Parallelizing RRT on large-scale distributed-memory architectures

Didier Devaurs, Thierry Simeon, Juan Cortés

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 (RRT) algorithm on large-scale distributed-memory architectures, using the message passing interface. We compare three parallel versions of RRT based on classical parallelization schemes. We evaluate them on different motion-planning problems and analyze the various factors influencing their performance.**

***Index Terms*—Distributed memory, message passing, parallel algorithms, path planning, rapidly-exploring random tree (RRT).**

I. Introduction

Due to a wide range of applications, sampling-based path planning has benefited from considerable research effort. It has proven to be an effective framework for various problems in domains as diverse 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) is a popular sampling-based algorithm applied to single-query path-planning problems [2]. It is suited to solve robot motion-planning problems that involve holonomic, nonholonomic, kinodynamic, or kinematic loop-closure constraints [2]–[4]. It is also applied to planning in discrete spaces or for hybrid systems [5]. In computational biology, it is used to analyze genetic network dynamics [6] or protein–ligand interactions [7]. However, when applied to complex problems, the growth of an RRT can become computationally expensive [8]–[11]. Some techniques have been proposed to improve the efficiency of RRT by controlling the sampling domain [8], reducing the complexity of the nearest neighbor search [9], or using gap reduction techniques [10].

Our objective is to further investigate improvements to RRT by exploiting speedup from parallel computation. Some results have been obtained in that direction (see Section II). Nevertheless, the existing study considers mainly shared-memory architectures and small-scale parallelism, up to 16 processors [12]–[14]. In this study, we are interested in what can be achieved by larger scale parallelism. We focus on parallelizing RRT on distributed-memory architectures, which requires the use of the message passing interface (MPI).

In this paper, we compare three parallel versions of RRT based on classical parallelization schemes: OR parallel RRT, distributed RRT, and manager–worker RRT. Besides the algorithms themselves, we also present the main technicalities involved in their development (see Section III). Our contribution focuses on evaluating these algorithms on several motion-planning problems and showing their differences in behavior (see Section IV). We also analyze their performance in order to understand the impact of several characteristics of the studied problems (see Section V). Our evaluation shows that parallelizing RRT with MPI can provide substantial performance improvement in two cases: 1) problems for which the variability in sequential runtime is high can benefit from the OR parallel RRT and 2) problems for which the computational cost of an RRT expansion is high can benefit from the distributed RRT and the manager–worker RRT. Even though this is not true for most academic motion-planning benchmarks, many robotic examples yield computationally expensive RRT expansions and can, thus, benefit from these parallel algorithms (see Section IV-F).

II. Related Work

A. Parallel Motion-Planning

The idea of improving motion-planning performance using parallel computation is not new. A survey of some early work proposes a classification scheme to review various motion-planning approaches and related parallel processing methods [15]. A more recent trend is to exploit the multicore technology available on many of today's PCs, which allows having multiple threads collaboratively solving a problem [16]. Another recent trend consists of using shared-memory models on many-core graphics processing units (GPUs) [17].

Among 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" [18]. The massive inherent parallelism of the basic PRM algorithm enables to reach a significant speedup, even with simple parallelization 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 providing a parallel formulation for the randomized path planner (RPP) [19]. Its principle is to have several processes running the same sequential randomized algorithm, where each one tries to build its own solution. The first process to reach a solution reports it and broadcasts a termination message. The idea is to minimize computing time by finding a small-sized solution. Despite its simplicity, this paradigm has been successfully applied to other randomized algorithms [20].

A more sophisticated approach is the *scheduler–processor scheme* that was developed to distribute the computation of the sampling-based roadmap of trees (SRT) algorithm [21]. In this scheme, the scheduler coordinates the processors constructing the milestones, which can be

This work was supported in part by 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) and in part by the French National Agency for Research under project GlucoDesign and by the European Community under Contract ICT 287617 "ARCAS."

The authors are with CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France, and also with Univ de Toulouse, LAAS, F-31400 Toulouse, France (e-mail: devaurs@laas.fr; nic@laas.fr; jcortes@laas.fr).

Digital Object Identifier 10.1109/TRO.2013.2239571

Algorithm 1: OR parallel RRT

```
   input : the configuration space C, the root q_init
   output: the tree T
1  T ← initTree(q_init)
2  while not stopCondition(T) or received(endMsg) do
3  |  q_rand ← sampleRandomConfiguration(C)
4  |  q_near ← findBestNeighbor(T, q_rand)
5  |  q_new ← extend(q_near, q_rand)
6  |  if q_new ≠ null 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 ← 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 ← sampleRandomConfiguration(C)
6  |  q_near ← findBestNeighbor(T, q_rand)
7  |  q_new ← extend(q_near, q_rand)
8  |  if q_new ≠ null then
9  |  |  addNewNodeAndEdge(T, q_near, q_new)
10 |  |  broadcast(nodeData(q_new, q_near))
11 if stopCondition(T) then
12 |  broadcast(endMsg)
```

RRTs or expansive space trees (ESTs), and the edges linking them. More generally, an approach based on the growth of several independent trees, such as the rapidly exploring random forest of trees [6] or RRTLocTrees [22], can lead to a straightforward parallelization. However, the focus of this paper lies elsewhere: Our objective is to provide a parallel version of the basic (single-tree) RRT algorithm. Furthermore, this study is not about parallelizing subroutines of RRT, as is done for collision detection in [17], nor about parallelizing specific variants of RRT, as is done for the *any-time RRT* in [23]. Finally, we aim to reduce the runtime of RRT and not to improve the quality of the paths it returns.

B. Parallel RRT

Only little work relates to parallelizing RRT [12]–[14]. The first approach applies the simple *OR parallel* and *embarrassingly parallel* paradigms, and a combination of both [12]. To benefit from the simplicity of the shared-memory model, the embarrassingly parallel algorithm is run on a single symmetrical multiprocessor node of a multinodes parallel computer. The only communication involved is a termination message that is broadcast when a solution is reached, and 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 central processing unit in [13], where they are renamed *OR* and *AND* implementations. In the Open Motion-Planning Library (OMPL), the AND paradigm is implemented via multithreading, and thus, for shared memory [24].

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 [14], 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, or how the tree is reconstructed from the parts built by the agents.

III. Parallelization of the Rapidly-Exploring Random Tree

For scalability purposes, we have parallelized RRT on distributed-memory architectures, using the message passing paradigm, one of the most widespread approaches in parallel programming. Since this paradigm imposes no requirement on the underlying hardware and requires us to explicitly parallelize algorithms, it enables a wide portability: any algorithm developed following this approach can also run on shared memory. 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 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 a few tasks synchronize to perform interactions as well.

A. OR Parallel RRT

The simplest way to parallelize RRT is to apply the OR parallel paradigm. Algorithm 1 shows the *OR parallel RRT*, as defined in [12]. 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 nonblocking 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 nonblocking receiving operation. Finally, in the case of several processes reaching a solution at the same time, the program ends with a collective operation for them to synchronize and agree on which one should report its solution. Note that communications are negligible in the total runtime.

B. Collaborative Building of a Single RRT

Instead of constructing several RRTs concurrently, another possibility is to have all processes collaborating to build a single RRT. Parallelization is then achieved by partitioning the building task into subtasks assigned to the various processes. We propose two ways of doing so, based on classical decomposition techniques. 1) Since the construction of an RRT consists of exploring a search space, we can use an *exploratory decomposition* [25]. 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 [26]. In the RRT algorithm, two kinds of subtasks can be distinguished: the ones that require to access 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

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 ← 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 ← sampleRandomConfiguration(C)
7          q_near ← findBestNeighbor(T, q_rand)
8          w ← 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 ← extend(q_near, q_rand)
15         if q_new ≠ null then
16             send(nodeData(q_new, q_near), mgr)
```

master–slave) scheme as the dynamic and centralized task-scheduling strategy: the manager maintains the tree, and the workers have no access to it.

1) Distributed RRT: Algorithm 2 presents our *distributed RRT*. 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) and, if so, adds them to its local copy of the tree (line 4). Then, it performs an expansion attempt (lines 5–7). If it succeeds (line 8), the process adds the new node to its local tree (line 9) and broadcasts the node (line 10). The addition of all the received nodes before attempting an expansion ensures that every process works with the most up-to-date state of the tree. 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 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 nonblocking MPI_Receive routines. That way, the `received(nodeData)` test (line 3) is performed by checking the status of the request associated with a nonblocking 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, we have to deal with the case of several processes reaching a solution at the same time. Finally, a universally unique identifier (UUID) is associated with each node to provide processes with a homogeneous way of referring to the nodes.

2) Manager–Worker RRT: Algorithm 3 introduces our *manager–worker RRT*. It contains the code of the manager (lines 2–10) and of the workers (lines 12–16). The manager is the only process accessing the tree. It delegates the expansion attempts to workers. The expansion is generally the most computationally expensive stage in the RRT construction because it involves motion simulation and validation. The manager could also delegate the sampling step, but this would be worthless 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 outweigh any potential benefit.

At each iteration of the construction loop (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 to the worker the data needed to perform an expansion attempt (line 9). Finally, when a stopping condition is reached, it broadcasts a termination message (line 10). Workers remain active until they receive this message (line 12), but 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 succeeds (line 15), it sends the new node to the manager (line 16).

Contrary to the previous ones, this algorithm does not require nonblocking receiving operations to broadcast 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 received message determines its next action: stop or attempt 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 subtask 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 new nodes when the expansion attempt succeeds and the other containing no data used otherwise. Second, on the manager side, two matching receiving operations are implemented via nonblocking MPI_Receive routines, allowing us to use MPI_Wait routines if necessary. This also enables us to implement the `received(nodeData)` test with an MPI_Test routine. These nonblocking receiving operations are initiated before entering the **while** loop, and reinitiated 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, thus avoiding the need for UUIDs.

C. Implementation Framework

Since the sequential implementation of RRT we wanted to parallelize was written in C++ and since MPI is 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 decided to exploit the higher level abstraction provided by the Boost.MPI library. Coupled with the Boost.Serialization library, it enables processes to easily exchange class instances, making the tasks of gathering, packing and unpacking the underlying data transparent to the programmer. We also used the implementation of UUIDs provided by the Boost library.

IV. Experimental Setup

Before presenting the results of the experiments, we 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

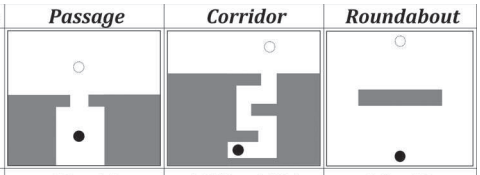
Problem name		*Passage*	*Corridor*	*Roundabout*
Problem type				
Sequential RRT	T_S (s)	48 ± 18	1250 ± 1001	38 ± 18
	N_S	644 ± 119	1027 ± 695	655 ± 336
	X_S	1807 ± 690	45374 ± 40855	1130 ± 456

Fig. 1. Schematic representation of the configuration spaces of the planning problems and results obtained with the sequential RRT including molecular energy computation (cf. Section IV-D). Average values over 100 runs (and standard deviation) are given for the sequential runtime, T_S (in seconds), the number of nodes in the tree N_S, and the number of expansion attempts, X_S.

performed and report general results. A detailed analysis of the performance of the algorithms will be the focus of Section V.

A. Performance Metrics

Aimed at assessing the performance gain achieved by a parallel algorithm run on p processors, the *speedup* S is defined as the ratio of the sequential runtime to the parallel runtime: $S(p) = T_S \,/\, T_P\,(p)$ [25], [26]. The parallel runtime $T_P\,(p)$ is measured on a parallel computer, using p processors, and the sequential runtime T_S is measured on a single processor of this 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. 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: $E(p) = S(p) \,/\, p$ [25], [26].

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.

C. Motion-Planning Problems

We have evaluated the parallel algorithms on three motion-planning problems involving molecular models, using the molecular motion-planning toolkit we are currently developing [7]. However, 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).[1] They are characterized by different configuration-space topologies (cf. Fig. 1). *Passage* is a protein-ligand exit problem: A ligand exits the active site of a protein through a relatively short and large pathway locally constrained by several side-chains. *Corridor* is a similar problem, but with a longer and narrower 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. For more details on these examples, see [7] and [28].

[1] Having a common dimensionality across examples facilitates the evaluation of the algorithms. Increasing dimensionality would mainly raise the computational cost of the nearest neighbor search. Note that, however, this cost becomes almost dimension independent when using projections on a lower dimensional space, without a significant loss in accuracy [27].

D. First Experiment—High Expansion Cost

Our first experiment aims at assessing the speedup achieved by the parallel variants of RRT. The tests are carried out while considering a computational cost for the RRT expansion that is significantly greater than the communication cost. This is a favorable situation for an MPI-based parallelization (as the results reported in Section IV-E will illustrate) because the communication overhead is outweighed by the sharing of high-cost workload units between processes [26]. Such a situation happens when planning motions of complex systems (robots or molecules), as discussed in Section IV-F. In the present context, the expansion cost is dominated by the energy evaluation of molecular motions, which replaces simple collision detection. This exemplifies the case of high-cost expansions.

Fig. 1 presents the results obtained with the sequential RRT in its Extend version [2] when molecular energy is computed. Fig. 2 shows the speedup achieved by the parallel algorithms on each problem. The OR parallel RRT always shows a poor speedup. On the other hand, the speedup achieved by the distributed RRT and the 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*, and then *Roundabout*. These results are further explained in the analysis presented in Section V.

E. Second Experiment—Variable Expansion Cost

In our second experiment, we study how the speedup achieved by the parallel algorithms evolves in relation to the computational cost of an RRT expansion. In parallel programming, speedup generally improves as the computational cost of a process workload unit increases w.r.t. the communication overhead [26]. To test that, we run a controlled experiment in which we artificially increase the cost of the RRT expansion. 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 c, we repeat t times the collision detection test in the *extend*() function. Note that we estimate c by dividing the sequential runtime by the number of expansion attempts. Finally, c is varied by varying t.

Fig. 3 shows how the speedup and efficiency achieved by the parallel algorithms vary with respect to the expansion cost c, when run on 32 processors. As the number of processors is fixed, efficiency is proportional to speedup. The speedup of the OR parallel RRT does not change with c. In other words, it is not influenced by the ratio between computation and communication costs. On the other hand, this ratio strongly impacts 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 speedup. The magnitude of this increase is strongly influenced by the 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 with computation load, the speedup reaches a plateau.

F. Robotic Examples

Our results show that the distributed and manager–worker RRT are beneficial on problems for which the computational cost of an RRT ex-

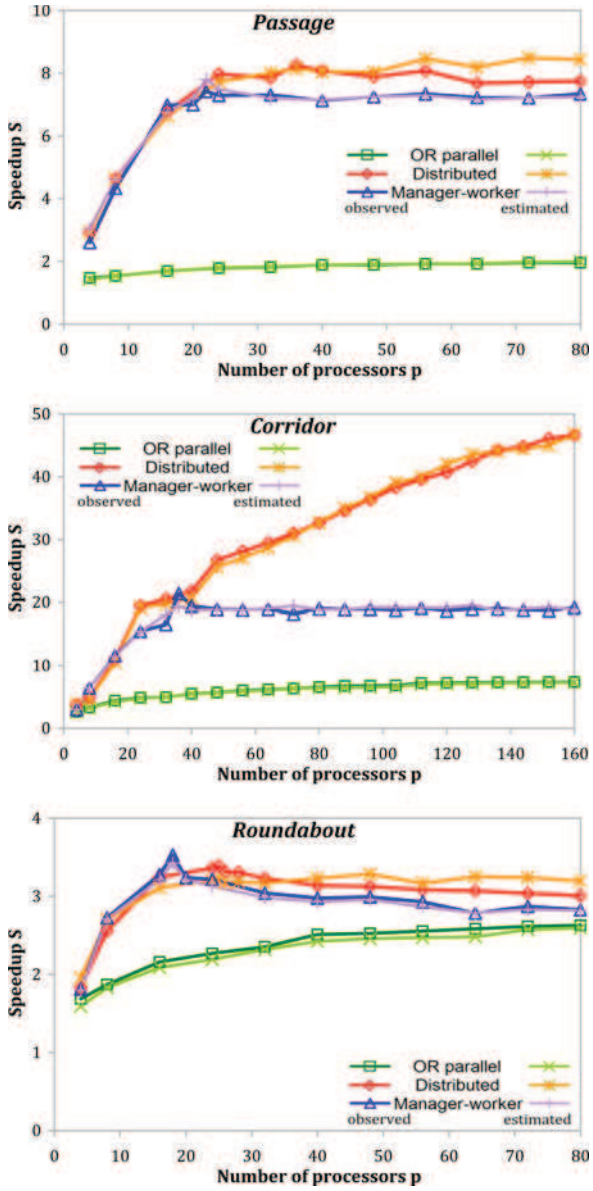


Fig. 2. Speedup (averaged over 100 runs) achieved by the parallel algorithms in relation to the number of processors on the *Passage, Corridor,* and *Roundabout* problems (first experiment). Both the observed speedup and the speedup estimated by the models presented in Section V are reported.

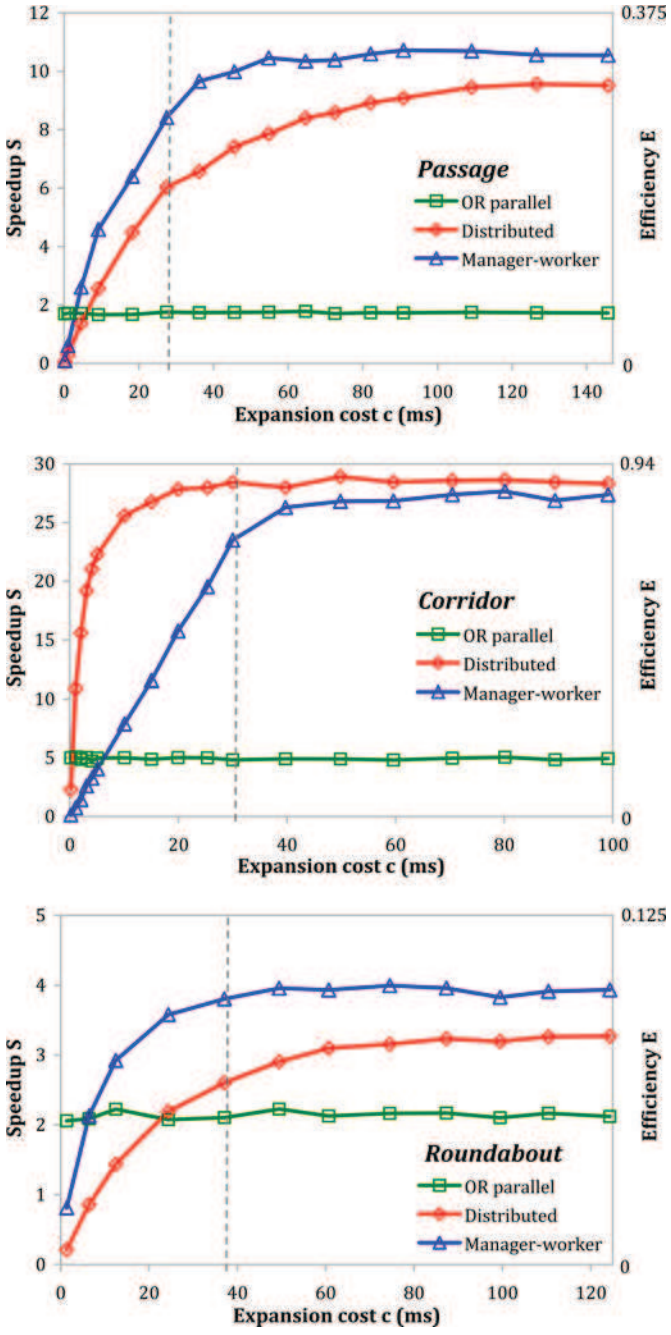


Fig. 3. Speedup and efficiency (averaged over 100 runs) of the parallel algorithms in relation to the computational cost of the RRT expansion (in milliseconds) when solving the *Passage, Corridor,* and *Roundabout* problems on 32 processors (second experiment). As a reference, the dashed vertical line shows the expansion cost value as estimated in the first experiment.

pansion c is significantly greater than the cost of a communication. The communication cost being about 1 ms on MareNostrum, we obtain a good speedup when c is greater than 25 ms (cf. Fig. 3). This means that most academic motion-planning benchmarks, such as the *Alpha* puzzle, cannot benefit from an MPI-based parallelization of RRT. Indeed, these examples often reduce motion validation to collision detection in geometrically simple scenes, leading to a fast RRT expansion. However, 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, when objects of the world are represented by large numbers of polyhedral faces. For example, c is about 27 ms on the *flange* benchmark [29] and about 28 ms on the *exhaust disassembly* problem [30], despite efficient collision detection. High-cost expansions may also occur on problems under kinodynamic constraints requiring to use a dynamic simulator [16]. Another case is when planning on constraint manifolds embedded in higher dimensional ambient spaces [31], especially with complex systems such as closed-chain mechanisms. For example, c is about 120 ms on a problem where the *Justin* robot transports a tray in a cluttered environment [32]. An even more complex case is task-based path planning involving humanoid robots with dynamic constraints [33], [34]. For example, c is greater than 1 s on a problem where two *HRP-2* robots collaboratively transport a table [34]. Due to their high expansion costs, all these examples would yield a similar or even higher speedup than those we have studied. This illustrates that a large class of practical problems involving complex environments and

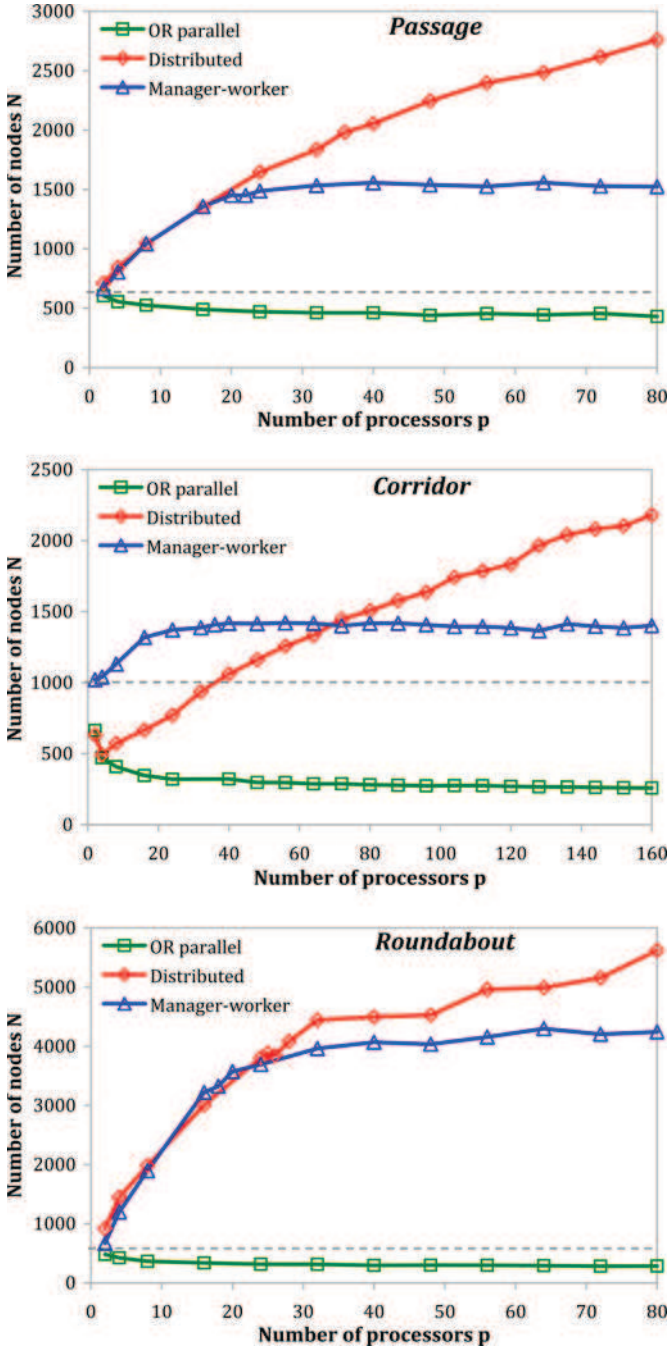


Fig. 4. Number of nodes (averaged over 100 runs) in the trees produced by the parallel algorithms in relation to the number of processors, on the *Passage*, *Corridor*, and *Roundabout* problems (first experiment). The dashed horizontal line shows the number of nodes in the trees generated by the sequential RRT.

complex robot systems can benefit from an MPI-based parallelization of RRT.

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. Nevertheless, 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 involved, the greater is the chance to find a solution quickly. On average, the number of expansions attempted by the OR parallel RRT on p processors $X_P(p)$ decreases with p. Similarly, the number of tree nodes $N_P(p)$ decreases with p (cf. Fig. 4). If we express the parallel runtime as $T_P(p) = X_P(p) \cdot c$, where c is the expansion cost, we get that $T_P(p)$ decreases with p. If the sequential runtime is similarly expressed as $T_S = X_S \cdot c$, where X_S is the number of expansions attempted by the sequential RRT, we have

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

Fig. 2 illustrates the evolution w.r.t. 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 well the observed data. Important features of the behavior of the OR parallel RRT are reflected in (1). First, S is independent from the expansion cost c because X is independent from it. This confirms what we could deduce from the fact that the efficiency curves of the OR parallel RRT are almost flat (cf. Fig. 3). Second, the only factor influencing the evolution of $S(p)$ is $X_P(p)$, which decreases with p and is lower bounded by the minimum number of expansion attempts required to reach a solution. This explains why $S(p)$ increases with p toward an asymptotic value $S_{\max}$ (equal to 2, 8, and 2.7 on *Passage*, *Corridor*, and *Roundabout*, respectively, as shown in Fig. 2). If we define the variability in sequential runtime by the ratio of the standard deviation to the mean of the runtime T_S reported in Fig. 1, we get the 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 $X_P(p) \cdot c$, where $X_P(p)$ decreases with p thanks to work sharing. A significant communication load is added to the global workload, but communications happen only after a new node is built. If we assume the tree construction is equally shared among processes, from the $N_P(p)$ tree nodes, each process will have contributed $N_P(p)/p$. Furthermore, each process sends this amount of nodes to, and receives this amount of nodes from, each of the $p-1$ other processes. The communication load can thus be estimated by $2(p-1) \cdot (N_P(p)/p) \cdot m$, where m is the cost of sending one node between two processes. Therefore, we have $T_P(p) = X_P(p) \cdot c + \frac{2(p-1)}{p} \cdot N_P(p) \cdot m$. This highlights the fact that the workload repartition between computation and communication mainly depends on the ratio $\frac{c}{m}$. Finally, we get

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

Fig. 2 illustrates the evolution w.r.t. p of both the observed speedup and the speedup estimated by (2) (computed using numbers of nodes and expansion attempts averaged over 100 runs). Knowing that

TABLE I
RESULTS OBTAINED WITH THE RRT VARIANTS ON MARENOSTRUM

		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

$T_P(2) = X_P(2) \cdot c + N_P(2) \cdot m$, we estimate m by running the distributed RRT on two processors. The graphs show that the estimated speedup provides a good fit to the observed speedup. The main factor allowing $S(p)$ to increase with p is work sharing, i.e., the decrease of $X_P(p)$. Another beneficial factor is what we call the "OR parallel effect": as each process performs its own sampling of the search space, when few processes are involved, the distributed RRT reaches smaller solutions than the sequential RRT. Fig. 4 shows that this happens 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 low. On the other hand, an important factor hampers the increase in speedup. When collaboratively building an RRT, a side effect of adding more processes is to change the balance between exploration and refinement (these terms being used as in [8]) in favor of refinement. Therefore, more expansions are attempted globally (i.e. $p \cdot X_P(p)$ increases with p), and larger trees are produced (i.e., $N_P(p)$ increases with p, as shown in Fig. 4). As a result, the overall computation load increases with p.

The denominator of (2) represents the workload of a single process. Even though the global computation load for all processes increases with p, the computation load for one process $X_P(p) \cdot c$ decreases with p. However, the communication load for one process $\frac{2(p-1)}{p} \cdot N_P(p) \cdot m$ increases with p because $N_P(p)$ increases with p and $\frac{2(p-1)}{p}$ increases with p in $[1, 2[$. The decrease in computation load seems to dominate, since Fig. 2 mainly shows an increase in speedup for the distributed RRT. However, it appears from the least constrained problem, *Roundabout*, that when p becomes too high the speedup decreases 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). Table I shows that $\bar{p}$ and $S_{\max}$ are strongly positively correlated: The more processes that can collaborate without increasing refinement too much, the higher $S_{\max}$ will be. The increase in refinement is observed through the increase in the number of nodes (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. For problems characterized by strong geometrical constraints, such as *Corridor*, the speedup scales better w.r.t. 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. Being empty, the message sent after a failed expansion can be ignored. In the trivial case of the manager using a single worker, communication and computation cannot overlap, and thus, $T_P(2) = X_P(2) \cdot c + (X_P(2) + N_P(2)) \cdot m$, where m is the cost of sending a message. We estimate m by running tests on two processors and using this formula. If more workers are available, two cases should be considered. First, if communication is more costly than computation (i.e., $m > c$), the manager can use at most two workers at a time: While it sends some data to a worker, the other worker has already finished its computation. In that case, we have $T_P(p) = (X_P(p) + N_P(p)) \cdot m > T_S$, and parallelization is useless. Second, if $c > m$, more than two workers can be used, but the manager is still a potential bottleneck depending on the ratio $\frac{c}{m}$: the less significant the communication cost compared with the expansion cost, the more workers can be used. For given values of c and m, 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 load is equally shared among workers, we have

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

The speedup estimated by (3) shows a good fit to the observed speedup of the manager–worker RRT (cf. Fig. 2). Equation (3) explains how the speedup evolves w.r.t. 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 becomes useless. 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*, respectively (cf. Fig. 2). In fact, $\bar{p}$ is 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). Moreover, the second experiment shows that $\bar{p}$ increases with c. This explains why we observe in 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 can no longer be increased. 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). Problems characterized by weak geometrical constraints, such as *Roundabout*, are more sensitive to the issue.

D. Discussion

To evaluate the influence of the architecture, we have performed the two previous experiments on another parallel platform, Cacao, available in our laboratory. Cacao is a small cluster composed of 24 HP servers including two 64-bit quad-core processors at 2.66 GHz, connected by a 10 Gbit/s InfiniBand switch, using OpenMPI. We aimed to assess i) the consistency of the performance of the parallel algorithms and ii) the goodness-of-fit of the models provided by (1)–(3). First, we observe that the models are robust and provide good estimations of the speedup achieved on Cacao. Second, the results obtained on Cacao and reported in Table II are similar to those obtained on MareNostrum (cf. Table I). The speedup of the OR parallel RRT is the same on both architectures because no communication is involved. The distributed RRT is more impacted than the manager–worker RRT by the choice of the architecture 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 observed on Cacao) the distributed RRT can be outperformed by the manager–worker RRT on less-constrained problems (such as

TABLE II
RESULTS OBTAINED WITH 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

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. Even though it should reduce communications, after evaluation this idea appears to yield mixed results. The drawback of this variant is to further worsen the main hindrance affecting the manager–worker RRT, namely the increase in refinement w.r.t. p. 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 ($N/X = 1/3$ for *Passage* and $1/2$ for *Roundabout*), using this modification reduces speedup, even with low values of k. Nevertheless, speedup increases slightly on the *Corridor* problem, where this success rate is much lower ($N/X = 1/50$), except when k becomes too high.

Algorithms building several RRTs can benefit from this work. For example, in the bidirectional-RRT variant where both trees are extended toward the same random configuration [2], processes can be separated into two groups applying our parallel algorithms, and getting random configurations from an extra process. More sophisticated variants of RRT, such as ML-RRT [11] or T-RRT [35], can be parallelized using the proposed schemes as such. Similar sampling-based tree planners, such as RRT* [36] or the one based on the idea of *expansive space* [37], can also benefit from this work. The latter can be parallelized exactly in the same way as RRT because the `propagate` function is the exact counterpart of the `extend` function of 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 higher cost of the expansion in RRT* than in RRT. However, as one RRT* expansion intertwines operations requiring or not access to the tree, a manager–worker version would not be very efficient.

VI. CONCLUSION

We have evaluated three parallel versions of RRT designed for distributed-memory architectures using MPI: OR parallel RRT, distributed RRT, and manager–worker RRT. The OR parallel RRT was first introduced in [12] and reused on shared memory in [13]. The distributed RRT and manager–worker RRT are the counterparts for distributed memory of the AND (or embarrassingly parallel) RRT used on shared memory [12], [13]. We have shown that parallelizing RRT with MPI can provide substantial performance improvement in two cases. First, problems whose variability in sequential runtime is high can benefit from the OR parallel RRT. Second, problems for which the computational cost of an RRT expansion is high can benefit from the distributed RRT and manager–worker RRT.

The empirical results and the performance analysis reveal that the best parallelization scheme depends on the studied problem, the computational cost of an RRT expansion, and the parallel architecture. The distributed RRT and manager–worker RRT provide a good speedup, except on problems with weak geometrical constraints. In that case, they suffer from an increase in refinement (versus exploration) translating into greater overall computation and communication loads. On problems showing a low variability in sequential runtime, depending on the architecture, the manager–worker RRT can outperform the distributed RRT. On the other hand, if the sequential runtime variability is high, the distributed RRT outperforms the manager–worker RRT thanks to its "OR parallel effect."

Based on these results, and as future work, we plan to improve the parallel schemes presented here. First, the distributed RRT can suffer from memory-overhead issues because each process maintains its own tree. To address this, we plan to better exploit the architecture of cluster platforms by combining message passing with multithreading and allowing the processes sharing the same memory to build a common tree. Second, in the manager–worker RRT, to avoid seeing the manager becoming a bottleneck, a hierarchical approach involving several managers can be developed. Third, we plan to investigate approaches combining several of 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 high variability in sequential runtime. Finally, instead of parallelizing RRT itself, we could also parallelize its most computationally expensive components, such as the collision detection, as done in [17].

REFERENCES

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

[2] S. LaValle and J. Kuffner, "Rapidly-exploring random trees: Progress and prospects," in *Proc. Algorithmic Comput. Robot.*, 2001, pp. 293–308.

[3] S. LaValle and J. Kuffner, "Randomized kinodynamic planning," *Int. J. Robot. Res.*, vol. 20, no. 5, pp. 378–400, May 2001.

[4] J. Cortés and T. Siméon, "Sampling-based motion planning under kinematic loop-closure constraints," in *Proc. VI Algorithmic Found. Robot.*, 2005, vol. 17, pp. 75–90.

[5] M. Branicky, M. Curtiss, J. Levine, and S. Morgan, "RRTs for nonlinear, discrete, and hybrid planning and control," in *Proc. IEEE Conf. Decision Control*, Dec. 2003, vol. 1, pp. 657–663.

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

[7] 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, no. 1, pp. i116–i125, 2005.

[8] L. Jaillet, A. Yershova, S. LaValle, and T. Siméon, "Adaptive tuning of the sampling domain for dynamic-domain RRTs," in *Proc. Intell. Robots Syst.*, Aug. 2005, pp. 2851–2856.

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

[10] P. Cheng, E. Frazzoli, and S. LaValle, "Improving the performance of sampling-based motion planning with symmetry-based gap reduction," *IEEE Trans. Robot.*, vol. 24, no. 2, pp. 488–494, Apr. 2008.

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

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

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

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

[15] D. Henrich, "Fast motion planning by parallel processing—A review," *J. Intell. Robot. Syst.*, vol. 20, pp. 45–69, 1997.

[16] I. Şucan and L. Kavraki, "Kinodynamic motion planning by interior-exterior cell exploration," in *Proc. Algo. Found. Robot. VIII*, 2010.

[17] J. Bialkowski, S. Karaman, and E. Frazzoli, "Massively parallelizing the RRT and the RRT*," in *Proc. Intell. Robots Syst.*, Sep. 2011, pp. 3513–3518.
[18] N. Amato and L. Dale, "Probabilistic roadmap methods are embarrassingly parallel," in *Proc. IEEE Int. Conf. Robot. Autom.*, 1999, vol. 1, pp. 688–694.
[19] D. Challou, D. Boley, M. Gini, V. Kumar, and C. Olson, "Parallel search algorithms for robot motion planning," in *Proc. Pract. Mot. Planning Robot. Curr. Appr. Future Direct.*, 1998.
[20] S. Caselli and M. Reggiani, "ERPP: An experience-based randomized path planner," in *Proc. ICRA*, 2000, vol. 2, pp. 1002–1008.
[21] E. Plaku, K. Bekris, B. Chen, A. Ladd, and L. Kavraki, "Sampling-based roadmap of trees for parallel motion planning," *IEEE Trans. Robot.*, vol. 21, no. 4, pp. 597–608, Aug. 2005.
[22] M. Strandberg, "Augmenting RRT-planners with local trees," in *Proc. ICRA*, Apr./May 2004, vol. 4, pp. 3258–3262.
[23] 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.
[24] [Online]. Available: http://www.ros.org/doc/api/ompl/html
[25] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, 2nd ed. Harlow, U.K.: Pearson, 2003.
[26] I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Reading, MA: Addison-Wesley, 1995.
[27] E. Plaku and L. Kavraki, "Quantitative analysis of nearest-neighbors search in high-dimensional sampling-based motion planning," in *Proc. Algorithm Found. Robot.*, 2008.
[28] (2013). [Online]. Available: http://homepages.laas.fr/jcortes/Papers/Parallel-RRT_suppl-mat.html
[29] S. Rodríguez, X. Tang, J.-M. Lien, and N. Amato, "An obstacle-based rapidly-exploring random tree," in *Proc. IEEE Int. Conf. Robot. Autom.*, May 2006, pp. 895–900.
[30] S. Dalibard and J.-P. Laumond, "Control of probabilistic diffusion in motion planning," in *Proc. 8th Algorithm Found. Robot.*, 2010.
[31] D. Berenson, S. Srinivasa, and J. Kuffner, "Task space regions: A framework for pose-constrained manipulation planning," *Int. J. Robot. Res.*, vol. 30, no. 12, pp. 1435–1460, Oct. 2011.
[32] M. Gharbi, "Motion planning and object manipulation by humanoid torsos," Ph.D. dissertation, Laboratoire d'analyse et d'architectures des systèmes, Cent. Nat. Recherche Scientifique, Toulouse, France, 2010.
[33] 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. Int. Conf. Humanoid Robot.*, Dec. 2009, pp. 355–360.
[34] K. Bouyarmane and A. Kheddar, "Static multi-contact inverse problem for multiple humanoid robots and manipulated objects," in *Proc. Int. Conf. Humanoid Robot.*, Dec. 2010, pp. 8–13.
[35] 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, Aug. 2010.
[36] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *Int. J. Robot. Res.*, vol. 30, no. 7, pp. 846–894, Jun. 2011.
[37] D. Hsu, R. Kindel, J.-C. Latombe, and S. Rock, "Randomized kinodynamic motion planning with moving obstacles," *Int. J. Robot. Res.*, vol. 21, no. 3, pp. 233–255, Mar. 2002.