Parallelizing RRT
on Large-Scale Distributed-Memory Architectures

Didier Devaurs, Thierry Siméon and Juan Cortés

***Abstract*—This paper addresses the problem of parallelizing the Rapidly-exploring Random Tree algorithm on large-scale distributed-memory architectures, using the Message Passing Interface. We propose three parallel versions of RRT, evaluate them on different motion planning problems, and thoroughly analyze the various factors influencing their performance.**

***Index Terms*—path planning, rapidly-exploring random tree, parallel algorithms, message passing, distributed memory**

I. Introduction

Due to a wide range of applications, sampling-based path planning has benefited from a considerable research effort [2], [3]. It has proven to be an effective framework suitable for a large class of problems in various domains such as autonomous robotics, aerospace, manufacturing, virtual prototyping, computer animation, structural biology, and medicine. These application fields yield increasingly difficult, highly-dimensional problems with complex geometric and differential constraints.

The Rapidly-exploring Random Tree (RRT) [4] has become a popular algorithm for solving single-query motion planning problems. It is suited to solve robot path planning problems involving holonomic, nonholonomic, kinodynamic, or kinematic loop-closure constraints [4]–[6]. It is also applied to planning in discrete spaces or for hybrid systems [7]. In computational biology, it is used to analyze genetic network dynamics [8] or protein-ligand interactions [9]. However, when applied to complex problems, the incremental growth of an RRT can become computationally expensive [10]–[13]. Some techniques have been proposed to improve the efficiency of RRT, by controlling the sampling domain [10], reducing the complexity of the nearest neighbor search [11], or using gap reduction techniques [12].

Our objective is to further investigate RRT improvement by exploiting speedup from parallel computation. Some results have been obtained in that direction (Section II). However, existing work considers mainly shared-memory architectures and small-scale parallelism, up to 16 processors [14]–[16]. In this work, we are interested in what can be achieved by larger-scale parallelism. We focus on parallelizing RRT on distributed-memory architectures, which require using the Message Passing Interface (MPI).

Our contribution is threefold. First, we propose three parallel versions of RRT, based on classical parallelization schemes: OR parallel RRT, Distributed RRT and Manager-worker RRT (Section III). Besides the abstract view provided by the algorithms themselves, we also present the main technicalities involved in their development. Then, we evaluate the algorithms on several motion planning problems and show their differences in behavior (Section IV). Finally, we analyze their performance in order to understand the impact of several characteristics of the studied problems (Section V). The parallel algorithms have been kept voluntarily simple to allow us to better isolate the effect of each of the influencing factors.

All authors are with CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France and Univ de Toulouse, LAAS, F-31400 Toulouse, France (e-mails: {devaurs, nic, jcortes}@laas.fr)

This work was carried out under the HPC-EUROPA project (RII3-CT-2003-506079), with the support of the European Community - Research Infrastructure Action under the FP6 "Structuring the European Research Area" Program). A preliminary version of this work has appeared in [1].

II. Related Work

A. Parallel Motion Planning

The idea of improving motion planning performance by using parallel computation has been raised in prior work. In a survey of some early work [17], a classification scheme is proposed to review different motion planning approaches and some related parallel processing methods. A more recent trend is to exploit the current multi-core technology available on many of today's PCs, which easily allows having multiple threads collaboratively solving a problem [18]. Another recent trend is the use of shared-memory models on many-core Graphics Processing Units (GPUs) [19].

Among the most classical approaches, the *embarrassingly parallel paradigm* exploits the fact that some randomized algorithms, such as the Probabilistic Road-Map (PRM), are what is termed "embarrassingly parallel" [20]. The massive inherent parallelism of the basic PRM algorithm enables a significant speedup, even with relatively simplistic parallelizing strategies, especially on shared-memory architectures. In this approach, computation time is minimized by having several processes cooperatively building the road-map.

Another simple approach is known as the *OR parallel paradigm*. It was first applied to theorem proving, before being used to provide a parallel formulation for the Randomized Path Planner (RPP) [21]. Its principle is to have several processes running the same sequential randomized algorithm, each one trying to build its own separate solution. The first process to reach a solution reports it and broadcasts a termination message. The idea here is to minimize computing time by finding a small-sized solution. Despite its simplicity, the OR parallel paradigm has been successfully applied to other randomized algorithms, such as in [22].

A more sophisticated approach is a *decentralized master-client scheme* developed to distribute the computation of the Sampling-based Roadmap of Trees (SRT) algorithm [23]. In a first step, several trees that can be RRTs or Expansive Space Trees (ESTs) are computed in parallel by all processes. In a second step, several master processes cooperate to evenly distribute the computation of edges linking these trees, among their respective client processes.

More generally, an approach based on growing several independent trees can lead to a straightforward parallelization. This is the case for the Rapidly exploring Random Forest of Trees (RRFT) [8], and for RRTLocTrees where several local trees are grown in difficult passages and connected to a global tree [24]. However, the focus of this paper lies elsewhere, our aim being to provide a parallel version of the basic (single-tree) RRT algorithm. Furthermore, this work is not about parallelizing subroutines of RRT, such as is done for collision detection in [19], nor about parallelizing specific variants of RRT, such as is done for the *any-time RRT* in [25]. Finally, our aim is to reduce the runtime of RRT and not to improve the quality of the paths it returns.

B. Parallel RRT

There is only little work related to parallelizing RRT [14]–[16]. The first one [14] applies the simple *OR parallel* and *embarrassingly parallel* paradigms, and a combination of both. To benefit from the simplicity of the shared-memory model, the embarrassingly parallel algorithm is run on a single symmetrical multiprocessor (SMP) node of a multi-nodes parallel computer. The only communication involved is a termination message that is broadcast when a solution is reached, but some coordination is required to avoid concurrent modifications of the tree. This scheme does not make use of the full computational power of the parallel platform, contrary to the OR parallel algorithm, which is run on all processors of all nodes. The same paradigms are also applied on a dual-core CPU in [15], where they are renamed

Algorithm 1: OR parallel RRT

```
input : the configuration space $C$, the root $q_{init}$
output: the tree $T$
1  $T \leftarrow$ initTree($q_{init}$)
2  while not stopCondition($T$) or received($endMsg$) do
3      $q_{rand} \leftarrow$ sampleRandomConfiguration($C$)
4      $q_{near} \leftarrow$ findBestNeighbor($T$, $q_{rand}$)
5      $q_{new} \leftarrow$ extend($q_{near}$, $q_{rand}$)
6      if not tooSimilar($q_{near}$, $q_{new}$) then
7          addNewNodeAndEdge($T$, $q_{near}$, $q_{new}$)
8  if stopCondition($T$) then
9      broadcast($endMsg$)
```

Algorithm 2: Distributed RRT

```
input : the configuration space $C$, the root $q_{init}$
output: the tree $T$
1  $T \leftarrow$ initTree($q_{init}$)
2  while not stopCondition($T$) or received($endMsg$) do
3      while received($nodeData$($q_{new}$, $q_{near}$)) do
4          addNewNodeAndEdge($T$, $q_{near}$, $q_{new}$)
5      $q_{rand} \leftarrow$ sampleRandomConfiguration($C$)
6      $q_{near} \leftarrow$ findBestNeighbor($T$, $q_{rand}$)
7      $q_{new} \leftarrow$ extend($q_{near}$, $q_{rand}$)
8      if not tooSimilar($q_{near}$, $q_{new}$) then
9          addNewNodeAndEdge($T$, $q_{near}$, $q_{new}$)
10         broadcast($nodeData$($q_{new}$, $q_{near}$))
11 if stopCondition($T$) then
12     broadcast($endMsg$)
```

OR and *AND* implementations. In the Open Motion Planning Library[1] (OMPL) of the ROS framework, the AND paradigm is implemented via multi-threading, and thus for shared memory.

To the best of our knowledge, there has been only one attempt to develop a parallel version of RRT on a distributed-memory architecture. In [16], the construction of the tree is distributed among several autonomous agents, using a message-passing model. However, no explanation is given on how the computation is distributed, and how the tree is reconstructed from the parts built by the agents.

III. Parallelizing RRT

For scalability purposes, we have chosen to parallelize RRT on distributed-memory architectures, using the message-passing paradigm, one of the most widespread approaches for programming parallel computers. Since this paradigm imposes no requirement on the underlying hardware and requires an explicit parallelization of the algorithms, it enables a wide portability. Any algorithm developed following this approach can also be run on a shared-memory architecture, even though it would mean not making an optimal use of this architecture. Besides, scalable distributed-memory architectures are rather commonly available, in the form of networks of personal computers, clustered workstations or grid computers. To develop our parallel algorithms, we have chosen to comply to the standard and widely-used Message Passing Interface[2] (MPI). Its logical view of the hardware architecture consists of p processes, each with its own exclusive address space. Our message-passing programs are based on the Single Program Multiple Data (SPMD) paradigm and follow a loosely synchronous approach: all processes execute the same code, containing mainly asynchronous tasks, but also a few tasks that synchronize to perform interactions.

A. OR Parallel RRT

The simplest way to parallelize RRT is to apply the OR parallel paradigm. Algorithm 1 presents our version of an *OR parallel RRT* that is similar to the one in [14]. Each process computes its own RRT (lines 1-7) and the first to reach a stopping condition broadcasts a termination message (lines 8-9). This broadcast operation cannot actually be implemented as a regular MPI_Broadcast routine, as this collective operation would require all processes to synchronize. Rather, the first process to finish sends a termination message to all others, using MPI_Send routines matched with MPI_Receive routines. As it is not known beforehand when these interactions should happen, a non-blocking receiving operation that will "catch" the termination message is initiated before entering the **while** loop. The received($endMsg$) operation is implemented as an MPI_Test routine checking the status (completed or pending) of the request generated by the non-blocking receiving operation. Finally, in case of several processes reaching a solution at the same time, the program ends with a collective operation for processes to synchronize and agree on which one should report its solution. Note that communications are negligible in the total runtime of the OR parallel RRT.

B. Collaborative Building of a Single RRT

Instead of constructing several RRTs concurrently, another possibility is to have all processes working collaboratively on building a single RRT. Parallelization is then achieved by partitioning the task of building an RRT into sub-tasks assigned to the various processes. We propose two ways of doing so, based on different decomposition techniques. (1) Since constructing an RRT consists in exploring a search space, we can use an *exploratory decomposition* [26]. Each process performs its own sampling of the search space – but without any space partitioning involved – and maintains its own copy of the tree, exchanging with the others the newly constructed nodes. This leads to a distributed (or decentralized) scheme where no task scheduling is required, aside from a termination detection mechanism. (2) Another classical approach is to perform a *functional decomposition* of the task [27]. In the RRT algorithm, two kinds of sub-tasks can be distinguished: the ones that require knowledge of the tree (initializing it, adding new nodes and edges, finding the best neighbor of q_{rand}, and evaluating the stopping conditions) and those that do not (sampling a random configuration and performing the extension step). This leads to the choice of a manager-worker (or master-slave) scheme as the dynamic and centralized task-scheduling strategy, the manager being in charge of maintaining the tree, and the workers having no knowledge of it. We now present both schemes in greater details.

1) Distributed RRT: Our version of a *Distributed RRT* is given by Algorithm 2. In each iteration of the tree construction loop (lines 2-10), each process first checks whether it has received new nodes from other processes (line 3). If this is the case, the process adds them to its local copy of the tree (line 4). Then, it performs its own expansion attempt (lines 5-7). If it is successful (line 8), the process adds the new node to its local copy of the tree (line 9) and broadcasts it (line 10). Adding all the received nodes before attempting an expansion ensures that every process works with the most up-to-date state of the tree. It is important to note that processes never wait for messages; they simply process them as they arrive. At the end, the first process to reach a stopping condition broadcasts a termination message (lines 11-12). This broadcast operation is implemented in the same way as for the OR parallel RRT. Similarly, the broadcast of

[1]http://www.ros.org/doc/api/ompl/html

[2]http://www.mpi-forum.org

Algorithm 3: Manager-worker RRT

```
input  : the configuration space $C$, the root $q_{init}$
output: the tree $T$
1  if $processID$ = $mgr$ then
2      $T \leftarrow$ initTree($q_{init}$)
3      while not stopCondition($T$) do
4          while received($nodeData(q_{new}, q_{near})$) do
5              addNewNodeAndEdge($T$, $q_{near}$, $q_{new}$)
6          $q_{rand} \leftarrow$ sampleRandomConfiguration($C$)
7          $q_{near} \leftarrow$ findBestNeighbor($T$, $q_{rand}$)
8          $w \leftarrow$ chooseWorker()
9          send($expansionData(q_{rand}, q_{near})$, $w$)
10     broadcast($endMsg$)
11 else
12     while not received($endMsg$) do
13         receive($expansionData(q_{rand}, q_{near})$, $mgr$)
14         $q_{new} \leftarrow$ extend($q_{near}$, $q_{rand}$)
15         if not tooSimilar($q_{near}$, $q_{new}$) then
16             send($nodeData(q_{new}, q_{near})$, $mgr$)
```

new nodes (line 10) is not implemented as a regular MPI_Broadcast routine, which would cause all processes to wait for each other. As a classical way to overlap computation with interactions, we again use MPI_Send routines matched with non-blocking MPI_Receive routines. That way, the received($nodeData$) test (line 3) is performed by checking the status of the request associated with a non-blocking receiving operation initiated beforehand, the first one being triggered before entering the **while** loop, and the subsequent ones being triggered each time a new node is received and processed. Again, the case of several processes reaching a solution at the same time has to be dealt with. Finally, a Universally Unique Identifier (UUID) is associated with each node, in order to provide processes with a homogeneous way of referring to the nodes.

2) Manager-Worker RRT: Algorithm 3 presents our version of a *Manager-worker RRT*. The program contains the code executed by the manager (lines 2-10) and the workers (lines 12-16). The manager is the only process having access to the tree. It performs the operations related to its construction, and delegates the expansion attempts to workers. In general, the expansion is the most computationally expensive stage in the RRT construction, since it involves motion simulation and validation. The manager could also delegate the sampling step, but this would not be worthwhile because of the low computational cost of this operation in our settings (i.e. in the standard case of a uniform random sampling in the whole search space): the additional communication cost would then outweigh any potential benefit.

At each iteration of the tree construction (lines 3-9) the manager first checks whether it has received new nodes from workers (line 4). If so, it adds them to the tree (line 5). Then, it samples a random configuration (line 6) and identifies its best neighbor in the tree (line 7). Next, it looks for an idle worker (line 8), which means potentially going through a waiting phase, and sends it the data necessary to perform an expansion attempt (line 9). Finally, when a stopping condition is reached, it broadcasts a termination message (line 10). On the other hand, workers are active as long as they have not received this message (line 12), though they can go through waiting phases. During each computing phase, a worker receives some data from the manager (line 13) and performs an expansion attempt (line 14). If it is successful (line 15), it sends the newly constructed node to the manager (line 16).

Contrary to the previous ones, this algorithm does not require non-blocking receiving operations for broadcasting the termination message. Workers being idle if they receive no data, there is no need to overlap computation with interactions. Before entering a computing phase, a worker waits on a blocking MPI_Receive routine implementing both the receive($expansionData$) operation and the received($endMsg$) test. The type of message received determines its next action: stopping or attempting an expansion. On the manager side, blocking MPI_Send routines implement the broadcast($endMsg$) and send($expansionData$) operations. The remaining question about the latter is to which worker should the data be sent. An important task of the manager is to perform load-balancing among workers, through the chooseWorker() function. For that, it keeps track of the status (busy or idle) of all workers and sends one sub-task at a time to an idle worker, choosing it in a round robin fashion. If all workers are busy, the manager waits until it receives a message from one of them, which then becomes idle. This has two consequences. First, on the worker side, the send($nodeData$) operation covers two MPI_Send routines: one invoked to send the new node when the expansion attempt is successful, and the other containing no data used otherwise. Second, on the manager side, two matching receiving operations are implemented via non-blocking MPI_Receive routines, allowing for the use of MPI_Wait routines if necessary. This also enables to implement the received($nodeData$) test with an MPI_Test routine. These non-blocking receiving operations are initiated before entering the **while** loop, and re-initiated each time the manager receives and processes a message. Finally, to reduce the communication costs of the send($nodeData$) operation, workers do not send back the configuration q_{near}. Rather, the manager keeps track of the data it sends to workers, which also releases us from having to use UUIDs.

C. Implementation Framework

Since the sequential implementation of RRT we wanted to parallelize was written in C++, and MPI being primarily targeted at C and Fortran, we had to use a C++ binding of MPI. We were also confronted with the low-level way in which MPI deals with communications, requiring the programmer to explicitly specify the size of each message. In our application, messages were to contain instances of high-level classes, whose attributes could be pointers or STL containers. Thus, we have decided to exploit the higher-level abstraction provided by the Boost.MPI library[3]. Coupled with the Boost.Serialization library[4], it enables processes to easily exchange class instances, making the tasks of gathering, packing and unpacking the underlying data transparent to the programmer. Finally, we have used the implementation of UUIDs provided by the Boost library[5].

IV. Experiments

Before presenting the results of the experiments themselves, we first introduce the metrics used to evaluate the parallel algorithms. We also present the parallel platform we have worked on, and the motion planning problems we have studied. We then explain the two experiments we have performed, and report the obtained results. A detailed analysis of the performance of the parallel algorithms will be the focus of Section V.

A. Performance Metrics

When evaluating a parallel algorithm on a given problem, we want to know how much performance gain it achieves over its sequential

[3]http://www.boost.org/doc/libs/1_47_0/libs/mpi
[4]http://www.boost.org/doc/libs/1_47_0/libs/serialization
[5]http://www.boost.org/doc/libs/1_47_0/libs/uuid

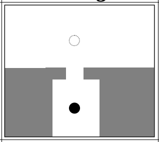
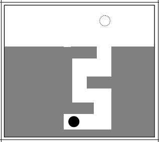
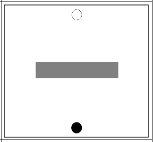
Problem name		***Passage***	***Corridor***	***Roundabout***
Problem type				
Sequential RRT	**T_S (s)**	48 ± 18	1250 ± 1001	38 ± 18
	N_S	644 ± 119	1027 ± 695	655 ± 336
	X_S	1807 ± 690	45374 ± 40855	1130 ± 456

Fig. 1. Simplified schematic representation of the configuration spaces of the three studied problems, and numerical results obtained with the sequential RRT including potential energy computation (first experiment). Average values over 100 runs (and standard deviation) are given for the sequential runtime, T_S (in seconds), the number of nodes in the final tree, N_S, and the number of expansion attempts, X_S.

counterpart. Aimed at measuring so, the *speedup* S of a parallel algorithm run on p processors is defined as the ratio of the runtime of its sequential counterpart to its own runtime: $S(p) = T_S \,/\, T_P(p)$ [26], [27]. The parallel runtime $T_P(p)$ is measured on a parallel computer, using p processors, and the sequential runtime T_S is measured on one processor of the same computer. We define $T_P(p)$ (resp. T_S) as the mean time needed to reach a solution, by averaging the runtimes obtained over 100 executions of a parallel (resp. sequential) algorithm. We then evaluate the *scalability* of a parallel algorithm, i.e. whether the speedup increases in proportion to the number of processors. Another common metric we use is the *efficiency* E of a parallel algorithm, which is defined as the ratio of the speedup to the number of processors: $E(p) = S(p) \,/\, p$ [26], [27]. For a given number of processors, we analyze the evolution of the efficiency with respect to the computational cost of the studied problem.

B. Parallel Computer Architecture

The numerical results presented in this paper have been obtained by running the algorithms on MareNostrum, the parallel platform of the Barcelona Supercomputing Center. It is an IBM® cluster platform composed of 2560 IBM® BladeCenter® JS21 blade servers connected by a Myrinet™ local area network warranting 2 Gbit/s of bandwidth. Each server includes two 64-bit dual-core PowerPC™ 970MP processors at 2.3 GHz, sharing 8 GB of memory. The implementation of MPI installed on this platform is MPICH2[6].

C. Motion Planning Problems Studied

We have evaluated the algorithms on three motion planning problems involving molecular models. The application we have used is the molecular motion planning toolkit we are currently developing [9]. However, it is important to note that these algorithms are not application-specific and can be applied to any kind of motion planning problem. The studied problems involve free-flying objects (i.e. six degrees of freedom)[7]. They are characterized by different configuration-space topologies (cf. Fig. 1). *Passage* is a protein-ligand exit problem, where a ligand exits the active site of a protein through a pathway that is relatively short and large but locally constrained by several side-chains. *Corridor* is a similar problem, but with a longer and very narrow exit pathway, i.e. more geometrically constrained than *Passage*. In *Roundabout*, a protein goes around another one in an empty space, thus involving the weakest geometrical constraints, but the longest distance to cover.

D. First Experiment - Speedup Measurements

The first experiment we have conducted aims at studying the speedup and scalability of the parallel versions of RRT. Tests have been carried out while considering a computational cost for the RRT expansion significantly greater that the communication cost. This is a favorable situation for parallelization using MPI (as will illustrate the results of the second experiment in Section IV-E) because the communication overhead can be outweighed by the sharing of high-cost workload-units between processes [27]. Such a situation happens when planning motions of complex systems (robots or molecules), as further discussed in Section V-D. In the present context, the expansion cost is dominated by the energy evaluation of molecular motions, which replaces simple collision detection, and exemplifies a planning problem requiring high-cost expansions.

Fig. 1 presents the results obtained when solving the three problems with the sequential RRT in its Extend version [4], and considering the aforementioned conditions (i.e. high expansion cost). Fig. 2 presents the scalability achieved by the three parallel algorithms on each problem. The OR parallel RRT always shows very poor speedup and scalability. On the other hand, the speedup achieved by the Distributed RRT and Manager-worker RRT can be really high. Differences between problems are significant, the best speedup being achieved on the most constrained problem, *Corridor*, then *Passage*, then *Roundabout*. These results are further explained in the analysis presented in Section V.

E. Second Experiment - Efficiency Measurements

In our second experiment, we analyze the evolution of the efficiency of the parallel algorithms in relation to the computational cost of an RRT expansion. In parallel programming, it is generally observed that efficiency improves as the computational cost of a process workload-unit increases wrt the communication overhead [27]. To test that, we run a controlled experiment in which we artificially increase the cost of the RRT expansion to emulate different settings. We start with a low-cost expansion setting (where motion validation is reduced to collision detection, i.e. without energy evaluation). To increase the expansion cost, we repeat t times the collision detection test in the `extend()` function. The expansion cost c can then be estimated by dividing the runtime of the sequential RRT by the number of expansion attempts. Finally, c is varied by varying t.

Fig. 3 shows how the speedup of the three algorithms scales with respect to the expansion cost, when run on 32 processors. As the number of processors is fixed, efficiency is proportional to speedup. The efficiency of the OR parallel RRT does not scale at all with respect to c: the ratio between computation and communication costs does not influence speedup. On the other hand, this ratio has a strong impact on the speedup of the Distributed RRT and Manager-worker RRT. They both achieve a very low speedup when c is low: the first point of each curve, obtained with $t = 1$, shows that in this case the parallel version is even slower than the sequential one (i.e. $S < 1$). When c increases, both algorithms show a similar and important increase in efficiency. The magnitude of this increase is strongly influenced by the studied problem: it is the greatest on the most constrained problem, *Corridor* (for which almost optimal efficiency is achieved), then *Passage*, then *Roundabout*. When c is high, making communication load insignificant compared to computation load, the efficiency reaches a plateau.

[6]http://www.mcs.anl.gov/research/projects/mpich2

[7]To facilitate the evaluation of the algorithms, we have chosen not to increase dimensionality. Increasing it would mainly raise the computational cost of the nearest neighbor search. Note that, however, the cost of this operation becomes almost dimension-independent when using projections on a lower-dimension space, without a significant loss in accuracy [28].

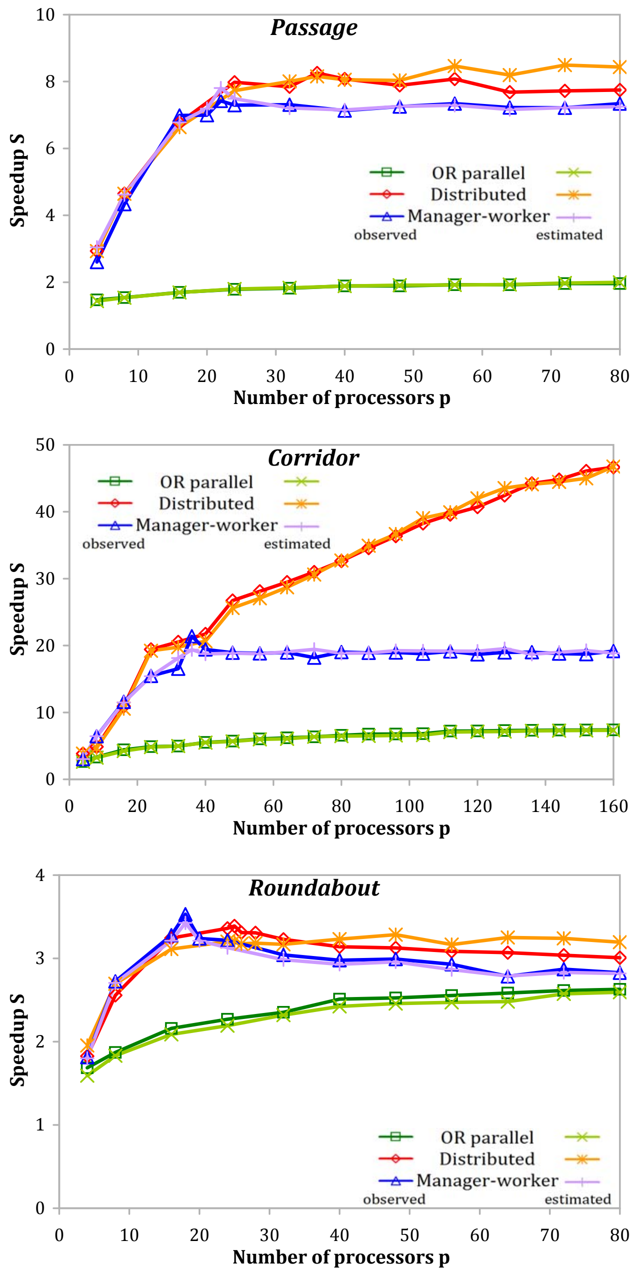


Fig. 2. Scalability of the three parallel algorithms on the *Passage*, *Corridor* and *Roundabout* problems (first experiment). Both the observed speedup and the speedup estimated by the models presented in Section V are reported.

V. Analysis of the Parallel Algorithms

The experiments we have presented provide the first clues on the differences in behavior between the parallel versions of RRT. However, the resulting speedup curves are not sufficient to understand performance variations due to the problem type, the number of processors involved or the computational cost of the RRT expansion. This is what we analyze now for each parallel algorithm.

A. OR Parallel RRT

The OR parallel RRT does not rely on sharing the computation load among processes, but on finding small-sized solutions that are faster to compute. The more processes are involved, the greater is the chance to find a solution that can be computed quickly. On average, the number of expansion attempts performed by the OR parallel RRT on p processors, $X_P(p)$, decreases with p. Similarly, the number of nodes in the tree, $N_P(p)$, decreases with p (cf. Fig. 4). If we express the parallel runtime as $T_P(p) = X_P(p) \times c$, where c is the computational cost of the RRT expansion, we get that $T_P(p)$ decreases with p. If the sequential runtime is similarly expressed as $T_S = X_S \times c$, with X_S the number of expansion attempts performed by the sequential RRT, we have:

$$S(p) = \frac{X_S}{X_P(p)} \tag{1}$$

Fig. 2 illustrates the evolution with respect to p of both the observed speedup (computed using runtimes averaged over 100 runs) and the speedup estimated by (1) (computed using values of X_S and $X_P(p)$ averaged over 100 runs). The graphs show that the estimated speedup values fit very well the observed data. Important features of the behavior of the OR parallel RRT are highlighted by (1). First, S does not depend on the expansion cost c because X_P does not depend on it. This confirms what could be intuitively deduced from the fact that the efficiency curves of the OR parallel RRT are more or less flat (cf. Fig. 3). Second, the only factor influencing the evolution of $S(p)$ is the variation of $X_P(p)$. $X_P(p)$ decreases with p, and its lower bound is the minimum number of expansion attempts required to reach a solution. This explains why $S(p)$ increases with p towards an asymptotic value S_{max} (equal to 2, 8 and 2.7 for *Passage*, *Corridor* and *Roundabout* respectively, as shown by Fig. 2). If we define the variability in runtime by the ratio of the standard deviation to the mean of the sequential runtime T_S reported in Fig. 1, we get as values 0.4, 0.8 and 0.5 for *Passage*, *Corridor* and *Roundabout* respectively. Table I shows that S_{max} is strongly positively correlated with this sequential runtime variability.

B. Distributed RRT

In the Distributed RRT, the computation load is shared among processes. It can again be expressed as $X_P(p) \times c$, where $X_P(p)$ decreases with p thanks to work sharing. On the other hand, a significant communication load is added to the global workload. However, communications happen only after a new node is built. If we assume that the tree construction is equally shared among processes, from the $N_P(p)$ nodes present in the solution, each process will have contributed $N_P(p)\,/\,p$. Furthermore, each process sends this amount of nodes to the others and receives this amount of nodes from each of the $p-1$ other processes. The communication load can thus be estimated by $2\,(p-1) \times (N_P(p)\,/\,p) \times m$, where m is the cost of sending one node between two processes. Thus: $T_P(p) = X_P(p) \times c + \frac{2\,(p-1)}{p} \times N_P(p) \times m$. This highlights the fact that the workload repartition between computation and communication mainly depends on the ratio $\frac{c}{m}$. Finally, we get:

$$S(p) = \frac{X_S \times c}{X_P(p) \times c + \frac{2\,(p-1)}{p} \times N_P(p) \times m} \tag{2}$$

Fig. 2 illustrates the evolution of both the observed speedup and the speedup estimated by (2) (computed using numbers of nodes and expansion attempts averaged over 100 runs). Here, m has been estimated by running the Distributed RRT on two processors, knowing that $T_P(2) = X_P(2) \times c + N_P(2) \times m$. The graphs show that the estimated speedup values provide a good fit to the observed speedup of the Distributed RRT. The main factor allowing $S(p)$ to increase with p is work sharing, i.e. the decrease of $X_P(p)$. Another factor increasing speedup is what we call the “OR parallel effect”: as each process performs its own sampling of the search space, when few processes are involved, on average the Distributed RRT reaches smaller-sized solutions than the sequential RRT. Fig. 4 shows that

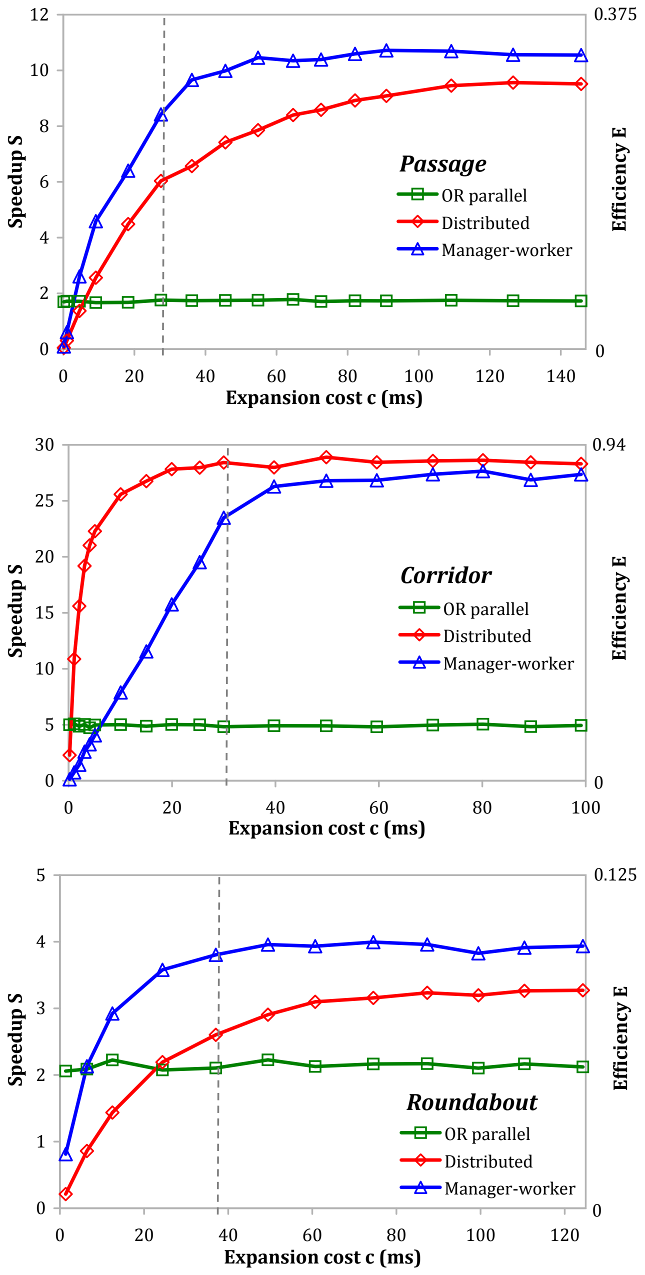


Fig. 3. Evolution of the speedup and efficiency of the parallel algorithms in relation to the computational cost of the RRT expansion, when solving the *Passage*, *Corridor* and *Roundabout* problems on 32 processors (second experiment). As a reference, the vertical line shows the expansion cost values corresponding to what is measured in the first experiment.

this phenomenon is observed mainly on problems whose sequential runtime variability is high, such as *Corridor*: in the middle graph, the curve representing $N_P(p)$ for the Distributed RRT is below the horizontal line representing N_S when p is not too high. On the other hand, an important factor hampers the increase in speedup. When an RRT is built collaboratively, a side-effect of adding more processors is to change the balance between exploration and refinement (these terms being defined as in [10]) in favor of more refinement. This translates into globally performing more expansion attempts (i.e. $p \times X_P(p)$ increases with p) and generating larger trees (i.e. $N_P(p)$ increases with p, as can be seen in Fig. 4). In other words, the overall computation load increases with p.

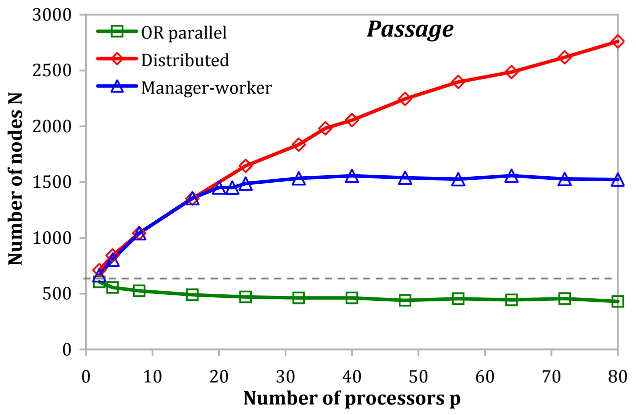
TABLE I
COMPARISON OF VARIOUS VARIABLES ASSOCIATED WITH THE PARALLEL ALGORITHMS APPLIED TO THE *Passage*, *Corridor* AND *Roundabout* PROBLEMS ON THE MARENOSTRUM PLATFORM.

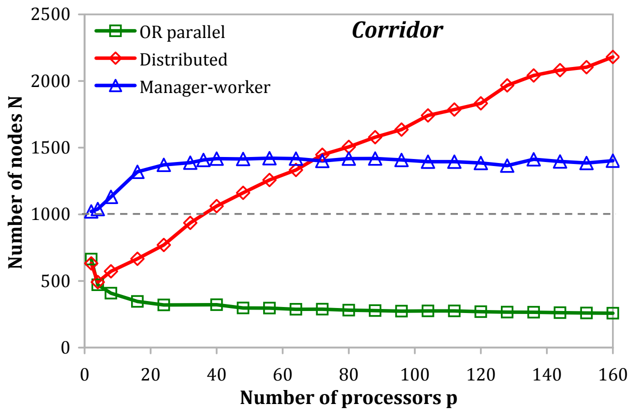
		Passage	*Corridor*	*Roundabout*
sequential runtime variability		0.4	0.8	0.5
OR parallel RRT	S_{max}	2	8	2.7
Distributed RRT	$\bar{p}$	36	> 160	25
	S_{max}	8.3	> 50	3.4
Manager-worker RRT	$\bar{p}$	22	36	18
	S_{max}	7.8	21.4	3.5

The denominator of (2) represents the workload of a single process. Even though the global (i.e. for all processes) computation load increases with p, the local (i.e. for one process) computation load $X_P(p) \times c$ decreases with p. However, the other term in the sum, representing the communication load, increases with p both because of the increase in $N_P(p)$ and also because $2\,(p-1)\,/\,p$ increases with p from 1 to 2. The decrease in computation load seems to dominate, since Fig. 2 mainly shows an increase in speedup with p for the Distributed RRT. However, it appears from the least constrained problem, *Roundabout*, that when p becomes too high the speedup starts to decrease slightly. The optimal observed speedup S_{max} is 8.3 and 3.4 for *Passage* and *Roundabout*, and seems to be greater than 50 for *Corridor* (cf. Fig. 2). It is achieved for an optimal value of p, denoted by $\bar{p}$, equal to 36 and 25 for *Passage* and *Roundabout*, and greater than 160 for *Corridor* (cf. Fig. 2). We also observe in Table I that $\bar{p}$ and S_{max} are strongly positively correlated: the more processes can collaborate without increasing too much the refinement, the higher S_{max} is. The magnitude of the increase in refinement can be observed through the magnitude of the increase in $N_P(p)$ with p (cf. Fig. 4). It appears that problems characterized by weak geometrical constraints, such as *Roundabout*, are more sensitive to this issue, leading to poor speedup. The increase in refinement also has a strong impact on the evolution of the efficiency of the Distributed RRT. For problems characterized by strong geometrical constraints, such as *Corridor*, the efficiency scales better with respect to the expansion cost c (cf. Fig. 3).

C. *Manager-Worker RRT*

In the Manager-worker RRT, each expansion attempt is preceded by a communication from the manager to a worker, and each successful expansion is followed by a communication from a worker to the manager. Since the message sent after a failed expansion is empty, it can be ignored. In the trivial case of the manager using a single worker, communication and computation cannot be overlapped, and thus $T_P(2) = (X_P(2) + N_P(2)) \times m + X_P(2) \times c$, where c is the expansion cost and m the cost of sending a message. Running tests on two processors and using this formula allowed us to estimate m. If more workers are available, two cases should be considered. First, if communication is more costly than computation (i.e. $m > c$) the manager cannot use more than two workers at a time: while the manager sends some data to a worker, the other worker has already finished its computation. In that case, we have $T_P(p) = (X_P(p) + N_P(p)) \times m > T_S$, and parallelization is therefore useless. Second, if $c > m$, more than two workers can be used, but the manager is still a potential bottleneck, depending on the ratio between c and m. On a given problem and for a given value of c, at most $\bar{p}$ processors can be used, and thus the number of workers effectively used is $min(p-1, \bar{p}-1)$. Assuming the computation





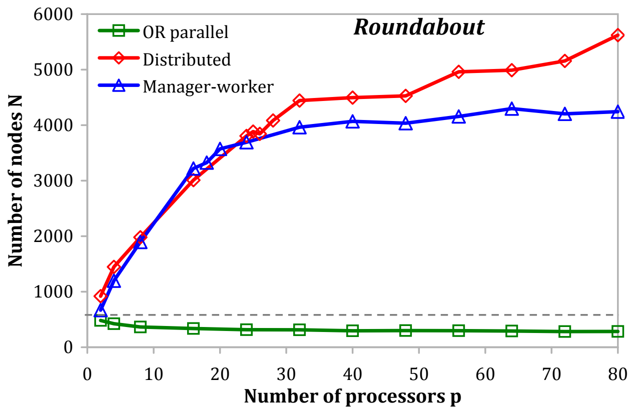


Fig. 4. Evolution of the number of nodes in the trees produced by the parallel algorithms in relation to the number of processors, when solving the *Passage*, *Corridor* and *Roundabout* problems (first experiment). The horizontal line represents the number of nodes generated by the sequential RRT.

load is equally shared among these workers, we have:

$$S(p) = \frac{X_S \times c}{(X_P(p) + N_P(p)) \times m + \frac{X_P(p)\times c}{min(p-1,\bar{p}-1)}} \quad (3)$$

Again, the speedup values estimated by (3) show a very good fit to the observed speedup of the Manager-worker RRT (cf. Fig. 2). The model provided by (3) explains the evolution of the speedup of the Manager-worker RRT with respect to p and c. When $p \leq \bar{p}$, $S(p)$ increases with p thanks to work sharing among workers. However, when $p > \bar{p}$, increasing p does not allow using more workers. Therefore, $S(p)$ reaches a plateau around a value S_{max} equal to 7.8, 21.4 and 3.5 for *Passage*, *Corridor* and *Roundabout* (cf. Fig. 2). $\bar{p}$ is given by the value of p for which $S(p)$ reaches S_{max}: it is equal to 22, 36 and 18 for *Passage*, *Corridor* and *Roundabout* (cf. Fig. 2). Obviously, S_{max} is strongly positively correlated with $\bar{p}$ (cf. Table I). The second experiment shows that, on a given problem, $\bar{p}$ increases with c: the less significant the communication cost is compared to the expansion cost, the more workers can be used. This explains why we observe on Fig. 3 that S increases with c at first, and then reaches a plateau: when $\bar{p}$ reaches 32 (the number of processors used in the experiment), S cannot increase anymore. Contrary to the Distributed RRT, the Manager-worker RRT does not benefit from the "OR parallel effect" (in Fig. 4 the curve of $N_P(p)$ is never below the horizontal line representing N_S). As a consequence, the Manager-worker RRT shows a lower speedup than the Distributed RRT on problems with a high variability in sequential runtime, such as *Corridor* (cf. Fig. 2). Besides, it suffers from the increase in refinement, which translates into $X_P(p)$ and $N_P(p)$ increasing with p, when $p \leq \bar{p}$ (cf. Fig. 4). Again, problems characterized by weak geometrical constraints, such as *Roundabout*, are more sensitive to the issue.

TABLE II
RESULTS OBTAINED BY THE PARALLEL ALGORITHMS ON CACAO.

		Passage	*Corridor*	*Roundabout*
OR parallel RRT	S_{max}	2	8	2.4
Distributed RRT	$\bar{p}$	22	> 160	28
	S_{max}	6.3	> 65	2.7
Manager-worker RRT	$\bar{p}$	25	31	23
	S_{max}	8.8	23.5	3.2

D. Discussion

To evaluate the influence of the system architecture, we have also performed the two experiments presented in Section IV on another parallel platform, Cacao, available in our laboratory. Cacao is a small cluster platform composed of 24 HP servers, each including two 64-bit quad-core processors at 2.66 GHz, connected by a 10 Gbit/s InfiniBand switch, and using OpenMPI. Our objective was to assess (i) the influence of the used architecture on the performance of the parallel algorithms and (ii) the goodness-of-fit of the models provided by (1), (2) and (3). First, we observe that the models are robust and also provide good estimations of the speedup achieved on Cacao. Second, the results obtained on Cacao and reported in Table II are very similar in terms of speedup and efficiency to those obtained on MareNostrum (cf. Table I). Speedup values reached by the OR parallel RRT are the same on both architectures because it involves no communication. The Distributed RRT is more impacted by the choice of the architecture than the Manager-worker RRT because its "n to n" communication scheme makes it more sensitive to the level of optimization of the MPI communications. As a result, when communications are less efficient (as has been observed on Cacao) the Distributed RRT can be outperformed by the Manager-worker RRT on less constrained problems (such as *Passage* and *Roundabout*) characterized by a low variability in sequential runtime.

One may wonder whether the Manager-worker RRT could be improved by assigning workers batches of multiple expansion attempts instead of single ones. We have evaluated this idea and obtained mixed results. The drawback of this variant is that it further worsens the main hindrance affecting the Manager-worker RRT, namely the loss of balance between refinement and exploration. If k is the size of a batch of expansion attempts, we observe that X_P and N_P increase with k. On problems for which the success rate of an RRT expansion is high ($N_S/X_S = 1/3$ for *Passage* and $1/2$ for *Roundabout*) this makes the speedup of this variant worse than that of the regular Manager-worker RRT, even with very low values of k. Nevertheless, we have noted that using batches of expansions slightly increases speedup on the *Corridor* problem, where the success rate of an RRT expansion is much lower ($N_S/X_S = 1/50$), except when k becomes too high.

Algorithms involving the construction of several independent RRTs can directly benefit from this work. For example, in the variant of the bidirectional-RRT [4] where both trees are extended toward the same random configuration, processes can be separated in two groups getting random configurations from an extra process. More sophisticated variants of RRT, such as ML-RRT [13] or T-RRT [29], can be parallelized using the proposed schemes as such. Similar sampling-based tree planners, such as RRT* [30] or the one based on the idea of *expansive space* [31], can also benefit from this work. Regarding the latter, since the `propagate` function is the exact counterpart of the `extend` function of RRT, it could be parallelized exactly in the same way as RRT. On the other hand, parallelizing RRT* would be much more involved, except for the OR parallel version. Besides new vertices, messages exchanged between processes should also include added and removed edges, which would increase the communication load. This could be balanced in the Distributed version by the fact that the expansion is more costly in RRT* than in RRT. However, since one RRT* expansion intertwines operations requiring or not access to the tree, a Manager-worker RRT* would likely be not very efficient.

An important question is about the generalizability of this work. Our results show that problems for which the variability in sequential runtime is high can benefit from the OR parallel RRT. Furthermore, problems for which the computational cost of an RRT expansion is high can benefit from the Distributed RRT and the Manager-worker RRT. Knowing that the communication cost on the parallel platforms we have used is about 1 ms, our results indicate that the computational cost of the RRT expansion has to be at least one order of magnitude greater. On the examples reported here, a good speedup is achieved for an RRT expansion cost greater than 25 ms. In the context of robot path planning, high-cost expansions may occur in various situations. The first one is the case of high geometric complexity. For example, on the *flange* benchmark from [32], the computational cost of the RRT expansion is about 27 ms, and on the *exhaust disassembly* problem from [33], it is about 28 ms on the KineoWorks™ platform. High-cost expansions may also occur for problems under kinodynamic constraints requiring a dynamic simulator to be used for the expansion [18]. Another situation is when path planning is performed on constraint manifolds embedded in higher-dimensional ambient spaces [34]. This is especially costly for complex systems such as closed-chain mechanisms. For example, on a problem from [35] where the *Justin* robot has to transport a tray in a cluttered environment, this cost is about 120 ms. Even more complex examples are task-based motion planning problems involving humanoid robots with dynamic constraints [36], [37]. For example, on a problem from [37] where two *HRP-2* robots have to collaboratively transport a table, the expansion cost is greater than one second. The expansion costs reported above indicate that these robotic examples would yield similar (or even higher) speedup than those we have analyzed. This illustrates that a large class of practical problems with complex environments and robot systems could potentially benefit for an MPI-based parallelization of RRT.

VI. Conclusion

We have proposed three parallel versions of RRT, designed for distributed-memory architectures using MPI: OR parallel RRT, Distributed RRT and Manager-worker RRT. Our OR parallel RRT is similar to the one in [14] and to those developed for shared memory [15]. Our Distributed RRT and Manager-worker RRT are the counterparts for distributed memory of the AND (or embarrassingly parallel) RRT [14], [15]. The experiments presented in this paper show that parallelizing RRT with MPI can provide substantial performance improvement on problems for which the computational cost of the RRT expansion is significantly higher than the cost of a communication. Furthermore, the empirical results and the performance analysis reveal that the best parallelization scheme depends on the studied problem, the computational cost of the RRT expansion and the parallel architecture.

The Distributed RRT and Manager-worker RRT provide a good speedup, except on problems characterized by weak geometrical constraints. In that case, they suffer from an important increase in refinement (vs. exploration) that translates into a significant increase of the overall computation and communication loads. On problems showing a low variability in sequential runtime, depending on the used architecture, the Manager-worker RRT can outperform the Distributed RRT. On the other hand, if the sequential runtime variability is high, the Distributed RRT greatly outperforms the Manager-worker RRT thanks to the "OR parallel effect".

Based on the results we have obtained, and as future work, we plan to improve the parallel schemes presented here. A limitation of the Distributed RRT is that it could suffer from memory-overhead issues because each process maintains its own tree. To address this point, we plan to better exploit the architecture of cluster platforms, by combining message passing with multi-threading, and allowing the processes sharing the same memory to work on a common tree. In the Manager-worker RRT, to avoid seeing the manager becoming a bottleneck, a hierarchical approach involving several managers could be developed. As part of our future work, we also plan to investigate approaches combining the three paradigms. For example, integrating the OR parallel RRT into the Manager-worker RRT could allow it to perform better on problems showing a great variability in sequential runtime. Instead of parallelizing RRT itself, we could also parallelize its most computationally expensive components, such as the collision detection, as is done in [19].

References

[1] D. Devaurs, T. Siméon, and J. Cortés, "Parallelizing RRT on distributed memory architectures," in *Proc. IEEE Int. Conf. Robot. Autom.*, 2011, pp. 2261–2266.

[2] H. Choset, K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun, *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, 2005.

[3] S. M. LaValle, *Planning Algorithms*. Cambridge University Press, 2006.

[4] S. M. LaValle and J. J. Kuffner, "Rapidly-exploring random trees: progress and prospects," in *Algorithmic and Computational Robotics: New Directions*. A K Peters, 2001, pp. 293–308.

[5] ——, "Randomized kinodynamic planning," *Int. J. Robot. Research*, vol. 20, no. 5, pp. 378–400, 2001.

[6] J. Cortés and T. Siméon, "Sampling-based motion planning under kinematic loop-closure constraints," in *Algorithmic Foundations of Robotics VI*. Springer-Verlag, 2005, pp. 75–90.

[7] M. S. Branicky, M. M. Curtiss, J. A. Levine, and S. B. Morgan, "RRTs for nonlinear, discrete, and hybrid planning and control," in *Proc. IEEE Conf. Decision Contr.*, 2003, pp. 657–663.

[8] C. Belta, J. M. Esposito, J. Kim, and V. Kumar, "Computational techniques for analysis of genetic network dynamics," *Int. J. Robot. Research*, vol. 24, no. 2-3, pp. 219–235, 2005.

[9] J. Cortés, T. Siméon, V. Ruiz de Angulo, D. Guieysse, M. Remaud-Siméon, and V. Tran, "A path planning approach for computing large-amplitude motions of flexible molecules," *Bioinformatics*, vol. 21 (Suppl. 1), pp. i116–i125, 2005.

[10] L. Jaillet, A. Yershova, S. M. LaValle, and T. Siméon, "Adaptive tuning of the sampling domain for dynamic-domain RRTs," in *Proc. IEEE/RSJ Int. Conf. Intell. Robot. Syst.*, 2005, pp. 2851–2856.

[11] A. Yershova and S. M. LaValle, "Improving motion planning algorithms by efficient nearest-neighbor searching," *IEEE Trans. Robot.*, vol. 23, no. 1, pp. 151–157, 2007.

[12] P. Cheng, E. Frazzoli, and S. M. LaValle, "Improving the performance of sampling-based planners by using a symmetry-exploiting gap reduction algorithm," in *Proc. IEEE Int. Conf. Robot. Autom.*, 2004, pp. 4362–4368.

[13] J. Cortés, L. Jaillet, and T. Siméon, "Disassembly path planning for complex articulated objects," *IEEE Trans. Robot.*, vol. 24, no. 2, pp. 475–481, 2008.

[14] S. Carpin and E. Pagello, "On parallel RRTs for multi-robot systems," in *Proc. Int. Conf. Italian Assoc. Artif. Intell.*, 2002, pp. 834–841.

[15] I. Aguinaga, D. Borro, and L. Matey, "Parallel RRT-based path planning for selective disassembly planning," *Int. J. Adv. Manufact. Technol.*, vol. 36, no. 11-12, pp. 1221–1233, 2008.

[16] D. Devalarazu and D. W. Watson, "Path planning for altruistically negotiating processes," in *Proc. Int. Symp. Collab. Technol. Syst.*, 2005, pp. 196–202.

[17] D. Henrich, "Fast motion planning by parallel processing - a review," *J. Intell. Robot. Syst.*, vol. 20, no. 1, pp. 45–69, 1997.

[18] I. A. Şucan and L. E. Kavraki, "Kinodynamic motion planning by interior-exterior cell exploration," in *Algorithmic Foundations of Robotics VIII*. Springer-Verlag, 2010, pp. 449–464.

[19] J. Bialkowski, S. Karaman, and E. Frazzoli, "Massively parallelizing the RRT and the RRT*," in *Proc. IEEE/RSJ Int. Conf. Intell. Robot. Syst.*, 2011, pp. 3513–3518.

[20] N. M. Amato and L. K. Dale, "Probabilistic roadmap methods are embarrassingly parallel," in *Proc. IEEE Int. Conf. Robot. Autom.*, 1999, pp. 688–694.

[21] D. Challou, D. Boley, M. Gini, V. Kumar, and C. Olson, "Parallel search algorithms for robot motion planning," in *Practical Motion Planning in Robotics: Current Approaches and Future Directions*. Wiley & Sons, 1998, pp. 115–131.

[22] S. Caselli and M. Reggiani, "ERPP: an experience-based randomized path planner," in *Proc. IEEE Int. Conf. Robot. Autom.*, 2000, pp. 1002–1008.

[23] E. Plaku and L. E. Kavraki, "Distributed sampling-based roadmap of trees for large-scale motion planning," in *Proc. IEEE Int. Conf. Robot. Autom.*, 2005, pp. 3868–3873.

[24] M. Strandberg, "Augmenting RRT-planners with local trees," in *Proc. IEEE Int. Conf. Robot. Autom.*, 2004, pp. 3258–3262.

[25] M. Otte and N. Correll, "Any-com multi-robot path-planning: maximizing collaboration for variable bandwidth," in *Proc. Int. Symp. Distrib. Auton. Robot. Syst.*, 2010.

[26] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, 2nd ed. Pearson Education, 2003.

[27] I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, 1995.

[28] E. Plaku and L. E. Kavraki, "Quantitative analysis of nearest-neighbors search in high-dimensional sampling-based motion planning," in *Algorithmic Foundations of Robotics VII*. Springer-Verlag, 2008, pp. 3–18.

[29] L. Jaillet, J. Cortés, and T. Siméon, "Sampling-based path planning on configuration-space costmaps," *IEEE Trans. Robot.*, vol. 26, no. 4, pp. 635–646, 2010.

[30] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *Int. J. Robot. Research*, vol. 30, no. 7, pp. 846–894, 2011.

[31] D. Hsu, R. Kindel, J.-C. Latombe, and S. Rock, "Randomized kinodynamic motion planning with moving obstacles," *Int. J. Robot. Research*, vol. 21, no. 3, pp. 233–255, 2002.

[32] S. Rodríguez, X. Tang, J.-M. Lien, and N. M. Amato, "An obstacle-based rapidly-exploring random tree," in *Proc. IEEE Int. Conf. Robot. Autom.*, 2006, pp. 895–900.

[33] S. Dalibard and J.-P. Laumond, "Control of probabilistic diffusion in motion planning," in *Algorithmic Foundations of Robotics VIII*. Springer-Verlag, 2010, pp. 467–481.

[34] D. Berenson, S. Srinivasa, and J. Kuffner, "Task space regions: a framework for pose-constrained manipulation planning," *Int. J. Robot. Research*, vol. 30, no. 12, pp. 1435–1460, 2011.

[35] M. Gharbi, "Motion planning and object manipulation by humanoid torsos," Ph.D. dissertation, LAAS-CNRS, Toulouse, 2010.

[36] S. Dalibard, A. Nakhaei, F. Lamiraux, and J.-P. Laumond, "Whole-body task planning for a humanoid robot: a way to integrate collision avoidance," in *Proc. IEEE-RAS Int. Conf. Humanoid Robot.*, 2009, pp. 355–360.

[37] K. Bouyarmane and A. Kheddar, "Static multi-contact inverse problem for multiple humanoid robots and manipulated objects," in *Proc. IEEE-RAS Int. Conf. on Humanoid Robot.*, 2010, pp. 8–13.