A Membrane Parallel Rapidly-Exploring Random Tree Algorithm for Robotic Motion Planning

Ignacio Pérez-Hurtado [a], Miguel Á. Martínez-del-Amor [a], Gexiang Zhang [b], Ferrante Neri [c,1] and Mario J. Pérez-Jiménez [a]

[a] *Research Group on Natural Computing, Dpt. Computer Science and Artificial Intelligence, School of Computer Engineering, Universidad de Sevilla, Seville, Spain*

[b] *College of Information Science and Technology, Chengdu University of Technology, Chengdu, People's Republic of China*

[c] *COL Labloratory, School of Computer Science, University of Nottingham, Nottingham, United Kingdom*

Abstract.
In recent years, incremental sampling-based motion planning algorithms have been widely used to solve robot motion planning problems in high-dimensional configuration spaces. In particular, the Rapidly-exploring Random Tree (RRT) algorithm and its asymptotically-optimal counterpart called RRT* are popular algorithms used in real-life applications due to its desirable properties. Such algorithms are inherently iterative, but certain modules such as the collision-checking procedure can be parallelized providing significant speedup with respect to sequential implementations. In this paper, the RRT and RRT* algorithms have been adapted to a bioinspired computational framework called Membrane Computing whose models of computation, a.k.a. P systems, run in a non-deterministic and massively parallel way. A large number of robotic applications are currently using a variant of P systems called Enzymatic Numerical P systems (ENPS) for reactive controlling, but there is a lack of solutions for motion planning in the framework. The novel models in this work have been designed using the ENPS framework. In order to test and validate the ENPS models for RRT and RRT*, we present two ad-hoc implementations able to emulate the computation of the models using OpenMP and CUDA. Finally, we show the speedup of our solutions with respect to sequential baseline implementations. The results show a speedup up to 6x using OpenMP with 8 cores against the sequential implementation and up to 24x using CUDA against the best multi-threading configuration.

Keywords. Optimal Motion Planning, Rapidly-exploring Random Tree, Membrane Computing, OpenMP, CUDA

1. Introduction

The development of modern autonomous robots poses difficult tasks associated with the interaction between the robot and the external world, see [25,64]. For example, a shape reconstruction system of 3D objects is proposed in [68] and a vision based navigation system for drones is proposed in [5]. Several studies on the modelling of the artificial visual system have been performed, e.g. a neural model [40] and methods on the functioning of artificial compound eye in [74,73]. The task of detecting and interpreting a moving object, that is motion tracking, is fundamental for the correct functioning of a robot, [8,76].

This article studies a crucially important problem related to motion tracking, i.e. motion planning. The motion planning problem consists of finding a trajectory to move an agent trough a complex environment from a starting point to a desired area avoiding any obstacle while considering constraints related to the agent such as shape, kinematics and others. This problem is critical in almost all robot applications since, by definition, a robot is a machine developing tasks in the real world. Furthermore, this problem is also relevant in many other research areas such as verification, computational biology, and computer animation [34]. The first computational complexity results

[1] Corresponding Author: School of Computer Science, Jubilee Campus, Wollaton Road, Nottingham NG8 1BB, United Kingdom, Email: ferrante.neri@nottingham.ac.uk.

for robot motion planning were due to J. Reif [63]. Specifically, he established that the cited problem is PSPACE-hard when the final positions of the obstacles are specified. Moreover, this problem belongs to the class PSPACE when it is expressed using polynomials with rational coefficient [12], that is, this variant is a PSPACE-complete problem. Nevertheless, the robot motion planning problem where the final positions of the obstacles are unspecified, is shown to be NP-hard [22].

Several approximated solutions have been proposed to the motion planning problem. For example, an incremental search algorithm called D* [70] has been used for path planning in real-time environments. In [15], an extension of the classical A* algorithm is given for grids with blocked and unblocked cells. In [48], the authors studies the motion planning problem in the challenging context of navigation through real-weld, where traditional on-site sign posts could be limited or not available.

An special mention should be given to a category of algorithms to build Rapidly-exploring Random Trees (RRTs) [35]. They are based on the randomized exploration of the configuration space by building a tree where nodes represent reachable states and edges represent transitions. In particular, the RRT* algorithm [29] is able to build an RRT whose paths asymptotically converge in time of computation to optimal motion planning solutions considering a cost function. Several variants of the RRT and RRT* algorithms have been designed. For example, RRT*-Smart [44] is a method for accelerating the convergence rate of RRT*. A*-RRT and A*-RRT* [10] are two-phase motion planning methods that use a graph search algorithm to search an initial feasible path in a low-dimensional space in a first phase and then focus the RRT* in the second phase. RRT*-FN [4] is a RRT* with a fixed number of nodes, which randomly removes a leaf node in every iteration. Informed RRT* [21] improves the convergence speed of RRT* by introducing a heuristic. While the base RRT algorithms are inherently sequential, there are modules that can be parallelized such as the obstacle collision detection [9,20]. An experimental comparison is proposed in [46]

Membrane Computing [58,24,61,62] is a computing paradigm inspired from the living cells and it provides distributed, massively parallel devices, see [1,66,67]. Models in Membrane Computing are generically called P systems and they have been used in different contexts. Among others, we stress the following: (a) showing the ability of some models to give polynomial time solutions to computationally hard problems, by trading space for time; (b) providing a new methodology to tackle the **P** versus **NP** problem [57,50,37,69]; (c) being a framework for modelling biomolecular processes as well as real ecosystems [60,23,7,14]; (d) incorporating fuzzy reasoning in models that mimic the way that neurons communicate with each other by means of short electrical impulses, and applying them to many different industrial applications related to fault diagnosis [72].

In this paper, Membrane Computing is used to design bio-inspired parallel RRT models that can be efficiently simulated by means of parallel software/ hardware architectures such as OpenMP [49] and CUDA [47]. A variant of P systems called Enzimatic Numerical P systems (ENPS, for short) has been used to model and simulate robot controllers [54,55,77,19]. An extension of ENPS, called RENPSM, was used for the first time in [56] to simulate basic RRT algorithms.

The main contribution of this work is to use the framework of ENPS for modelling the RRT and the RRT* algorithms. It is worth pointing out that the major advantage of using P systems is the inherent parallelism associated with the theoretical computing devices that they provide. Moreover, no additional *ingredients* to the ENPS framework has been included, as in [56], where features such as proteins and shared memory were used. In consequence, the presented models are compatible with existing ENPS robot controllers given that they belong exactly to the same framework. In order to validate and test these models, two ad-hoc simulators have been implemented in OpenMP and CUDA, showing experimental results in four scenarios. Henceforth, the parallel implementations show that the introduced ENPS-based models can be easily parallelized for multicores and manycores at high-end servers as well as on-board platforms.

The rest of this paper is structured as follows. Next section summarizes some preliminary concepts. Section 3 is devoted to present two specific ENPS models for RRT and RRT*. The simulators implemented in OpenMP and CUDA are described in Sec-

tion 4. The experimental results for validation and testing are presented in Section 5. The paper ends with the main conclusions of the work.

2. Preliminaries

This Section provides the reader with the basic concepts and notation used throughout this paper.

2.1. Motion planning

In general terms, the problem of motion planning can be defined in the configuration space of a mobile agent as follows. Given:

- An initial configuration state.
- A set of valid final configuration states.
- A map of obstacles in the environment.
- A description of the agent shape.
- A description of the agent kinematics.

Find a sequence of configuration states through the configuration space, a.k.a. trajectory or plan, from the initial state to one of the final states, which does not touch any obstacle in the environment considering the agent shape and kinematics.

There are two variants of the problem:

- **The feasibility problem** is to find a feasible trajectory, if one exists, and report failure otherwise.
- **The optimality problem** is to find a feasible trajectory with minimal cost where the cost of a trajectory is given by a computable function.

In a robot navigation software architecture, the module in charge of solving the motion planning problem is called *Global Planner*. It computes trajectories by means of an anytime algorithm, i.e, an algorithm that can return a valid solution to the problem even if it is interrupted before it ends. After that, the *Local Planner* module generates motion commands to follow the trajectory in a safe manner considering the information given by the sensors in real-time. Finally, the *Controller* module manages the power of the actuators to fit each motion command and maintain a constant velocity until the next command, see [11].

2.2. Rapidly-exploring random trees

An RRT [35] is a randomized tree structure for rapidly exploring the obstacle-free configuration space. It has successfully been used to solve nonholonomic and kinodynamic motion planning problems [36]. Nodes in an RRT represent possible reachable states, edges represent transitions between states. The root of an RRT is the initial state. Each state in an RRT can be reached by following the trajectory from the root to the corresponding node, as can be seen in Figure 1. Algorithms used in robotics to generate RRTs are known to be anytime and any-angle, i.e, the produced trajectories could contain turns in any valid angle considering the robot constraints and kinodynamics.

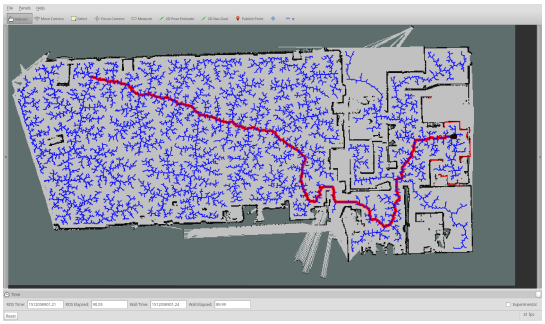


Figure 1. A trajectory conducted by an RRT

2.2.1. The RRT algorithm

The first algorithm to generate RRTs, called the RRT algorithm [35], was able to solve the feasibility problem for motion planning in robotics.

2.2.2. The RRT algorithm*

The RRT* algorithm [29] is a variant of the previous algorithm which is able to build an RRT whose branches asymptotically converge in time of computation to optimal motion planning solutions with respect to a cost function, i.e, it is able to solve the optimality problem using an infinite time of computation. Nevertheless, the algorithm solves the feasibility problem and provides *good* solutions with respect to the cost function in an anytime and any-angle fashion [29].

2.3. Membrane Computing

P systems are theoretical computing devices biologically inspired from the structure and processes taking place in living cells [58]. As mentioned above, Membrane Computing is the field that studies these devices. The syntactic ingredients of P systems are an abstraction of chemical components and compartments of cells: a set of *membranes* delimiting regions where *multisets of objects* reside and evolve according to a set of *rewriting rules*. The type of membrane structure and rules depends on the class of P system, which can be *cell* (rooted tree), *tissue* (undirected graph) or *neural-like* (directed graph). So far, a wide variety of *variants* have been define within each class that answers to different semantics. In general, a computation of a P system is a sequence of *configurations* provided by *transitions steps* where the rules are executed and modify the state of the system. P systems are inherently non-deterministic and massively parallel devices, given that different sets of rules might be applicable at the same time, and each of such sets includes a high amount of rules that are executed in parallel. A global clock takes the control of each transition step.

Rules can modify only the objects in the region where they are defined, or exchange objects with a neighbor region, or modify the membrane structure by dividing or dissolving compartments, etc. Membranes can also be enriched by associating electrical charges to them. The objects that can appear in the membranes are defined in an alphabet. Several different semantics have been defined for P systems. Usually, the rules of a P system are executed in a maximally parallel way; that is, any rule that can be executed in a transition step must be executed and cannot remain unselected. On the other hand, it is possible to use a minimal parallel policy of execution; that is, at least one rule within each region must be executed. Some types of rules might be incompatible with the execution of other (e.g. division and send-out rules in active membranes), and this is given by a derivation mode.

Main research concerning P systems focuses on their computational power and efficiency, see [27,6, 39]. In this sense, they have been used as a tool to attack the P vs NP problem by sharpening the frontiers [57,50,37]. However, their flexibility and massively parallelism has been employed to define modelling frameworks for real-life phenomena, such as biological systems at both micro (e.g. signally pathways, bacteria colony) [60,23] and macro (e.g. ecosystems, population dynamics) levels [7,14], physical systems, image processing, robot control, economic systems, etc. In general, solutions based on P systems must implement parallelism, given that this component is natural and inherent. These applications have been accompanied normally with software simulators in order to ease validation and virtual experimentation processes [16,71]. A recent trend is to accelerate these simulations by using parallel devices like GPUs [41,42,13].

2.4. Numerical P systems

Numerical P systems (NPS) are P systems, see [52, 75], introduced in [59] to model economical and business processes. In NPS, the concept of multisets of objects is replaced by numerical variables that evolve from initial values by means of production functions and repartition protocols. A numerical P system is formally expressed by:

$$\Pi = (m, H, \mu, (Var_1, Pr_1, Var_1(0)), \ldots, (Var_m, Pr_m, Var_m(0)))$$

where

- m is the number of membranes; $m \geq 1$;
- H is an alphabet of labels, containing m symbols;
- μ is the membrane structure;
- Var_i is a set of variables for compartment i, being $Var_i(0)$ their initial values;
- Pr_i is a set of programs for compartment i. A program has the following syntax:

$$F(x_1, \ldots, x_k) \rightarrow c_1|v_1 + \ldots + c_n|v_n$$

where

  * The left-hand-side of the program is called *production function* and the right-hand-side is called *repartition protocol*.
  * $F(x_1, \ldots, x_k)$ is a function $\mathbb{R}^k \rightarrow \mathbb{R}$ using variables $x_1, \ldots, x_k$.

* $v_1, \ldots, v_n$ are output variables. The output value of F will be distributed among the output variables according to the repartition protocol given by $c_1, \ldots, c_n$.
* $c_1, \ldots, c_n$ are numeric values representing the portion of the output which is going to be assigned to the corresponding variable.

For the sake of simplicity, in the rest of this paper, we will use the next syntax:

$$F(x_1, \ldots, x_k) \rightarrow v$$

when there is one and only one output variable in the repartition protocol.

2.5. Enzymatic numerical P systems

Enzimatic numerical P systems (ENPS) are an extension of NPS introduced in [54] for modeling and simulation of membrane controllers for autonomous mobile robots. The main difference of ENPS with respect to NPS is the concept of *enzyme* which is used to write conditions related to programs. The formal definition of the ENPS is the following:

$$\Pi = (m, H, \mu, (Var_1, E_1, Pr_1, Var_1(0)), \ldots, (Var_m, E_m, Pr_m, Var_m(0)))$$

where

- m, H, μ, Var_i and $Var_i(0)$ have the same meaning than explained in Subsection 2.4.
- E_i is a set of variables $E_i \subseteq Var_i$ called enzymes.
- Pr_i is a set of programs for compartment i. The syntax of a program is the following:

$$F(x_1, \ldots, x_k)|_{Cond(e_1, \ldots, e_r)} \rightarrow c_1|v_1 + \ldots + c_n|v_n$$

where

- $F(x_1, \ldots, x_k)$ and $c_1|v_1 + \ldots + c_n|v_n$ have the same meaning than explained in Subsection 2.4.

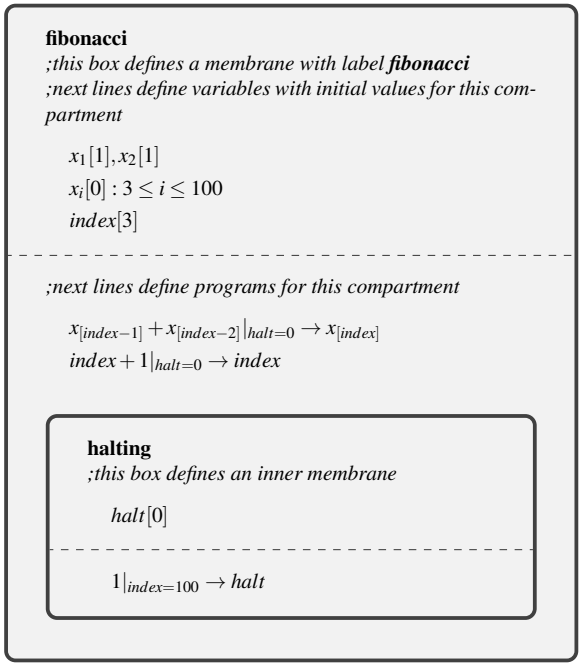


Figure 2. ENPS generating the Fibonacci sequence

- $Cond(e_1, \ldots, e_r)$ is a condition function $\mathbb{R}^r \rightarrow \{true, false\}$ using enzymes $e_1, \ldots, e_r$. The program is disabled when the output of such a function is $false$.

For the sake of simplicity, it is not necessary to write the condition function when it is the constant function $Cond() = true$.

Figure 2 shows an ENPS model to generate the 100 first terms of the fibonacci succession as a *toy example* to illustrate how ENPS models are going to be written in this paper.

The membrane structure is designed to organize programs and variables in modules. All the variables are considered global, i.e, any program can read or write a variable regardless of the compartment where the variable is defined. The definition of variables as well as their initial values is given by the syntax $x[v]$ where x is a variable and v is a numeric value. The reserved word *input* can be written instead of a numeric value for v when the initial value should be read from external inputs, e.g, robot sensors. In this paper, we will use the syntax $a_{[b]}$ in order to refer to variable a_i where i is the value stored in variable b.

2.6. Parallel Computing: OpenMP and CUDA

Since the advent of the transistor, computer processors have been increasing the amount of resources in order to achieve higher efficiency, and so, power [2,65,32]. Recent trend is to incorporate more computing cores within the same chip, so today we can easily find multicore (up to dozens of cores) and manycore (up to thousands of cores) processors in the market [3,53]. The former concerns current CPUs, and the latter involves mainly GPUs. The main standard to harness the parallelism in multicore processors is *OpenMP* [49], which is managed by the OpenMP ARB consortium. First introduced in the late 90's, it provides an API for common languages such as Fortran, C and C++, and that is supported in the majority of systems. The main advantage of OpenMP is its ease of use, given that the programming is made through directives on top of the code. The execution model is fork-join, where a master takes control of the children and synchronizes the threads created. This model is also based on shared memory (all threads have access to the same memory system), although there are variants for distributed memory as well. OpenMP has worked also effectively with other standards for distributed computing such as MPI [28], and in fact, this combination is widely used in today clusters and supercomputers [30].

Nowadays, GPUs can be used as manycore processors to accelerate scientific computation [31,18]. CUDA [47] is the most widely used programming model, given that it was first introduced in 2007 by NVIDIA, and it has been strongly supported by this company. Although OpenCL is the standard introduced by Kronos for GPU computing, CUDA is still better supported by NVIDIA GPUs. In any case, similar concepts can be found in OpenCL and in CUDA. It is a heterogeneous programming model where CPU (host) and GPU (device) are separated, and so, has different memory spaces. GPUs contain several multiprocessors including computing units and fast shared memory system, and all multiprocessors are interconnected with a global memory that can be also accessed by the CPU, specially to copy and retrieve data. CUDA provides an abstraction of the GPU in form of threads that execute the same code which is a function called kernel [45]. Threads are grouped in blocks, so each block is assigned to a multiprocessor, being able to efficiently cooperate locally inside the blocks. CUDA programmers have to manage these aspects manually: thread grid and memory layout. Moreover, threads are executed completely in parallel when they are fully synchronized in the code; that is, they follow a SIMD fashion (single instruction multiple data). Moreover, accesses to memory get optimized when threads query contiguous positions of data [45,51]. Therefore, parallel algorithms have to be carefully adapted to this model bearing in mind several efficiency aspects. Some algorithms fit very well to the GPU architecture and so they are employed as primitives and building blocks, such as map (application of a function to all elements of a vector), reduction [33] (calculation of a function over all elements of a vector), scan (accumulative application of a function to elements of a vector), etc. [31,51].

3. ENPS models for RRT algorithms

In this section, four ENPS models are presented in order to simulate the behavior of the RRT and RRT* algorithms taking advantage of the inherent parallelism level existing in the membrane computing framework. We have divided the whole problem in the next subproblems:

- Find the nearest point to a given point according to the Euclidean distance.
- Determine if a given trajectory is obstacle free.
- Simulate the RRT algorithm.
- Simulate the RRT* algorithm.

The main ideas of the P system design are the following:

- To use enzymes in order to synchronize the processes.
- To compute Euclidean distances in parallel.
- To use a parallel reduction process in order to compute minimum distances.
- To add a node to the RRT in each iteration.

3.1. Finding the nearest point

Given a set of points $X = \{(x_i, y_i)\} : 1 \leq i \leq 2^n$ and a target point (x_t, y_t), find the nearest point (x, y) in X to the target point according to the Euclidean distance.

3.1.1. Solution for n=3

The solution of the nearest point for $n = 2$ is shown in Figure 3 Where

$$min(a,b) = \begin{cases} a & a < b \\ b & a \geq b \end{cases}$$

$$min^*(a,b,c,d) = \begin{cases} a & c < d \\ b & c \geq d \end{cases}$$

Nearest(3)

$x_i[input], y_i[input], d_i[0], x'_i[0], y'_i[0] : 1 \leq i \leq 8$
$x_t[input], y_t[input], x_{nearest}[0], y_{nearest}[0], \alpha[1], halt[0]$

$(x_i - x_t)^2 + (y_i - y_t)^2|_{\alpha=1} \rightarrow d_i : 1 \leq i \leq 8$
$x_i|_{\alpha=1} \rightarrow x'_i : 1 \leq i \leq 8$
$y_i|_{\alpha=1} \rightarrow y'_i : 1 \leq i \leq 8$
$min(d_i, d_{i+4})|_{\alpha=2} \rightarrow d_i : 1 \leq i \leq 4$
$min^*(x'_i, x'_{i+4}, d_i, d_{i+4})|_{\alpha=2} \rightarrow x'_i : 1 \leq i \leq 4$
$min^*(y'_i, y'_{i+4}, d_i, d_{i+4})|_{\alpha=2} \rightarrow y'_i : 1 \leq i \leq 4$
$min(d_i, d_{i+2})|_{\alpha=3} \rightarrow d_i : 1 \leq i \leq 2$
$min^*(x'_i, x'_{i+2}, d_i, d_{i+2})|_{\alpha=3} \rightarrow x'_i : 1 \leq i \leq 2$
$min^*(y'_i, y'_{i+2}, d_i, d_{i+2})|_{\alpha=3} \rightarrow y'_i : 1 \leq i \leq 2$
$min(d_1, d_2)|_{\alpha=4} \rightarrow d_1$
$min^*(x'_1, x'_2, d_1, d_2)|_{\alpha=4} \rightarrow x_{nearest}$
$min^*(y'_1, y'_2, d_1, d_2)|_{\alpha=4} \rightarrow y_{nearest}$
$\alpha + 1|_{halt=0} \rightarrow \alpha$
$1|_{\alpha=4} \rightarrow halt$

Figure 3. Nearest(3) procedure

The P system computes in one step of computation the squared Euclidean distance for all the points in X to the target point. After that, a reduction operation is conducted in 3 steps to compute the minimum distance as well as the nearest point in X. The computation stops after 4 steps, then *halt* is set to 1 and the nearest point is stored in $(x_{nearest}, y_{nearest})$. Variable α is used as an step counter.

3.1.2. General solution

The general solution, for n points is given in Figure 4

After $n+1$ steps, the nearest point is stored in $(x_{nearest}, y_{nearest})$.

Nearest(n)

$x_i[input], y_i[input], d_i[0], x'_i[0], y'_i[0] : 1 \leq i \leq 2^n$
$x_t[input], y_t[input], x_{nearest}[0], y_{nearest}[0], \alpha[1], halt[0]$

$(x_i - x_t)^2 + (y_i - y_t)^2|_{\alpha=1} \rightarrow d_i : 1 \leq i \leq 2^n$
$x_i|_{\alpha=1} \rightarrow x'_i : 1 \leq i \leq 2^n$
$y_i|_{\alpha=1} \rightarrow y'_i : 1 \leq i \leq 2^n$
$min(d_i, d_{i+2^{n-j}})|_{\alpha=j+1} \rightarrow d_i : 1 \leq i \leq 2^{n-j}, 1 \leq j \leq n$
$min^*(x'_i, x'_{2^{n-j}}, d_i, d_{2^{n-j}})|_{\alpha=j+1} \rightarrow x'_i : 1 \leq i \leq 2^{n-j}, 1 \leq j < n$
$min^*(y'_i, y'_{2^{n-j}}, d_i, d_{2^{n-j}})|_{\alpha=j+1} \rightarrow y'_i : 1 \leq i \leq 2^{n-j}, 1 \leq j < n$
$min^*(x'_1, x'_2, d_1, d_2)|_{\alpha=n+1} \rightarrow x_{nearest}$
$min^*(y'_1, y'_2, d_1, d_2)|_{\alpha=n+1} \rightarrow y_{nearest}$
$\alpha + 1|_{halt=0} \rightarrow \alpha$
$1|_{\alpha=n+1} \rightarrow halt$

Figure 4. General Nearest(n) procedure

ObstacleFree(m,ξ)

$a_i[input], b_i[input], d'_i[0] : 1 \leq i \leq 2^m$
$x_0[input], y_0[input], x_1[input], y_1[input]$
$collision[0], \alpha[1], halt[0]$

$pDist(a_i, b_i, x_0, y_0, x_1, y_1)|_{\alpha=1} \rightarrow d'_i : 1 \leq i \leq 2^m$
$min(d'_i, d'_{i+2^{m-j}})|_{\alpha=j+1} \rightarrow d'_i : 1 \leq i \leq 2^{m-j}, 1 \leq j < m$

$min(\xi - d'_1, \xi - d'_2)|_{\alpha=m+1} \rightarrow collision$
$\alpha + 1|_{halt=0} \rightarrow \alpha$
$1|_{\alpha=m+1} \rightarrow halt$

Figure 5. Procedure for the detection of an obstacle free trajectory

3.2. Obstacle free trajectories

Given a set of obstacle points $O = \{(a_i, b_i)\} : 1 \leq i \leq 2^m$, a starting point (x_0, y_0) and an ending point (x_1, y_1), determine if the trajectory following a straight line from (x_0, y_0) to (x_1, y_1) is obstacle free. A trajectory is obstacle free if the distance from the nearest obstacle to the trajectory is greater or equal than a given parameter ξ, see Figure 5.

With reference to Figure 5:

- $pDist(c_x, c_y, a_x, a_y, b_x, b_y)$ returns the squared Euclidean distance from the point (c_x, c_y) to the segment $[(a_x, a_y), (b_x, b_y)]$.
- After $m+1$ steps, the variable *collision* contains a value equal or less than zero if the trajectory is obstacle free.

Algorithm 1 shows the pseudocode of the *pDist* function.

Algorithm 1 $pDist(c_x, c_y, a_x, a_y, b_x, b_y)$

$u \leftarrow (c_x - a_x) \cdot (b_x - a_x) + (c_y - a_y) \cdot (b_y - a_y)$;
$u \leftarrow u/[(b_x - a_x)^2 + (b_y - a_y)^2]$;
if $u < 0$ **then**
  return $(a_x - c_x)^2 + (a_y - c_y)^2$
end if
if $u > 1$ **then**
  return $(b_x - c_x)^2 + (b_y - c_y)^2$
end if
$p_x \leftarrow a_x + u \cdot (b_x - a_x)$;
$p_y \leftarrow a_y + u \cdot (b_y - a_y)$;
return $(p_x - c_x)^2 + (p_y - c_y)^2$

The P system computes in one step of computation the squared Euclidean distance for all the obstacles to the segment given by $[(x_0, y_0), (x_1, y_1)]$. After that, a reduction operation is conducted in m steps of computation, obtaining the minimum distance. In the last step, variables *collision* and *halt* are set.

3.3. The RRT algorithm

For the following set of parameters

- An initial robot position (x_1, y_1).
- A set of obstacle points $\{(a_i, b_i)\} : 1 \leq i \leq 2^m$.
- The size (p, q) of the scenario.
- Parameter n, where the number of points in the RRT will be 2^n.
- Parameter ξ as explained in subsection 3.2.
- Parameter δ giving the length of the edges in the RRT.

the RRT algorithm is illustrated in Figure 6.

The inner modules are defined as shown in Figures 7, 8, and 9 where

- *random*() returns a random number independent identically distributed (i.i.d.) in $[0,1] \cap \mathbb{R}$

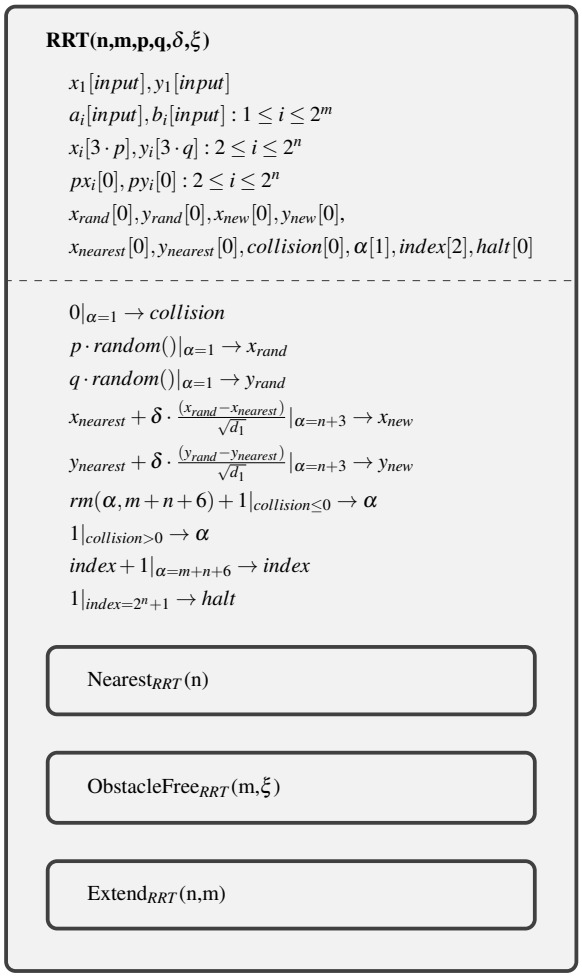


Figure 6. Rapidly-exploring Random Tree

- $rm(x, y)$ returns the remainder of the integer division between x and y.
- The coordinates of the RRT nodes will be $\{(x_i, y_i)\} : 1 \leq i \leq 2^n$
- The coordinates of the RRT parent nodes will be $\{(px_i, py_i)\} : 2 \leq i \leq 2^n$
- The RRT is completely generated and the computation stops when $halt = 1$.

The P system generates the point (x_{rand}, y_{rand}) in one step of computation, after that, the module $Nearest_{RRT}(n)$ computes the point $(x_{nearest}, y_{nearest})$ in $n+1$ steps of computation as explained in subsection 3.1. Then, the (x_{new}, y_{new}) point is computed ac-

Nearest$_{RRT}$(n)

$d_i[0], x'_i[0], y'_i[0] : 1 \leq i \leq 2^n$

$(x_i - x_{rand})^2 + (y_i - y_{rand})^2|_{\alpha=2} \rightarrow d_i : 1 \leq i \leq 2^n$

$x_i|_{\alpha=2} \rightarrow x'_i : 1 \leq i \leq 2^n$

$y_i|_{\alpha=2} \rightarrow y'_i : 1 \leq i \leq 2^n$

$min(d_i, d_{i+2^{n-j}})|_{\alpha=j+2} \rightarrow d_i : 1 \leq i \leq 2^{n-j}, 1 \leq j \leq n$

$min^*(x'_i, x'_{2^{n-j}}, d_i, d_{2^{n-j}})|_{\alpha=j+2} \rightarrow x'_i : 1 \leq i \leq 2^{n-j}, 1 \leq j < n$

$min^*(y'_i, y'_{2^{n-j}}, d_i, d_{2^{n-j}})|_{\alpha=j+2} \rightarrow y'_i : 1 \leq i \leq 2^{n-j}, 1 \leq j < n$

$min^*(x'_1, x'_2, d_1, d_2)|_{\alpha=n+2} \rightarrow x_{nearest}$

$min^*(y'_1, y'_2, d_1, d_2)|_{\alpha=n+2} \rightarrow y_{nearest}$

Figure 7. Nearest$_{RRT}$ Procedure

ObstacleFree$_{RRT}$(m,ξ)

$d'_i[0] : 1 \leq i \leq 2^m$

$pDist(a_i, b_i, x_{nearest}, y_{nearest}, x_{new}, y_{new})|_{\alpha=n+4} \rightarrow d'_i : 1 \leq i \leq 2^m$

$min(d'_i, d'_{i+2^{m-j}})|_{\alpha=j+n+4} \rightarrow d'_i : 1 \leq i \leq 2^{m-j}, 1 \leq j < m$

$min(\xi - d'_1, \xi - d'_2)|_{\alpha=m+n+4} \rightarrow collision$

Figure 8. ObstacleFree$_{RRT}$ Procedure

Extend$_{RRT}$(n,m)

$x_{new}|_{\alpha=m+n+6} \rightarrow x_{[index]}$

$y_{new}|_{\alpha=m+n+6} \rightarrow y_{[index]}$

$x_{nearest}|_{\alpha=m+n+6} \rightarrow px_{[index]}$

$y_{nearest}|_{\alpha=m+n+6} \rightarrow py_{[index]}$

Figure 9. Extend$_{RRT}$ Procedure

cording to the δ parameter. The next module to be executed is the $ObstacleFree_{RRT}(n, \xi)$ module, see subsection 3.2, computing the squared Euclidean distance of the nearest obstacle point to the segment $[(x_{nearest}, y_{nearest}), (x_{new}, y_{new})]$, if this value is less than ξ parameter, then the variable *collision* will contain

RRT*(n,m,p,q,δ,ξ)

$x_1[input], y_1[input], c_1[0]$

$a_i[input], b_i[input] : 1 \leq i \leq 2^m$

$x_i[3 \cdot p], y_i[3 \cdot q], c_i[9 \cdot (p^2 + q^2)] : 2 \leq i \leq 2^n$

$px_i[0], py_i[0] : 2 \leq i \leq 2^n$

$x_{rand}[0]$, $y_{rand}[0]$, $x_{new}[0]$, $y_{new}[0]$,

$x_{nearest}[0], y_{nearest}[0], collision[0], index[2], halt[0]$

$0|_{\alpha=1} \rightarrow collision$

$p \cdot random()|_{\alpha=1} \rightarrow x_{rand}$

$q \cdot random()|_{\alpha=1} \rightarrow y_{rand}$

$x_{nearest} + \delta \cdot \frac{(x_{rand} - x_{nearest})}{\sqrt{d_1}}|_{\alpha=n+3} \rightarrow x_{new}$

$y_{nearest} + \delta \cdot \frac{(y_{rand} - y_{nearest})}{\sqrt{d_1}}|_{\alpha=n+3} \rightarrow y_{new}$

$rm(\alpha, 2 \cdot m + 2 \cdot n + 10) + 1|_{collision \leq 0} \rightarrow \alpha$

$1|_{collision > 0} \rightarrow \alpha$

$index + 1|_{\alpha=2 \cdot m+2 \cdot n+10} \rightarrow index$

$1|_{index=2^n+1} \rightarrow halt$

Nearest$_{RRT}$(n)

ObstacleFree$_{RRT}$(m,ξ)

Extend$_{RRT*}$(n,m,p,q,ξ)

Figure 10. Asymptotically optimal Rapidly-exploring Random Tree (RRT*)

a value greater than 0 after $m+1$ steps of computation. Finally, if the segment is obstacle free, the module $Extend(n,m)$ updates the RRT. In this way, each node is added to the RRT in $m+n+6$ computation steps if the corresponding edge is obstacle free.

3.4. The RRT algorithm*

The asymptotically-optimal version of the RRT algorithm, i.e. the RRT* algorithm is outlined in Figure 10.

The first $m+n+5$ steps of computation for the ENPS RRT* algorithm are the same to those given in the ENPS RRT algorithm, see subsection 3.3. In the

$m+n+6$ step, the module $Extend_{RRT*}(n,m,p,q,\xi)$ generates in one step of computation the squared Euclidean distances from all the obstacles to all the possible segments $[(x_j,y_j),(x_{new},y_{new})]$ being (x_j,y_j) all the nodes in the RRT, i.e, parent candidates for the new edge. After that, a reduction process computes the minimum distance for each parent candidate in m steps of computation. Then, the costs for all the possible trajectories to (x_{new},y_{new}) are computed where trajectories passing through obstacles will have a larger cost value than obstacle-free trajectories. After that, the minimum cost as well as the best parent candidate are computed in n steps of computation by applying a reduction process. The RRT will be updated with the new node as well as a new edge from the best parent candidate to the new node. The cost of the new node will be also stored. Finally, each node (x_i,y_i) in the RRT computes in one step of computation an alternative cost considering (x_{new},y_{new}) as its parent node. If the alternative cost is less than the current cost and the segment $[(x_i,y_i),(x_{new},y_{new})]$ is obstacle-free, then the parent of (x_i,y_i) is updated to (x_{new},y_{new}). In this way, each node is added to the RRT in $2\cdot m+2\cdot n+10$ steps of computation if the corresponding edge is obstacle free. The Extend$_{RRT*}$ is shown ins Figure 11.
where

- $max(a,b)=\begin{cases}a & a>b\\ b & a\leq b\end{cases}$
- c_i is the cost associated to node (x_i,y_i).

Extend$_{RRT^*}$(n,m,p,q,ξ)

$d''_{i,j}[0]:1\leq i\leq 2^m,1\leq j\leq 2^n$
$x''_i[0],y''_i[0],c'_i[0]:1\leq i\leq 2^n$

$x_i|_{\alpha=m+n+6}\rightarrow x''_i:1\leq i\leq 2^n$
$y_i|_{\alpha=m+n+6}\rightarrow y''_i:1\leq i\leq 2^n$
$pDist(a_i,b_i,x_j,y_j,x_{new},y_{new})|_{\alpha=m+n+6}\rightarrow d''_{i,j}:1\leq i\leq 2^m,1\leq j\leq 2^n$
$min(d''_{i,j},d''_{i+2^{m-k},j})|_{\alpha=k+m+n+6}\rightarrow d''_{i,j}:1\leq i\leq 2^{m-k},1\leq j\leq 2^n,1\leq k<m$
$max(0,min(\xi-d''_{1,j},\xi-d''_{2,j}))|_{\alpha=2\cdot m+n+6}\rightarrow d''_{1,j}:1\leq j\leq 2^n$
$c_i+(x_{new}-x_i)^2+(y_{new}-y_i)^2+9\cdot d''_{1,i}\cdot(p^2+q^2)|_{\alpha=2\cdot m+n+7}\rightarrow c'_i:1\leq i\leq 2^n$
$min(c'_i,c'_{i+2^{n-j}})|_{\alpha=j+2\cdot m+n+7}\rightarrow c'_i:1\leq i\leq 2^{n-j},1\leq j\leq n$
$min^*(x''_i,x''_{2^{n-j}},c'_i,c'_{2^{n-j}})|_{\alpha=j+2\cdot m+n+7}\rightarrow x''_i:1\leq i\leq 2^{n-j},1\leq j\leq n$
$min^*(y''_i,y''_{2^{n-j}},c'_i,c'_{2^{n-j}})|_{\alpha=j+2\cdot m+n+7}\rightarrow y''_i:1\leq i\leq 2^{n-j},1\leq j\leq n$
$x_{new}|_{\alpha=2\cdot m+2\cdot n+8}\rightarrow x_{[index]}$
$y_{new}|_{\alpha=2\cdot m+2\cdot n+8}\rightarrow y_{[index]}$
$x''_1|_{\alpha=2\cdot m+2\cdot n+8}\rightarrow px_{[index]}$
$y''_1|_{\alpha=2\cdot m+2\cdot n+8}\rightarrow py_{[index]}$
$c'_1|_{\alpha=2\cdot m+2\cdot n+8}\rightarrow c_{[index]}$
$c_i-c_{[index]}-(x_{new}-x_i)^2-(y_{new}-y_i)^2|_{\alpha=2\cdot m+2\cdot n+9}\rightarrow c'_i:2\leq i\leq 2^n$
$x_{new}|_{(\alpha=2\cdot m+2\cdot n+10)and(c'_i>0)and(d''_{1,i}=0)}\rightarrow px_i:2\leq i\leq 2^n$
$y_{new}|_{(\alpha=2\cdot m+2\cdot n+10)and(c'_i>0)and(d''_{1,i}=0)}\rightarrow py_i:2\leq i\leq 2^n$

Figure 11. Extend$_{RRT^*}$ procedure

4. Developed software

We have implemented two specific (ad-hoc) simulators using OpenMP and CUDA, each one is able to simulate the ENPS-RRT and ENPS-RRT* models described in 3. Let us recall that a specific simulator aims at simulating a certain P system model (as in this case), instead of simulating a whole P system variant (i.e. a generic simulator) [42]. In a specific simulator, the developer encode the P system directly in the source code, and can implement assumptions for simplicity and efficiency. In any case, the simulator must simulate the model; that is, there should be a way to know the state of the P system at any time. The software can be downloaded at `https://github.com/Ignacio-Perez/enps_rrt`, where the master is the OpenMP simulator and the CUDA branch is the GPU version.

The conventional RRT and RRT* algorithms have also been implemented as baseline software for testing and validation using C++ and OpenMP. It is available at `https://github.com/Ignacio-Perez/openmprrt`.

4.1. An OpenMP simulator

The OpenMP simulator is able to manage image files in PGM format defining the obstacle maps, four pre-

defined scenarios have been included with the software as it will be explained in Section 5. For each one, a PGM file is included along with a fixed initial position for the robot. The resolution for all maps is $5cm^2/pixel$, which is the resolution of the LIDAR sensor used to generate the *map1* and *ccia_h* maps.

The software provides a command-line interface in order to specify the scenario to be used, as well as the model (ENPS-RRT or ENPS-RRT*) and the random seed. The output is a new PGM file where an RRT has been drawn over the original map. The number of CPU threads to be used by the OpenMP simulator is configured by the environment variable OMP_NUM_THREADS.

The software allows to set all the parameter values. For the experiments in this paper, the next set of values has been used:

- $n = 12$, i.e, 2^{12} RRT nodes will be generated. It produces a large enough tree to stress the parallel simulators.
- $\delta = 15cm$, i.e, the RRT edges will have $15cm$ length.
- $\varepsilon = 20cm$, the robot radius.
- m, p, q depend on the specific PGM file.

The selected values for δ and ε are common values for indoor platforms such as the Pioneer DX-2 robot.

The simulator uses data structures to store the value of the ENPS variables in run-time and programming sentences in C with OpenMP in order to simulate the behaviour of the ENPS programs. That is, the simulator allocates arrays storing the objects for:

- obstacle positions (a and b).
- RRT points and their parents, (x, y, px and py).
- auxiliary objects for reduction at nearest (xp, yp and d) and at obstacleFree modules (d').

Moreover, the simulator is written in a modular way, where it is easy to identify the different modules of the model:

- Computing (x_{rand}, y_{rand}) on the CPU.
- *nearest*, in order to compute ($x_{nearest}$, $y_{nearest}$). This is done by:
  1. computing squared distances from all points in RRT to (x_{rand}, y_{rand})
  2. computing minimum distance and nearest point in RRT
- *obstacleFree*, in order to compute obstacle collision. This is done by:
  1. compute distances from all obstacles to segment [($x_{nearest}$, $y_{nearest}$),(x_{new}, y_{new})]
  2. Compute minimun distance to check if there is collision, by comparing with the epsilon variable
- *Extend RRT*, if using RRT algorithm.
- *Extend RRT**, if using RRT* algorithm. This is implemented by:
  1. computing squared distances from all obstacles to all segments [(x,y),(x_{new}, y_{new})] where (x,y) are points in RRT
  2. for each point (x,y) in RRT, compute the minimum distance to all obstacles
  3. compute new cost for all points in RRT
  4. compute the minimum cost and the variable with that value (that is, argmin operation)
  5. fix the edges in RRT

As an example, the next C/OpenMP code computes the squared distances from all the points in the RRT to the $(x_{nearest}, y_{nearest})$ point.

```
#pragma omp parallel for
for (int i=0;i<vars->index;i++) {
  vars->d[i] = (vars->x[i] - vars->x_rand) *
               (vars->x[i] - vars->x_rand) +
               (vars->y[i] - vars->y_rand) *
               (vars->y[i] - vars->y_rand);
}
```

In order to compute minimums in a whole array, we can use reduction pattern. That is, calculate a value from a whole collection, in this case, a minimum. Reduction is implemented in OpenMP by a special directive extension [17]. For instance, the next code shows the computation of the minimum distance and nearest point:

```
XYD value = {0,0,INF};
#pragma omp parallel for reduction(xyd_min:value)
for (int i=0;i<vars->index;i++) {
  XYD new_value = {vars->x[i],vars->y[i],vars->d[i]};
  value = xyd_min2(value,new_value);
}
```

4.2. A CUDA simulator

Additionally, we have outsourced the ad-hoc simulator to the GPU. This enables its parallelization for both high-end and low-end GPUs by using CUDA. Let us recall that this technology provides automatic scalability and portability of the code. Thus, the most computing-intensive part of the simulation is executed on device, what will allow to accelerate its execution significantly, as we will discuss in the results. First of all, we have replicated the data structures employed on the OpenMP simulator on the GPU.

Simple map functions applied to objects in parallel are implemented by CUDA kernels, optimized by launching 256 threads per block (we have tuned this number experimentally), a number of thread blocks multiple of the amount of streaming multiprocessors on the specific device, and looping by tiles over the objects (contiguous threads work with contiguous positions of arrays, and iterate in this manner until covering all positions). The following parts are implemented in this way:

- At nearest module, computing squared distances from all points in RRT to (x_{rand}, y_{rand}).
- At obstacleFree module, computing the distances from all obstacles to segment [($x_{nearest}$, $y_{nearest}$), (x_{new}, y_{new})], by using pDist.
- At RRT* extend module:
  - computing squared distances, using pDist, from all obstacles to all segments [(x,y), (x_{new},y_{new})], where (x,y) are points in RRT. This kernel specifically employs a 2-dimensional grid of thread blocks to cover the two nested loops in parallel.
  - computing new cost for all points in RRT, while checking the minimun distance is greater than epsilon.
  - Fixing edges.

As made for the OpenMP implementations, global minimum calculations can be efficiently implemented by the reduction operation. In CUDA, reduction is a parallel primitive, and there is a plethora of optimized implementations. We have focused only on reduction with minimum operation for floating numbers. We have tested three different libraries in order to select the fastest one: (1) reduction example from CUDA Toolkit SDK 10.1 [47], (2) Thrust (as redistributed in CUDA 10.1) [26], and (3) CUB 1.8 [43]. Implementation (1) is based on the optimization ideas using per-warp shuffle operations [38], available first in Kepler architecture. We have modified the implementation in the SDK by replacing the sum (+) operation by the built-in fminf function in CUDA. Concerning implementation (2), we have used the STL-like library Thrust, which offers an optimized minimum reduction function. We have ensured that the function is launched on the device. Finally, we have also tested CUB primitives library by NVIDIA labs in implementation (3). This is a set of header files that do not require to compile, but just including them in our code. Both Thrust and CUB also have a variant for argmin operation, required in order to calculate in which position of the array the minimum takes place. Finally, as for reduction with minimum operation, CUB outperformed the other two, (3) against (1) by three times, and (3) against (2) slightly better.

We have therefore used CUB to compute the following reduction operations:

- At nearest module: compute minimum distance and nearest point by using argmin reduction.
- At obstacleFree module: compute minimum distance by using min reduction.
- At RRT* extend module:
  - Compute the minimum distance for each point (x,y) in RRT to all obstacles. This requires to construct a matrix with a row per RRT point and columns the number of obstacles. We have used the segmented reduction minimum variant to calculate all the minimums per row in parallel. This variant performs reduction globally but it only takes effect inside local segments that are previously defined.
  - Compute minimum cost by argmin.

Finally, in order to keep all the computation on the GPU and hence, avoiding transferring data from the GPU to the CPU at every iteration, store also the scalar values of $x_{nearest}$, $y_{nearest}$, x_{new}, y_{new} on device. These are updated by kernels with single thread. The objective is the reduce the amount of memory transfer as much as possible, even if just one thread is in charge of updating single objects.

5. Experimental results

In this section, we will show the experiments conducted to test the performance of the ad-hoc simulators for both OpenMP and CUDA. We first start analyzing the results for the multicore version, selecting the best configuration. After that, we show the runtimes obtained with GPU version, and conclude with a comparison of configurations.

The source code of the simulators is written in C/C++ programming language, and binaries are compiled with GCC 7.4 and NVCC 10.1 using the flag -O3. The random numbers are always generated on the CPU by using the seed 1. Execution times are measures with time bash instruction, so that we consider the whole time to read and write the maps and the simulation itself with all the overheads. We have used the hardware setup shown below, where we looked for a low-end and a high-end GPU, and employed two versions of Intel CPUs corresponding to the processors where the GPUs are plugged. In this way, we validate our simulators and its acceleration by emulating two different situations: (a) when the RRT is pre-computed on a server with a high-end GPU and transmitted to the robot before starting to move, and (b) when the RRT is pre-computed using the low-end GPU on-board the robot like a NVIDIA Jetson.

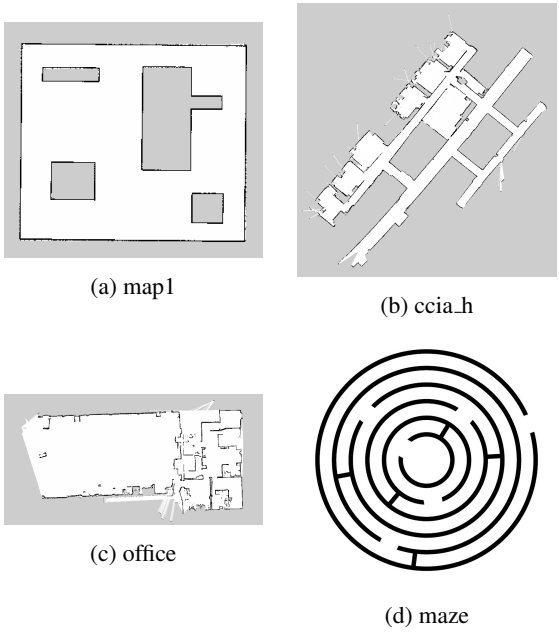
- **i7-8700**: Intel Core i7-8700 CPU with 12 cores (6x2 hyperthreading) at 3.2Ghz 12MB of cache
- **i7-9700**: Intel Core i7-9700K CPU with 8 cores (4x2 hyperthreading) at 3.6Ghz and 12MB of cache
- **GTX1050**: NVIDIA GeForce GTX 1050Ti GPU with 768 cores at 1.42Ghz and 1MB of cache, plugged to i7-8700.
- **RTX2080**: NVIDIA GeForce RTX2080 GPU with 2944 cores at 1.85Ghz and 4MB of cache, plugged to i7-9700.

We have used four different maps for the experiments. These are shown in Figure 12. We have sorted them by increasing complexity, so that we expect that RRT and RRT* will require more iterations for maze than for map1 for example. For all experiments, we provide two fixed distant points in the map. The baseline software has been executed using the i7-9700 hardware with configurations of 4 and 8 cores. The speedup for each map, each algorithm and each CPU configuration is shown in Table 1. An average speedup of 1.39x was obtained using the baseline RRT algorithm with 8 cores with respect to the sequential implementation. The speedup was 2.57x in the case of the RRT* algorithm.

In Figure 13, the runtimes of the multicore simu-

Table 1. Speedup using OpenMP for different multithreading configurations with respect to the sequential implementation for each baseline algorithm and for each map.

Map	CPUs	Speedup RRT	Speedup RRT*
map1	4	1.2678	1.8715
map1	8	1.4634	3.1708
ccia_h	4	1.1585	1.6139
ccia_h	8	1.2664	2.2785
office	4	1.2977	1.6894
office	8	1.4159	2.4494
maze	4	1.2757	1.6696
maze	8	1.4338	2.4055



(a) map1 (b) ccia_h (c) office (d) maze

Figure 12. Maps employed in the experiments. Maps are sorted by complexity: (a) is **map1** (a simple room), (b) is **ccia_h** (scanned map of the computer science department's wing H at University of Seville), (c) is **office** (a set of rooms in an office), and (d) is **maze** (a labyrinth composed of concentric circles).

lator for RRT are displayed. We increase the number of threads by 2, starting in 1 and until the total amount of cores in the corresponding CPU. We can see how it scales well and the runtime drops proportionally to the number of threads executed, specially when going from just 1 thread to the half amount of cores in each CPU. Runtimes from half to all cores remain almost constant. Only in maze we can experiment some improvement. Low-complexity map1 is handled fast in all CPUs, so running several threads is not worthy. Finally, runtimes are similar in both CPUs, and is always below 2 seconds when using all cores. For i7-8700, a speedup from 1.8x (map1) to 4.5x (maze) is obtained when using all cores against just one core (sequential version). For i7-9700, a speedup from 2.2x (map1) to 4.9x (maze) is achieved.

We can see some differences when running the multicore simulator with RRT*, given its higher complexity. The runtimes are plotted in Figure 14. For i7-8700, half of the cores (6) are enough to saturate the processor. For i7-9700, the best runtime is achieved with full occupancy of the cores (8). We can see how the runtime decrease is proportional to the amount of cores again. Runtime for i7-9700 is slightly lower than for i7-8700, being below 200 seconds against 300 seconds in the latter. The map maze requires the higher computation time. For i7-8700, a speedup from 4x (maze) to 5.2x (map1) can be obtained when using all cores against the sequential version. For i7-9700, a speedup of around 6.7x for all maps is reported.

In Figure 15, we compare the runtimes obtained with the GPU against the CPU. We consider for this experiment the best configuration for both CPUs, that is using all the cores in each processor. For RRT*, the GPU outperforms the CPU in all maps, with alwarys very low runtimes. The example of maze stands out, because the RTX2080 scales very well and requires just 13.21 seconds to construct the RRT* tree, whereas the CPUs requires 2 to 4 minutes to complete. In this specific map, the GTX1050 still behaves better than the CPU. Note that we do not report the results for RRT, for which the GPU is always slower than the CPU mainly because the overhead to perform reduction on device is not worthy for such small amount of item. For RRT*, we need to execute a reduction much larger, so we can see the good effect of using the GPU.

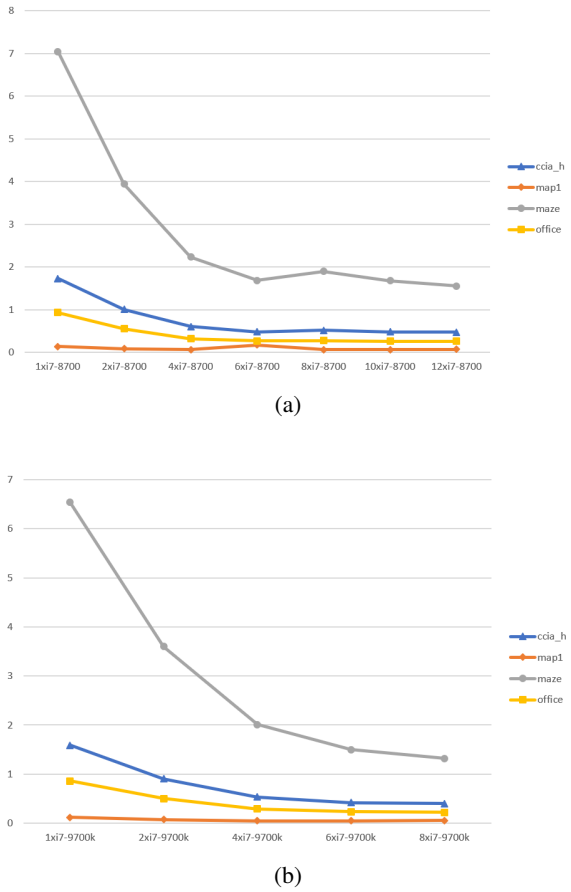
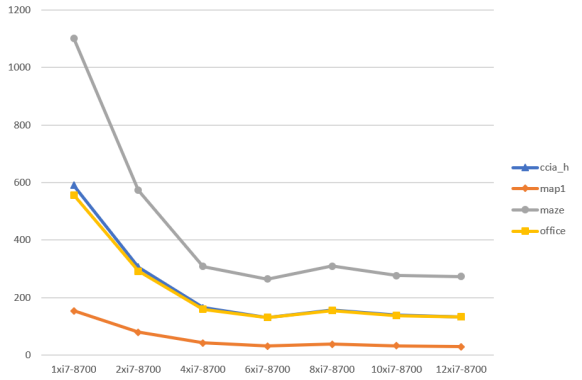
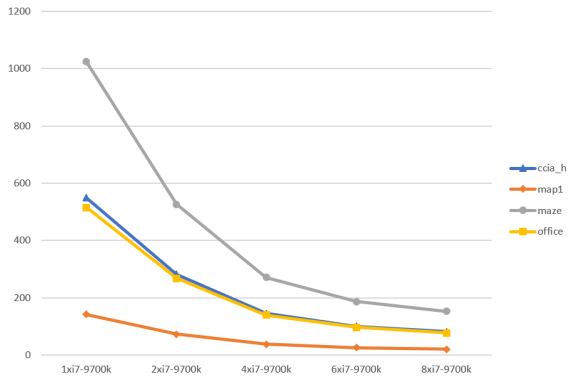


Figure 13. Runtimes of the OpenMP simulator for RRT on the i7-8700 (a) and i7-9700 (b) CPU. X-axis shows the number of cores used during execution (1 to 12 for i7-8700, and 1 to 8 for i7-9700, both by step of 2). Y-axis shows the time in seconds.

Finally, let us assume the best and worst device: **fastest CPU** is i7-9700 with 8 cores, **slowest CPU** is i7-8700 with 12 cores, **fastest GPU** is RTX2080 and **slowest GPU** is GTX1050. In Figure 16, we show the speedups obtained by comparing one by one the devices tested for RRT*. We can see that the slowest GPU still outperforms the fastest GPU by 1.5x to 2.3x. On the contrary, the fastest GPU against the slowest CPUs leads to very high speedups, of around 20x. Finally, the fastest CPU and GPU comparison gives speedups of around 14x for the GPU. It is interesting to see that the GPU always get higher speedups for the office-type maps (ccia_h and office).



(a)



(b)

Figure 14. Runtimes of the OpenMP simulator for RRT* on the i7-8700 (a) and i7-9700 (b) CPU. X-axis shows the number of cores used during execution (1 to 12 for i7-8700, and 1 to 8 for i7-9700, both by step of 2). Y-axis shows the time in seconds.

In light of the results, we can draw the following conclusions: CPU is the best candidate if we want to run the adhoc simulator for RRT, and the GPU is always the best candidate to run the ad-hoc simulator for RRT*; and for the CPU, it is always worthy using at least half of the cores in the processor.

6. Conclusions

In this work we present two ENPS models emulating the computation of the RRT and RRT* algorithms in order to solve the motion planning problem in robotics. The ENPS framework was successfully used

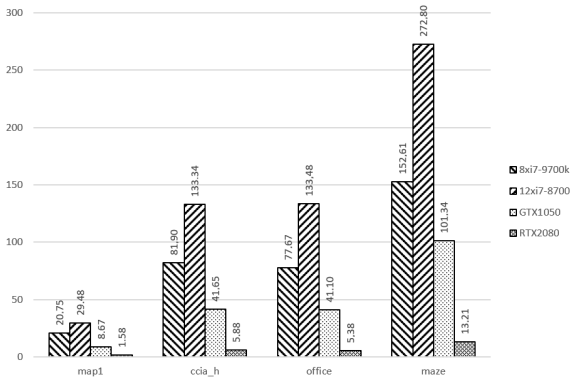


Figure 15. Runtime of the CUDA simulator for RRT* on GTX1050 and RTX2080 compared to the best multi-threading configuration for the OpenMP simulator with the two CPUs.

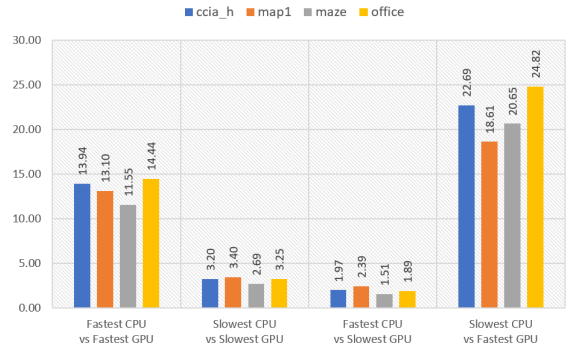


Figure 16. Seedups obtained comparing the best and worst configurations for both CPU and GPU.

in previous works to design and simulate robot controllers, but there is a lack of solutions for global path planning using the framework. The RRT and RRT* algorithms are inherently iterative, but they have modules that can run in parallel, such as the obstacle collision detection and others. By using a massively parallel model of computation, we provide a solution able to add nodes to an RRT in constant time. Two simulators have been implemented using OpenMP and CUDA, i.e., two parallel computing frameworks *in silico*. Several processes in the models, such as the computation of the nearest point to a given point by using reduction, have been translated to the simulators in a natural way. Finally, we have validated and tested the models and simulators by using four scenarios. The experimental results show a speedup up to 6x us-

ing OpenMP with 8 cores against the sequential implementation and up to 24x using CUDA against the best multi-threading configuration. The best speedup using a baseline OpenMP implementation with 8 cores against the corresponding baseline sequential implementation was 2.57x. Such results show a better parallel scalability when using the membrane computing paradigm which is a maximal parallel model of computation.

Acknowledgements

This work is supported by the research project TIN2017-89842-P (MABICAP), co-financed by *Ministerio de Economía, Industria y Competitividad (MINECO)* of Spain, through the *Agencia Estatal de Investigación (AEI)*, and by *Fondo Europeo de Desarrollo Regional (FEDER)* of the European Union.

This work is supported by the *National Natural Science Foundation of China* (61972324, 61672437, 61702428), and also funded by Beijing Advanced Innovation Center for Intelligent Robots and Systems (2019IRS14), Artificial Intelligence Key Laboratory of Sichuan Province (2019RYJ06), the New Generation Artificial Intelligence Science and Technology Major Project of Sichuan Province (2018GZDZX0043) and the Sichuan Science and Technology Program (2018GZ0185, 2018GZ0086).

References

[1] H. Adeli and S. L. Hung, "A concurrent adaptive conjugate gradient learning algorithm on MIMD machines," *Journal of Supercomputer*, vol. 7, no. 2, pp. 155–166, 1993.

[2] H. Adeli and S. Kumar, "Distributed finite element analysis on a network of workstations - algorithms," *Journal of Structural Engineering*, vol. 121, no. 10, pp. 1448–1455, 1995.

[3] H. Adeli and S. Kumar, *Distributed Computer-Aided Engineering: For Analysis, Design, and Visualization*, 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 1998.

[4] O. Adiyatov and H. A. Varol, "Rapidly-exploring random tree based memory efficient motion planning," in *2013 IEEE International Conference on Mechatronics and Automation*, Aug 2013, pp. 354–359.

[5] A. Al-Kaff, J. M. Armingol, and A. de la Escalera, "A vision-based navigation system for unmanned aerial vehicles (uavs)," *Integrated Computer-Aided Engineering*, vol. 26, no. 3, pp. 297–310, 2019.

[6] B. Aman and G. Ciobanu, "Synchronization of rules in membrane computing," *Journal of Membrane Computing*, vol. 1, no. 4, pp. 233–240, 2019.

[7] R. Barbuti, P. Bove, P. Milazzo, and G. Pardini, "Minimal probabilistic P systems for modelling ecological systems," *Theoretical Computer Science*, vol. 608, pp. 36 – 56, 2015.

[8] J. Benito-Picazo, E. Domínguez, E. J. Palomo, E. López-Rubio, and J. M. Ortiz-de-Lazcano-Lobato, "Motion detection with low cost hardware for PTZ cameras," *Integrated Computer-Aided Engineering*, vol. 26, no. 1, pp. 21–36, 2019.

[9] J. Bialkowski, S. Karaman, and E. Frazzoli, "Massively parallelizing the RRT and the RRT*," in *Proceedings of the 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sep. 2011, pp. 3513–3518.

[10] M. Brunner, B. Brüggemann, and D. Schulz, "Hierarchical rough terrain motion planning using an optimal sampling-based method," *2013 IEEE International Conference on Robotics and Automation*, pp. 5539–5544, 2013.

[11] C. Buiu and A. George, "Membrane computing models and robot controller design , current results and challenges," *Journal of Membrane Computing*, vol. 1, no. 4, pp. 262–269, 2019.

[12] J. F. Canny, *The Complexity of Robot Motion Planning*. MIT Press, 1988.

[13] J. M. Cecilia, J. M. García, G. D. Guerrero, M. A. Martínez-del-Amor, I. Pérez-Hurtado, and M. J. Pérez-Jiménez, "Simulation of P systems with active membranes on CUDA," *Briefings in Bioinformatics*, vol. 11, no. 3, pp. 313–322, 2010.

[14] M. A. Colomer, A. Margalida, and M. J. Pérez-Jiménez, "Population Dynamics P System (PDP) Models: A Standardized Protocol for Describing and Applying Novel Bio-Inspired Computing Tools," *PLOS ONE*, vol. 8, no. 5, p. e60698, 2013.

[15] K. Daniel, A. Nash, S. Koenig, and A. Felner, "Theta*: Any-Angle Path Planning on Grids," *CoRR*, vol. abs/1401.3843, 2014. [Online]. Available: http://arxiv.org/abs/1401.3843

[16] D. Díaz-Pernil, C. Graciani-Díaz, M. A. Gutiérrez-Naranjo, I. Pérez-Hurtado, and M. J. Pérez-Jiménez, *Software for P systems*. Oxford University Press, 2010, ch. 17, pp. 437–454.

[17] A. Duran, R. Ferrer, M. Klemm, B. R. de Supinski, and E. Ayguadé, "A proposal for user-defined reductions in openmp," in *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More*, M. Sato, T. Hanawa, M. S. Müller, B. M. Chapman, and B. R. de Supinski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 43–55.

[18] A. C. Elster and S. Requena, "Parallel computing with gpus," in *Parallel Computing: From Multicores and GPU's to Petascale, Proceedings of the conference ParCo 2009, 1-4 September 2009, Lyon, France*, 2009, pp. 533–535. [Online]. Available: https://doi.org/10.3233/978-1-60750-530-3-533

[19] A. Florea and C. Buiu, *Membrane computing for distributed control of robotic swarms: Emerging research and opportunities*, 03 2017.

[20] C. Ford, "Rrt-gpu and minecraft: Hardware accelerated

rapidly exploring random trees in three dimensions," Ph.D. dissertation, 06 2018.

[21] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot, "Informed rrt*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sep. 2014, pp. 2997–3004.

[22] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[23] M. Gheorghe, N. Krasnogor, and M. Camara, "P systems applications to systems biology," *Biosystems*, vol. 91, no. 3, pp. 435 – 437, 2008.

[24] Gheorghe Păun, *Membrane Computing: An Introduction*. Berlin, Heidelberg: Springer-Verlag, 2002.

[25] M. Gutierrez and H. Adeli, "Recent advances in control algorithms for smart structures and machines," *Expert Systems*, vol. 34, no. 2, 2017.

[26] J. Hoberock and N. Bell, "Thrust Parallel Algorithms Library," https://thrust.github.io, Online (accesed August 2019).

[27] Y. Jiang, Y. Su, and F. Luo, "An improved universal spiking neural P system with generalized use of rules," *Journal of Membrane Computing*, vol. 1, no. 4, pp. 270–278, 2019.

[28] H. Jin, D. Jespersen, P. Mehrotra, R. Biswas, L. Huang, and B. Chapman, "High performance computing using mpi and openmp on multi-core parallel systems," *Parallel Computing*, vol. 37, no. 9, pp. 562 – 575, 2011, emerging Programming Paradigms for Large-Scale Scientific Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819111000159

[29] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011.

[30] M. M. Khan and A. C. Elster, "Characterizing numascale clusters with gpus: Mpi-based and gpu interconnect benchmarks," in *2016 International Conference on High Performance Computing Simulation (HPCS)*, July 2016, pp. 840–847.

[31] D. B. Kirk and W. W. Hwu, *Programming massively parallel processors: a hands-on approach*, 3rd ed. Morgan Kaufmann Publishers Inc., 2016.

[32] S. Kumar and H. Adeli, "Distributed finite element analysis on a network of workstations - implementation and applications," *Journal of Structural Engineering*, vol. 121, no. 10, pp. 1456–1462, 1995.

[33] R. W. Larsen and T. Henriksen, "Strategies for regular segmented reductions on gpu," in *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, ser. FHPC 2017. New York, NY, USA: ACM, 2017, pp. 42–52. [Online]. Available: http://doi.acm.org/10.1145/3122948.3122952

[34] J. Latombe, "Motion Planning: A Journey of Robots, Molecules, Digital Actors, and Other Artifacts," *The International Journal of Robotics Research*, vol. 18, no. 11, pp. 1119–1128, 1999.

[35] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," Tech. Rep., 1998.

[36] S. M. LaValle and J. J. Kuffner, "Randomized Kinodynamic Planning," *The International Journal of Robotics Research*, vol. 20, no. 5, pp. 378–400, 2001.

[37] A. Leporati, L. Manzoni, G. Mauri, A. E. Porreca, and C. Zandron, "A survey on space complexity of P systems with active membranes," *International Journal of Advances in Engineering Sciences and Applied Mathematics*, vol. 10, no. 3, pp. 221–229, 2018.

[38] J. Luitjens, "CUDA blog, faster parallel reductions on Kepler," https://devblogs.nvidia.com/faster-parallel-reductions-kepler, Online (accesed August 2019).

[39] V. Manca, "Metabolic computing," *Journal of Membrane Computing*, vol. 1, no. 3, pp. 223–232, 2019.

[40] P. Martínez-Cañada, C. A. Morillas, and F. J. Pelayo, "A neuronal network model of the primate visual system: Color mechanisms in the retina, LGN and V1," *Int. J. Neural Syst.*, vol. 29, no. 2, p. 1850036, 2019.

[41] M. A. Martínez-del-Amor, M. García-Quismondo, L. F. Macías-Ramos, L. Valencia-Cabrera, A. Riscos-Núñez, and M. J. Pérez-Jiménez, "Simulating P systems on GPU devices: a survey," *Fundamenta Informaticae*, vol. 136, no. 3, pp. 269–284, 2015.

[42] M. Á. Martínez-del-Amor, D. Orellana-Martín, I. Pérez-Hurtado, L. Valencia-Cabrera, A. Riscos-Núñez, and M. J. Pérez-Jiménez, "Design of Specific P Systems Simulators on GPUs," in *Membrane Computing*, ser. Lecture Notes in Computer Science, T. Hinze, G. Rozenberg, A. Salomaa, and C. Zandron, Eds., vol. 11399. Springer International Publishing, 2019, pp. 202–207.

[43] D. Merrill, "CUB Cuda UnBound," https://nvlabs.github.io/cub, NVIDIA Corporation, Online (accesed August 2019).

[44] J. Nasir, F. Islam, U. Malik, Y. Ayaz, O. Hasan, M. Khan, and M. S. Muhammad, "Rrt*-smart: A rapid convergence implementation of rrt*," *International Journal of Advanced Robotic Systems*, vol. 10, no. 7, p. 299, 2013. [Online]. Available: https://doi.org/10.5772/56718

[45] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 40–53, 2008. [Online]. Available: https://doi.org/10.1145/1365490.1365500

[46] I. Noreen, A. Khan, K. Asghar, and Z. Habib, "A path-planning performance comparison of RRT*-AB with MEA* in a 2-dimensional environment," *Symmetry*, vol. 11, no. 945, 2019.

[47] "NVIDIA CUDA Toolkit," https://developer.nvidia.com/cuda-toolkit, NVIDIA Corporation, Online (accesed August 2019).

[48] J. I. Olszewska and J. Toman, "OPEN: New path-planning algorithm for real-world complex environment," in *Research and Development in Intelligent Systems XXXIII*, M. Bramer and M. Petridis, Eds. Cham: Springer International Publishing, 2016, pp. 237–244.

[49] "OpenMP specification, version 4.5," https://www.openmp.

org/specifications, The OpenMP ARB (Architecture Review Boards), Online (accesed August 2019).

[50] D. Orellana-Martín, L. Valencia-Cabrera, A. Riscos-Núñez, and M. J. Pérez-Jiménez, "A path to computational efficiency through membrane computing," *Theoretical Computer Science*, vol. 777, pp. 443 – 453, 2019.

[51] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008. [Online]. Available: https://doi.org/10.1109/JPROC.2008.917757

[52] L. Pan, G. Puaun, G. Zhang, and F. Neri, "Spiking neural *P* systems with communication on request," *Int. J. Neural Syst.*, vol. 27, no. 8, pp. 1–13, 2017.

[53] H. S. Park and H. Adeli, "Distributed neural dynamics algorithms for optimization of large steel structures," *Journal of Structural Engineering*, vol. 123, no. 7, pp. 880–888, 1997.

[54] A. Pavel, O. Arsene, and C. Buiu, "Enzymatic numerical p systems - a new class of membrane computing systems," in *2010 IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA)*, Sep. 2010, pp. 1331–1336.

[55] A. Pavel and C. Buiu, "Using enzymatic numerical p systems for modeling mobile robot controllers," *Natural Computing*, vol. 11, no. 3, pp. 387–393, 2012.

[56] I. Pérez-Hurtado, M. J. Pérez-Jiménez, G. Zhang, and D. Orellana-Martín, "Simulation of Rapidly-Exploring Random Trees in Membrane Computing with P-Lingua and Automatic Programming," *International Journal of Computers, Communications and Control*, vol. 13, no. 6, pp. 1007–1031, 2019.

[57] M. J. Pérez-Jiménez, A. Riscos-Núñez, L. Valencia-Cabrera, and D. Orellana-Martín, *Results on Computational Complexity in Bio-inspired Computing*. World Scientific, 2019, ch. 2, pp. 33–73.

[58] G. Păun, "Computing with Membranes," *Journal of Computer and System Sciences*, vol. 61, no. 1, pp. 108 – 143, 2000.

[59] G. Păun and R. Păun, "Membrane Computing and Economics: Numerical P Systems," *Fundamenta Informaticae*, vol. 73, no. 1,2, pp. 213–227, 2006.

[60] G. Păun and F. J. Romero-Campero, "Membrane Computing as a modeling framework. Cellular systems case studies," in *Formal Methods for Computational Systems Biology*, ser. Lecture Notes in Computer Science, M. Bernardo, P. Degano, and G. Zavattaro, Eds. Springer Berlin Heidelberg, 2008, vol. 5016, pp. 168–214.

[61] G. Păun, G. Rozenberg, and A. Salomaa, *The Oxford Handbook of Membrane Computing*. New York, NY, USA: Oxford University Press, Inc., 2010.

[62] G. Păun, "A quick introduction to membrane computing," *The Journal of Logic and Algebraic Programming*, vol. 79, pp. 291 – 294, 2010.

[63] J. H. Reif, "Complexity of the mover's problem and generalizations," in *20th Annual Symposium on Foundations of Computer Science*, Oct. 1979, pp. 421–427.

[64] D. Rodrigues, J. P. Papa, and H. Adeli, "Meta-heuristic multi- and many-objective optimization techniques for solution of machine learning problems," *Expert Systems*, vol. 34, no. 6, 2017.

[65] A. Saleh and H. Adeli, "Parallel algorithms for integrated structural and control optimization," *Journal of Aerospace Engineering*, vol. 7, no. 3, pp. 297–314, 1994.

[66] A. Saleh and H. Adeli, "Parallel eigenvalue algorithms for large-scale control-optimization problems," *Journal of Aerospace Engineering*, vol. 9, no. 3, pp. 70–79, 1996.

[67] A. Saleh and H. Adeli, "Robust parallel algorithms for solution of the riccati equation," *Journal of Aerospace Engineering*, vol. 10, no. 3, pp. 126–133, 1997.

[68] M. Slembrouck, P. Veelaert, D. V. Cauwelaert, D. V. Hamme, and W. Philips, "Cell-based shape reconstruction from incomplete silhouettes," *Integrated Computer-Aided Engineering*, vol. 26, no. 3, pp. 257–271, 2019.

[69] P. Sosík, "P systems attacking hard problems beyond NP: a survey," *Journal of Membrane Computing*, vol. 1, no. 3, pp. 198–208, 2019.

[70] A. T. Stentz, "The Focussed D* Algorithm for Real-Time Replanning," in *Proceedings of the International Joint Conference on Artificial Intelligence*, Aug. 1995.

[71] L. Valencia-Cabrera, D. Orellana-Martín, M. A. Martínez-del-Amor, and M. J. Pérez-Jiménez, "An interactive timeline of simulators in membrane computing," *Journal of Membrane Computing*, vol. 1, no. 3, pp. 209–222, 2019.

[72] T. Wang, G. Zhang, J. Zhao, Z. He, J. Wang, and M. J. Pérez-Jiménez, "Fault diagnosis of electric power systems based on fuzzy reasoning spiking neural p systems," *IEEE Transactions on Power Systems*, vol. 30, no. 3, pp. 1182–1194, 2015.

[73] S. Wu, G. Zhang, F. Neri, M. Zhu, T. Jiang, and K. Kuhnert, "A multi-aperture optical flow estimation method for an artificial compound eye," *Integrated Computer-Aided Engineering*, vol. 26, no. 2, pp. 139–157, 2019.

[74] S. Wu, G. Zhang, M. Zhu, T. Jiang, and F. Neri, "Geometry based three-dimensional image processing method for electronic cluster eye," *Integrated Computer-Aided Engineering*, vol. 25, no. 3, pp. 213–228, 2018.

[75] T. Wu, F. Bîlbîe, A. Paun, L. Pan, and F. Neri, "Simplified and yet turing universal spiking neural P systems with communication on request," *Int. J. Neural Syst.*, vol. 28, no. 8, p. 1850013, 2018.

[76] T. Yang, C. Cappelle, Y. Ruichek, and M. E. Bagdouri, "Multi-object tracking with discriminant correlation filter based deep learning tracker," *Integrated Computer-Aided Engineering*, vol. 26, no. 3, pp. 273–284, 2019.

[77] G. Zhang, M. Pérez-Jiménez, and M. Gheorghe, *Real-life Applications with Membrane Computing*. Springer, 2017, vol. 25.