# Parallelizing RRT on distributed-memory architectures

Didier Devaurs, Thierry Simeon, Juan Cortés

# Parallelizing RRT on Distributed-Memory Architectures

Didier Devaurs, Thierry Siméon and Juan Cortés

***Abstract*— This paper addresses the problem of improving the performance of the Rapidly-exploring Random Tree (RRT) algorithm by parallelizing it. For scalability reasons we do so on a distributed-memory architecture, using the message-passing paradigm. We present three parallel versions of RRT along with the technicalities involved in their implementation. We also evaluate the algorithms and study how they behave on different motion planning problems.**

## I. Introduction

Due to a wide range of applications, sampling-based path planning has benefited from a considerable research effort [1], [2]. It has proven to be an effective framework suitable for a large class of problems in domains such as autonomous robotics, manufacturing, virtual prototyping, computer graphics, structural biology, and medicine. These application fields yield increasingly difficult, highly-dimensional problems with complex geometric and kinodynamic constraints.

The Rapidly-exploring Random Tree (RRT) has become a popular algorithm for solving single-query motion planning problems [3]. It is suited to solve robot motion planning problems involving holonomic, non-holonomic, kinodynamic, or kinematic closure constraints [3]–[5]. It is also applied to the validation and control of hybrid systems [6], [7]. In biology, it is used to analyze genetic network dynamics [8] or protein-ligand interactions [9], [10]. However, when applied to complex problems, the incremental growth of an RRT can become computationally expensive [9], [11], [12]. Some techniques have been proposed to improve the efficiency of RRT, by controlling the sampling domain [12], reducing the complexity of the nearest neighbor search [13], or employing gap reduction techniques [11].

Our objective is to further investigate RRT improvement by exploiting speedup from parallel computation. Some results have been obtained in that sense (Section II). However, existing work considers mainly shared-memory architectures and thus small-scale parallelism, up to 16 processors [14]–[17]. In this work, we are interested in what can be achieved by larger-scale parallelism. We focus on parallelizing RRT on distributed-memory architectures, using the message-passing paradigm. Our contribution is three-fold. First, we propose three parallel versions of RRT, based on classical parallelization schemes: OR parallel RRT, Distributed RRT and Manager-worker RRT (Section III). Second, beside the abstract view provided by the algorithms themselves, we present the main technicalities involved in their development (Section III). Third, we evaluate the algorithms on several motion planning problems and show their differences in behavior, depending on the problem type (Section IV).

All authors are with CNRS ; LAAS ; 7 avenue du colonel Roche, F-31077 Toulouse Cedex 4, France and Université de Toulouse ; UPS, INSA, INP, ISAE ; UT1, UTM, LAAS ; F-31077 Toulouse Cedex 4, France {devaurs,nic,jcortes}@laas.fr

## II. Related Work

### A. Parallel Motion Planning

The idea of improving motion planning performance by using parallel computation was raised in prior work. In a survey of some early work [18], a classification scheme was 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, that easily allows having multiple threads collaboratively solving a problem [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 of 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 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 algorithms, such as in [22].

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

An approach based on growing several independent trees can lead to a straightforward parallelization. This is the case for RRTLocTrees [24] and for the Rapidly exploring Random Forest of Trees (RRFT) [7], [8]. However, the focus of this paper lies elsewhere, our aim being to provide a parallel version of the basic (single-tree) RRT algorithm.

### B. Parallel RRT

There is relatively little work related to parallelizing RRT [14]–[17]. 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 case, the embarrassingly parallel algorithm is run on a single SMP (symmetrical multiprocessor) 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 *OR* and *AND* implementations. In the Open Motion Planning Library[1] (OMPL) of the ROS framework, the AND paradigm is implemented via multi-threading, thus for shared memory. In [16], the OR paradigm is used on 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 [17], 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 will parallelize RRT on a distributed-memory architecture, 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 this 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*, which is similar to the one in [14].

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

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

**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 not tooSimilar(q_near, q_new)[3] then
7          addNewNodeAndEdge(T, q_near, q_new)
8  if stopCondition(T) then
9      broadcast(endMsg)
```

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 we do not know 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 these processes to synchronize and agree on which one should report its solution.

### 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* [25]. Each process performs its own sampling of the search space[4] 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 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).

[3]Two configurations are deemed too similar if the distance between them is less than the minimum validation step-size along the path.

[4]Note that space partitioning would be possible here, but is not required.

**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 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)
```

**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 not tooSimilar(q_near, q_new) then
16             send(nodeData(q_new, q_near), mgr)
```

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 5-10). 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. 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 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. Note also that a Universally Unique Identifier (UUID) is associated with each node, in order to provide processes with a homogeneous way of referring to the nodes. Finally, the case of several processes reaching a solution at the same time has to be dealt with.

*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 communication cost would then outweigh any potential benefit. At each iteration of the tree building (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 algorithms, this one 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 each worker, which also releases us from having to use UUIDs.

### C. Implementation Framework

Among the various implementations of MPI, we have chosen OpenMPI[5]. 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 were often pointers or STL containers. Thus, we have decided to exploit the higher-level abstraction provided by the C++ library Boost.MPI[6]. Coupled with the Boost.Serialization library[7], Boost.MPI enables processes to exchange instances of high-level classes in a straightforward manner, making the tasks of gathering, packing and unpacking the underlying data transparent to the programmer. Finally, we have used Qt's implementation of UUIDs[8].

## IV. Experiments

### 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 counterpart. Aimed at measuring so, the *speedup* $S$ of a parallel algorithm is defined as the ratio of the runtime of its sequential counterpart to its own runtime: $S(p) = T_S \,/\, T_P(p)$ [25], [26]. In theory $S(p)$ is bounded by $p$, but in practice super-linear speedup ($S(p) > p$) can be observed. 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 runtime obtained over 100 executions of a parallel (resp. sequential) algorithm. We can then evaluate the *scalability* of a parallel algorithm, i.e. study whether the speedup increases in proportion to the number of processors. We can also measure the *efficiency* of a parallel algorithm ( $E(p) = S(p) \,/\, p$ ) which is a decreasing function of $p$ theoretically having values in $[0, 1]$ [25], [26].

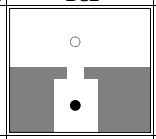
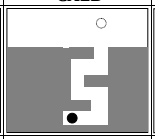
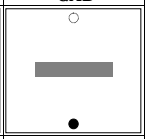
| Problem name | | BCL | CALB | GAB |
|---|---|---|---|---|
| Problem type | |  |  |  |
| Sequential RRT | $T_S$ (s) | 1.4 ± 0.81 | 148 ± 129 | 62 ± 12 |
| | N | 46 ± 19 | 1629 ± 1365 | 615 ± 90 |
| | E | 821 ± 474 | 81023 ± 69917 | 770 ± 121 |

Fig. 1. Simplified schematic representation of the configuration spaces of our three problems, and numerical results obtained with the sequential RRT. 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, $N$, and the number of expansion attempts, $E$.

### B. Parallel Computer Architecture

The numerical results presented in this section have been obtained by running the algorithms on an HP cluster platform composed of 24 HP ProLiant DL160 G5 servers connected by a high-speed InfiniBand™ switch warranting 10 Gbit/s of bandwidth. Each server includes two 64-bit quad-core Intel® Xeon® E5430 processors at 2.66 GHz, with 12 MB of L2 cache, and sharing 7.79 GB of memory.

### C. Motion Planning Problems Studied

We have evaluated the algorithms on three motion planning problems involving molecular models[9]. However, it is important to note that our 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[10]) and are characterized by different configuration-space topologies (cf. Fig. 1). BCL 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. CALB is a similar problem, but with a longer and very narrow exit pathway, i.e. more geometrically constrained than BCL. In GAB, a protein goes around another one in an empty space, thus involving the weakest geometrical constraints, but the longest distance to cover of all problems. Fig. 1 also presents the numerical results obtained when solving these problems with the sequential RRT.

[5]http://www.open-mpi.org

[6]http://www.boost.org/doc/libs/1_43_0/doc/html/mpi.html

[7]http://www.boost.org/doc/libs/1_43_0/libs/serialization

[8]http://doc.trolltech.com/4.3/quuid.html

[9]The application we have used is the molecular motion planning toolkit we are currently developing [27].

[10]To facilitate the algorithms' evaluation, 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-dimensional space, without a significant loss in accuracy [28].

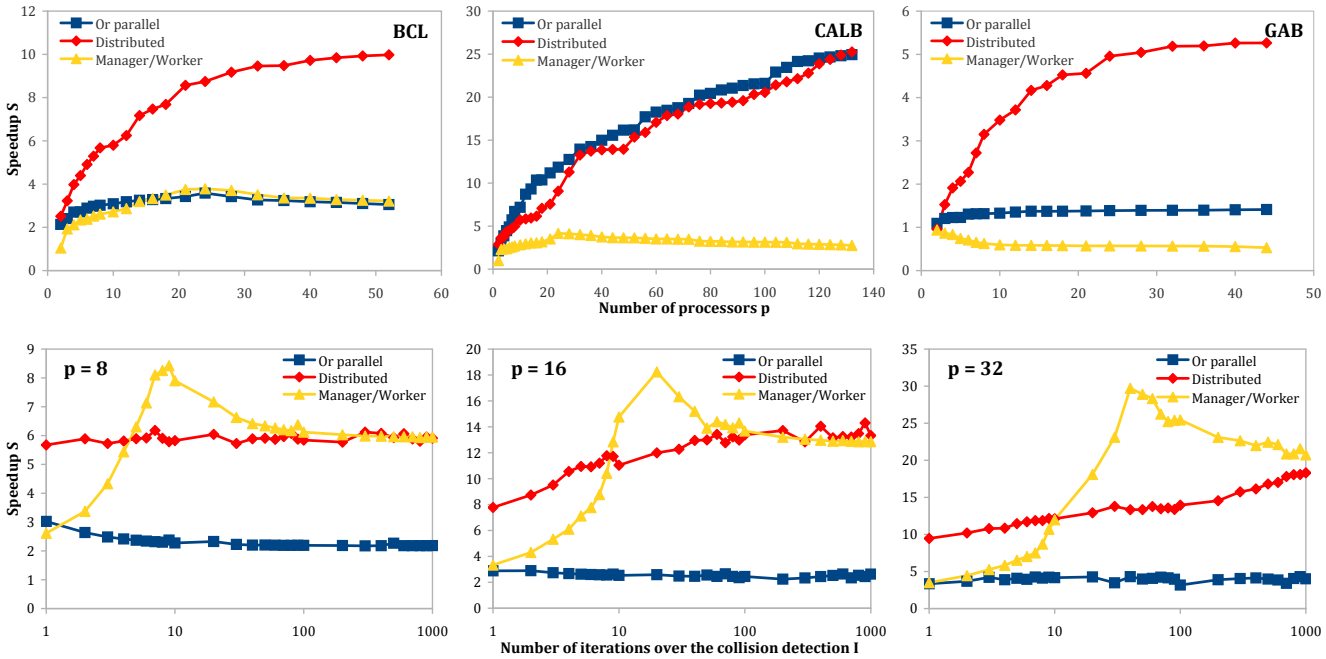



Fig. 2. First row: scalability of our algorithms on the BCL, CALB and GAB problems. Second row: evolution of the algorithms' speedup (and efficiency, as they are proportional in that case) in relation to the expansion cost, while solving the BCL problem on 8, 16 and 32 processors.

### *D. Speedup Achieved by the Parallel Algorithms*

The first row of Fig. 2 presents the scalability achieved by the algorithms on each problem. Unsurprisingly, the scalability of the OR parallel RRT is strongly correlated with the variability of the computing time, measured by the ratio of the standard deviation to the mean of the runtime $T_S$ (cf. Fig. 1). This algorithm can achieve good results only on problems where this variability is large, such as CALB. When this variability is low, e.g. in GAB, it provides almost no improvement over the sequential RRT. The Manager-worker RRT shows a very poor speedup on all problems. This is partly explained by the fact that it involves much more communication than the other schemes. Each expansion attempt is preceded and followed by a communication between the manager and a worker, contrary to the Distributed RRT, in which communications between processes happen only after a new node is built. In the Distributed scheme, the total number of messages exchanged over the network increases linearly with $p$, but at each processor's level, the number of messages is bounded by $N$. Thus, as long as the network bandwidth can withstand the communication load, the Distributed RRT can show a good scalability.

Although speedup curves of the Distributed RRT flatten when $p$ increases, we would have to use many more processors to see a decrease, contrary to other schemes. The best speedup it achieves on BCL, CALB and GAB is 10, 25.3 and 5.3, which correspond to quite a low efficiency of 0.2, 0.2 and 0.1 respectively. The greatest number of processors for which its efficiency is greater than 0.5 is 14, 10 and 3 respectively. Several factors contribute to this low efficiency. (1) Runtime is quite short on these problems, especially BCL. When more and more processors are added, the communication load increases significantly, thus outweighing the reduction in computing time and leading to a smaller increase in speedup. (2) 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 [12]) in favor of more refinement. This translates into generating larger trees (i.e. the number of nodes $N$ increases with $p$), thus reducing the increase in speedup, especially on not very constrained problems, such as GAB.

Generally, efficiency improves as the problem difficulty increases. In artificially increasing it, we will also show that the Manager-worker RRT can perform better in some settings. Intuitively, it is worth using this scheme when the manager can delegate costly sub-tasks to its workers. However, in our settings the cost of the expansion step is quite low, as $q_{new}$ is generated by a simple linear interpolation between $q_{near}$ and $q_{rand}$, and motion validation is limited to collision detection. Expansion could be much more expensive, e.g. if a dynamic simulator was producing robot motions, or if some potential energy was computed in the case of molecular models. To test whether this could have an impact on the algorithms' performance, we have run a controlled experiment in which we have artificially increased the cost of the expansion step to emulate different settings. To do so, during an expansion attempt we repeat $I$ times the collision detection routine in the `extend()` function. Tests were performed on the BCL problem, as it is characterized by a medium-level difficulty in its configuration-space topology. The second row of Fig. 2 shows the evolution of the algorithms' speedup in relation to $I$, on 8, 16 and 32 processors. As $I$ goes up, we observe first a dramatic increase in the speedup of the

Manager-worker RRT, followed by a slower decrease due to the fact that the manager becomes a bottleneck waiting for busy workers. This higher speedup is enabled by the growth in computational load making the communication load not significant anymore. The maximum speedup corresponds to an optimal use of this scheme, which depends on $p$: when $p$ is increased, this maximum raises and is reached for higher values of $I$. The best efficiency values obtained for $p = 8$, 16 and 32 are 1.1, 1.1 and 0.9 respectively. Similarly, though not so dramatically, an increase in the expansion cost also translates into a better use of the Distributed RRT, which is more visible as $p$ goes up. As expected, no benefit is observed for the OR parallel RRT, whose optimal use relates to variability in runtime and not to computational load.

## V. Conclusion

We have proposed three parallel versions of the RRT algorithm, designed for distributed-memory architectures using message passing: 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], [16]. Our Distributed RRT and Manager-worker RRT are the counterparts for distributed memory of the AND (or embarrassingly parallel) RRT [14], [15]. None of these algorithms can be held as the best parallelization of RRT: it really depends on the studied problem. The Distributed RRT shows the most consistent results across experiments, but its efficiency does not scale well when the problem becomes more difficult. It could also suffer from memory scalability issues, since each process maintains its own tree. It is outperformed by the OR parallel RRT on problems yielding a great variability in computing time. It is also outperformed by the Manager-worker RRT in settings involving high expansion costs. The Manager-worker RRT shows the best efficiency scalability when the problem difficulty increases.

This paper was focused on a high-level parallelization of RRT. It could be extended by parallelizing its sub-routines, such as the nearest neighbor search. For that, we could use data decomposition and evaluate the speedup achieved depending on the search paradigm (brute force, kd-trees, etc.). Algorithms involving the construction of several independent RRTs can directly benefit from this work. For example, in the simple variant of the bidirectional-RRT where both trees are extended toward the same random configuration, processes can be separated in two building groups getting random configurations from an extra process. When the RRTs are not independently built, specific algorithms have to be developed.

As part of our future work, we plan to investigate approaches combining the three paradigms. We are currently extending our molecular motion planning application to allow for potential energy computation, in order to pursue our study started by artificially increasing the tree expansion costs. We also plan to better exploit the architecture of our cluster platform, by combining multi-threading and message passing approaches. Allowing the eight processes sharing the same memory to work on a common tree would mitigate the memory scalability issue of the Distributed RRT.

## References


[1] 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.

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

[3] 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.

[4] ——, "Randomized kinodynamic planning," *Int. J. Robot. Research*, vol. 20, no. 5, 2001.

[5] 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.

[6] 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.

[7] J. M. Esposito, J. Kim, and V. Kumar, "Adaptive RRTs for validating hybrid robotic control systems," in *Algorithmic Foundations of Robotics VI*. Springer-Verlag, 2005, pp. 107–121.

[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, 2005.

[9] J. Cortés, L. Jaillet, and T. Siméon, "Molecular disassembly with RRT-like algorithms," in *Proc. IEEE ICRA*, 2007.

[10] J. Cortés, D. T. Le, R. Iehl, and T. Siméon, "Simulating ligand-induced conformational changes in proteins using a mechanical disassembly method," *Phys. Chem. Chem. Phys.*, vol. 12, no. 29, 2010.

[11] 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 ICRA*, 2004.

[12] 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.

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

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

[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, 2008.

[16] S. Sengupta, "A parallel randomized path planner for robot navigation," *Int. J. Adv. Robot. Syst.*, vol. 3, no. 3, 2006.

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

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

[19] 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.

[20] N. M. Amato and L. K. Dale, "Probabilistic roadmap methods are embarrassingly parallel," in *Proc. IEEE ICRA*, 1999.

[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 ICRA*, 2000.

[23] E. Plaku and L. E. Kavraki, "Distributed sampling-based roadmap of trees for large-scale motion planning," in *Proc. IEEE ICRA*, 2005.

[24] M. Strandberg, "Augmenting RRT-planners with local trees," in *Proc. IEEE ICRA*, 2004.

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

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

[27] 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), 2005.

[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.