High Performance Computing Project

Parallelization of the Probabilistic Roadmap Method with GPU Acceleration

Manuel Demmeler

1 Motivation

In trajectory optimization with obstacles, it is often necessary to provide feasible initial trajectories. For example in robot motion planning, where the dimensions of the underlying spaces are rather high, this can be expensive. A further problem is that because of the transformation from work into configuration space, the obstacles are often no longer given in a closed form. An efficient approximative solution for this is using probabilistic roadmaps (PRMs).

In this project, a parallelized version of the PRM method has been developed in general. The robotics application then was implemented as an special case of this with GPU support.

2 Introduction

2.1 Problem Statement and the Sequential PRM Method

The abstract problem treated here is to find a path in a d-dimensional space, which avoids intersecting obstacles. The area occupied by obstacles is given by an indicator function

$$I : \Omega \to \{0, 1\}$$

on a region $\Omega = [q_{min,1}, q_{max,1}] \times ... \times [q_{min,d}, q_{max,d}] \subset \mathbb{R}^d$. Hence, the goal is to find a continuous curve

$$\Gamma : [0, 1] \to \Omega$$

between a given start and endpoint, $\Gamma(0) = q_b,\ \Gamma(1) = q_e$, such that

$$I(\Gamma(s)) = 0 \text{ f.a. } s \ \in [0, 1].$$

The PRM algorithm now grows a graph $G = (V, E)$ form points q_s and q_e by randomly sampling nodes $v \in \Omega$ and inserting them into V. Two nodes $v, w \in V$ are connected by an edge, if they are near enough each other and the line between them is free from obstacles. The algorithm terminates, when there exists a path from q_s to q_e on G.

Definition. *For two points q_1 and q_2 we define the connection with stepsize $h > 0$ as*

$$[q_1, q_2]_h := \{q = \lambda q_1 + (1-\lambda) q_2 \mid \lambda \in [0,1], \|q - q_1\| \in \mathbb{N}_0 h\}.$$

We say, q_1 and q_2 are connected with stepsize $h > 0$, if $I(q) = 0$ for all $q \in [q_1, q_2]_h$.

Algorithm 1: Probabilistic Roadmap Algorithm

```
Data: $q_s, q_e \in \Omega, h > 0, d_0 > 0$
Result: $q_s = q_1, ..., q_n = q_e \in \Omega$, such that all $q_i$ and $q_{i+1}$ are connected with
        stepsize $h$
Initialize graph $G = (V, E)$ with $V = \{q_s, q_e\}$
If $q_s$ and $q_e$ are connected, $E \leftarrow \{q_s, q_e\}$
while there exists no path in $G$ between $q_s$ and $q_e$ do
    Sample $q \in \Omega$ from a probability density $p(\,\cdot\,, G)$, resample until $I(q) = 0$
    for all nodes $\tilde{q} \in V$, with $\|q - \tilde{q}\| \leq d_0$ do
        if $q$ and $\tilde{q}$ connected with stepsize $h$ then
            $E \leftarrow \{q, \tilde{q}\}$
        end
    end
    $V \leftarrow q$, if at least one edge was inserted
end
Determine the shortest path $q_1, ..., q_n \in V$ on $G$ from $q_s$ to $q_e$.
```

The probability density $p(\,\cdot\,, G)$ (depending on the actual graph) can be for example first selecting randomly a node $q_0 \in V$ and then sampling a new point q in the neighbourhood of q_0. Theory about success probabilities and convergence results can be found in [1], [2] or [3].

2.2 Application to Robotics

In this project, the PRM algorithm is used to find trajectories for a robot arm, which avoid self collisions and collisions with obstacles in the environment. Such trajectories can be expressed as functions describing the joint variables changing over time, that means as a curve in the configuration space with dimension equal to the number of degrees of freedom of the robot. With this association, the trajectory finding problem becomes a path finding problem in the configuration space exactly of the form defined in the previous section. The real obstacles transform into obstacles in the configuration space, given by the indicator function, that returns for a given vector of joint angles, if it is feasible or leads to a collision.

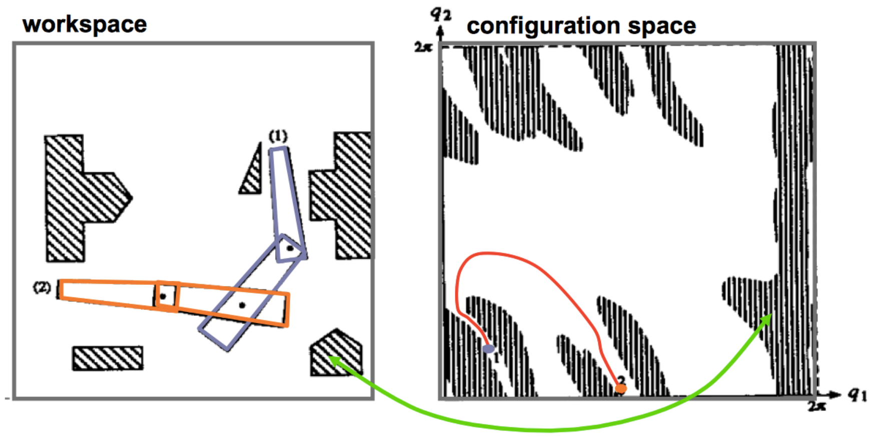
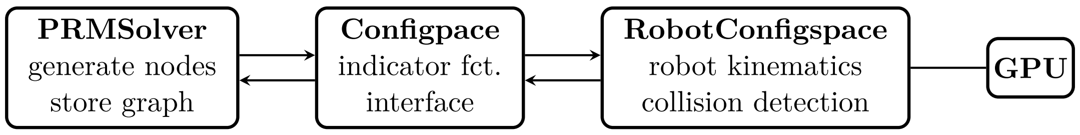


Figure 1: Example from [1] for an indicator function in robotics. The real obstacles on the left are transformed into occupied areas of the configuration space on the right.

2.3 Project Overview

Like in the stated robotics example, in many cases the most costly operation of the PRM algorithm are the many evaluations of the indicator function $I(q)$, which however can be done independently for every new q. Therefore it is implemented in a vectorized way as a function $(q_1, ..., q_P) \mapsto (I(q_1), ..., I(q_P))$ by a CUDA kernel, where every GPU thread computes one evaluation. In the robotics case, this means, the direct kinematic and collision calculation are made on the GPU parallelly for different joint angle configurations. Although this will the only major application in the project, the PRM solver and the configuration space are implemented separated from each other and connected by an interface to be free to simply add other applications. As an example and for testing, additionally a simple 2D configuration space is provided, whose indicator function is read from a black-and-white image.



While the distribution of the indicator function to different CUDA threads is a first level of parallelization, a second level can be achieved by sampling and connecting several new nodes parallelly. This is more difficult, because all new nodes have to be connected and the graph data has to be updated on all processes. For this, different approaches have been implemented, which are described in chapter 4.

Before that, chapter 3 gives more details about the robotics indicator function.

3 Implementation of an Indicator Function for Robot Arms

The probabilistic roadmap algorithm is applied to create collision-free trajectories of a robot arm. For a robot with given joint angles q_i one can calculate straightforwardly all positions and orientations of the single parts of the robot. For this, coordinate frames according to the Denavit-Hartenberg convention are used. The coordinate frames of the single parts are described by transformation matrices $T_i \in R^{4x4}$, which transform points from the body frame into the world frame. They depend on the state $q = (q_1, ..., q_d)$ of the robot and on the geometry of the joint angles given by the Denavit-Hartenberg parameters of the robot.

The geometry of the robot parts is defined by a set of convex polytopes, each of them belonging to one of the robots coordinate frame. Applying the corresponding transformation T_i, its vertices can be transformed into the world frame. Environment obstacles are also defined by convex polytopes, given in the world frame.

Now the movement of the robot can be described by a curve $q(t) \in \mathbb{R}^d$ in the configuration space. It is collision-free, if at no time, any two polytopes of the robot intersect with each other or with the environment after being transformed into the world frame. To apply the probabilistic roadmap algorithm, we express this by the indicator function

$$I(q) = \begin{cases} 0, & \text{for no collision for all given pairs of polytopes} \\ 1, & \text{if at least one collision occurs} \end{cases}$$

The indicator function is implemented in a vectorized way as an CUDA kernel, which means that, given set of states $q^1, .., q^M$, the indicator function is evaluated for each state by one CUDA thread. Therefore, the kinematics and collision algorithm are implemented as device code for the use in the kernel.

It has to be noted that the trajectories $q(t)$, the PRM algorithm will generate, are piecewise linear and therefore not differentiable. Hence, they have to be smoothed before the use on a robot. This is not part of the project. The task is only to generate feasible trajectories to be passed to an optimizer.

3.1 Direct Kinematics

The Denavit-Hartenberg (DH) convention is a method to describe the geometry of a robots joints, that is, how the transformations between the robots coordinate frames look like in dependency of its joint values $q_1, ..., q_d$. A robot consists of $d + 1$ bodies each of which has a body fixed coordinate frame B_i, which is described by a transformation matrix T_i into the world frame. It is assumed that each joint is either rotational, that means an rotation around a fixed axis, or prismatic, a translation along an axis. These joints may now be defined through DH parameters $a_{i-1}, \alpha_{i-1}, d_i, \theta_i \in \mathbb{R}$, $i = 1, ..., d$, by setting the relative transformations $T_{i-1,i} = T_{i-1}^{-1} T_i$ between two bodies to

$$T_{i-1,i} = R_z(\theta_i) T_z(d_i) R_x(\alpha_{i-1}) R_x(a_{i-1}) \tag{1}$$

where R_x, R_z are rotations and T_x, T_z translations around the x and z axes. With this and $T_0 = I$, all transformations T_i are determined.

The other way round, one can now show, that by an proper choice of the body frames, every robot with rotational or prismatic joints can be described by such a set of DH parameters, i.e. it exists a set of parameters, such that the above definition leads to the correct transformations for this robot. For the exact calculations, see [4].

3.2 2D and 3D Geometry Library

For the implementation of the kinematics and later the collision algorithm, a geometry library was written, based on the CUDA datatypes float2 for 2D vectors and float4 for 3D vectors. To guarantee aligned access on the GPU, float4 is used instead of float3. Furthermore, as the geometric calculations made here are numerically not very critical, the usage of double precision has turned out to be not necessary. The library contains hybrid host and device inline functions for common vector operations such as

```
__host__ __device__ inline float4& operator += (float4& u, const float4& v);
__host__ __device__ inline float4& operator *=  (float4& u, const float f);
__host__ __device__ inline float4& add(const float4& u, const float4& v, float4& w);
```

Operators like $+$ or $*$ are not overloaded to prevent unnecessary temporary variables.

Transformation matrices are implemented as float4 arrays with inline functions for common matrix operations. Again operators with temporaries were avoided, wherever possible.

```
class trafo4{
  public:
    float4 col[4];
    __host__ __device__ trafo4(){}
    __host__ __device__ trafo4(float a, float alpha, float q, float d);
    __host__ __device__ inline float4& apply(float4& u) const;
    __host__ __device__ inline float4& apply(const float4& u, float4& Tu) const;
    .
    .
    .
}
```

The second constructor implements the relative DH transformations.

3.3 Polytopes

To calculate the intersection between two convex polytopes, the here used Chung-Wang algorithm only needs their vertices and the data, which of them are connected by an edge. For this the adjacency matrix of the edge graph is stored compactly in compact row storage format.

```
struct polytope4{
    float4* vertices;   //length n
    int n;
    //edges
    int* dsp;    //length n
    int* cnt;    //length n
    int* dest;   //length m
    int m;
};
```

It means, that for $i = 0, ..., n-1$ the vertices $dest[dsp[i]], ..., dest[dsp[i] + cnt[i] - 1]$ are the ones connected with vertex i by an edge. The polytopes used in this project are given by the user as convex hull of a list of vertices. However, to determine the edge graph only from the vertices is a nontrivial task, which is done by Matlab here using provided high level functions.

3.4 Chung-Wang Collision Algorithm

To check if two polytopes collide an algorithm by Chung and Wang is used here. It uses the separating vector theorem and tries to find a separating vector by iterative search. Furthermore, a sub-algorithm determines in each iteration from the candidate history, if it is still theoretically possible to find a separating vector. If not, it is returned a collision. If on the other side a separating vector is found, the algorithm can return no collision. For detailed informations see [5] and [6], pp. 410-412. The method is implemented as device code for the use on GPU kernels.

```
__host__ __device__ int separating_vector_algorithm(const polytope4& P, const polytope4& Q,
                                                    const trafo4& tp, const trafo4& tq);
```

In our application, the polytopes are mostly parts of the robot, which are given in a body frame and have to be transformed into world frame according to section 3.1 first. However, from the structure of the Chung-Wang algorithm drops out, that mostly only a small part of the polytopes vertices are needed for the computations. Therefore, it would be rather time wasting to transform all vertices before passing them to the algorithm. To avoid this, all polytopes are passed in their body frames together with their transformations and the algorithm transforms a vertex, whenever it occurs in calculations. This is also the version proposed in [5].

3.5 Robot Kernel

The robot configuration space is implemented in the form

```
template<int ndof>
class RobotConfigspace : public Configspace<ndof>
{
public:
  RobotConfigspace(const Robot<ndof>* robot_,
                   const polytope *polys_,
                   const int* sys_,
                   const int N_,
                   const int *from_, const int *to_,
                   const int M_,
                   const float* mins_, const float* maxs_, const float dq_,
                   const int nbuf_);
  int init(const int ressource_rank=0, const int ressource_size=1);
  int indicator2(const float* qs, const float* qe, int *res, const int N, const int offset);
  .
  .
  .
private:
  const Robot<ndof>* robot; //host object
  Robot<ndof>* robotdev;    //GPU object
  collision4::polytope4data* polydata;    //host object
  collision4::polytope4data* polydatadev; //GPU object
  .
  .
  .
}
```

```
__global__ void kernel_indicator2(const Robot<ndof>* robot,
                                  const collision4::polytope4data* polydata,
                                  const float* qs, int offsets,
                                  const float* qe, int offsete,
                                  int* res,
                                  const int* testpos, const int* testnum,
                                  int N, int numthreads);
```

where the constant geometry and robot data is loaded once at the beginning and copied to the GPU by an init function. The indicator function gets as input a list of border points $q^j_{start}, q^j_{end}, j = 0, ..., N$ and checks now for each pair, if the indicator function is always 0 on the line in between.

$$res[j] = \begin{cases} 0 & \text{if } I(q) = 0 \text{ f.a. } q \in [qs[j], qe[j]]_{\Delta q} \\ 1 & else \end{cases} \tag{2}$$

The border points are stored in structures of arrays.

$$\begin{aligned} q^j_{start} &= (qs[j], qs[j + \text{offset}], ..., qs[j + (d-1) * \text{offset}])^T \\ q^j_{end} &= (qe[j], qe[j + \text{offset}], ..., qe[j + (d-1) * \text{offset}])^T \end{aligned} \tag{3}$$

As the PRMSolver only needs to know, which node points can be connected by a line, and not, which intermediate points are free in detail, this version delivers exactly the minimal necessary information. Furthermore, it saves CPU and communication time, because the intermediate points on the lines can be directly computed by the GPU threads and do not need to be exchanged.

Algorithms 2 and 3 show in pseudo code, how the work is split between host and device.

Algorithm 2: Host indicator function

```
Data: qs, qe, N, offset
Result: res
compute distances $d_j = \left| q^j_{end} - q^j_{start} \right|$ ;
testnum[j] = $d_j / \Delta q$: number of threads for each line ;
testpos[j] = thread displacements ;
reset res ;
call kernel ;
```

In order to overlap computation time of GPU and CPU, an asynchronous version of the indicator function was written. Here the function is divided into everything until the kernel launch and a function, which waits, until the kernel has finished and receives the result data.

```
int indicator2_async(const float* qs, const float* qe, int *res,
                     const int N, const int offset, int &request);
int indicator2_async_wait(int request);
```

The implementation also allows overlapping of multiple requests, for example two calls of the launch function and after this the two corresponding waiting calls. With this, it is possible to use the full computation time of the GPU.

Algorithm 3: Robot kernel: code for one thread

```
Data: arrays qs, qe, testpos, testnum and threadindex
Result: res
compute associated edge number $j$, defined by
$testpos[j] \leq threadindex \leq testpos[j] + testnum[j]$ ;
$c = (threadindex - testpos[j])/(testnum[k] - 1)$ ;
$q = cq^j_{start} + (1 - c)q^j_{end}$ ;
calculate robot kinematics $T_0(q), ..., T_d(q)$;
$res_{tmp} = 0$; ;
for all registered pairs pairs of polytopes $P, Q$ do
    $res_{tmp} = separating\_vector\_algorithm(P, T_{i_P}, Q, T_{i_Q})$ ;
    if $res_{tmp} \neq 0$ then
        $res[j] = res_{tmp}$ ;
        break ;
    end
end
```

4 The PRM Solver

The solver builds a first phase the roadmap graph from both the start and the end node, until it is connected. It is always ensured that the graph consists of two connected components containing the start resp. the end node. With this, it can be easily determined at each time, if the whole graph has become connected.

At the end, in a second phase, a shortest path from the start to the end node is searched by the Dijkstra algorithm. It has shown, that this part needs a neglectable amount of time compared to building the graph. Therefore, it is not parallelized here and the focus lies on the first phase.

4.1 Storing the Graph

Apart from the indicator function evaluation, the main problem in building the probabilistic roadmap is, that for each new node q it have to be determined all possible neighbors. That is the set of all nodes $\tilde{q}$ of the graph, with $\|\tilde{q} - q\| \leq D$. If no special order of the nodes is given, one has to calculate this norms for all nodes of the graph, which is rather time consuming.

Therefore, a special indexing is used here. The nodes are stored in a vector, which is organized in blocks with fixed sizes. The blocks are referred to by ids which are stored in a map.

```
struct block{
    int pos;      //! position of first vertex
    int num;      //! current number of vertices stored
    block* next; //! if num==blocksize -> pointer to next block
};
```

```
struct graph{
    std::map<int,block*> map; //map for accessing blocks
    std::vector<block> blocks;
    std::vector<float> qstorage; //length ndof*N
    int newblockpos;  //position of next new block
    int blocknum;     //number of used blocks
    .
    .
};
```

The key of an block is computed by the mapping

$$key(q) = \left\lfloor \frac{q_1}{H} \right\rfloor$$

such that $H > 0$ becomes a parameter to adjust the grid density. This has to be chosen so large that enough nodes are stored in one block in order to have efficient memory access. On the other side, the smaller it is, the less unnecessary computations have to be done for the neighborhood requests. For illustration of the structure, the following pseudocode shows the insertion of a node.

Algorithm 4: Inserting a node

```
Data: $q \in \mathbb{R}^d$
Compute key=key(q)
if key exists in map then
    block=map[key]
    while block full do
        block=block->next
    end
else
    insert new block at map[key]
end
Insert node into block
```

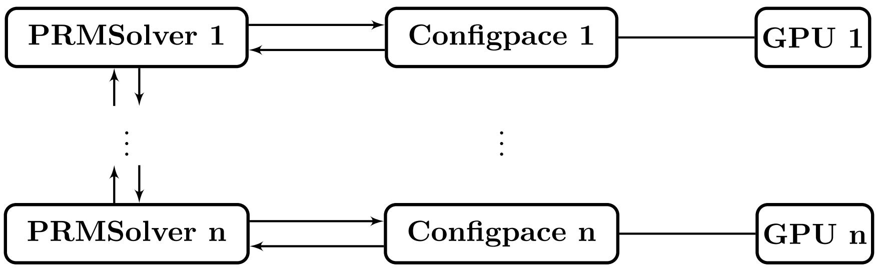
For getting the neighbors of a node q it is then enough to look at the keys from $key(q - cD)$ to $key(q + cD)$. For $D = H$, $c = 1$ is possible. The key mapping could be generalized by taking more components of q into account, but that was not done here.

4.2 Parallelization Models

The parallelization is done with MPI. All processes store their own versions of the graph, which are hold up to date between each other. Furthermore, it is assumed that every process has its own GPU. The work of the single MPI processes could be parallelized further with OpenMP, but this has not been done here. Instead, different approaches to distribute the work over several GPUs have been investigated.

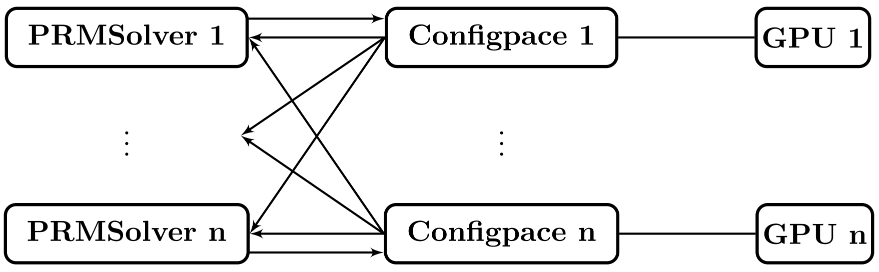
Version 1

In the first approach, that has been evaluated, every process generates new nodes independently from each other. Then the possible neighbors are determined and the connections, which have to be tested. The indicator function is called and returns the edge data. After this, the graph must be synchronized between all processes, which needs an all-to-all communication. As the graph building phase does not need the edge data, it is sufficient to exchange only the nodes and collect the edges at the end.



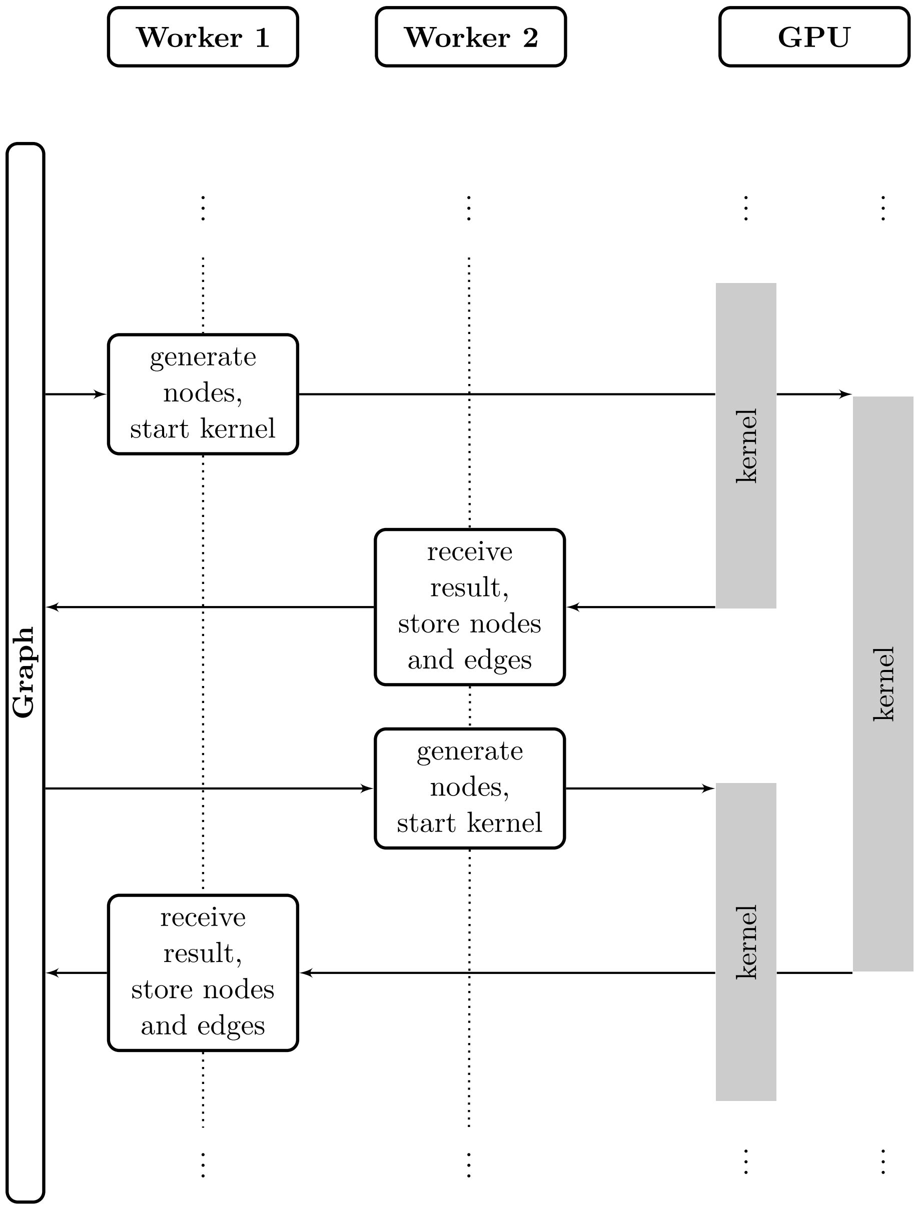
Version 2

The second approach uses the fact, that by synchronizing the random seed, different processes can produce the same sequence of random numbers independently from each other. By using this, every process can generate all new nodes and determine the connection data. Then the indicator function computation is split over their configuration space instances and the returned data shared afterwards with each other. With this data, all graphs can be updated and it is guaranteed, that they grow exactly in the same way on each process. The purpose of this version compared to the first one is, that less data has to be sent. Furthermore, in the first version it can occur, that the different generated nodes can result in very different counts of neighbors and therefore uneven loads on the GPUs. Here, the distribution of the computation amount over the GPUs can be chosen almost freely.



Version 3

The third version does a further optimization by using the asynchronous version of the indicator function. Here, every process runs two workers overlapped. In the time, when the first worker has generated his indicator function request, launched the kernel and now is waiting until it has finished, the second worker updates the graph and sends his new request, and the other way round. With this it is possible, that the computation time of the GPU is fully used, and overlapped with the communication.



Worker 1
Worker 2
GPU
Graph
generate nodes, start kernel
kernel
receive result, store nodes and edges
kernel
generate nodes, start kernel
receive result, store nodes and edges
kernel

5 Results

Tests have been made for a scenario with a robot with 4 joints. For creating and plotting the environment and the resulting trajectories, Matlab functions are provided. Because of the dependency on random numbers, the algorithm takes different amounts of time for different runs. Therefore, 10 runs were made for each version and number of processes. However, the different versions lead to a different average amount of GPU threads needed. Hence, to be able to better compare how they scale, the time per number of GPU threads is used as a quantity. Table 1 shows the results. One can see, that the version with node communication has a better time to GPU threads ratio, but leads to a greater amount of threads on average. The versions with less communication, however, have better absolute times. Furthermore, one sees the improvement from version 2 to version 3 from the asynchronous indicator function.

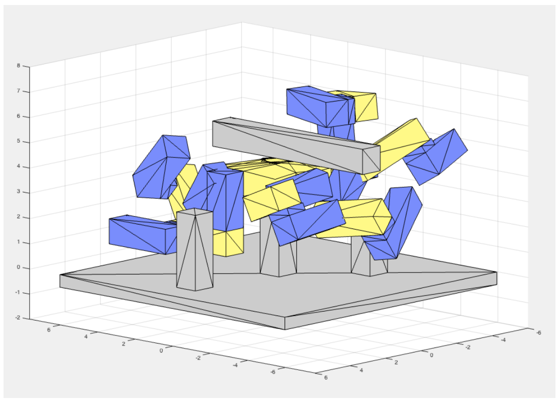


Figure 2: PRM generated collision-free movement for a 4-axis robot

version	processes/ GPUs	time (ms)	GPU threads	time per 1e6 GPU threads (ms)
1	1	1002	696413	1448
	2	763	838938	918
	4	706	1113781	647
2	1	900	608172	1490
	2	647	608172	1070
	4	568	608172	947
3	1	696	599847	1181
	2	554	599847	951
	4	500	599847	864

Table 1: Timing results for the parallelization versions from chapter 4 on a cluster node with 2x Intel Xeon Six-Core CPU X5650 @2.67GHz and 4x GeForce GTX 480

As the indicator function has also been implemented on the CPU, all versions could be run without GPU. In most cases, the GPU code was around 10 times faster. However, the robot code is not optimized for the CPU version, such that this comparison is not quite fair.

References

[1] D. Burschka, Lecture Robot Motion Planning, source: http://robvis01.informatik.tu-muenchen.de/courses/wegtraj/index.html

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

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

[4] T. Buschmann, Skript zur Vorlesung: Roboterdynamik SS14, Lehrstuhl für Angewandte Mechanik, TU München, 2014

[5] K. Chung, Wenping Wang, Quick Collision Detection of Polytopes in Virtual Environments, Proceedings of the ACM Symposium on Virtual Reality Software and Technology (VRST 96), Mark Green (ed.), 1996

[6] C. Ericson, Real-Time Collision Detection, Elsevier, 2005