

# Massively Parallelizing the RRT and the RRT\*

Joshua Bialkowski

Sertac Karaman

Emilio Frazzoli

**Abstract**—In recent years, the growth of the computational power available in the Central Processing Units (CPUs) of consumer computers has tapered significantly. At the same time, growth in the computational power available in the Graphics Processing Units (GPUs) has remained strong. Algorithms that can be implemented on GPUs today are not only limited to graphics processing, but include scientific computation and beyond. This paper is concerned with massively parallel implementations of incremental sampling-based robot motion planning algorithms, namely the widely-used Rapidly-exploring Random Tree (RRT) algorithm and its asymptotically-optimal counterpart called RRT\*. We demonstrate an example implementation of RRT and RRT\* motion-planning algorithm for a high-dimensional robotic manipulator that takes advantage of an NVIDIA CUDA-enabled GPU. We focus on parallelizing the collision-checking procedure, which is generally recognized as the computationally expensive component of sampling-based motion planning algorithms. Our experimental results indicate significant speedup when compared to CPU implementations, leading to practical algorithms for optimal motion planning in high-dimensional configuration spaces.

## I. INTRODUCTION

Given a description of the robot, an initial configuration, a set of goal configurations, and a set of obstacles, the robot motion planning problem is to find a path that starts from the initial configuration and reaches a goal configuration while avoiding collision with the obstacles. This problem of navigating through a complex environment has been widely studied in the robotics literature for at least three decades and has several applications outside the domain of robotics [1].

Although the motion planning problem is known to be challenging from a computational point of view [2], several practical algorithms have been proposed in the literature. Arguably, one of the most widely-used class of practical motion planning algorithms is the sampling-based methods, introduced by Kavraki et al. [3], under the name of Probabilistic RoadMaps (PRMs). Incremental sampling-based algorithms, such as the Rapidly-exploring Random Tree (RRT) algorithm [4], have emerged as online counterparts of PRMs, tailored mainly for single-query applications.

Both the PRM and RRT algorithms are *probabilistically complete* in the sense that the probability that these algorithms return a solution, if one exists, converges to one as the number of samples approaches infinity, under mild

technical assumptions. Since their introduction in the literature, sampling-based motion algorithms have been extended in several directions (see e.g., [5]). In particular, the RRT algorithm was showcased in major robotics events [6].

In most applications of motion planning, finding not only a feasible solution, but also one that has good quality, e.g., in terms of a cost function, is highly desired. In this context, a sampling-based motion planning algorithm is said to be *asymptotically optimal*, if the solution returned by the algorithm converges to an optimal solution almost surely as the number of samples approaches infinity [7]. A recently proposed incremental sampling-based algorithm, called RRT\*, was shown to have the asymptotic optimality property, which the RRT algorithm lacked, while incurring no substantial computational cost when compared to RRT [7].

Traditionally, these motion planning algorithms have been designed in a serial manner, leading to direct implementation on commodity CPUs. However, especially recently, the computational power embedded in the commercially-available CPUs, e.g., in terms of clock frequency, has tapered significantly. On the other hand, dedicated computational architectures such as GPUs continue to provide rapidly-increasing computational power by making use of massively parallel architectures in which thousands of data-parallel logical processing units are common. In fact, a recent trend in computing technology has been fueled by increasing the number of dedicated processing units embedded into a single chip. To benefit from this trend, however, next generation motion planning algorithms have to be able to process large blocks of data in parallel in order to provide better performance as the number of processing units increase.

Roughly speaking, *massively parallelizable* algorithms are those whose throughput scales well (e.g., almost linearly) with the number of available logical processors. In this paper, we investigate massively parallel implementations of the RRT and RRT\* algorithms, and describe their implementation for a robotic manipulator with several links operating in two dimensions in the presence of axis-aligned rectangular obstacles. We focus on the first step of a massively parallel implementation by addressing the primary bottleneck of many practical motion-planning problems, i.e., collision checking.

Due to the specifics of GPU hardware, and the infancy of the general purpose GPU (GP-GPU) technology, the tools available to the programmer are far less sophisticated than those available for traditional (CPU) programming. The hardware and conceptual models supported by manufacturers continue to change meaning that the implementation of a particular planning algorithm will necessarily be specialized

J. Bialkowski is with the Department of Aeronautics and Astronautics, Massachusetts Institute of Technology, Cambridge, MA 02139, USA [jbialk@mit.edu](mailto:jbialk@mit.edu)

S. Karaman is with the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, USA [sertac@mit.edu](mailto:sertac@mit.edu)

E. Frazzoli is with the Department of Aeronautics and Astronautics, Massachusetts Institute of Technology, Cambridge, MA 02139, USA [frazzoli@mit.edu](mailto:frazzoli@mit.edu)

for the problem class at hand and the hardware chosen to address the problem. This paper will address the design issues involved in implementing RRT and RRT\* on an NVidia<sup>®</sup> CUDA<sup>®</sup>-enabled GPU in the hopes that it will provide a template for those who attempt to implement these algorithms for other problems on platforms with similar models of parallel computation.

Some early work has been done on implementing parallel versions of the RRT algorithm, such as [8], and [9]. A parallel RRT/PRM hybrid is presented in [10]. Yet these implementations focus on a distributed memory model of parallel computation. While this model has the advantage of being highly scalable, computing clusters are expensive and specialized. This paper addresses instead a shared memory model of computation allowing for significant improvements in runtime on cheaper and more readily available equipment.

## II. PROBLEM DESCRIPTION

We address the problem of feasible motion planning for a multi-link robotic manipulator in a 2D workspace, as shown in Fig. 1. We assume the manipulator is fully actuated, and that each link arm of the manipulator is the same length,  $l$ . The manipulator's configuration space is denoted as  $\mathcal{X} = S^d$ , where the configuration is expressed in coordinates as  $x = (\theta_1, \dots, \theta_d)$ , and  $\theta_i \in [-\pi, \pi)$  is the angle of the  $i$ -th joint. Given a configuration  $x \in \mathcal{X}$ , the set of all points occupied by the links of the manipulator is denoted by  $\Gamma(x) \subset \mathbb{R}^2$ . A path of the manipulator is a continuous function from the real interval  $[0, 1]$  to the set  $\mathcal{X}$  of all configurations, usually denoted by  $\sigma : [0, 1] \rightarrow \mathcal{X}$ .

Given an initial configuration  $x_{\text{init}}$ , a goal set  $\mathcal{X}_{\text{goal}} \subset \mathbb{R}^2$ , and an obstacle set  $\mathcal{X}_{\text{obs}} \subset \mathbb{R}^2$ , the *feasible motion planning problem* for the manipulator is to find a path  $\sigma$  that (i) starts from the initial configuration, i.e.,  $\sigma(0) = x_{\text{init}}$ , (ii) reaches the goal region, i.e.,  $\sigma(1) \in \mathcal{X}_{\text{goal}}$ , and (iii) avoids collision with obstacles, i.e.,  $\Gamma(\sigma(\tau)) \cap \mathcal{X}_{\text{obs}} = \emptyset$  for all  $\tau \in [0, 1]$ . Given also a cost function  $c(\cdot)$  that maps each path to a non-negative cost, the *optimal motion planning problem* is to find a path  $\sigma^*$  that solves the feasible motion planning problem such that  $\sigma^*$  minimizes the cost function  $c(\cdot)$ , i.e.,  $\sigma^* = \arg \min_{\sigma} c(\sigma)$ . We assume throughout the paper that  $\mathcal{X}_{\text{goal}}$  and  $\mathcal{X}_{\text{obs}}$  are open sets.

## III. ALGORITHMS

### A. The RRT and the RRT\* algorithms

Before presenting the RRT and the RRT\* algorithms, we discuss a set of primitive procedures that they rely on. Implementation details of these procedures for the manipulator link example are explained in the next section.

*a) Sampling:* The *Sample* procedure returns a configuration sampled uniformly at random from the configuration space.

*b) Nearest Vertex:* Given a finite set  $V$  of configurations and a configuration  $x \in \mathcal{X}$ , the *Nearest* procedure returns the configuration  $\bar{x} \in V$  that is closest to  $x$  in terms of e.g. geodesic distance.

### Algorithm 1: RRT and RRT\* Algorithms

```

1  $V \leftarrow \{x_{\text{init}}\}; E \leftarrow \emptyset; i \leftarrow 0;$ 
2 while  $i < N$  do
3    $x_{\text{rand}} \leftarrow \text{Sample}(i);$ 
4    $(V, E) \leftarrow \text{Extend}((V, E), x_{\text{rand}});$ 
5    $i \leftarrow i + 1;$ 

```

*c) Near Vertices:* Given a set  $V$  of configurations and a configuration  $x \in \mathcal{X}$ , the *Near* procedure returns the set of all vertices that are within the ball of volume  $(\log n)/n$  centered at  $x$ , i.e.,

$$\text{Near}(V, x) = \left\{ x' \in V : \|x' - x\| \leq \left( \frac{\log n}{n} \right)^{1/d} \right\},$$

where  $n = |V|$  is the cardinality of  $V$ .

*d) Steering:* Given two configurations  $x, x' \in \mathbb{R}^n$ , the *Steer* procedure returns a straight path  $\sigma$  that connects  $x$  and  $x'$ . That is,  $\sigma(\tau) = \tau x + (1 - \tau)x'$  for all  $\tau \in [0, 1]$ . Note that we are considering a kinematic model.

*e) Collision Checking:* Given a path  $\sigma$ , the *ObstacleFree* procedure returns true if the  $d$ -link manipulator executing this path avoids collision with the obstacles, i.e. if  $\Gamma(\sigma(\tau)) \cap \mathcal{X}_{\text{obs}} = \emptyset$  for all  $\tau \in [0, 1]$ , and returns false otherwise.

The RRT [4] and RRT\* [7] algorithms are summarized in Algorithm 1. Both algorithms maintain a tree of trajectories, denoted by  $G = (V, E)$ , where  $V$  is the set of vertices and  $E$  is the set of edges. Initially, the set of vertices contains only  $x_{\text{init}}$  while the set of edges is empty (Line 1). In each iteration, both algorithms first randomly sample a configuration from the configuration space using the *Sample* procedure (Line 3), and then extend the tree towards this sample using the *Extend* procedure (Line 4).

The RRT and the RRT\* algorithms differ in the way that they handle the extension procedure. The extension subroutine of the RRT algorithm is shown in Algorithm 2. The procedure finds the configuration  $x_{\text{nearest}} \in V$  that is closest to the sampled configuration, and checks whether the straight path connecting  $x_{\text{nearest}}$  and the sampled configuration is

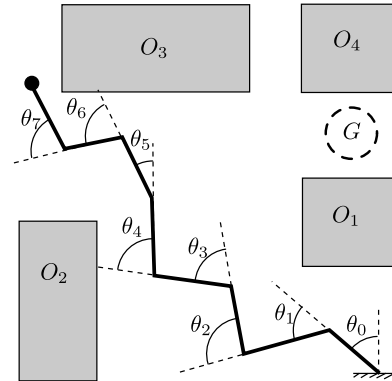


Fig. 1. Robotic Manipulator

**Algorithm 2:**  $\text{Extend}_{\text{RRT}}((V, E), x)$ 

```

1  $V' \leftarrow V; E' \leftarrow E;$ 
2  $x_{\text{nearest}} \leftarrow \text{Nearest}(G, x);$ 
3  $\sigma \leftarrow \text{Steer}(x_{\text{nearest}}, x);$ 
4 if  $\text{ObstacleFree}(\sigma)$  then
5    $V' \leftarrow V' \cup \{x_{\text{new}}\};$ 
6    $E' \leftarrow E' \cup \{(x_{\text{nearest}}, x_{\text{new}})\};$ 
7 return  $G' = (V', E')$ 

```

**Algorithm 3:**  $\text{Extend}_{\text{RRT}^*}((V, E), x)$ 

```

1  $V' \leftarrow V; E' \leftarrow E;$ 
2  $x_{\text{nearest}} \leftarrow \text{Nearest}(G, x);$ 
3  $\sigma \leftarrow \text{Steer}(x_{\text{nearest}}, x);$ 
4 if  $\text{ObstacleFree}(\sigma)$  then
5    $V' \leftarrow V' \cup \{x\};$ 
6    $x_{\text{min}} \leftarrow x_{\text{nearest}};$ 
7    $c_{\text{min}} \leftarrow \text{Cost}(x_{\text{nearest}}) + c(\sigma);$ 
8    $X_{\text{near}} \leftarrow \text{Near}(G, x, |V|);$ 
9   for all  $x_{\text{near}} \in X_{\text{near}}$  do
10     $\sigma \leftarrow \text{Steer}(x_{\text{near}}, x);$ 
11    if  $\text{ObstacleFree}(\sigma)$  and
12       $\text{Cost}(x_{\text{near}}) + c(\sigma) < c_{\text{min}}$  then
13       $x_{\text{min}} \leftarrow x_{\text{near}};$ 
14       $c_{\text{min}} \leftarrow \text{Cost}(x_{\text{near}}) + c(\sigma);$ 
15    $E' \leftarrow E' \cup \{(x_{\text{min}}, x)\};$ 
16   for all  $x_{\text{near}} \in X_{\text{near}} \setminus \{x_{\text{min}}\}$  do
17      $\sigma \leftarrow \text{Steer}(x, x_{\text{near}});$ 
18     if  $\text{ObstacleFree}(\sigma)$  and
19        $\text{Cost}(x_{\text{near}}) > c_{\text{min}} + c(\sigma)$  then
20        $x_{\text{parent}} \leftarrow \text{Parent}(x_{\text{near}});$ 
21        $E' \leftarrow E' \setminus \{(x_{\text{parent}}, x_{\text{near}})\};$ 
22        $E' \leftarrow E' \cup \{(x_{\text{new}}, x_{\text{near}})\};$ 
23 return  $G' = (V', E')$ 

```

collision free. If so, the sampled configuration is added to the tree as a vertex along with an edge connecting it to  $x_{\text{nearest}}$ .

The extension sub-routine of the RRT\* (Algorithm 3) is slightly more involved. The same operations are performed as in the RRT up to the point of checking that the path is collision free. However, before inserting the sampled configuration in the graph, the procedure runs through the set of all near nodes around the sampled configuration. The near vertex  $x_{\text{near}}$  that reaches the sampled configuration with the smallest accumulated cost is picked for connection, and the sampled configuration is added to the tree connected to this near vertex with an edge. Then, for each vertex in the set of near vertices, the algorithm checks whether the trajectory connecting the sampled configuration and  $x_{\text{near}}$  is collision-free and the cost of the trajectory that connects  $x_{\text{near}}$  to the root vertex through the sampled configuration is lower than current path that reaches  $x_{\text{near}}$ . If so, the incoming edge to  $x_{\text{near}}$  is “re-wired” through  $x_{\text{new}}$ . The reader is referred to [7] for a more elaborate description of the RRT and the RRT\* algorithms.

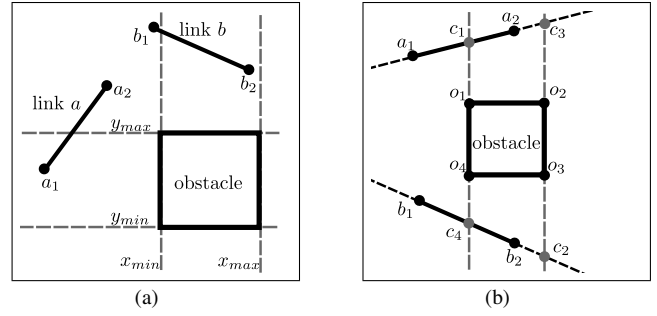


Fig. 2. Collision Checking Formula

**B. The manipulator link domain**

1) *Sampling and Steering:* For the manipulator case, in both algorithms our sampling, steering, and extension routines are the same. We sample points from the state space independently according to a uniform distribution with the range of each joint in  $[-\pi, \pi]$ .

2) *Graph Operations:* We maintain the graph for both the RRT and RRT\* via a balanced KD-tree [11]. Let us note that, for a given graph-size of  $n$  nodes, the expected time complexity for finding the nearest neighbor and for node insertion is  $O(\log(n))$  for approximate queries [12].

3) *Collision Checking:* By modeling the manipulator arms as line segments and the obstacles as axis-aligned rectangles checking whether a particular link is in collision with a particular obstacle can be carried out as follows. First, consider the four lines coincident with the four edges of the obstacle. Check to see if both end-points of the link lie on the outer side of any one of these lines, indicating the link is not in collision. See Fig. 2(a) for an illustration. In the figure, it is found that link  $a$  is not in collision with the obstacle because, for each end-point,  $x < x_{\text{min}}$  indicating that the entire link lies to the left of the minimum  $x$  extent. Second, if these first four tests are inconclusive then examine the line in the plane coincident with link. If all corners of the obstacle lie on the same side of that line the link is not in collision with that obstacle. We perform this check by calculating the  $y$  coordinates on that line of the two points whose  $x$  coordinates correspond with the  $x$  extents of the obstacle, and checking to make sure that it is either above, or below both  $y$  extents of the obstacle, as illustrated in Fig 2(b). Finally, conclude that a given configuration is in collision with a particular obstacle if the at least one link is in collision given that configuration.

Checking whether a particular path is collision-free is carried out by discretizing the path uniformly according to the length of the path in the  $d$ -dimensional space. That is for  $m$  discretization points along a path between two configurations, say  $x_1$  and  $x_2$ , the configuration  $j$  for link  $i$  is the joint angle  $\theta_{i,j} = \theta_{i,1} + (j/m)(\theta_{i,2} - \theta_{i,1})$ , where  $\theta_{i,1}$  and  $\theta_{i,2}$  are the joint angles for the  $i$ th joint given configurations  $x_1$  and  $x_2$ , respectively.

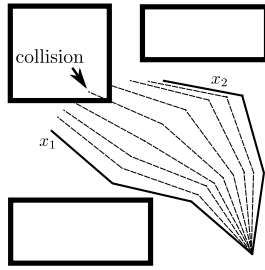


Fig. 3. Discretization of Trajectory

### C. Implementation

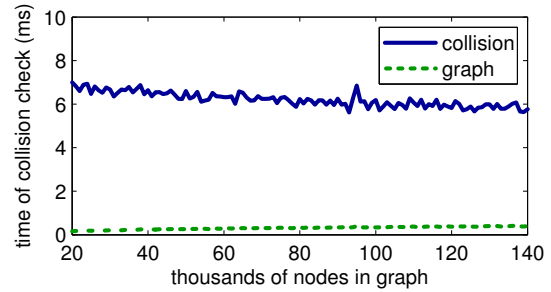
The algorithms for the manipulator case are implemented using the Sampling-based Motion Planning (SMP) C++ library maintained by the ARES group at MIT, and available at <http://ares.lids.mit.edu/software/>. SMP is a C++ template library that provides baseline templates for common modules of motion planning algorithms including samplers, distance metrics, reachability criteria, collision checkers and planners. To implement this test case, we use SMP along with a specialized collision checker and reachability criteria with a serial and parallel version. Both the serial and parallel versions of the code used for the manipulator example are available as a part of SMP.

## IV. PARALLELIZATION OF THE ALGORITHMS

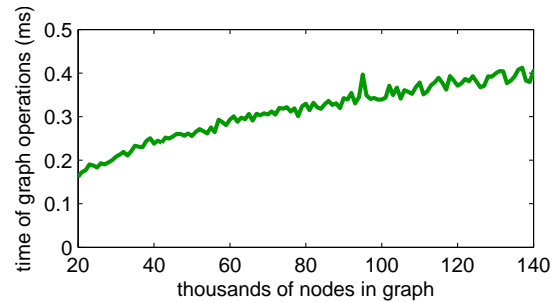
### A. Finding the Bottleneck

For a static environment, the time complexity of the RRT is governed by the graph operations, such as evaluating nearest neighbors, since collision checking is a constant-time operation, while the time complexity of the nearest-neighbor search and insertion grows as  $O(\log n)$ . However, while the graph operations may be the asymptotic bottleneck, in many cases, such as the manipulator case, the collision checking consumes most of the computational time.

In fact, profiling the SMP implementation in Valgrind's Callgrind [13] shows that, within the planner's `iterate` method, 99% of the instructions executed are within the collision-checker. More illustrative though is a profiling of the computation time spent in the collision-checker routines versus that spent in the graph operation routines, as in Fig. 4(a). The RRT was executed for one million iterations for a nine-link manipulator with four obstacles and a discretization count of 100. Fig. 4(a) shows the per-iteration time spent on finding the nearest neighbor, and inserting a new node, along with the time spent doing collision checking, plotted against the graph size. It is clear that for the manipulator the collision checker is the bottleneck. Fig. 4(b) shows the per-iteration time spent on graph operations alone, illustrating its logarithmic increase, but note that after a million iterations the RRT had found 40,000 solutions and yet the graph operations consumed less than 1% of the time used by the collision checker. Note that in these figures recorded times are averaged over 100 runs to smooth out the variance and illustrate the trends.



(a) Computation time of collision checking and graph operations



(b) Computation time of graph operations only

Fig. 4. Computation time of sub-algorithms.

For the RRT\*, the number of collision checks that are required per iteration scales by  $O(\log n)$ , which is, in fact, by design [7]. Given that the range-nearest-neighbor search grows with the same complexity, it is clear that collision-checking for the RRT\* will be the bottleneck.

### B. NVidia CUDA

The CUDA model of parallel programming has a three level thread-hierarchy. At the hardware level, threads are grouped into *warps* of at most 32 threads. Within a single warp, threads are executed by Single Instruction Multiple Data (SIMD). At the lower of the two logical levels, threads are organized in a one, two, or three dimensional *block*. At the top level, blocks are organized into a one or two dimensional *grid*. The logical levels of the hierarchy allow for execution in the Single Instruction Multiple Thread (SIMT) whereby individual warps are scheduled separately allowing kernels to branch when necessary, and only incurring branch-overhead when threads within the same warp diverge. An exhaustive description of the functional considerations of programming in CUDA is given in [14].

The CUDA memory model exposes three levels of memory. Global memory is a large-volume, persistent, high-latency memory. Texture caches are a mapped store of persistent memory that is optimized and aligned for lower-latency access of large arrays. Shared and per-thread memory are low-latency on-chip small memory caches. Shared memory is mapped per thread-block, and thread-memory is mapped per-thread.

CUDA C functions that are executed on the GPU are called *kernels*. For a particular kernel execution, the maximum block-size and grid-size are determined by a couple of different factors. For a particular compute-capability, there is

an upper limit for the dimensions of both thread block and block grids, as well as an upper limit for the number of total threads per block. In addition, the number of threads per-block can be limited by the shared and per-thread memory required by the kernel.

### C. Parallel Collision Checking

The algorithms are implemented and tested on a laptop computer with an Intel® Core i7 at 1.73 GHz with 4GB of main memory and an NVidia® Quadro FX 1800M with compute capability 1.2, clock rate of 1.1Ghz, 9 Streaming Multiprocessors (72 CUDA Cores), and 1GB of global memory. For this machine the maximum grid dimensions are  $[65535 \times 65535 \times 1]$ , while the maximum block dimensions are  $[512 \times 512 \times 64]$  with a maximum number of threads per block of 512. The total amount of shared memory and per-thread memory is 16kB.

In designing the collision-checking kernel, we assume that the dimension of the state space and number of obstacles is much smaller than the number of discretizations required for checking. Given that the CUDA thread block has three indexed dimensions, one dimension enumerates discretizations of the trajectory. The second dimension enumerates over obstacles to check. Thus, for  $n_d$  discretizations and  $n_o$  obstacles we have the constraint that  $n_d \times n_o < 512$ . For the test case, we check against 4 obstacles meaning we can split the path into a maximum of 128 configurations, although the implementation uses 100. The thread-limit prevents us from further enumerating dimensions of the configuration space over the third dimension of the block. Ignoring memory-transfer overhead, with this kernel we can perform  $n_d \times n_o$  configuration collision checks roughly in the same amount of computation time spent for a single configuration. We do this by utilizing a shared-memory flag such that, if any thread finds a collision, all threads in the block exit immediately. Note that for CUDA-enabled NVidia hardware with compute capability 1.3, there exists a block-voting function that can reduce the overhead of a serialized-writes to a shared memory flag.

For RRT alone, the capacity of the collision checking kernel could be increased by expanding over several blocks as described above. However, to the authors experience communication between blocks is quite slow because of serialized global-memory access, which is low latency. Instead, the excess capability is utilized by processing new samples in batches. The first dimension of the block grid enumerates configuration pairs (paths). For RRT\*, a single sample requires collision checking with several potential start configurations as the algorithm considers all nodes in  $X_{\text{near}}$ . Therefore, the kernel enumerates over the second dimension for batches of several configuration pairs.

Input to the kernel is organized in two arrays of global memory. The first is the configuration batches. Because RRT\* checks collisions in batches where the start node is the same, the implementation saves some memory by allocated enough memory for  $(n_p + 1) \times n_b$  configurations, where  $n_p$  is the number of pairs processed in an RRT\* batch, and  $n_b$

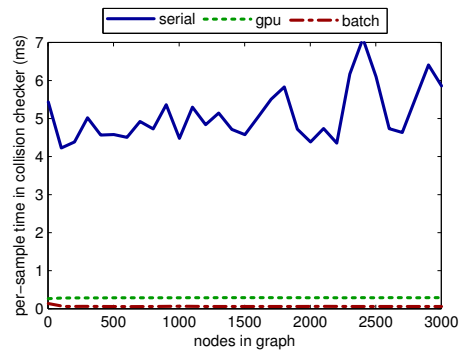


Fig. 5. Comparison of parallel and serial implementations.

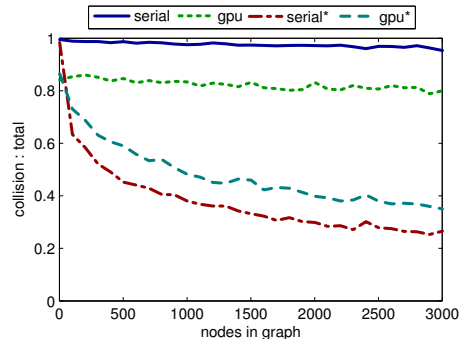


Fig. 6. Fraction of time spent for collision checking.

is the number of batches processed per call to the kernel. Output from the kernel is stored in a third array in global memory. The  $(i, j)$ th element of this array is set to 1 if any thread in the  $(i, j)$ th block finds a collision.

## V. RESULTS

### A. Collision Checker

With the described kernel, the improvement in the run-time of the collision checker is apparent. Profiling the collision checker over 40000 iterations of the RRT shows a speed up of about  $10\times$ . Fig. 5 shows the amount of time spent in the collision check per trajectory checked. For the CPU implementation the checker takes on average about 5ms per trajectory, while the GPU implementation spends about 500ns. More importantly, the kernel succeeds alleviating the collision checking bottleneck. Fig. 6 illustrates the relative amount of time spent in the collision checker as compared to the total time to perform one iteration of the planner. Note that all the data is given in wall-clock-time for the collision checking procedure averaged over 100 runs. Profiling is not measured in CPU-time because it does not count time spent waiting for the kernel to finish executing.

### B. Batching

Batching the collision checking also has a significant impact on reducing the runtime of the RRT. As shown in Fig. 5, the batched kernel achieves about  $25\times$  speedup over the serial version. The actual runtime of the kernel is larger than that of the non-batched kernel, but the increase is smaller than the batch size, so the overall per-sample time is lower.

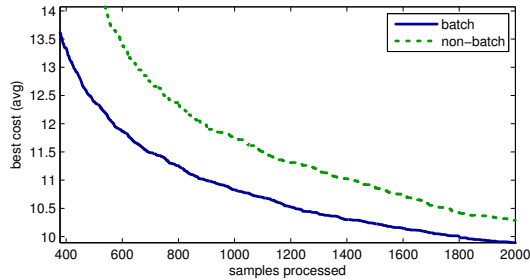


Fig. 7. Batched and Non-Batched RRT\*

It is likely that the real advantage of the batched kernel, though, will come from improvements to the RRT\* implementation. Fig. 7 illustrates the averaged convergence of the RRT\* implementation for both the batched and non-batched kernels for the number of samples processed. For this case the implementation uses a batch size of 20 samples. Note that the number of top-level iterations for the batched RRT\* processes 20 new samples while the non-batched version samples only one. Further note that, because the size of the ball used for nearest neighbor searching is calculated at the start of each top-level iteration, the batched kernel considers more nodes than necessary for the sample batch. For 40000 samples, the serial RRT\* takes 6.62 minutes, the RRT\* with the GPU collision checker takes 25 seconds, and the batched RRT\* with the GPU collision checker takes 90 seconds. Note that the total runtime of the batched version is longer, however, SMP makes use of `std::list` objects to pass data between the different SMP components. Profiling the code indicates that most of the time in the batched implementation is spent on iterating over the lists, indicating that the batched version would likely run faster than the non-batched version once the CPU-side code is optimized. However, the larger-than-necessary ball size of the batched RRT\*, which leads to faster convergence is an effect that will propagate to future development where the graph operations are parallelized as well.

Fig. 8 illustrates the ratio of the per-iteration time of the RRT\* to the RRT algorithm for the three different variants. Note that the GPU implementation of the collision checker reduces the ratio between the two algorithms significantly, and that the batched version brings the ratio quite close to unity. As mentioned, the bottleneck in the batched version is clearly non-cached CPU memory access during list iteration, so the precision of this ratio may not be accurate for an optimized implementation.

## VI. CONCLUSIONS

In this paper, we have discussed a data-parallel implementation of two sampling-based motion planning algorithms, namely RRT and RRT\*. We have shown that, for the particular problem at hand, the computationally most expensive procedure is that of collision checking, and we have proposed a parallel implementation of this procedure for a manipulator with several degrees of freedom. Our experiments indicate

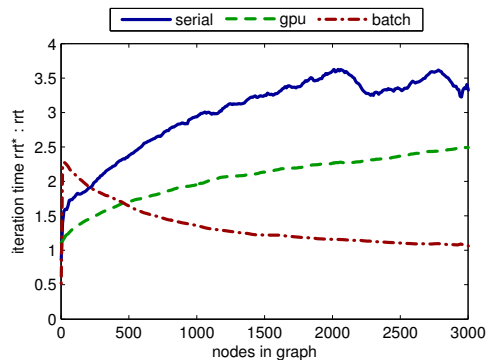


Fig. 8. Ratio of RRT\* to RRT runtime

that both the RRT and the RRT\* algorithms largely benefit from a parallel implementation. Moreover, the results show that parallel implementations can marginalize the differences in run-time between the RRT and RRT\* algorithms. Indeed, should the run-time gap between the RRT and the RRT\* be further reduced, designers will not have to make significant sacrifices to move from the realm of feasibility planning, into the realm of optimal planning. Future work includes optimizing the CPU side of the batched code, and design and development of parallel implementations of graph search procedures.

## REFERENCES

- [1] J. C. Latombe, "Motion planning: A journey of robots, molecules, digital actors, and other artifacts," *International Journal of Robotics Research*, vol. 18, no. 11, pp. 1119–1128, 1999.
- [2] J. H. Reif, "Complexity of the mover's problem and generalizations," in *Proceedings of the IEEE Symposium on Foundations of Computer Science*, 1979.
- [3] L. E. Kavraki, P. Svestka, J. C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.
- [4] S. LaValle and J. J. Kuffner, "Randomized kinodynamic planning," *International Journal of Robotics Research*, vol. 20, no. 5, pp. 378–400, 2001.
- [5] S. LaValle, *Planning Algorithms*. Cambridge University Press, 2006.
- [6] Y. Kuwata, J. Teo, G. Fiore, S. Karaman, E. Frazzoli, and J. How, "Real-time motion planning with applications to autonomous urban driving," *IEEE Transactions on Control Systems*, vol. 17, no. 5, pp. 1105–1118, 2009.
- [7] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *International Journal of Robotics Research (to appear)*, 2011.
- [8] S. Carpin and E. Pagello, "On parallel RRTs for multi-robot systems," in *Proc. 8th Conf. Italian Association for Artificial Intelligence*, Pisa, Italy, Sep 2003, p. pages?
- [9] S. Sengupta, "A parallel randomized path planner for robot navigation," *International Journal of Advanced Robotic Systems*, vol. 3, pp. 256–266, Sep 2006.
- [10] E. Plaku, K. Bekris, B. Chen, A. Ladd, and L. Kavraki, "Sampling-based roadmap of trees for parallel motion planning," *Robotics, IEEE Transactions on*, vol. 21, no. 4, pp. 597 – 608, Aug 2005.
- [11] H. Samet, *Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1989.
- [12] S. Arya, D. M. Mount, R. Silverman, and A. Y. Wu, "An optimal algorithm for approximate nearest neighbor search in fixed dimensions," *Journal of the ACM*, vol. 45, no. 6, pp. 891–923, November 1999.
- [13] Valgrind Developers. (2011, Jul) Callgrind manual. [Online]. Available: <http://valgrind.org/docs/manual/cl-manual.html>
- [14] *NVIDIA CUDA C Programming Guide*, NVIDIA, Nov 2010.