Resource-Aware Motion Planning

Manfred Kröhnert, Raphael Grimm, Nikolaus Vahrenkamp and Tamim Asfour

***Abstract*— We address the question of how resource-aware concepts can be utilized in motion planning algorithms. Resource-awareness facilitate better resource allocation on global system level, e.g. when a humanoid robot needs to distribute and schedule a wide variety of concurrent algorithms. We present a motion planning approach that employs self-monitoring concepts in order to identify the difficulty of the planning problem. Resources are requested dynamically and adapted based on problem difficulty and current planning progress. We show how dynamic adaptation of resource allocation on algorithmic level can reduce the system workload as compared to static resource allocation while meeting Quality of Service (QoS) measures such as average workload or efficiency. We evaluate our approach both in several synthetic setups with varying difficulty and with the humanoid robot ARMAR-4.**

I. INTRODUCTION

Complex robot systems, such as humanoid robots, utilize a wide variety of algorithms, ranging from low level control, image processing to visual perception and symbolic planning. During execution, these algorithms share a limited set of resources (CPU, memory, communication bandwidth, power). In general, an operating system schedules requested resources fairly between concurrent components, while the algorithms themselves act in a greedy manner, ignoring the current workload of the system. This can result in non-optimal resource allocation due to several issues. First, it might be required to limit the greedy nature of algorithms (i.e. limit their resource allocation) in order to improve global resource availability and to avoid workload peaks affecting other components. Second, context dependent task priorities can not be taken into account by a static scheduler operating on system level. For example, a speech recognition and dialogue management system should run with high priority when commands are triggered, while a smalltalk situation could result in lower priorities and lower resource demands. Finally, unused system resources could be assigned to computationally intensive modules to speed up calculations or improve QoS.

Resource-aware algorithms can be used to deal with changing and non-optimal resource allocations. Based on the current context, they adapt their resource demands to the current system load. Such resource-aware concepts are for example investigated by the research project Invasive Computing [1]. These concepts are helpful in programming, monitoring, and supervising distributed and component-based systems and hence, in fine tuning the overall system performance. Benefits and advances of resource-awareness have been shown for perceptual algorithms in [2], [3].

The authors are with the Institute for Anthropomatics and Robotics, Karlsruhe Institute of Technology (KIT), Germany, {kroehnert,asfour}@kit.edu

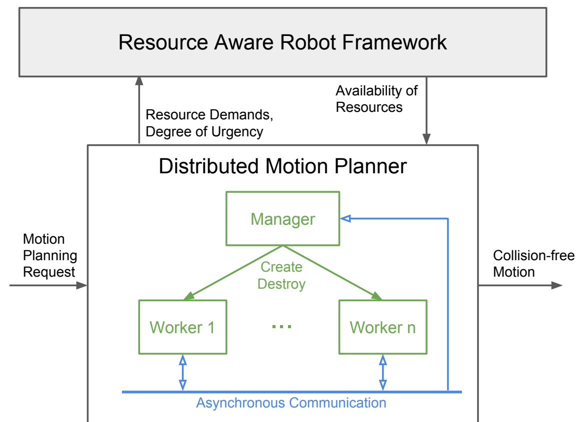

Fig. 1: The proposed motion planning algorithm can be embedded within a resource-aware robot software framework. the planner is equipped with self-monitoring concepts and the capability to adaptively change its resource demand.

Especially autonomous humanoid robots benefit from resource-aware concepts. Their resources are always limited due to many concurrent algorithms and the constraints of available battery power. Hence, it makes sense for algorithms to only request resources when actually needed. Ideally, the result quality of such algorithms improves by adding more resources or stays stable with less resources available. However, not all algorithms share these properties. To actually equip a robot with resource-aware components, self-monitoring concepts are required in order to allocate available resources to algorithms based on the current system status.

In this work we present a distributed motion planner which is able to request more resources at runtime (see Fig. 1). We focus on CPU utilization as the most important resource aspect for motion planning algorithms. Further, we show how self-monitoring strategies are incorporated to identify the difficulty of a planning problem in order to dynamically increase the requested amount of system resources.

The algorithm is executed on a network of PCs running standard Ubuntu 12.04. Evaluation is performed on both an artificial setup allowing variations in the planning problem difficulty and a realistic setup with the humanoid robot ARMAR-4 [4].

II. RELATED WORK

The motion planning problem can be addressed by a wide variety of algorithms. High dimensional problems, as they are present in humanoid robotics, can be solved efficiently with randomized approaches like Probabilistic Roadmaps (PRM) [5] and Rapidly-exploring Random Trees (RRT) [6]. In this work, we focus on variants of the RRT approach, since it allows for efficient solving of single query planning problems while maintaining probabilistic completeness. Since the original version of the RRT algorithm, many improvements have been published which can be separated into three functional groups according to their goals.

A. Improving the quality of a found solution

RRT* [7] makes two modifications to the original *RRT* to provide asymptotic optimality, in addition to the probabilistic completeness of *RRT*.

First, the parent node selection is changed to create an intermediate configuration c when trying to connect the nearest neighbor nn to the sampled random configuration. The algorithm then searches for a configuration in c's vicinity, instead of connecting c to nn. The line segment from c to the found configuration must be collision free and result in a path with the lowest cost from the start node to c.

The second modification optimizes path costs during a rewiring step, which is executed each time a new configuration is added to the search tree. Each node in the new node's vicinity is checked to determine whether the cost of its current path is higher than the cost of a path including the new configuration. If this is the case and the line segment connecting the node and the new configuration is collision free, the node's old path is removed and a path via the new configuration is added.

Informed RRT* [8] improves *RRT**'s [7] convergence towards the optimal solution. Each time a better solution s is found, the sampling domain c_{space} changes to $c_{space} \bigcap P_s$, with P_s being the prolate spheroid containing all solutions with a cost less or equal to s's cost. P_s's focal points are ρ_{start} and ρ_{goal} and its polar diameter is set to the cost of s. This change causes *Informed RRT** to concentrate on improving the path $\rho_{start} \rightarrow \rho_{goal}$ instead of improving the path from ρ_{start} to every configuration $\rho \in c_{space}$.

B. Changing the sequential algorithm to find solutions faster

Dynamic-Domain RRT [9] tries to limit the sampling domain to the search tree's visible Voronoi region. Since determining the visible Voronoi region is a hard problem, this region is only approximated. For this approximation to work, each node ρ is assigned a radius with infinity as initial value. Once a connection attempt fails, the node radius will be set to r_{border}. In this case, the intermediate configuration $\rho_{reached} \leftarrow Steer(\rho, \rho_{rnd})$ [1] is identical to ρ and the node is then called a boundary node. When a node ρ_{nn} is the nearest neighbor to a random configuration ρ_{rnd}, the function $Steer(\rho_{nn}, \rho_{rnd})$ is only called, if $Distance(\rho_{nn}, \rho_{rnd})$ is less than ρ_{nn}'s radius. Since boundary nodes lie on the border between c_{free} and $c_{obstacle}$, their visible Voronoi region is smaller than their total Voronoi region. Visible Voronoi region of all other nodes is mostly identical to their total Voronoi region. The smaller radius of boundary nodes expresses this fact. This property stops *Dynamic-Domain RRT* from trying to expand towards unreachable nodes and benefits the algorithm in presence of small passages by reducing the number of collision checks. The correct value for parameter r_{border} is important for the visible Voronoi region's close approximation. It has to be small enough to exclude regions outside and large enough to include most of the visible Voronoi region.

Adaptive Dynamic-Domain RRT [10] is a modification of *Dynamic-Domain RRT* making it more robust against poor choices of parameter r_{border}. The robustness is achieved by further changing the radius of boundary nodes, after it was set to r_{border}. For each additional failed connection attempt the node's radius is decreased by an α value, while it is increased by α for each successful connection attempt. To preserve probabilistic completeness, a node's radius will not get lower than r_{min}. This adapting *Dynamic-Domain* causes border nodes with a high percentage of $c_{obstacle}$ in their vicinity to have a lower radius than other boundary nodes, and thus improves the approximation of visible Voronoi region.

C. Parallelizing the algorithm

Several parallel RRT implementations have been proposed to decrease planning time and making it possible to deal with more difficult planning problems.

Sampling-Based Roadmap of Trees (SRT) [11] creates a global PRM of local RRT's resulting in a more performant and significantly more decoupled planner than just PRM and sampling-based tree planners. The more powerful local planer allows using more complex milestones and to distribute the workload almost evenly among processors while keeping communication low.

Manager Worker RRT [12], [13] uses functional decomposition to collaboratively build a search tree. The planing task is split into two types of subtasks: operations with search tree access (e.g. nearest neighbor search) and operations without (e.g. collision checking). A manager process executes the first type of subtask while it outsources collision checks to multiple worker processes. This strategy can become inefficient if the two subtask types are intertwined (e.g in *RRT**).

Distributed RRT [12], [13] collaboratively builds a search tree through exploratory decomposition by using multiple identical workers. Each worker owns a copy of the sampling tree and performs sampling and collision checking. Before each planning iteration, a worker applies all updates received from other workers. A worker broadcasts its progress asynchronously to all other workers after each iteration. Once a solution is found, a stop signal is broadcast and all workers terminate.

[1] $Steer(\rho_f, \rho_t)$ returns the point ρ_r from the line segment (ρ_f, ρ_t), closest to ρ_t. The line segment (ρ_f, ρ_r) has to be collision free.

pSBMP-RRT and ***pSBMP-PRM*** [14] provide a planning framework compatible with sampling-based algorithms such as PRMs and RRTs. Better scalability than other methods such as SRT is achieved through applying C-space subdivision. An independent roadmap is construct in each subdivided region in parallel. Afterwards, regional roadmaps are connected to other nearby roadmaps to form a roadmap of the entire C-space.

Bulk Synchronous Distributed RRT [15] is a modification of *Distributed RRT*. It reduces communication overhead by only sending updates after a bulk of iterations have been performed by a worker.

Using graphics coprocessors can speed up collision checking and thus the whole algorithm [16].

Utilizing lock free concurrency and cache effects can result in superlinear speedup [17].

Other types of parallel RRT algorithms exist which were not used in this work, such as ***Or parallel RRT*** [12], [13], ***RRTLocTrees*** [18], and ***Blind RRT*** [19].

III. Resource-Aware RRT*

Already proposed RRT algorithms do not consider adapting the algorithm's resource usage to the problem's requirements. Additionally, the parallelized RRT algorithms use a static number of processing units for calculation and do neither consider system state nor availability of resources.

As a first step in this direction, we propose a distributed resource-aware RRT algorithm which observes its own state and requests more resources if the current problem demands so. It uses *Informed RRT** for its asymptotic optimality and improved convergence, *Adaptive Dynamic-Domain RRT* for its sampling domain limiting and small passage handling, and *Bulk Synchronous Distributed RRT* for its distributed computation and reduced communication overhead. Since one implemented resource allocation strategy utilizes runtime tracking of planning progress, the used planners must be able to provide some kind of failure metric.

A. The planning algorithm

The planning algorithm consists of one manager and identical multiple worker processes as shown in Fig. 1. Since sampling time is comparatively low, the main idea is to parallelize work associated with costly collision checks instead of optimizing specific subparts of the algorithm.

Algorithm 1 describes the dedicated manager process which monitors the planning state, analyzes the sampling tree, and determines if additional workers should be started. However, the manager does not perform any kind of sampling or collision checking. The manager starts a new worker node every time the $RequiresAdditionalResources(s)$ function evaluates to $true$. Internally, resource allocation strategies as described in section III-B are used to determine if additional resources are required. The maximum worker count $workerCnt_{max}$ is designed to be influenced from outside the algorithm. If $workerCnt_{max}$ is reduced, the manager will stop workers, starting with the newest one until their number is below the new maximum. Planning fails when no solution was found after the specified $timeout$ passes. After planning has completed or failed, the manager terminates all workers and returns the result.

Algorithm 1 ManagerLoop

Require: start and goal configuration $\rho_{start}, \rho_{goal} \in c_{space}$, strategy s, bulk size $m \in \mathbb{N}$, $timeout$, initial and maximum worker count $workerCnt_{initial}, workerCnt_{max}$, adaptive dynamic domain parameters α, r_{border}, $r_{min} \in \mathbb{R}$

```
1: τ : Tree
2: AddNode(τ, ρ_start)
3: for i ← 0 to workerCnt_initial do
4:    StartAdditionalWorker()
5: end for
6: while not (HasPath(τ, ρ_start, ρ_goal) or isTimeout())
   do
7:    Wait()
8:    ApplyPendingUpdates(τ)
9:    UpdateStrategy(s)
10:   if CurrentWorkerCnt() < workerCnt_max then
11:      if RequiresAdditionalResources(s) then
12:         StartAdditionalWorker()
13:      end if
14:   end if
15:   while CurrentWorkerCnt() > workerCnt_max do
16:      StopNewestWorker()
17:   end while
18: end while
19: ShutdownAllWorkers()
20: ApplyPendingUpdates(τ)
21: return Path(τ, ρ_start, ρ_goal)
```

Algorithm 2 describes the combination of *Informed RRT**, *Adaptive Dynamic-Domain RRT*, and *Bulk Synchronous Distributed RRT* executed by each worker. The functions $SelectParent_{RRT^*}$ and $Rewire_{RRT^*}$ select a node's parent and perform rewiring analogous to *RRT**.

B. Resource Request Strategies

The implemented algorithm uses different strategies to determine whether more computational power is required to solve the current problem. Every strategy to be used with the planner must have the following properties: independence of and thus not posing restrictions on the planning algorithm and short computation time to only cause minimal overhead.

1) Possible Strategies: A strategy can track an expansion metric of the tree and use this to determine the rate of expansion or the ratio between current and maximal tree expansion. Two groups of expansion metrics exist.
The first group examines the path length or path cost. Expansion can be determined by considering the length of the longest path in the tree, or the average cost to reach a leaf node. The current expansion rate can be estimated by observing the average length of the last N edges added to the tree.
The second group examines the volume of the tree. One

Algorithm 2 WorkerLoop

Require: goal configuration $\rho_{goal} \in c_{space}$, bulk size $m \in \mathbb{N}$, adaptive dynamic domain parameters: α, r_{border}, $r_{min} \in \mathbb{R}$

1: $\tau \leftarrow GetCurrentTree()$
2: **while not** $ReceivedShutdownRequest()$ **do**
3: **for** $i = 0$ to m **do**
4: $ApplyPendingUpdates(\tau)$
5: **repeat**
6: $\rho_{rnd} \leftarrow Sample(c_{space})$
7: $\rho_{nn} \leftarrow NearestNeighbour(\tau, \rho_{rnd})$
8: **until** $\rho_f.r > Distance(\rho_f, \rho_{rnd})$
9: $\rho_{reached} \leftarrow Steer(\rho_{nn}, \rho_{rnd})$
10: **if** $\rho_{reached} \neq \rho_{nn}$ **then**
11: $V_{NNs} \leftarrow NearestNeighbors(\tau, \rho_{reached})$
12: $\rho_f \leftarrow SelectParent_{RRT^*}(\rho_{reached}, \rho_n, V_{NNs})$
13: $AddNode(\tau, \rho_{reached})$
14: $AddEdge(\tau, (\rho_f, \rho_{reached}))$
15: $\rho_{reached}.r \leftarrow \infty$
16: $\rho_{nn}.r \leftarrow \rho_{nn}.r * (1 + \alpha)$
17: $Rewire_{RRT^*}(\tau, \rho_{reached}, V_{NNs})$
18: $UpdateMinimalPathLength(\tau)$
19: **else**
20: **if** $\rho_{nn}.r = \infty$ **then**
21: $\rho_{nn}.r \leftarrow r_{border}$
22: **else**
23: $\rho_{nn}.r \leftarrow \max(r_{min}, \rho_{nn}.r * (1 - \alpha))$
24: **end if**
25: **end if**
26: **end for**
27: $SendUpdate(\tau.currentUpdate)$
28: **end while**

approach is to use the volume of the axis aligned bounding box (AABB) containing all nodes. However, this strategy is unstable, since is not able to track the planning progress if the solution is contained inside the AABB's volume.

A different approach determines the ratio of failed creations of intermediate configurations to the amount of tries of creating them over the last N iterations. If a method such as *Dynamic-Domain* is used, the ratio approximates the relation of boundary surface between tree region + $c_{obstacle}$ and the tree region itself. In this case, resources will only be requested if the ratio is high, meaning that the problem is most probably hard.

A third approach is to measure the algorithms runtime and request more computation power after a time delta ΔT has passed since the last request for additional computation power. This approach is robust, since planning problems are guaranteed to be hard if they take longer to solve. On the other hand, workers are started too frequently or infrequently if ΔT is chosen too small or too big. Similar to the r_{border} parameter of *Dynamic-Domain RRT* this weakness can be addressed by adding an additional parameter to adapt ΔT. To balance out weaknesses of other approaches, a time delta should always be added to any strategy.

2) Implemented Strategies: Two strategies were implemented after evaluating proposed possible candidates. Both produce a binary response of whether additional resource should be requested or not. No distance based metric was implemented, since their values are input dependent and would require normalization. Due to this dependency, it is impossible to come up with a general, meaningful, and problem independent normalization strategy.

Since planning time is a direct indicator of the problem difficulty, the first evaluated strategy uses the ΔT-approach. In general, it is referred to as ΔT-strategy and as ΔTy for a parameter $\Delta T = y$.

$$T_{last} + \Delta T \leq T_{now} \tag{1}$$

If equation (1) evaluates to $true$, the ΔT-strategy decides to request additional computational power.

Wrong choices of the ΔT parameter can lead to late resource acquisition for difficult problems. Hence, the second strategy modifies ΔT depending on the rate φ of failed creations of intermediate configurations. High values of φ indicate that the algorithm is most probably searching in a difficult boundary area where using more resources would be beneficial. The added parameter σ determines how strongly the rate φ influences ΔT. In general, this strategy is referred to as *NN*ΔT-strategy (No Node ΔT) and as *NNx*ΔTy for a set of parameters $\sigma = x$ and $\Delta T = y$

$$\Delta T' = \frac{\Delta T}{1 + \sigma * \varphi} \tag{2}$$

$$T_{last} + \Delta T' \leq T_{now} \tag{3}$$

Equation (2) and (3) describe whether the *NN*ΔT-strategy decides to request additional computation power. The *NN*ΔT-strategy will request an additional worker at least every ΔT but not faster than every $\Delta T/(1 + \sigma)$. The effect of σ on ΔT is shown in Fig. 2.

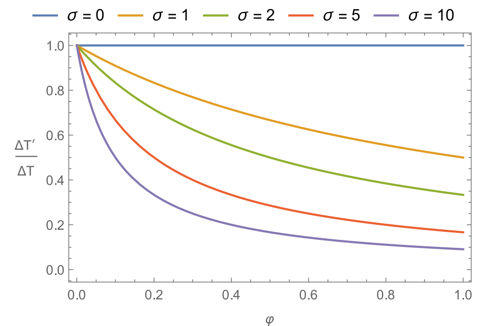

Fig. 2: Effect of σ on the ratio $\frac{\Delta T'}{\Delta T}$ as a function of φ.

IV. Evaluation

The two implemented strategies are evaluated with four synthetic setups and one realistic setup with the humanoid robot ARMAR-4. All test cases are executed 100 times and runtime is measured in wall-clock-time. The value of the

bulk size parameter m has been experimentally determined and is set to 10 in all cases. During test execution, the system never attempts to remove resources from the algorithm. For the majority of time the manager process waits for updates of the worker nodes. Thus, it is excluded from the process count since it only causes an insignificant amount of CPU utilization as opposed to the 100% CPU utilization of each worker node. To eliminate outliers, the 10% trimmed mean is used for all values.

A. Test platform

All test cases are executed on up to four machines connected via Gigabit Ethernet. The average CPU speed of 3.38 GHz is realized through the following *Intel(R) Core(TM)* processors: 1 x *i7-4770* (3.4 GHz), 2 x *i7-4790* (3.6 GHz) and 1 x *i7 CPU 870* (2.93 GHz). *Ubuntu 12.04.5 LTS* is used as operating system which ships version 4.6.3 of the *g++* compiler. No additional tasks are running on the computers and the connection network is free from additional load. Each machine starts a maximum number of four workers during execution.

B. Test Case 1: SerialWalls

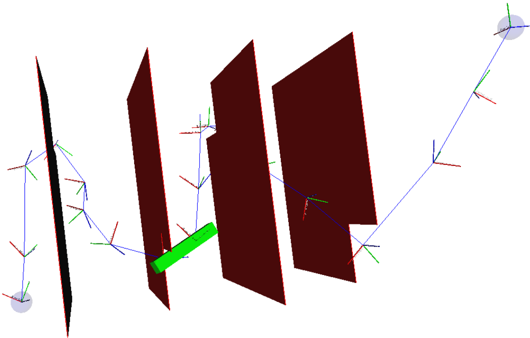

Fig. 3: The test case $SerialWalls4$ consisting of four walls and the moving object.

The $SerialWallsN$ test case is used to evaluate performance in a similar setup with increasing difficulty. The synthetic setups consist of $N \in \{1, 2, 3, 4\}$ walls with holes in opposite corners. The task is to plan a collision-free motion that moves a cuboid from one side through all holes to the other side. The hardest test case $SerialWalls4$ is shown in Fig. 3.

The following strategies are applied in order to evaluate the overall system performance:

- **Static Resource Allocation**: A constant number of workers is started at the beginning. The first worker immediately starts the planning algorithm. Evaluation is performed with 1, 4, and 16 workers.
- **ΔT**: The ΔT-strategy is applied with $\Delta T \in \{5, 10, 20, 30\}$. After ΔT seconds, a new worker process is started.
- **NNΔT**: The $NN\Delta T$-strategy is used with $\Delta T = 30$ and $\sigma \in \{1, 5, 10\}$. It adaptively requests new workers when the planning progress is weak.

C. Test Case 2: Box

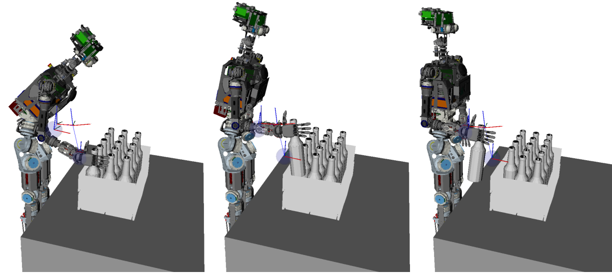

Fig. 4: The test case Box. ARMAR-4 pulls a bottle out of a box of bottles.

In this test case, a collision-free motion for the humanoid robot ARMAR-4 has to be planned for pulling a bottle out of a box of bottles. The kinematic chain used for grasping contains 10 degrees of freedom, covering two torso and 8 arm joints. This test case contains a narrow passage at the start configuration and is used to test whether the $NN\Delta T$-strategy is robust against weak selections of parameter ΔT. Chosen parameters are $\Delta T \in \{5, 30\}$ for the ΔT-strategy; $\Delta T = 30$ and $\sigma \in \{1, 5, 10\}$ for the $NN\Delta T$-strategy.

The best strategy for the Box test case would be to start as many workers as fast as possible, since it poses a difficult problem.

D. Evaluation of the Static Resource Allocation Strategy

Fig. 5 compares the $SerialWalls$ test cases executed with a static number of workers. It shows the average execution time t_{solve} until the first solution was found and the time $t_{initial}$ required to start all initial workers. The bars in each test case are normalized to the runtime of one worker, to provide better comparison of the planning time improvements, Absolute planning times are listed in TABLE I.

Fig. 5 shows, that, relative to the one worker case, both execution time and $t_{initial}/t_{solve}$ ratio decrease with increasing difficulty. Further, it shows that a constant resource allocation is not optimal for simple problems. The time required to start 16 worker processes in the $SerialWalls1$ test case almost equals the time required by four workers to solve the problem. In this test case, $t_{initial}$ is quite large, since each worker loads its models over network and workers are started sequentially. All workers run independent of the manager process and directly start planning after being initialized. Thus, the execution of the 16 worker case terminates after all workers were started, since a solution was already found by a previous worker. Comparison between the 4 workers and the 16 workers setup, shows that using all available computation power for a simple problem results in a low speedup due to the static overhead of starting additional threads and copying required data.

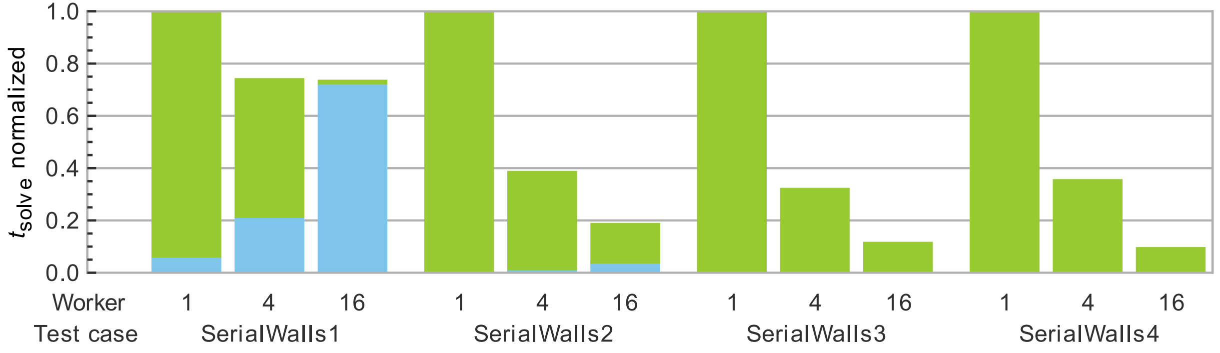

Fig. 5: Static resource allocation in the $SerialWalls$ test cases. t_{solve} is the average execution time until the first solution was found (green) and $t_{initial}$ the time required to start all initial workers (blue). Detailed planning times are listed in TABLE I.

TABLE I: The execution time average t_{solve} and standard deviation ($SD_{t_{solve}}$) until the first solution was found and the time $t_{initial}$ and standard deviation ($SD_{t_{initial}}$) required to start all initial workers.

Testcase	SerialWalls1			SerialWalls2			SerialWalls3			SerialWalls4		
Worker	1	4	16	1	4	16	1	4	16	1	4	16
Solved	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	42.0%	94.0%	100.0%
t_{solve} $[s]$	0.95	0.71	0.70	21.10	8.20	4.00	198.31	64.21	23.28	671.90	239.81	65.26
$SD_{t_{solve}}$ $[s]$	0.006	0.031	0.009	1.103	0.397	0.131	7.571	2.511	0.709	8.368	6.962	1.438
$t_{initial}$ $[ms]$	56.6	200.7	687.5	59.9	218.1	768.9	67.8	155.8	909.8	74.3	160.4	935.7
$SD_{t_{initial}}$ $[ms]$	0.66	11.64	1.94	0.75	12.00	1.78	0.74	6.39	1.85	0.88	1.15	1.42
$t_{initial}/t_{solve}$	5.9%	28.3%	97.9%	0.3%	2.7%	19.2%	0.0%	0.2%	3.9%	0.0%	0.1%	1.4%

For harder test cases this static overhead is negligible since t_{solve} increases while $t_{initial}$ stays mostly unchanged.

E. Evaluation of Dynamic Resource Allocation Strategies

In order to evaluate and compare dynamic resource allocation strategies, we introduce the following measures to obtain comparable numbers:

- **Average Workload $\mathbf{W_{avg}}$**: Since our algorithm uses strategies to request additional workers at runtime, the total numbers for started workers (W_{total}) are incomparable. Instead, a comparable value of average number of workers (W_{avg}) is calculated.

$$W_{avg} := \frac{t_{total}}{t_{solve}} \quad (4)$$

  t_{solve} is the execution time until the first solution was found (measured in the manager process) and t_{total} the accumulated execution time of all workers during this period (measured in each worker process). If only one single worker is started, its execution time equals t_{total}. Thus, the value of W_{avg} can be less than one, since t_{solve} includes additional work done by the manager process before any workers are started (like setting up the environment), .
- **Efficiency $\mathbf{t_{total}}$**: The efficiency can be expressed with t_{total}, the accumulated overall computation time of all workers required to generate the planning result. The smaller t_{total}, the more efficient the computation power is used.

Fig. 6 shows W_{total} and W_{avg} for all combinations of $SerialWalls$ test cases and executed strategies. It can be seen that all strategies decide not to request additional workers for $SerialWalls1$ while all strategies decide to start more workers for harder test cases. The lower the parameter ΔT, the more workers are started by the ΔT-strategy. The higher the parameter σ, the more workers are started by the $NN\Delta T$-strategy. The $NN10\Delta T30$-strategy compensates the relatively high (and therefore not optimally chosen) ΔT value and starts more workers than the $\Delta T20$-strategy.

To measure the effect of strategies on execution time, t_{solve} can be compared (see Fig. 7(a)). It can be seen that the simple test case $SerialWalls2$ is solved fastest with static resource allocation strategies. On the other hand, efficiency of static strategies gets worse (see Fig. 7(b)), since a part of t_{solve} is spent on initializing workers. In particular, the 16 workers setup shows bad efficiency. In contrast, the dynamic resource allocation strategies run a bit slower but with higher efficiency.

The evaluation of the hard problem $SerialWalls4$ shows that performance of the dynamic resource allocation strategies comes close to the static 16 worker strategy while the measured efficiency remains better.

In general, it can be stated that the efficiency of all dynamic resource allocation strategies is better than static approaches with more than 1 worker. This shows that using strategies results in a more efficient usage of computation power. Differences in efficiency get smaller with increasing toughness of the planning problem.

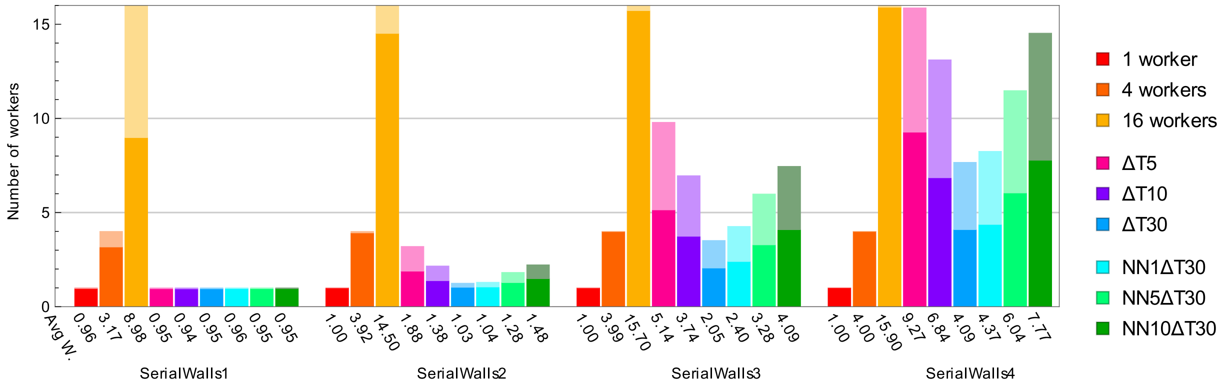

Fig. 6: The values of W_{total} (light color) and W_{avg} (dark color) for the $SerialWalls$ test cases and the used strategies. New resources are only allocated for harder test cases.

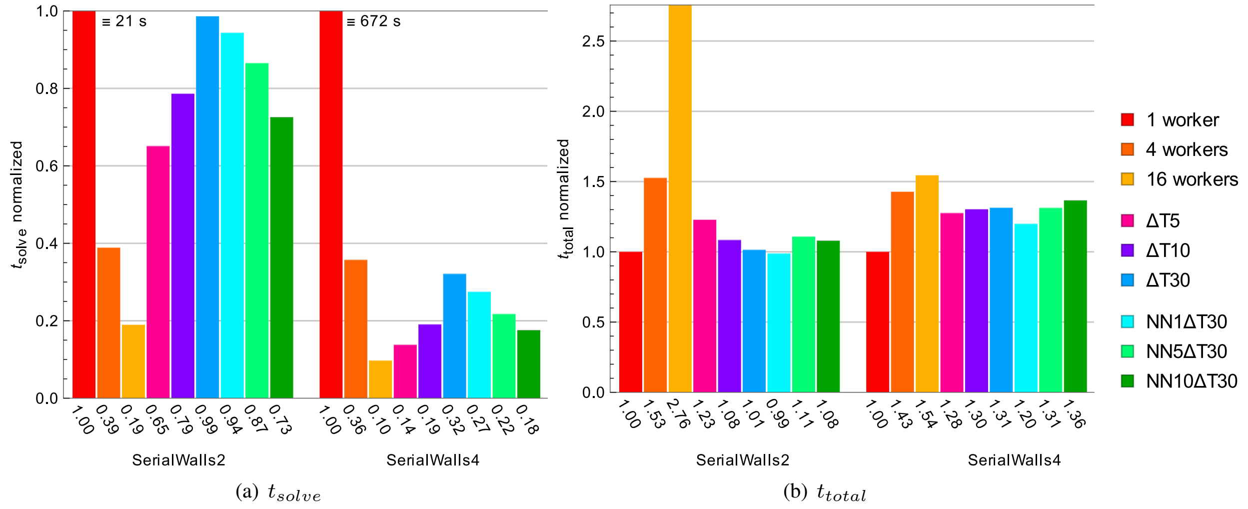

Fig. 7: Execution time t_{solve} and efficiency t_{total} for the test cases $SerialWalls2$ and $SerialWalls4$ per evaluated strategy. For each test case, the values are normalized to the execution time with a single worker.

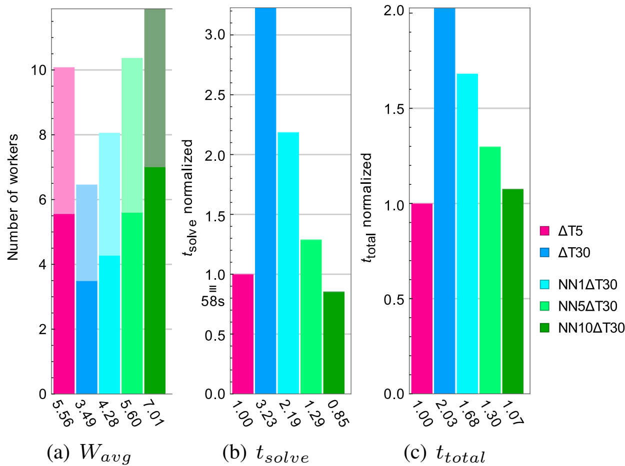

Fig. 8: W_{avg}, normalized t_{solve} and normalized t_{total} for the test case Box per evaluated strategy.

F. Evaluation with the humanoid robot ARMAR-4

The test case Box was chosen as a realistic setup which can show how well a badly selected parameter ΔT can be compensated by the $NN\Delta T$ strategy. Metrics for this test case are presented in Fig. 8 and the concrete numbers for average and standard deviation of t_{solve} for each strategy were $\Delta T5$ = (58,2 s / 3,8 s), $\Delta T30$ = (187,8 s / 7,3 s), $NN1\Delta T30$ = (127,2 s / 5,2 s), $NN5\Delta T30$ = (75,0 s / 3,7 s), and $NN10\Delta T30$ = (49,7 s / 3 s). The serial method is not shown, since only 33% of the test runs found a valid plan in a maximum of 6 minutes.

The $\Delta T30$ strategy starts the least amount of workers of all strategies due to the bad choice of $\Delta T = 30$. Its runtime is more than three times the runtime of $\Delta T5$, while t_{total} is two times larger. All $NN\Delta T$ strategies compensate this by starting more workers, causing both t_{solve} and t_{total} to decrease. $NN10\Delta T30$ solves the problem 15% faster than $\Delta T5$ while having a 7% higher efficiency value t_{total}. This again shows the ability of the $NN\Delta T$ strategy to compensate weak choices of parameter ΔT.

V. CONCLUSION

In this work, we presented a distributed resource-aware RRT algorithm capable of requesting additional computing resources based on strategies which monitor the current planning progress. Different resource request strategies have been evaluated in a series of synthetic motion planning problems with increasing difficulty as well as with a realistic setup with the humanoid robot ARMAR-4.

Overall we showed, that depending on the planning problem, static allocation or over allocation of resources often lead to inefficiencies and unnecessary increases in system workload. Thus, the amount of used computational resources should be directly adapted according to the problem difficulty in order to increase resource usage efficiency. The evaluated resource request strategies were shown to provide the required information to perform such adaptation by determining if and when additional resources should be requested. However, a slight increase in execution time can be observed in some cases, making the concrete choice of strategy parameters a tradeoff between efficiency and speed.

In the future, we plan to implement and evaluate other strategies to make the algorithm fully resource-aware. For example, a more sophisticated strategy would additionally include the current system workload in the decision making process in order to not impair the system by requesting more resources than available. Another possible strategy would be to use space partitioning for dimensionality limiting (like an octree) and to determine the tree expansion by counting occupied voxels. This approach requires a transformation from the configuration space to the work space, as well as an additional parameter to adapt the internals of this metric to the current planning problem. This parameter could be user-defined or problem derived, such as volume or diagonal lengths of the configuration space.

Communication (as opposed to collision checking) was not a bottleneck due to high bandwidth network connections. Thus, a future research direction is to examine the relationship between the number of workers and communication costs. Different levels of resource requirement urgencies as well as indicating the willingness to release held resources, could be achieved by transforming the current binary resource request value to a continuous one. The system can then utilize these urgency values in combination with priorities to better distribute system resources and keep the average level of satisfaction as high as possible.

Reducing or closing the gap in execution time and thus reducing tradeoff penalties is another topic for future work as well as widening the range of managed resources to include network transfer rates and memory consumption. Special hardware characteristics such as graphics processing units can be considered, too.

ACKNOWLEDGEMENT

The research leading to these results was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre Invasive Computing (SFB/TR 89) and has received funding from the European Unions Horizon 2020 Research and Innovation programme under grant agreement No 643950 (SecondHands).

REFERENCES

[1] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, and G. Snelting, "Invasive computing: An overview," in *Multiprocessor System-on-Chip*. Springer, 2011, pp. 241–268.

[2] J. Paul, W. Stechele, M. Kröhnert, and T. Asfour, "Resource-aware programming for robotic vision," *CoRR*, vol. abs/1405.2908, 2014.

[3] J. Paul, W. Stechele, M. Kröhnert, T. Asfour, B. Oechslein, C. Erhardt, J. Schedel, D. Lohmann, and W. Schröder-Preikschat, "Resource-Aware Harris Corner Detection based on Adaptive Pruning," in *Proceedings of the Conference on Architecture of Computing Systems (ARCS)*, 2014, pp. 1–12.

[4] T. Asfour, J. Schill, H. Peters, C. Klas, J. Bücker, C. Sander, S. Schulz, A. Kargov, T. Werner, and V. Bartenbach, "ARMAR-4: A 63 DOF Torque Controlled Humanoid Robot," in *IEEE/RAS International Conference on Humanoid Robots (Humanoids)*, 2013, pp. 390–396.

[5] L. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *Robotics and Automation, IEEE Transactions on*, vol. 12, no. 4, pp. 566–580, Aug 1996.

[6] J. Kuffner and S. LaValle, "RRT-connect: An efficient approach to single-query path planning," in *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on*, vol. 2, 2000, pp. 995–1001 vol.2.

[7] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011.

[8] J. Gammell, S. Srinivasa, and T. Barfoot, "Informed RRT*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic," in *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, Sept 2014, pp. 2997–3004.

[9] A. Yershova, L. Jaillet, T. Simeon, and S. LaValle, "Dynamic-Domain RRTs: Efficient Exploration by Controlling the Sampling Domain," in *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, April 2005, pp. 3856–3861.

[10] L. Jaillet, A. Yershova, S. La Valle, and T. Simeon, "Adaptive tuning of the sampling domain for dynamic-domain RRTs," in *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on*, Aug 2005, pp. 2851–2856.

[11] E. Plaku, K. Bekris, B. Chen, A. Ladd, and L. Kavraki, "Sampling-Based Roadmap of Trees for Parallel Motion Planning," *IEEE Transactions on Robotics*, vol. 21, no. 4, pp. 597–608, Aug. 2005.

[12] D. Devaurs, T. Simeon, and J. Cortes, "Parallelizing RRT on distributed-memory architectures," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, May 2011, pp. 2261–2266.

[13] ——, "Parallelizing RRT on Large-Scale Distributed-Memory Architectures," *Robotics, IEEE Transactions on*, vol. 29, no. 2, pp. 571–579, April 2013.

[14] S. A. Jacobs, K. Manavi, J. Burgos, J. Denny, S. Thomas, and N. M. Amato, "A Scalable Method for Parallelizing Sampling-Based Motion Planning Algorithms." IEEE, May 2012, pp. 2529–2536.

[15] S. Jacobs, N. Stradford, C. Rodriguez, S. Thomas, and N. Amato, "A scalable distributed RRT for motion planning," in *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, May 2013, pp. 5088–5095.

[16] J. Bialkowski, S. Karaman, and E. Frazzoli, "Massively parallelizing the RRT and the RRT*," in *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, Sept 2011, pp. 3513–3518.

[17] J. Ichnowski and R. Alterovitz, "Scalable Multicore Motion Planning Using Lock-Free Concurrency," *Robotics, IEEE Transactions on*, vol. 30, no. 5, pp. 1123–1136, Oct 2014.

[18] M. Strandberg, "Augmenting RRT-planners with local trees," in *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*, vol. 4, April 2004, pp. 3258–3262 Vol.4.

[19] C. Rodriguez, J. Denny, S. Jacobs, S. Thomas, and N. Amato, "Blind RRT: A probabilistically complete distributed RRT," in *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, Nov 2013, pp. 1758–1765.