC version 11.4.0 with all compiler optimizations enabled). In both cases, the OPEN priority queue is implemented using a `std::set`. The remaining data structures (P, G_{OP}, and G_{CL}) are implemented using standard C++ vectors to allow for dynamic memory management. In OPMOS, threads are spawned using the POSIX thread (`pthreads`) library, with the associated data structures in shared memory. As in OPMOS, runtime does not include initialization of the data structures (and threads), with timing measurements collected using the C++ `chrono`

Table 1: TMPLAR (TMP) and road-network (NY) graphs with number of objectives (O). Origin and destination are the location (TMP) and node IDs (NY). Hops-to-goal (HTG) represents the shortest path length from the origin to the destination.

Graph	Origin	Dest.	Nodes	Edges	O	HTG
TMP1	Roanoke Isl.	Bahamas	471	4394	12	7
TMP2	Alaska	San Diego	1610	10019	4	10
TMP3	Alaska	Seattle	461	2610	12	6
TMP4	Guam	Sasebo	201	2476	12	6
TMP5	Str. of Gibr.	Roanoke Isl.	778	7787	6	15
NY6	172882	189944	264346	730100	4	272
NY15	35170	237017	264346	730100	4	265
NY20	242644	163590	264346	730100	4	271
NY31	25610	143842	264346	730100	4	210
NY50	61414	50367	264346	730100	4	205

library. The number of labels extracted from the OPEN queue is collected using performance counters as a metric of work efficiency.

MPMOS is implemented in CUDA C++ using CUDA 12.9. The maximum number of concurrent warps is spawned, with 32 warps per block, and 2 blocks per SM for a total of 8448 warps. The GPU kernels from Algorithm 1 rely on implicit barriers for synchronizations. The labels are lexicographically ordered in OPEN using the proposed efficient sort-and-merge strategy. A global sorting is also implemented for comparison, where all non-extracted and newly inserted labels are sorted in OPEN. The dynamic memory allocator is necessary for all evaluated graphs to complete. To evaluate the efficacy of the proposed distribution of labels for processing, dominance and pruning operators, and compaction of pruned/deleted labels, MPMOS is compared against variants with each proposed optimization removed.

OPMOS evaluates parallel MOS performance using real-world graphs for maritime vessel routing generated by the Tool for Multi-objective Planning and Asset Routing (TMPLAR) [20, 30, 37]. TMPLAR supports a spatio-temporal setting and state-space reduction techniques to generate graphs with up to 12 objectives using a variety of dynamic weather and ship datasets. MPMOS uses the same graphs and objectives as evaluated in OPMOS, as shown in Table 1. The TMPLAR maritime graphs are shown with the prefix TMP. MOS algorithm literature also uses the DIMACS road network multi-attribute graphs [6]. MPMOS uses the New York (NY) graph with the same four objectives as evaluated in the MOS literature, i.e., travel distance, travel time, the number of edges, and the path length [24, 25]. The source and destination nodes for MOS are randomly selected from [17], representing variability in sequential MOS runtime. These graphs are shown in Table 1 with the prefix NY, and the sequence number corresponds to the instance in [17]. While there are implications with increasing graph sizes and degree, the primary cause of the exploding complexity of MOS is the number of objectives. All TMPLAR graphs are evaluated with up to 12 objectives to illustrate the computational and memory scalability of MPMOS. The TMP and NY graphs sufficiently stress the limits of MOS with variability in objectives, nodes, degree, and diameter (hop to goal-node).

The kernel completion time and breakdowns are used as the performance metric to evaluate MPMOS. The number of labels extracted from OPEN for processing is used as a metric of work efficiency. However, due to the high variability in label computations, the number of check operations performed is also established as a key metric for efficiency. This metric counts each time an objective is compared between two labels, including checks to determine if a label is valid. To gain further insights, GPU occupancy and the memory system are profiled using NVIDIA's Nsight Compute tool.

Table 2: Runtime of Sequential NAMOA*, OPMOS, and MPMOS, with the increase in labels extracted shown. Geometric means are shown for the TMPLAR (TMP) graphs, New York road networks (NY), and overall.

Graph	Runtime (s) & Speedup (over Seq.)			Labels Processed [mil.]	
	Seq.	OPMOS	MPMOS	Seq.	MPMOS
TMP1	251	9.96 (25×)	0.16 (1571×)	0.563	1.086 (1.93×)
TMP2	8368	419.9 (20×)	2.92 (2866×)	8.51	18.66 (2.19×)
TMP3	0.081	0.01 (8.1×)	0.006 (13.4×)	0.009	0.059 (6.88×)
TMP4	4.04	0.12 (33×)	0.02 (248×)	0.050	0.146 (2.94×)
TMP5	1778	83.56 (21×)	0.51 (3501×)	1.286	3.941 (3.06×)
GMean	65.73	3.36 (20×)	0.118 (555×)	0.305	0.93 (3.04×)
NY6	3781	1124 (3.4×)	5.19 (720×)	23.9	36.61 (1.53×)
NY15	2.94	0.80 (3.7×)	0.134 (21.9×)	0.743	6.614 (8.90×)
NY20	399.1	128.1 (3.1×)	1.155 (346×)	4.185	14.03 (3.35×)
NY31	3041	226.4 (13×)	3.346 (909×)	16.39	24.51 (1.49×)
NY50	39.26	8.51 (4.6×)	0.316 (124×)	2.904	10.51 (3.62×)
GMean	221.2	46.7 (4.7×)	0.968 (229×)	5.13	15.43 (3.01×)
GMean	120.5	12.53 (10×)	0.339 (356×)	1.25	3.788 (3.03×)

6 Evaluation

The performance of sequential NAMOA*, OPMOS, and MPMOS is shown in Table 2. The sequential execution times range from milliseconds to seconds to hours, with OPMOS reducing that range to milliseconds to minutes. MPMOS further reduces that range to seconds to milliseconds, executing with 356× geometric mean speedup over sequential NAMOA* and 37× speedup over the CPU parallel OPMOS. However, these speedups cover a wide range, from as low as 13.4× on TMP3 to as high as 3501× on TMP5. The primary cause for the lack of speedups on certain graphs (e.g., TMP3 and NY15) is the exploitable parallelism in the graph. The graphs with more labels processed (and therefore more available parallelism) have the highest speedups. Further contributing to the performance variation is work efficiency: the graphs with less additional work performed over sequential observe higher speedups.

To understand where the performance gains originate, the following MPMOS characteristics are evaluated: 1) finding the right balance between parallelism and ordering of labels, 2) the work-efficiency benefits of label-level vectorization for the dominance and pruning operators, 3) an analysis of the compaction of data structures, and 4) the impact of increasing objectives on scalability.

6.1 Label Ordering and Parallelism

Figure 9 shows the runtime distribution of MPMOS, where 3% is spent in label extraction and 81% in label processing/expansion. The label sorting and compaction kernels take 6% and 10%, respectively. The sorting of labels is triggered on average 53% (and up to 85%) of iterations. The 10% sorting overhead is attributed to the proposed sorting algorithm, where only the newly inserted labels are sorted and merged with the existing labels in OPEN. This overhead diminishes to under 2% for graphs with increasing search space (labels processed). However, the worst-case graph (NY15) spends 30% of

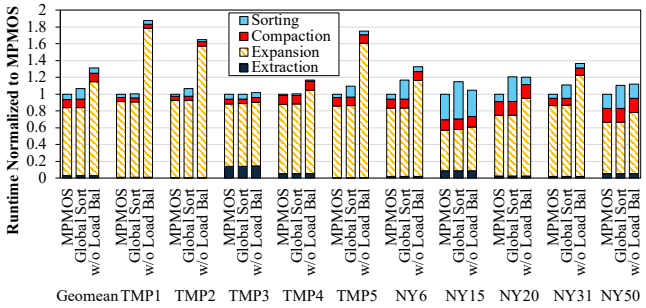


Figure 9: Normalized runtime distributions for MPMOS with and without full global sorting and the load balancer.

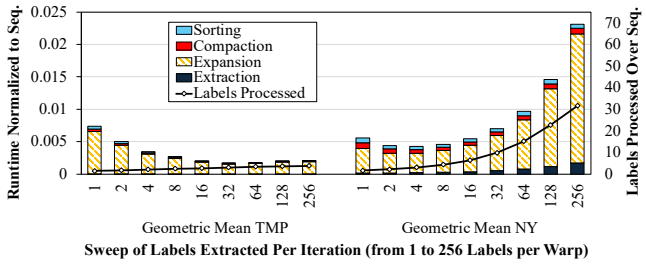


Figure 10: Geometric mean runtime distributions of MPMOS with labels extracted per iteration swept for TMP and NY graphs from 1 label per warp to 32.

the runtime in sorting. This is justified since NY15 processes 212× more labels (data not shown) when no sorting is applied, leading to a 25× loss in performance compared to the proposed ordering of labels. These observations underpin the importance of sorting labels in OPEN, even if it comes with overhead. To evaluate the efficacy of the proposed intelligent sorting of labels, Figure 9 shows a variant where all non-extracted and newly inserted labels (the entire OPEN) are sorted. Across all graphs, an average of 2× reduction in sorting overhead is observed with the merge strategy when compared to global sorting.

Although label ordering plays a key role in unlocking work efficiency, MPMOS relies on extracting massive label parallelism for performance. This tradeoff is controlled with the system parameter, NUM_EXTRACT, which is set as a factor of the number of warps spawned in the GPU. Extracting too few labels does not unleash enough parallelism, and performance suffers. However, extracting too many labels introduces wasteful computations, adversely impacting performance. Figure 10 shows the geometric mean performance for both TMP and NY graphs under varying NUM_EXTRACT settings. The NUM_EXTRACT is correlated with the hops to the goal node (HTG), where an increasing HTG causes more labels to be explored in the search space, increasing the number of redundant and unwanted labels being processed. This work-efficiency tradeoff with label parallelism is used to tune NUM_EXTRACT when significant variability in HTG is observed between NY and TMP graphs.

The low HTG in TMP graphs results in minimal work efficiency impacts, with only 2.5× work performed with 256 labels extracted from OPEN per warp. This allows TMP to unlock more parallelism, and while the runtime does begin to increase after 32 extracted labels per warp, the impact is minimal. The NY graphs observe a significant impact on work efficiency, with over 30× labels processed at 256 labels per warp. This rapid increase in work efficiency prevents scaling with increased parallelism, with an optimal setting at 4 labels per warp. While an optimal point could be found for both the TMP and NY graphs, the big gap in graph characteristics prevents a unifying NUM_EXTRACT from being set in the system. The prior analysis and the remainder of this section assume NUM_EXTRACT is set with 32 labels per warp for TMP graphs, and 4 labels per warp for NY graphs. This is a key knob for MPMOS runtime performance, and future work can explore static and dynamic methods for determining the optimal NUM_EXTRACT settings.

Table 3: Number of objectives compared per-label in Obj-Vectorized (Obj), label-vectorized (Obj/Label), and the resulting theoretical work efficiency (W.E.) for TMP and NY graphs.

Graph	Obj	Obj/Label	W.E.	Graph	Obj	Obj/Label	W.E.
TMP1	12	1.9	6.3×	**NY6**	4	2.2	1.8×
TMP2	4	1.8	2.2×	**NY15**	4	1.8	2.2×
TMP3	12	2.6	4.6×	**NY20**	4	2.2	1.8×
TMP4	12	2.5	5.1×	**NY31**	4	2.3	1.7×
TMP5	6	1.6	3.8×	**NY50**	4	1.8	2.2×

MPMOS unlocks massive parallelism with high-priority labels being processed in each iteration. However, label processing observes high variability across the control and dataflow of the underlying dominance and pruning operators. As observed in Figure 9, the label processing/expansion consumes the majority of runtime. MPMOS proposes a dynamic label-aware work-stealing approach that operates at the warp granularity. A static round-robin label distribution is compared to evaluate the efficacy of the proposed scheduler, where an average 38% performance improvement is observed in the label processing component. For small graphs (such as NY15 and TMP3), where the labels per iteration do not fully saturate the massive parallelism of the GPU, minimal performance gains are observed as expected. However, for large graphs (such as NY6 and TMP2), up to 2× performance gains are observed when using the work-stealing approach.

6.2 Efficient Dominance and Pruning Operators

MPMOS label vectorization packs all lanes of a warp with labels, where each label independently performs the dominance and pruning checks per objective. If the label determines its dominance condition earlier in the objectives, the operator quits early. However, the baseline objective vectorization maps all objectives of a label on the warp lanes, while packing multiple labels per warp to saturate hardware resources. If not all objectives participate in evaluating the check outcome, the baseline strategy performs wasteful computations. Table 3 shows the average number of participating objectives per label, which ranges from 1.6 to 2.6 objectives. When compared to the total number of objectives per graph, the work efficiency improvements from the proposed label vectorization range from 1.7× in the NY31 graph with 4 objectives to 6.3× in the TMP1 graph with 12 objectives. In general, as the number of objectives

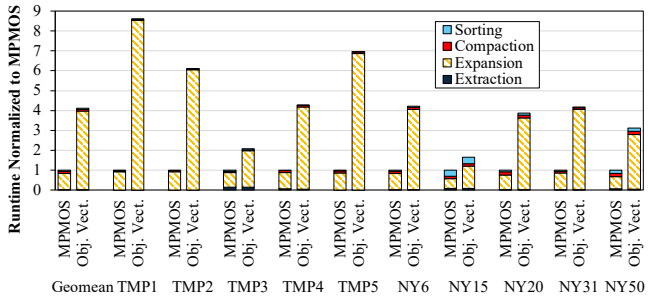


Figure 11: Normalized runtime distributions for MPMOS with label-vectorization and objective-vectorization.

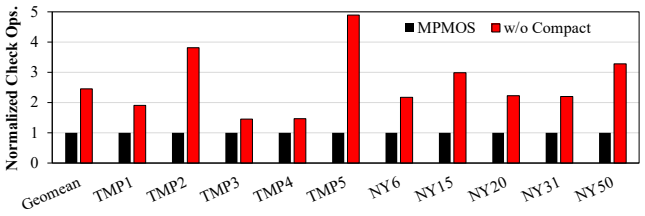


Figure 12: Number of label check operations performed with and without smart label compaction normalized to MPMOS.

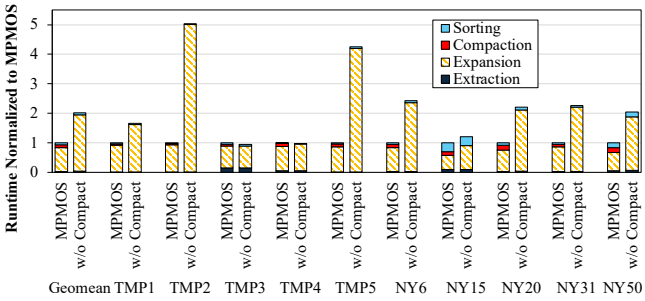


Figure 13: Normalized runtime distributions for MPMOS with and without smart label compaction.

increases, so does the work efficiency potential with the proposed label vectorization for the dominance and pruning operators.

Figure 11 shows that the performance gains from label versus objective vectorization not only depend on the improvements in work efficiency, but also on the effective achieved parallelism at runtime and the resulting stress on the memory and compute hardware resources from the wasteful checks. In cases where graphs are small and/or many iterations of label processing are performed with plentiful available warps, the unnecessary objective checks are performed in parallel without significantly impacting performance. This behavior is observed in TMP3 and TMP4 graphs with a significantly lower number of labels processed compared to other TMP graphs (cf. Table 2). Although the potential work efficiency gains are 4.6× and 5.1× for these graphs, the observed performance gains are 2× and 4× for TMP3 and TMP4, respectively. A similar trend is observed in the NY15 graph, which processes the smallest number of labels among the NY graphs.

When graphs process more labels at runtime, the parallelism from the available warps saturates. In these graphs, when wasteful labels are processed, fewer labels effectively complete their work in parallel. Moreover, the unnecessary checks incur memory accesses, computations, and warp-level synchronizations that further diminish the performance when baseline objective vectorization is used. Consequently, graphs with a large number of active warps achieve more performance gains than work efficiency alone with label vectorization. This trend is observed in all graphs except for the small TMP3, TMP4, and NY15 graphs. For example, in TMP2, more than half of the objectives are processed unnecessarily in the baseline approach. However, the gains from unlocking more parallelism and efficient memory and compute accesses from the label vectorization achieve 6× improvement in performance. Overall, the label vectorization for dominance and pruning checks achieves a geometric mean 4.1× speedup over the baseline objective vectorization.

6.3 Smart Data Structure Compaction

When labels are removed (during extraction) or pruned (during expansion) from the per-node data structures, MPMOS invalidates them during label processing. At the end of an iteration, these invalidated labels can remain in the data structures or be removed using the proposed parallel compaction kernel. As discussed earlier, the compaction kernel incurs a 10% average overhead in MPMOS runtime. This is justified if compacting labels in the per-node arrays results in improved work efficiency. Figure 12 shows the number of dominance check operations performed as a measure of compaction efficiency. In small graphs, such as TMP3, processing fewer labels leads to fewer opportunities for holes to be generated, reducing the necessity and impact of data compaction. However, on average a 2.5× compaction efficiency is observed. This results in a geometric mean 2× performance boost with up to 5× in TMP2, as observed in Figure 13. This result neatly justifies that the performance gains from the proposed compaction strategy result in far more gains than the 10% loss of performance from the compaction overhead.

6.4 Scalability at Increasing Objectives

A key aspect of the MOS problem is the rapid increase in labels and runtime as the number of objectives increases. So far, the TMP2 and TMP5 graphs are evaluated using 4 and 6 objectives, respectively. However, these graphs can potentially execute with up to the 12 available objectives, given their runtimes remain tractable and/or the open and closed label sets and OPEN do not run out of the available GPU memory during processing.

Figure 14 shows the runtime (left) and memory requirements (right) of the TMP5 graph at an increasing number of objectives. The runtime rises rapidly as the number of objectives increases from 2 to 12. However, all objectives finish in approximately 37 minutes using the H100 GPU. To evaluate the efficacy of the proposed dynamic memory allocator, a static allocation policy is introduced. The static is an oracle policy that first tracks the number of labels per node and per data structure to determine the maximum memory that needs to be allocated for each node in a given set. Once the maximum number of buckets is allocated, the graph is executed, and the consumed memory requirement is plotted. The static memory requirement

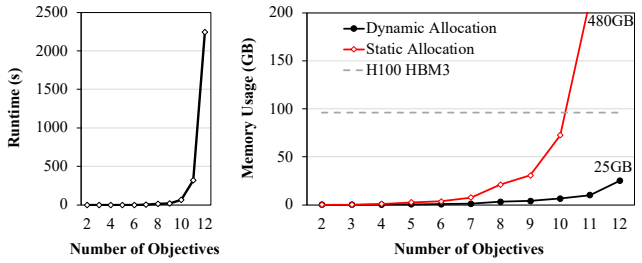


Figure 14: Runtime (left) and memory consumption (right) for TMP5 graph at increasing objectives.

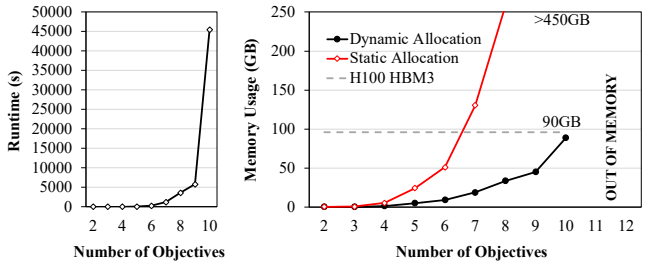


Figure 15: Runtime (left) and memory consumption (right) for TMP2 graph at increasing objectives.

is observed to increase exponentially, and after 10 objectives, it surpasses the available 96 *GB* GPU memory. However, the proposed dynamic allocator determines the need for bucket allocation at a per-node granularity and completes the TMP5 graph for all 12 objectives. The memory consumption is observed to grow rapidly from 0.76 *GB* at 6 objectives to 25 *GB* at 12 objectives; however, the proposed dynamic allocator is 19× memory efficient than the static allocator, highlighting its efficacy.

Figure 15 shows the same scaling study for the TMP2 graph. Only ten objectives finish execution for this graph; at eleven objectives, the GPU runs out of memory even when using the proposed dynamic memory allocator. However, at ten objectives, the runtime explodes to over twelve hours, increasing from just seconds at four objectives. Additionally, although the dynamic allocator cannot complete at eleven objectives, the static allocation policy prevents memory scaling beyond six objectives. Using the proposed dynamic memory allocator, an additional four objectives are unlocked, operating at over 8× memory efficiency compared to the static allocator, again highlighting its efficacy.

7 Future Directions

MPMOS faces three key outstanding challenges: 1) the explosion in memory requirements to track the open and closed sets and the OPEN queue, 2) the highly variable and irregular dominance and pruning operators on unstructured data, and 3) managing the right tradeoff between parallelism and ordered processing of labels. All operators and memory/compute requirements lie in the expansion phase, lending label expansion to be the ideal target for future performance gains.

As shown in Figures 14 and 15, memory grows by two orders of magnitude while runtime increases by five to six orders of magnitude from two to twelve objectives. Thus, computational intractability remains the key bottleneck. For example, TMP2 is computationally intractable at ten objectives (taking over twelve hours to converge), while still fitting within the memory bounds of a single GPU. To further exploit GPU parallelism, architectural hardware-centric optimizations can be explored. The throughput of label processing can be improved through methods that make use of specialized computational units like Tensor cores, programmable indexing hardware accelerators, independent thread scheduling, asynchronous memory operations, and efficient use of shared memory for data reuse and overlap. Inevitably, as we have already shown, the memory of a single GPU will be exhausted. At this point, alternative systems need to be explored. One possibility is to explore GPU-CPU heterogeneity. For example, utilizing the GH200 NVLink connection between the H100 GPU and the host Grace CPU, providing over 500 GB/s bandwidth connection to the 480 *GB* CPU memory (over 5× the GPU memory). The key question is: can the extra or extended memory be exploited to unlock the performance potential we have shown with the GPU memory, while enabling the processing of larger problem sizes?

Beyond a single superchip, multi-GPU and disaggregated GPUs with expanded memory can also be explored to accelerate MPMOS. However, these systems come at the cost of power efficiency and communication overheads. Additionally, a hardware accelerator approach is envisioned, where custom vector hardware devices tuned for the unique dominance and pruning operators have access to high-bandwidth, high-capacity 3D-stacked memory. These are some avenues of research we plan to explore in future work.

8 Conclusion

This work introduces MPMOS, a novel GPU-accelerated algorithm that navigates the high variability in computation and memory access that has traditionally prevented efficient, massively parallel solutions for the exact multi-objective shortest-path problem. To address the computational and memory complexity, dynamic parallel label distribution, work-efficient dominance and pruning operators, and runtime compaction of underlying data structures are proposed. A novel dynamic memory allocator is proposed to harness the unpredictable storage requirements for the data structures at runtime. Evaluation using the NVIDIA GH200 Superchip shows that MPMOS achieves an order of magnitude speedup over state-of-the-art CPU-parallel MOS and two orders of magnitude speedup over sequential MOS across real-world maritime vessel routing and road network graphs. MPMOS enables GPU acceleration for exact multi-objective optimization at unprecedented computational scales.

Acknowledgments

This research is supported by the U.S. Government under a grant by the U.S. Naval Research Laboratory. This material is also based upon work supported by the U.S. National Science Foundation under Grant No. 2429516. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation and the Naval Research Laboratory.

References

[1] Saman Ahmadi, Nathan R. Sturtevant, Andrea Raith, Daniel Harabor, and Mahdi Jalili. 2025. Parallelizing multi-objective A* search. In *Proceedings of the Thirty-Fifth International Conference on Automated Planning and Scheduling* (Melbourne, Victoria, Australia) *(ICAPS '25)*. AAAI Press, Article 15, 9 pages. doi:10.1609/icaps.v35i1.36109

[2] Faez Ahmed and Kalyanmoy Deb. 2011. Multi-objective path planning using spline representation. In *2011 IEEE International Conference on Robotics and Biomimetics*. 1047–1052. doi:10.1109/ROBIO.2011.6181426

[3] Adam Bienkowski, Lingyi Zhang, David Sidoti, and Krishna R. Pattipati. 2026. Intelligent Search and Multiobjective Path Planning With Application to Ship Routing. *IEEE Journal of Oceanic Engineering* 51, 1 (2026), 370–388. doi:10.1109/JOE.2025.3604173

[4] Thomas Breugem, Twan Dollevoet, and Wilco van den Heuvel. 2017. Analysis of FPTASes for the multi-objective shortest path problem. *Computers & Operations Research* 78 (2017), 44–58. doi:10.1016/j.cor.2016.06.022

[5] Kalyanmoy Deb and Himanshu Jain. 2014. An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints. *IEEE Transactions on Evolutionary Computation* 18, 4 (2014), 577–601. doi:10.1109/TEVC.2013.2281535

[6] Camil Demetrescu, Andrew Goldberg, and David Johnson (Eds.). 2009. *The shortest path problem.* American Mathematical Society, Providence, RI.

[7] Matthias Ehrgott. 2005. Multicriteria Optimization. In *Multicriteria Optimization.*

[8] Jonathon Evans, Michael Andersch, Vikram Sethi, Gonzalo Brito, and Vishal Mehta. 2022. NVIDIA Grace Hopper Superchip Architecture In-Depth. https://developer.nvidia.com/blog/nvidia-grace-hopper-superchip-architecture-in-depth/

[9] Leo Gold, Adam Bienkowski, David Sidoti, Krishna Pattipati, and Omer Khan. 2025. OPMOS: Ordered Parallel Algorithm for Multi-Objective Shortest-Paths. In *Proceedings of the 2025 ACM International Conference on Supercomputing (ICS '25)* (Salt Lake City, UT, USA) *(ICS '25)*. Association for Computing Machinery, New York, NY, USA, 16 pages. doi:10.1145/3721145.3725781

[10] Pierre Hansen. 1980. Bicriterion Path Problems. In *Multiple Criteria Decision Making Theory and Application*, Günter Fandel and Tomas Gal (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 109–127.

[11] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. 2011. Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming* (San Antonio, TX, USA) *(PPoPP '11)*. Association for Computing Machinery, New York, NY, USA, 3–12. doi:10.1145/1941553.1941557

[12] Carlos Hernández, William Yeoh, Jorge A. Baier, Ariel Felner, Oren Salzman, Han Zhang, Shao-Hung Chan, and Sven Koenig. 2023. Multi-objective Search via Lazy and Efficient Dominance Checks. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI-23*, Edith Elkind (Ed.). International Joint Conferences on Artificial Intelligence Organization, 7223–7230. doi:10.24963/ijcai.2023/850 Main Track.

[13] Carlos Hernández Ulloa, Han Zhang, Sven Koenig, Ariel Felner, and Oren Salzman. 2024. Efficient Set Dominance Checks in Multi-Objective Shortest-Path Algorithms via Vectorized Operations. *Proceedings of the International Symposium on Combinatorial Search* 17, 1 (Jun. 2024), 208–212. doi:10.1609/socs.v17i1.31560

[14] J. Horn, N. Nafpliotis, and D.E. Goldberg. 1994. A niched Pareto genetic algorithm for multiobjective optimization. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence.* 82–87 vol.1. doi:10.1109/ICEC.1994.350037

[15] Himanshu Jain and Kalyanmoy Deb. 2014. An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point Based Nondominated Sorting Approach, Part II: Handling Constraints and Extending to an Adaptive Approach. *IEEE Transactions on Evolutionary Computation* 18, 4 (Aug 2014), 602–622. doi:10.1109/TEVC.2013.2281534

[16] Seunghwa Kang, Chuck Hastings, Joe Eaton, and Brad Rees. 2023. cuGraph C++ primitives: vertex/edge-centric building blocks for parallel graph computing. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 226–229. doi:10.1109/IPDPSW59300.2023.00045

[17] E. Machuca and L. Mandow. 2012. Multiobjective heuristic search in road maps. *Expert Systems with Applications* 39, 7 (2012), 6435–6445. doi:10.1016/j.eswa.2011.12.022

[18] Lawrence Mandow and José. Luis Pérez De La Cruz. 2008. Multiobjective A* search with consistent heuristics. *J. ACM* 57, 5, Article 27 (June 2008), 25 pages. doi:10.1145/1754399.1754400

[19] Ernesto Queirós Vieira Martins. 1984. On a special class of bicriterion path problems. *European Journal of Operational Research* 17, 1 (1984), 85–94. doi:10.1016/0377-2217(84)90011-0

[20] Manisha Mishra, David Sidoti, Gopi Vinod Avvari, Pujitha Mannaru, Diego Fernando Martínez Ayala, Krishna R. Pattipati, and David L. Kleinman. 2017. A Context-Driven Framework for Proactive Decision Support With Applications. *IEEE Access* 5 (2017), 12475–12495. doi:10.1109/ACCESS.2017.2707091

[21] C.H. Papadimitriou and M. Yannakakis. 2000. On the approximability of trade-offs and optimal access of Web sources. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*. 86–92. doi:10.1109/SFCS.2000.892068

[22] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 12–25. doi:10.1145/1993498.1993501

[23] Francisco-Javier Pulido, Lawrence Mandow, and José-Luis Pérez de-la Cruz. 2015. Dimensionality reduction in multiobjective shortest path search. *Computers & Operations Research* 64 (2015), 60–70. doi:10.1016/j.cor.2015.05.007

[24] Zhongqiang Ren, Carlos Hernández, Maxim Likhachev, Ariel Felner, Sven Koenig, Oren Salzman, Sivakumar Rathinam, and Howie Choset. 2025. EMOA*: A framework for search-based multi-objective path planning. *Artificial Intelligence* 339 (2025), 104260. doi:10.1016/j.artint.2024.104260

[25] Zhongqiang Ren, Richard Zhan, Sivakumar Rathinam, Maxim Likhachev, and Howie Choset. 2022. Enhanced Multi-Objective A* Using Balanced Binary Search Trees. *Proceedings of the International Symposium on Combinatorial Search* 15, 1 (July 2022), 162–170. doi:10.1609/socs.v15i1.21764

[26] Oren Salzman, Ariel Felner, Carlos Hernández, Han Zhang, Shao-Hung Chan, and Sven Koenig. 2023. Heuristic-Search Approaches for the Multi-Objective Shortest-Path Problem: Progress and Research Opportunities. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI-23*, Edith Elkind (Ed.). International Joint Conferences on Artificial Intelligence Organization, 6759–6768. doi:10.24963/ijcai.2023/757 Survey Track.

[27] Peter Sanders and Lawrence Mandow. 2013. Parallel Label-Setting Multi-objective Shortest Path Search. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 215–224. doi:10.1109/IPDPS.2013.89

[28] Paolo Serafini. 1987. Some Considerations about Computational Complexity for Multi Objective Combinatorial Problems. In *Recent Advances and Historical Development of Vector Optimization*, Johannes Jahn and Werner Krabs (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 222–232.

[29] S. M. Shovan, Arindam Khanda, and Sajal K. Das. 2025. Parallel Multi Objective Shortest Path Update Algorithm in Large Dynamic Networks. *IEEE Transactions on Parallel and Distributed Systems* 36, 5 (2025), 932–944. doi:10.1109/TPDS.2025.3536357

[30] David Sidoti, Gopi Vinod Avvari, Manisha Mishra, Lingyi Zhang, Bala Kishore Nadella, James E. Peak, James A. Hansen, and Krishna R. Pattipati. 2017. A Multiobjective Path-Planning Algorithm With Time Windows for Asset Routing in a Dynamic Weather-Impacted Environment. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 47, 12 (2017), 3256–3271. doi:10.1109/TSMC.2016.2573271

[31] Bradley S. Stewart and Chelsea C. White. 1991. Multiobjective A*. *J. ACM* 38, 4 (Oct. 1991), 775–814. doi:10.1145/115234.115368

[32] Robert Endre Tarjan. 1983. *Data structures and network algorithms.* Society for Industrial and Applied Mathematics, USA.

[33] Carlos Hernández Ulloa, William Yeoh, Jorge A. Baier, Han Zhang, Luis Suazo, and Sven Koenig. 2020. A Simple and Fast Bi-Objective Search Algorithm. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS)*. AAAI Press, 143–151.

[34] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. 2017. Gunrock: GPU Graph Analytics. *ACM Trans. Parallel Comput.* 4, 1, Article 3 (Aug. 2017), 49 pages. doi:10.1145/3108140

[35] Arthur Warburton. 1987. Approximation of Pareto Optima in Multiple-Objective, Shortest-Path Problems. *Oper. Res.* 35, 1 (Feb. 1987), 70–79.

[36] Yuan Yao, Zhe Peng, and Bin Xiao. 2018. Parallel Hyper-Heuristic Algorithm for Multi-Objective Route Planning in a Smart City. *IEEE Transactions on Vehicular Technology* 67, 11 (2018), 10307–10318. doi:10.1109/TVT.2018.2868942

[37] Lingyi Zhang, Adam Bienkowski, Matthew Macesker, Krishna R. Pattipati, David Sidoti, and James A. Hansen. 2021. Many-Objective Maritime Path Planning for Dynamic and Uncertain Environments. In *2021 IEEE Aerospace Conference (50100)*. 1–10. doi:10.1109/AERO50100.2021.9438262

[38] E. Zitzler and L. Thiele. 1999. Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. *IEEE Transactions on Evolutionary Computation* 3, 4 (1999), 257–271. doi:10.1109/4235.797969