# Comparison of Sampling v Parallelized Grid Planners in High DOF Environments

Jacob Chisholm
*Electrical & Computer Engineering*
*Queen's University*
Kingston Ontario, Canada
21jc138@queensu.ca

***Abstract*—This work evaluates the performance and scalability of grid-based and sampling-based motion planning algorithms in high-dimensional configuration spaces. Breadth-First Search (BFS) and A* are implemented using a shared, dimension-agnostic, MPI-parallel grid-search framework, while Fast Marching Tree (FMT*) and Batch Informed Trees (BIT*) are evaluated using implementations from OMPL. All planners are benchmarked in a consistent, procedurally generated environment across dimensions ranging from 2D to 10D, using identical collision checking and cost models. Results show that A* consistently outperforms all other planners in planning time and reliability across tested dimensions, while BIT* produces competitive early solutions and, in very high dimensions, higher-quality final paths. BFS becomes infeasible beyond low-dimensional spaces due to state-space explosion. Analysis of parallel scaling demonstrates that the hash-based MPI partitioning strategy provides effective workload balance, though communication overhead limits scalability at higher core counts. Collision checking costs are comparable across planners for a given dimension, with grid-based planners benefiting from distributed collision evaluation. These results highlight the tradeoffs between informed grid search and sampling-based planning in high-dimensional environments and demonstrate the effectiveness of hash-based parallelization for scalable motion planning.**

***Index Terms*—HPC, Supercomputing, MPI, BIT*, FMT*, BFS, A*, Robotics, High-DOF, Grid-search, Random Sampling**

## I. Introduction

Optimal motion planning for robotic manipulators involves finding the lowest cost feasible path between a start and goal configuration. In high degree of freedom (DOF) environments, this problem presents a significant challenge commonly referred to as the *curse of dimensionality*. As the number of dimensions in the problem grow linearly, the size of the configuration space grows exponentially, rendering exhaustive search methods computationally infeasible, even with aggressive pruning or heuristics.

Therefore, to address this challenge, algorithms often switch from a fixed, grid-based search to a sampling based technique. Rather than explicitly enumerating the entire search space, these algorithms operate on randomly sampled configurations and incrementally construct graphs that approximate the underlying connectivity of the free space. Algorithms such as the Rapidly-exploring Random Tree (RRT), Probabilistic Road-Map (PRM*), Fast Marching Tree (FMT*) [1] and Batch Informed Tree (BIT*) [2] are all examples of sampling based approaches used to reduce the computational complexity of exploring high-DOF spaces.

However, while random sapling algorithms reduce computational complexity, they often sacrifice or compromise on key algorithmic guarantees, relying on probabilistic definitions for optimality and completeness [3]. Moreover, the performance of sampling based methods is deeply dependent on the sampling method, connection radius, and collision checking resolution.

This inherent trade-off often forces practitioners to accept suboptimal solutions, or endure excessive computation time to achieve convergence in high-DOF spaces. Therefore, the objective of this work is twofold. First, we systematically analyze the impact of dimensionality on both sampling and grid-search planners in controlled, reproducible environments ranging from 2 to 10 dimensions. Second, we explore the extent to which High Performance Computing (HPC) and large scale parallelization can enhance the speed and accuracy of grid-based planners in high-DOF spaces.

The primary contributions of this paper are as follows:

1) A minimal, deterministic, and procedurally generated benchmark environment suitable for high-dimensional motion planning
2) A direct comparison of two sampling-based and two grid-based planners across a range of dimensions
3) Strong and weak scaling analysis of grid-based planners in high-dimensional configuration spaces
4) An evaluation of parallel graph search as a means to mitigate dimensionality-induced computational complexity

## II. Problem Formulation

### A. Planning Objective

The objective of this work is to compute a collision-free path between a specified start state $x_s$ and goal state $x_g$ in a $d$-dimensional configuration space. The path is required to minimize a cumulative cost defined as the sum of Euclidean distances between successive states along the path.

A solution is considered **valid** if it satisfies the following conditions:

1) The path connects the start and goal states through a sequence of intermediate configurations.
2) The configurations along the path lie within the free configuration space and do not intersect any obstacles.

3) The total path cost is finite and compared using the Euclidean metric in $\Re^d$.

For grid-based planners, paths are represented as sequences of adjacent lattice states, while sampling-based planners produce piece-wise-linear paths connecting sampled configurations. In both cases, cost evaluation and collision validation are performed using a shared environment and collision checking model to ensure comparability.

### B. Configuration Space Representation

The configuration space is defined as a real-valued vector space

$$X = \{x \in \Re^d | x_i \in [0, 100)\}$$

corresponding to a $d$-dimensional hypercube. Experiments are conducted for $d \in \{2, 4, 6, 8, 10\}$ to evaluate the impact of dimensionality on planner performance.

For grid-based planners, the configuration space is uniformly discretized along each dimension into 100 equally sized intervals, henceforth called voxels. This results in a discretized search space containing $100^d$ vertices. Adjacency relationships are defined using local unit moves in the discretized configuration space. From any state, neighbouring states are generated by applying increments along either one or two distinct coordinate axes, subject to boundary and collision constraints. Diagonal moves are permitted only when all intermediate sates along the move are collision free. This allows full diagonal connectivity in two-dimensional spaces and partial diagonal connectivity in higher dimensions. Under this model, each state has at most $2d + 2d(d-1)$ neighbours, corresponding to axis-aligned moves and pairwise diagonal moves, respectively. All grid-based planners evaluated in this work operate on the same adjacency structure, ensuring performance differences arise solely from search strategy rather than graph connectivity.

Sampling-based planners operate on the continuous configuration space, but are subject to the same bounds and collision checking model. This unified representation ensures all planners are evaluated within identical geometric constraints.

## III. Environment & Collision Model

### A. Robot & Search Space Abstraction

The robot is modeled as a single point within a $d$-dimensional configuration space with no kinodynamic or holonomic constraints imposed. Planning is performed entirely in configuration space, and feasibility is determined solely by collision constraints. This abstraction isolates the effects of dimensionality and environment structure from robot-specific dynamics, allowing a direct comparison between planning algorithms.

Experiments are conducted for configuration spaces of dimensionality $d \in \{2, 4, 6, 8, 10\}$. In all cases, the robot occupies a single configuration $x \in \Re^d$ within the bounded search space defined in Section II-B.

### B. Start & Goal State Selection

The nominal start and goal configurations are fixed at

$$x_s = (0, ..., 0), x_g = (99, ..., 99).$$

To ensure that both states lie within free space, each configuration is iteratively adjusted along each dimension until a collision free state is reached. This process incrementally advances the configuration by one voxel at a time along individual axes and terminates once a valid state is found.

Due to a structural property of the obstacle generation process, the free configuration space is fully connected. As a result, a feasible path between the start and goal states is guaranteed to exist for all tested environments.

### C. Obstacle Generation

Initial experiments using explicit hyper-spherical and hypercubic obstacles revealed significant computational overhead during collision checking, particularly as dimensionality increased. Evaluating configurations against large numbers of explicit geometric primitives was found to scale poorly and dominate planning runtime in high-dimensional spaces.

To address this limitation, a procedural obstacle generation method was developed, henceforth refereed to as the Geometric Obstacle Generator (GOG). Rather than explicitly storing obstacle geometry, the GOG determines obstacle membership implicitly using a deterministic, hash-based function applied independently along each coordinate axis (Algorithm 1).

**Algorithm 1** Deterministic Obstacle Hash

**Require:** Point $x \in \Re^d, x = (x_0, ..., x_{d-1})$,

**Require:** Hash seeds $h = (h_0, ..., h_{d-1})$,

**Require:** Frequency Bits $b_p, b_v$

**Require:** Collision Threshold Value $c_v$

$s \leftarrow 0$
**for** $i \in$ range(0, $d-1$) **do** ▷ *For all axis*
  $v_i \leftarrow x[i] \oplus h[i]$ ▷ *Randomize the value with XOR*
  $p \leftarrow x[i] >> b_p$
  $v_i \leftarrow (p + v_i) >> b_v$ ▷ *Further Randomization*
  $s \leftarrow (v_i \ \texttt{\&} \ \texttt{0x03}) + s$
**return** $s \geq c_v$

For a given configuration $x \in \Re^d$, a small hash function is applied to each coordinate, producing a two-bit output per axis ($v_i$). The outputs are then summed and compared against the fixed threshold $c_v$. Configurations whose summed value exceeds this threshold are classified as invalid. Obstacle density and effective obstacle size are controlled through a combination of this threshold and the selective bits $b_p$, and $b_v$.

This approach enabled collision checking with a time complexity linear in the number of dimensions, $O(d)$, requiring only 4 bitwise and 2 addition operations per dimension. No explicit obstacle data structures are stored in memory, allowing all tested environments to be represented in less

than a single cache line worth of memory on a modern CPU. As a result, collision checking incurs minimal memory traffic and is bounded primarily by speed of arithmetic and bitwise operations.

### D. Collision Checking

Collision checking is performed using the same procedural model as obstacle generation, ensuring consistency across all planners. A configuration is deemed valid if and only if it satisfies the hash based threshold condition described above. This collision checking mechanism is shared by both grid-based and sampling-based planners to ensure a fair comparison.

Sampling based planners perform additional internal path validation to verify that local connections between sampled states are collision free. However, all validity checks ultimately rely on the same underlying collision model.

## IV. Algorithms

This section describes the four motion planning algorithms evaluated in this work: two sampling-based planners, Fast Marching Tree (FMT*) and Batch Informed Trees (BIT*), and two grid-based planners, Breadth-First Search (BFS) and A*. The planners are grouped into sampling-based and grid-based approaches, and further classified as uninformed or informed based on whether a heuristic is used to guide the search.

### A. Sampling-Based Planners

Sampling based planners explore the configuration space by generating a finite set of samples and constructing a graph or tree that approximates the connectivity of the free space. In this work, all sampling-based planners are evaluated using their reference implementations provided by the Open Motion Planning Library (OMPL). These methods are generally more scalable in high-dimensional environments but do not guarantee completeness for a fixed number of samples.

*1) Fast Marching Tree (FMT*):* Fast Marching Tree (FMT*) is an uninformed sampling-based planner that incrementally constructs a tree over a fixed set of randomly generated samples. The algorithm operates by expanding a wavefront outwards from the start configuration, connecting samples in order of increasing cost-to-come. A sample is added to the tree only if a collision free connection exists and yields the lowest observed cost.

Unlike the grid-based breadth-first search, FMT* does not explicitly enumerate all reachable states. Instead, its performance is highly dependent on the number and distribution of sampled points. As a result, FMT* cannot determine whether a solution exists if no valid path is found within the sampled set.

FMT* offers favorable asymptotic optimality properties as the number of samples increases, but its fixed-sample nature imposes several limitations. Because of search methodology, the samples must be generated prior, and additional samples cannot be introduced if the initial set proves insufficient. Therefore, to maintain a consistent exploration effort across increasing dimensionality, the number of samples used by FMT* was chosen as an explicit function of the state space dimensions. Specifically, the number of samples $N(d)$ is defined in Equation 1.

$$N(d) = 2000 + 1500 * (\text{STATESPACE DIMS} - 2)^2 \quad (1)$$

This scaling was selected to reflect a practical trade-off between coverage and computational cost, rather than attempting to approximate asymptotic optimality guarantees.

*2) Batch Informed Trees (BIT*):* Batch Informed Trees (BIT*) is an informed sampling based planner that combines ideas from heuristic graph search and sampling based planning. BIT* incrementally constructs a tree over batches of randomly sampled configurations, using an admissible heuristic to guide exploration toward regions of the space that are likely to improve the current solution.

The planner operates by alternating between sampling batches of configurations and performing a graph search over the implicit random geometric graph induced by those samples. Only nodes and edges that can potentially improve the current best solution are considered. These nodes are processed in order of solution quality, with collision checking being performed lazily, significantly reducing the number of collision checks and graph expansions.

This informed exploration strategy makes BIT* conceptually closer to A* than uninformed planners such as FMT*. As better solutions are discovered, the heuristic increasingly restricts exploration to a subset of the configuration space, enabling more efficient use of computational resources in high dimensions.

### B. Grid-Based Planners

Both grid-based planners, Breadth-First Search and A*, are implemented using a shared, dimension-agnostic graph search framework. The framework is designed to scale with high-dimensional spaces while avoiding the memory explosion typically associated with explicit grid representations.

The discretized configuration space is treated as an implicit graph, where vertices correspond to grid states and are generated on-the-fly during search. For any given state, neighboring states are produced by applying $\pm 1$ offsets along either one or two distinct coordinate axes (Algorithm 2).

This adjacency rule enables partial diagonal connectivity in higher dimensions while allowing the graph to be traversed without constructing an explicit grid, significantly reducing both the memory footprint and bandwidth requirements.

To enable scalable parallel execution utilizing MPI, the grid based planners employ a SplitMix64-style hash function (Algorithm 3) to produce a 64-bit identifier for each configuration state.

The resulting identifier is used for two purposes. First, it serves as the key for a process-local hash map that records visited states, eliminating the need to allocate a dense $100^d$ occupancy grid. Second, the hash value is used to determine state ownership by mapping it to an MPI rank, with each process responsible for expanding and storing only the states assigned to its partition.

**Algorithm 2** Implicit Graph Traversal

**Require:** Starting point $x \in \Re^d, x = (x_0, ..., x_{d-1})$,
**for** $i \in$ range(0, $d-1$) **do** ▷ *First Axis Offset*
  **for** $j \in \{-1, 1\}$ **do**
    $s_1 \leftarrow x$
    $s_1[i] \leftarrow s_1[i] + j$
    **if** IsValid($s_1$) **then**
      SendVisit($s_1$, OWNER($s_1$))
    **for** $k \in$ range(i+1, $d-1$) **do** ▷ *Second Axis Offset*
      **for** $l \in \{-1, 1\}$ **do**
        $s_2 \leftarrow s_1$
        $s_2[k] \leftarrow s_2[k] + l$
        **if** IsValid($s_2$) **then**
          SendVisit($s_2$, OWNER($s_2$))

**Algorithm 3** SplitMix64-style Hash

**Require:** Point $x \in \Re^d, x = (x_0, ..., x_{d-1})$,
$p \leftarrow$ AsUint64($x$)
$p \leftarrow (p \oplus (p >> 30)) \times$ `0xbf58476d1ce4e5b9ULL`
$p \leftarrow (p \oplus (p >> 27)) \times$ `0x94d049bb133111ebULL`
$p \leftarrow p \oplus (p >> 31)$
**return** $p$

This hash-based ownership scheme ensures that each state is expanded by exactly one MPI rank, balancing the load across processes, and minimizing inter-process communication. Frontier states whose ownership lies on a remote rank are communicated asynchronously via visit handlers.

Within this shared grid-search framework, individual planners are defined solely by their frontier prioritization and visit handling logic. Breadth-First Search (BFS) and A* therefore share identical state representations, adjacency generation, collision checking, hash-based ownership, and parallelization strategies. The only substantive difference between the two algorithms lies in how newly discovered states are prioritized and inserted into the search frontier. This separation allows the impact of heuristic guidance to be evaluated independently of implementation or parallelization effects.

*1) Breadth-First Search:* Bread-first Search (BFS) is an uninformed graph based planner (Algorithm 4) that explores configuration space by expanding states in increasing order of graph distance from the start. The BFS visit handler is shown in Algorithm 5. Within the shared grid-search framework described in Section IV-B, BFS assigns equal priority to all frontier states, resulting in a uniform wavefront expansion across the grid. When a solution exists, BFS guarantees the shortest cost path will be found.

Conceptually, BFS exhibits a wavefront expansion pattern similar to that of FMT*, but differs fundamentally in that it operates over an explicit discretization of the configuration space rather than a randomly sampled subset. As a result, BFS can determine whether a solution exists, at the cost of exploring a rapidly growing number of states. Therefore, BFS is primarily evaluated in lower-dimensional environments and serves as a baseline for comparison.

**Algorithm 4** Breadth-First Search

**Require:** Start configuration $x_s$, Target configuration $x_g$
**if** OWNER($x_s$) **then**
  QueuePush($s_1, x_s$)
  local_work = 1
**else**
  $s_1 \leftarrow []$
  local_work = 0
$s_2 \leftarrow []$
global_work $\leftarrow 1$
**while** global_work $> 0$ **do**
  **while** local_work $> 0$ **do**
    $x \leftarrow$ QueuePop($s_1$)
    TraverseNeighbours($x$)
  global_work $\leftarrow$ local_work $\leftarrow$ Length($s_2$)
  MPI_AllSum(global_work)
  MPI_Synchronize(MPI_COMM_WORLD)
  $s_1 \leftrightarrow s_2$
**return** Path($h, x_s, x_g$)

**Algorithm 5** Breadth-First Search Visit Handler

**Require:** Visited Nodes Hash Map $h$

**Require:** Processing Queue $s_2$

**Require:** Newly Visited Node $x$
**if** NOT OWNER($x$) **then**
  **return**
**if** NOT Visited($x$) **then**
  Insert($h, x$)
  QueuePush($s_2, x$)

*2) A*:* A* is an informed grid-based planner that guides exploration using a heuristic estimate of the remaining cost to the goal. In this work, A* uses the same adjacency generation, cost model, visited-state tracking, and hash-parallelization strategy as BFS, but differs in how newly discovered states are prioritized for expansion (Algorithm 6). Additionally, while BFS performs process synchronization upon the conclusion of every frontier, A* performs synchronization after a fixed number of states have been expanded.

To efficiently support a heuristic-driven exploration while minimizing heap reordering overhead, A* employs two heap-based priority queues. The first queue is used as a fast track queue, containing only states whose estimated total cost indicates a high likelihood of improving the current best solution. The second queue serves as a reserve queue for any states that do not meet the criterion to be added to the fast track queue.

During search, states from the fast-track queue are expanded preferentially. The reserve queue is processed only when the fast track queue becomes empty before a synchronization period is reached. This design enables informed exploration without excess heap reordering, ensuring that the algorithm

**Algorithm 6** A* Search

**Require:** Start configuration $x_s$, Target configuration $x_g$

**if** OWNER($x_s$) **then**
    QUEUEPUSH($s_1, x_s$)
    local_work = 1
**else**
    $s_1 \leftarrow []$
    local_work = 0
$s_2 \leftarrow []$
global_work $\leftarrow 1$
**while** global_work $> 0$ **do**
    **for** iteration $<$ N_ITERATIONS **do**
        **if** LENGTH($s_1$) $> 0$ **then**
            $x \leftarrow$ QUEUEPOP($s_1$)
        **else**
            $x \leftarrow$ QUEUEPOP($s_2$)
        TRAVERSENEIGHBOURS($x$)
    global_work $\leftarrow$ local_work $\leftarrow$ LENGTH($s_2$) + LENGTH($s_1$)
    MPI_ALLSUM(global_work)
    MPI_SYNCHRONIZE(MPI_COMM_WORLD)
**return** PATH($h, x_s, x_g$)

degrades gracefully towards uninformed search behavior when the heuristic become ineffective.

The A* visit handling logic, including the dual-queue insertion policy, is outlined in Algorithm 7. Aside from this prioritization mechanism, A* relies entirely on the shared grid-search framework described in Section IV-B.

**Algorithm 7** A* Search Visit Handler

**Require:** Visited Nodes Hash Map $h$

**Require:** Processing Queues $s_1, s_2$

**Require:** Newly Visited Node $x$

**if** NOT OWNER($x_s$) **then**
    **return**
**if** $x$.cost $<$ COST($h, x$) **then**
    **if** $x$.cost $<$ QUEUEPEEK($s_1$).cost **then**
        QUEUEPUSH($s_1, x$)
    **else**
        QUEUEPUSH($s_2, x$)
**if** NOT VISITED($x$) **then**
    INSERT($h, x$)

## V. EXPERIMENTAL SETUP

This section describes the benchmarking methodology used to evaluate all motion planning algorithms considered in this work. The experimental design ensures that performance differences arise from algorithmic behavior rather than environmental, hardware, or implementation artifacts.

### A. Benchmarking Environment

All planners are evaluated using environments generated by the Geometric Obstacle Generator (GOG) with a fixed hash seed across all experiments. For any given dimensionality $d$, the same seed produces a deterministic and reproducible obstacle configuration.

As the dimensionality changes, the planning environment changes **only** as a consequence of increasing the dimensionality of the configuration space, not due to randomization. That is, each $d$-dimensional benchmark represents a deterministic extension of the same underlying underlying obstacle generation process, ensuring consistency across dimensions while preserving reproducability.

This design ensures that:

1) Comparisons between planners at a fixed dimensionality are performed on identical environments
2) Comparisons across dimensionalities reflect the impact of increased degrees of freedom rather than changes in obstacle randomness

No environment specific tuning or reseeding is performed when transitioning between environments; the same generator and seed are used for all benchmarks.

### B. Hardware Platform

All experiments are conducted across two compute nodes equipped with dual AMD EPYC 7532 (Zen 2) processors. For parallel planners, processor counts are explicitly controlled and reported. No other user workloads were present during benchmarking.

### C. Timing and Metrics

Only planning time is considered in all reported results. Initialization, including memory allocation, and teardown costs are exluded to focus solely on algorithmic performance during search.

The following metrics are collected for each trial:

- Time to solution
- Solution path length, measured as the Euclidean length of the returned path ($l_p$)
- Solution Quality (defined in Equation 2)
- Number of collision checking calls

$$q = \frac{l_p}{\sqrt{\sum_{i=0}^{d} (x_g[i] - x_s[i])^2}} \tag{2}$$

These metrics allow evaluation of computational speed, solution quality and collision checking impact.

### D. Sampling-Based Planners (OMPL)

All sampling-based planners are implemented using the Open Motion Planning Library. For each planner and problem configuration:

- 100 independent trials are executed
- Each trial is run with a fixed termination time $t \in \{0.5, 1, 2, 5, 10, 20\}$ seconds
- Path validity checking is handled internally by OMPL
- Collision checking sampling resolution is set to $\frac{0.01}{10^{\text{dims}/2}}$

This setup allows consistent comparison across planners with differing convergence configurations and solution times.

*1) BIT* Specific Metrics:* For Batch Informed Trees (BIT*), additional timing metrics are recorded to capture its anytime planning behavior and solution refinement over time:

- Time and quality of the first solution
- Time and quality of the "best" solution
- Solution quality at the termination cutoff

The "best" solution is defined as the point at which the solution quality reaches the midpoint between the quality of the first solution and that of the optimal solution for the problem instance. This criterion provides a consistent reference for comparing how quickly BIT* improves the solution quality.

### *E. Grid-Based Planners*

The grid-based planners (BFS and A*) are evaluated using an MPI-based parallel implementation.

- Processor counts are varied;

$$p \in \{1, 2, 4, 8, 16, 32, 64, 128\}$$

- 10 trials are performed for each processor count and dimensionality
- For each trial, workload distribution statistics are collected

These measurements are used to assess parallel scalability and load balancing in addition to solution performance.

## VI. Results

This section evaluates planner performance across increasing state space dimensionality, parallel scalability, and collision checking cost. Results are reported using median planning time, success rate, normalized solution quality, and workload distribution metrics. Grid-based planners (BFS and A*) are evaluated under MPI parallel execution, while sampling-based planners are evaluated using OMPL under fixed time budgets. For anytime planners, multiple performance points are reported to capture both early feasibility and solution refinement behavior.

Unless otherwise stated, reported planning times for MPI-based planners correspond to the best-performing processor count for each planner and dimension, as determined by minimum median solve time.

### *A. Impact of Dimensionality on Planner Performance*

Table I reports the median planning time as a function of state space dimensionality. For MPI-based planners (BFS and A*), the reported time for each dimension corresponds to the fastest observed configuration across all tested processor counts, reflecting the best achievable performance under parallel execution. Unsolved configurations, due to either memory or time limits, are represented by dashes.

TABLE I
Planning Time v State Space Dimension

| **Planner** | **Median Plannning Time (ms)** | | | | |
|---|---|---|---|---|---|
| | ***2D*** | ***4D*** | ***6D*** | ***8D*** | ***10D*** |
| FMT | 30.0 | 759.5 | 22603.0 | - | - |
| BIT* - First | 16.8 | 56.2 | 68.8 | 2166.6 | 5605.1 |
| BIT* - Best | 351.3 | 592.1 | 347.1 | 9471.5 | 12130.6 |
| BIT* - Final | 500.0 | 1005.0 | 2010.0 | 20091.0 | 20123.0 |
| A* | 0.24 | 129.5 | 212.5 | 1086.7 | 1252.8 |
| BFS | 0.98 | 1344.6 | - | - | - |

Table II reports the fraction of trials in which a feasible solution was found within the maximum allotted budget of 20 seconds. For BIT*, a trial is considered successful if any feasible solution is found before timeout.

TABLE II
Success Rate v State Space Dimension

| **Planner** | **Success Rate** | | | | |
|---|---|---|---|---|---|
| | ***2D*** | ***4D*** | ***6D*** | ***8D*** | ***10D*** |
| FMT | 1.00 | 1.00 | 1.00 | 0.00 | 0.00 |
| BIT* | 1.00 | 0.99 | 0.98 | 0.81 | 0.68 |
| A* | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| BFS | 1.00 | 1.00 | 0.00 | 0.00 | 0.00 |

Table III reports normalized solution quality, defined as the ratio between path length and straight-line distance. For BIT*, quality is shown at first solution, intermediate optimization, and final timeout.

TABLE III
Solution Quality v State Space Dimension

| **Planner** | **Solution Quality** | | | | |
|---|---|---|---|---|---|
| | ***2D*** | ***4D*** | ***6D*** | ***8D*** | ***10D*** |
| FMT | 1.178 | 1.228 | 1.168 | - | - |
| BIT* - First | 1.301 | 1.936 | 1.900 | 2.168 | 2.127 |
| BIT* - Best | 1.160 | 1.557 | 1.521 | 1.715 | 1.828 |
| BIT* - Final | 1.137 | 1.498 | 1.394 | 1.583 | 1.5983 |
| A* | 1.08 | 1.41 | 1.73 | 2.00 | 2.22 |
| BFS | 1.08 | 1.41 | - | - | - |

### *B. Scaling Analysis of Grid-Based Planners*

Figure VI-B shows A* planning time as a function of processor count for each dimension. This figure represents both the strong scaling analysis, which occurs as the processor count is increased for a given dimension, and the weak scaling analysis, which occurs when the processor count is fixed and the dimensionality is scaled. The BFS planning time is not shown due to only 2D and 4D results being possible.

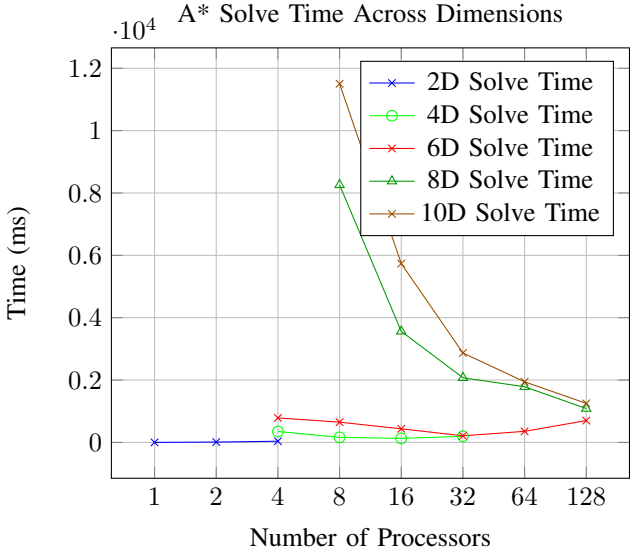


Table IV reports A* workload deviation across MPI ranks for each processor count and dimension. Calculated as the difference between the maximum and minimum workload percentages for all processors, the deviation shows load imbalance between processors. Lower deviation indicates improved parallelization while higher deviation indicates poor parallelization and program slowdown due to MPI barrier overhead.

TABLE IV
A* WORKLOAD DISTRIBUTION

| Processor Count | Workload Deviation | | | | |
|---|---|---|---|---|---|
| | *2D* | *4D* | *6D* | *8D* | *10D* |
| 2 | 47.6 | - | - | - | - |
| 4 | 21.6 | 0.49 | 2.04 | - | - |
| 8 | - | 0.54 | 1.68 | 0.10 | - |
| 16 | - | 2.29 | 0.69 | 0.08 | 0.07 |
| 32 | - | 0.50 | 2.62 | 0.02 | 0.06 |
| 64 | - | - | 0.31 | 0.01 | 0.03 |
| 128 | - | - | 0.21 | 0.02 | 0.03 |

### C. Collision Checking Benchmarks

Table V reports the total number of collision checks performed by each planner. For MPI-based planners, the number of collision checks is the total number of checks performed across all processors.

TABLE V
COLLISION CHECKS PER ALGORITHM

| Planner | Number Of Collision Checks | | | | |
|---|---|---|---|---|---|
| | *2D* | *4D* | *6D* | *8D* | *10D* |
| FMT | $21 \times 10^5$ | $11 \times 10^6$ | $61 \times 10^7$ | - | - |
| BIT-First | $66 \times 10^2$ | $16 \times 10^5$ | $50 \times 10^5$ | $14 \times 10^7$ | $31 \times 10^7$ |
| BIT-Best | $23 \times 10^4$ | $50 \times 10^5$ | $23 \times 10^6$ | $59 \times 10^7$ | $68 \times 10^7$ |
| BIT-Final | $31 \times 10^4$ | $62 \times 10^5$ | $55 \times 10^6$ | $10 \times 10^8$ | $11 \times 10^8$ |
| A* | $56 \times 10^2$ | $41 \times 10^7$ | $23 \times 10^6$ | $69 \times 10^7$ | $92 \times 10^7$ |
| BFS | $32 \times 10^3$ | $26 \times 10^8$ | - | - | - |

## VII. DISCUSSION

### A. Relative Planner Performance

Across all evaluated dimensions, A* consistently achieves the lowest median planning time among the planners considered in this study. The sole exception occurs when considering BIT*'s time to first feasible solution, where BIT* occasionally produces an initial path more quickly, albeit with substantially poorer solution quality. When accounting for both planning time and solution quality, A* remains the dominant performer across the benchmarked environments.

In higher-dimensional settings, particularly in 10D, BIT* attains marginally shorter final path lengths than A*. This behavior is likely attributable to differences in connectivity: The grid-based planners restrict state expansion to movements along at most two coordinate axes per step, limiting the ability to exploit diagonal structure in high-dimensional spaces. In contrast, BIT* operates over a continuous sampling-based graph, enabling more flexible transitions that can yield shorter paths despite higher planning times. This highlights a fundamental tradeoff observed in the results, A* prioritizes computational speed and scalability, while BIT* can achieve higher-quality solutions in very high dimensions at increased computational cost.

### B. Parallelization & Scalability

The grid-based planners exhibit strong parallel scalability up to a dimension-dependent saturation point, beyond which additional processors provide diminishing or negative returns. In lower-dimensional problems, the workload is insufficient to effectively utilize large core counts. However, in higher dimensions, the larger search frontier is too large for small processor counts, but provides a sufficient workload to balance across many processors. For this reason, certain planner–dimension combinations were not evaluated beyond a limited number of processes, as increasing, or decreasing, the core count no longer improved performance.

Workload distribution measurements indicate that the hash-based ownership strategy achieves a moderately well-balanced partitioning of the search space across MPI ranks. Across most tested dimensions and processor counts, the observed workload deviation remains low, demonstrating that the dual-purpose hash function effectively distributes states while preserving locality for visited-state tracking. These results confirm that the parallelization strategy scales efficiently within its practical operating regime and that performance limitations at higher core counts are primarily due to algorithmic and communication constraints rather than load imbalance.

### C. Collision Checking

Across all planners, the total number of collision checks scales roughly with the dimensionality of the problem, and is relatively consistent for a given dimension. Grid-based planners, however, are able to exploit parallelism to distribute collision checking across multiple MPI ranks, effectively reducing the per-rank computation time. This parallel evaluation likely

contributes to their comparatively lower wall-clock planning times.

## VIII. Conclusion

Overall, the findings highlight clear tradeoffs: A* provides the most reliable and high-quality paths assuming that the processor count is correctly scaled for the workload. In higher dimensions, BIT* can produce better solutions, at the cost of planning time. BFS rapidly becomes intractable above 4D, serving primarily as a baseline. The hash-based parallelization strategy is effective for distributing work and mitigating memory constraints, enabling grid-based planners to scale to higher dimensions where feasible. Future work could explore adaptive adjacency strategies for A* or further tuning of BIT* sampling parameters to improve performance in very high-dimensional narrow-passage environments.

## References


[1] L. Janson, E. Schmerling, A. Alark, and M. Pavone "Fast Marching Tree: a Fast Marching Sampling-Based Method for Optimal Motion Planning in Many Dimensions" Febuary, 2015

[2] J. Gammell, T. Barfoot, and S. Srinivasa "Batch Informed Trees (BIT*): Informed Asymptotically Optimal Anytime Search" 2017

[3] S. Karaman and E. Frazzoli " Sampling-based Algorithms for Optimal Motion Planning "

[4] A. Ghosh, M. Ojha, and K. P. Singh " A Comparative Study on Grid-Based and Non-grid-based Path Planning Algorithm " 2023

[5] J. Luo and K. Hauser " An Empirical Study of Optimal Motion Planning "