Ruch na Rondzie - Projekt I

Systemy Równoległe i Rozproszone

Michał Loska
Jerzy Kłos
28.04.2020

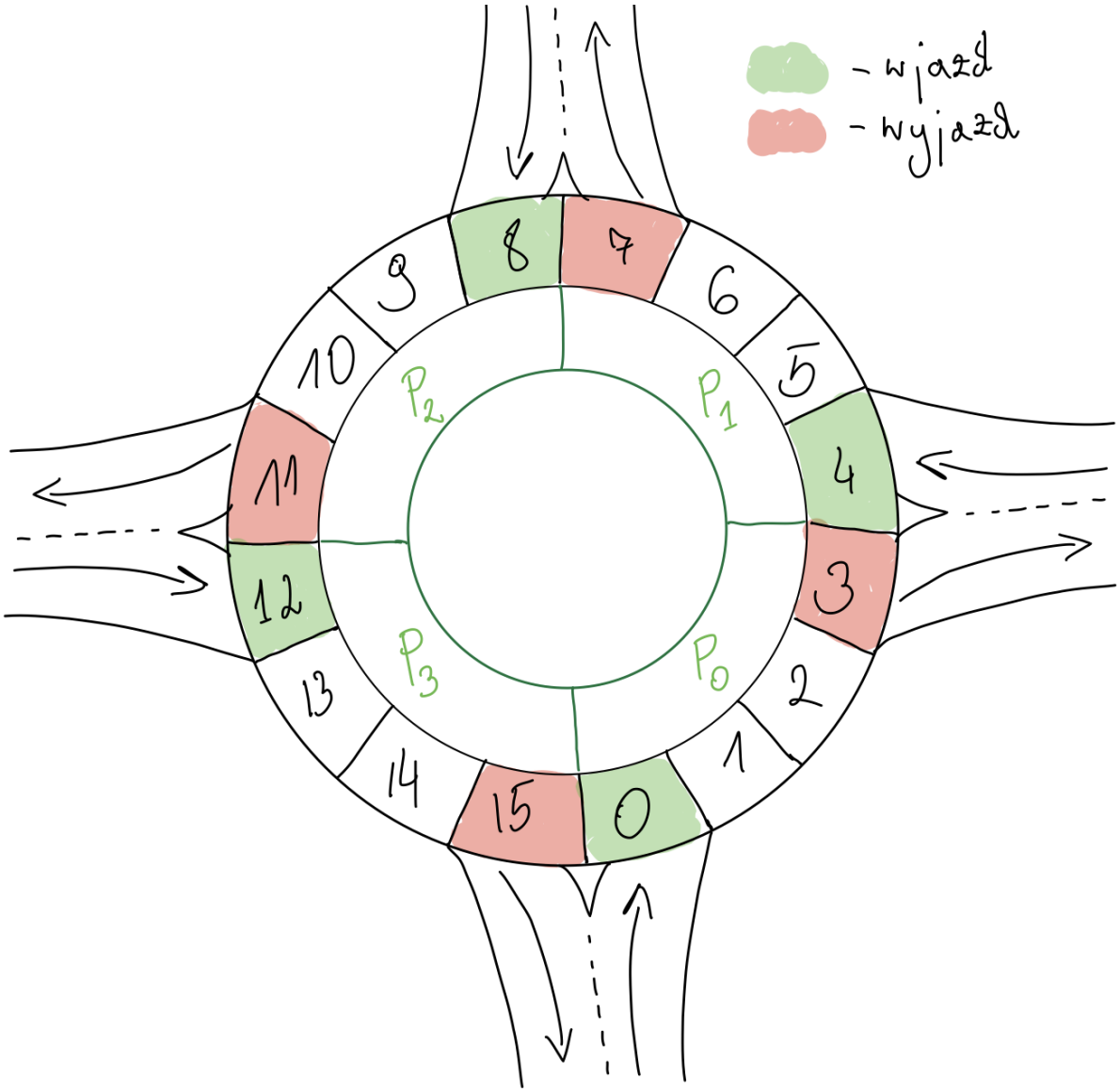
Spis Treści

1. Opis Projektu

Celem projektu było zaimplementowanie programu obsługującego/symulującego ruch na rondzie. Program ten miał zostać stworzony dla systemów rozproszonych w technologii *MPI* oraz języka *C++*.

Program ten opiera się na pseudolosowości, jest to bowiem metoda Monte Carlo. Program pozwala symulować ruch na rondzie dla jego dowolnego rozmiaru. Możemy bowiem manipulować jego parametrami:

- Ilość wjazdów i wyjazdów (ustalana przy uruchomieniu programu jako ilość procesów)
- Odległość między wjazdami i wyjazdami (zawsze stała) (ustalana w kodzie jako *ROUNDABOUT_PART_SIZE*)
- Prawdopodobieństwo pojawienia się samochodu na danym wjeździe (ustalana w kodzie jako *CAR_PROBABILITY*)



Przykład struktury ronda dla 4wjazdów/wyjazdów (4 procesów)

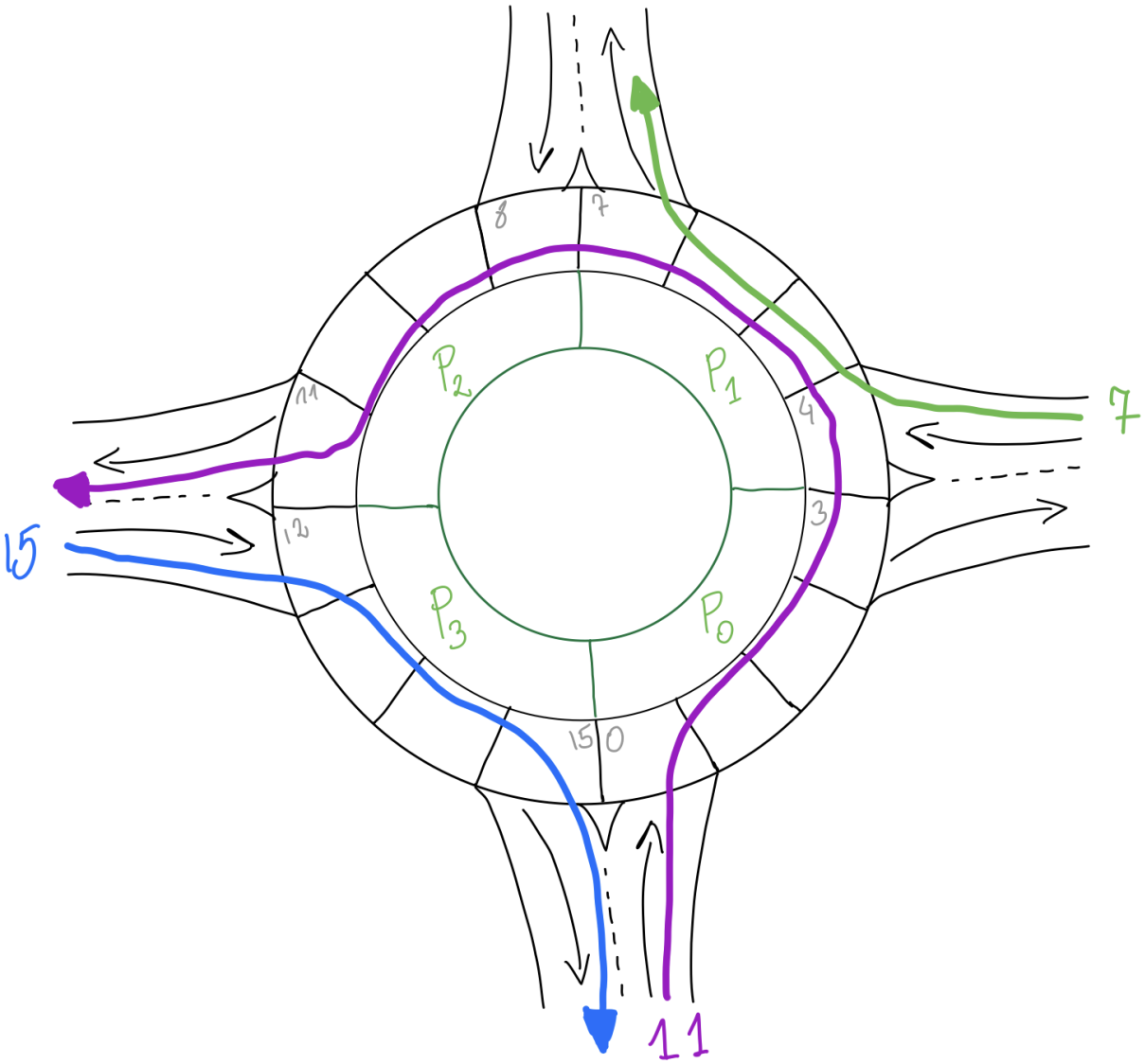
Powyższy rysunek przedstawia strukturę logiczną implementacji ronda dla problemu. Widzimy, że przedstawione rondo to klasyczny przykład dla 4 wjazdów i wyjazdów. Program jest

zaimplementowany w ten sposób, że jeden proces zawsze obsługuje jeden wjazd i jeden (najbliższy mu) wyjazd. Na zamieszczonym rysunku widzimy podział ronda między 4 procesy (P_0, P_1, P_2, P_3).

Ruch na rondzie zaczynamy gdy jest ono puste idąc od wjazdu Południowego zgodnie z ruchem prawostronnym (w przeciwnym kierunku wskazówek zegara).

2. Implementacja Techniczna

W programie odległość między wjazdem a wyjazdem ustalana jest arbitralnie jako 2 pola. Każdy proces ma do dyspozycji kontener o długości 4(std::vector<int>) do przechowywania informacji o samochodach znajdujących się w części ronda, za którą odpowiada dany proces. Początkowo elementy te wypełnione są wartością *NONE* czyli -1. Z biegiem czasu gdy samochody zaczną wjeżdżać na rondo, kontener będzie wypełniany indeksami elementów ronda odpowiadającymi wyjazdom. np: przez wjazd 0 wjedzie samochód, który chce wyjechać wyjazdem 11. Zatem element 0 ronda będzie miał w chwili czasowej t_0 wartość 11. Wraz z biegiem czasu element ten będzie przesuwany zgodnie z kierunkiem ruchu prawostronnego w stronę wyjazdu numer 11.



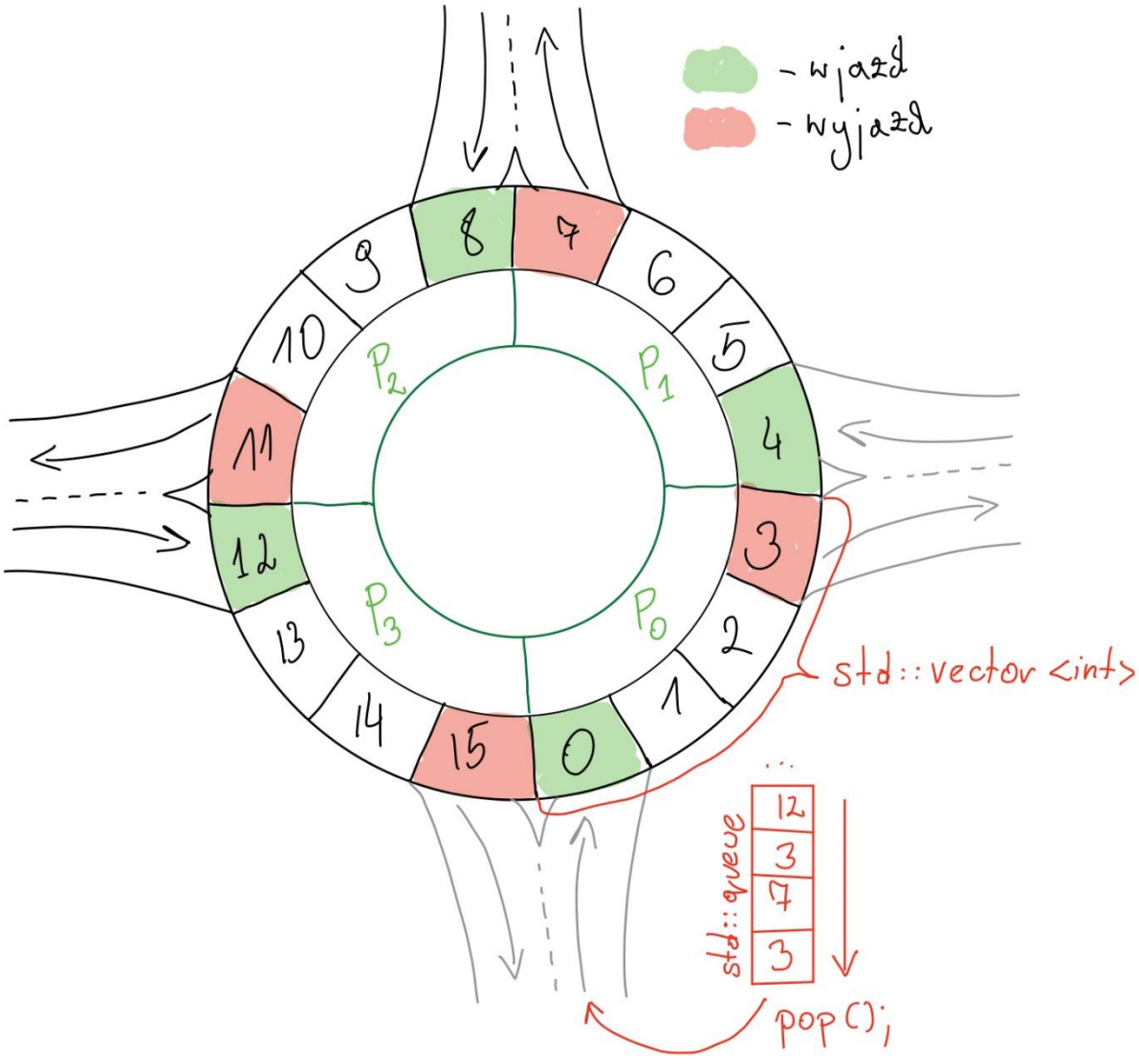
Przykładowy ruch na rondzie

Na powyższym schemacie widzimy, że na rondo w danym momencie chcą wjechać 3 samochody. Zakładamy, że rondo w danym momencie jest puste, zatem samochody wjeżdżają bezkolizyjnie. Samochód 11 potrzebuje 12 iteracji by dotrzeć do swojego wyjazdu, oraz kolejne analogicznie potrzebują 4 iteracje. Zgodnie z założeniem programu równoległego już po 4 iteracjach na rondzie pozostanie tylko jeden samochód.

Dany proces może przyjąć nowy samochód na dwa sposoby:

- Samochód z poprzedniego procesu nie dotarł jeszcze do swojego zjazdu więc przekazujemy go do kolejnego
- Element wjazdowy dla danego procesu jest pusty więc możemy przyjąć samochód oczekujący na wjazd (zakładając, że takowy istnieje).

Samochody znajdujące się na rondzie mają pierwszeństwo!



Implementacja kontenerów obsługujących rondo

Kolejki oczekujące są indywidualnie implementowane dla każdego wjazdu/procesu. Za logikę odpowiada kontener std::queue (carsAtEntrance) przechowujący indeksy wyjazdów samochodów oczekujących na wjazd. Natomiast część ronda przypisaną do danego procesu obsługuje std::vector

(*currentRoundaboutPart*). **Ilość wjazdów/wyjazdów(procesów), ogległości między wjazdami i wjazdami możemy zmienić elastycznie bez ingerencji w kod programu.** Program jest wykonywany przez arbitralnie ustaloną ilość iteracji (default = 20).

3. Komunikacja Procesów

Program MPI opiera się na wymianie wiadomości między programami. W tym przypadku każdy proces **wysyła** lub **odbiera** jakąś wiadomość. W każdej iteracji (**oprócz pierwszej i ostatniej)** każdy proces wysyła ostatni samochód z zakresu swojej części ronda (std::vector.back()). Element ten zostaje odebrany w kolejnym procesie oraz wpisany jako pierwszy element części ronda kolejnego procesu.

Komunikacja ta opiera się na instrukcjach MPI_Send oraz MPI_Recv. Jest to model blokującej komunikacji Point-To-Point. Znaczy to, że jeżeli jeden proces wyśle wiadomość to program nie będzie wykonywany do póki wiadomość ta nie zostanie odebrana przez adresata. Z tego właśnie powodu pierwsza oraz ostatnia iteracja różni się nieco od każdej innej(kwestia ta zostanie opisana później).

Kolejność komunikacji wygląda następująco:

1. Proces odbiera wjeżdżający samochód z poprzedniej części ronda.
2. Indeks samochodu zostaje zapisany w zmiennej tymczasowej (incomingCar)
3. Wykonywany zostaje ruch samochodów
4. Wpisujemy odebrany samochód jako pierwszy w części ronda
5. Ostatni samochód wyjeżdża z ronda(outgoingCar) lub kontynuuje podróż na rondzie – zgodnie z tym przesyłamy kolejnemu procesowi albo numer indeksu samochodu lub *NONE* oznaczający, że pole jest puste.

Wysyłanie wiadomości:

```
MPI_Send(&outgoingCar, 1, MPI_INT, myId + 1, myId + 1, MPI_COMM_WORLD);
```

Odbieranie wiadomości:

```
MPI_Recv(&incomingCar, 1, MPI_INT, myId - 1, myId, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
```

W powyższy sposób komunikują się ze sobą procesy w iteracjach programu poza pierwszą oraz ostatnią.

3.1 Komunikacja Procesów – pierwsza iteracja

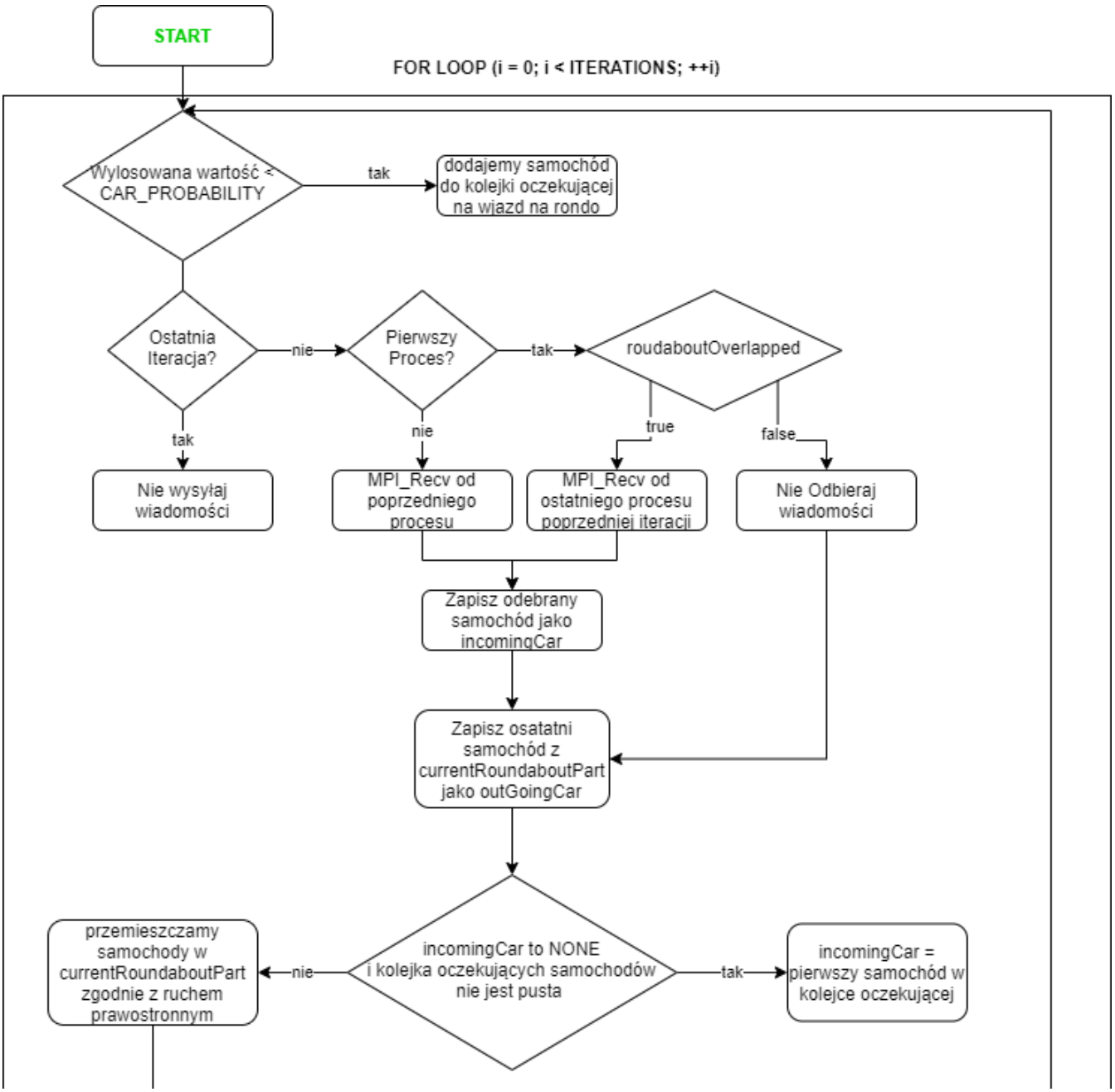
Według wcześniej wspomnianej kolejności komunikacji pierwszą czynnością jest odbiór wiadomości od poprzedniego procesu. Z tego powodu nie będzie to możliwe dla pierwszego procesu, dlatego, że taka wiadomość nie będzie istnieć – zatem w pierwszej iteracji programu, pierwszy proces nie będzie odbierał żadnych wiadomości, a jedynie wysyłał. Gdyby proces ten miał polecenie służące do odbioru – czekałby on w nieskończoność na wiadomość, która nigdy nie zostałaby wysłana. Z tego powodu w pierwszej iteracji ostatni proces zmienia flagę pomocniczą (*roudaboutOvelapped)* z *false* na *true* i przesyła tę wiadomość w formie Broadcast (Point-To-Many Communication). W tym przypadku ostatni proces zmienia wartość flagi oraz jej zmienioną wartość przesyła w sposób nieblokujący do wszystkich innych procesów. Dzięki temu od kolejnych iteracji pierwszy proces będzie już odbierał wiadomość od ostatniego procesu (0 -> 1 -> 2 -> 3 -> 0 -> …).

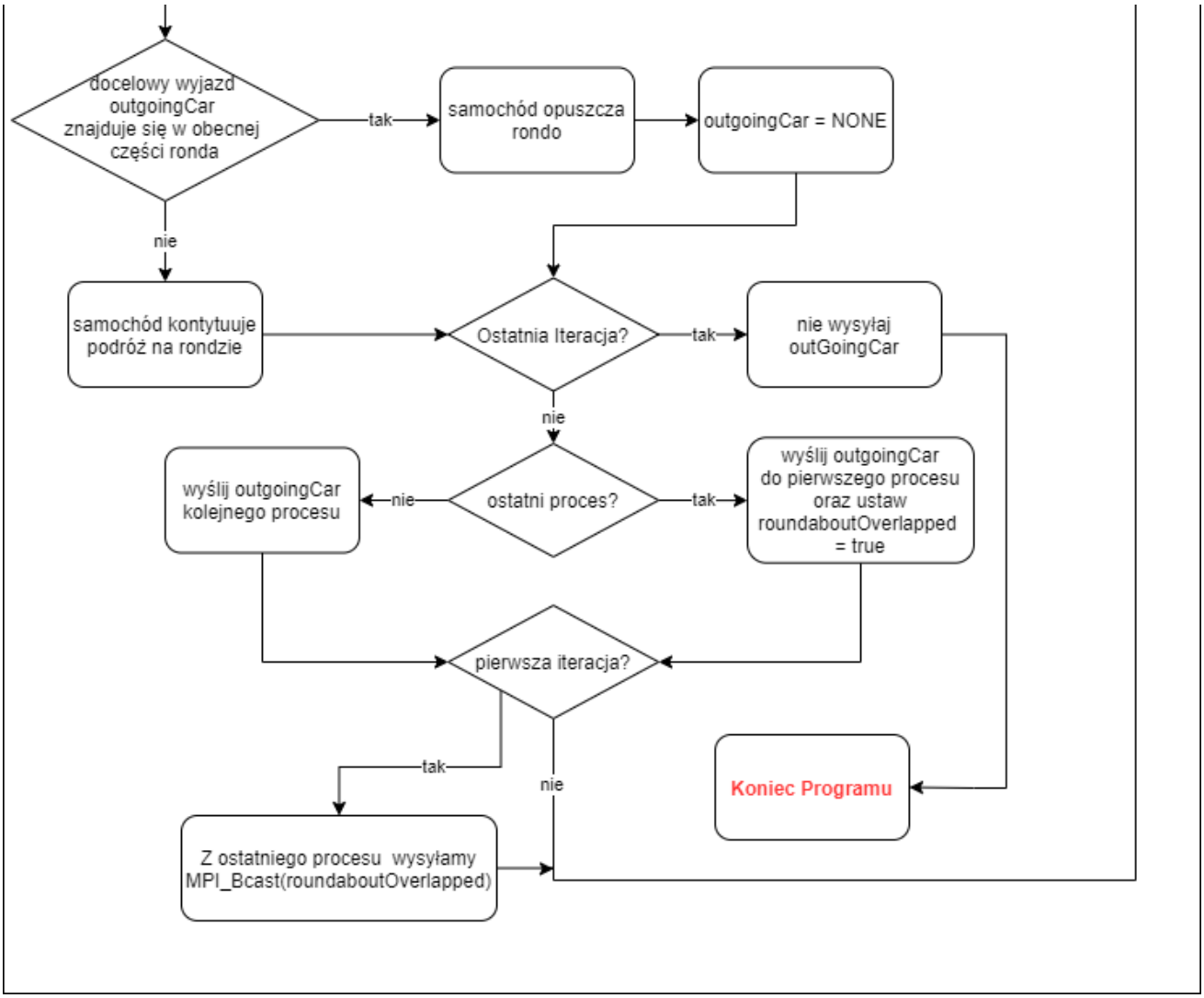
```
MPI_Bcast(&roundaboutOverlapped, 1, MPI_INT, numOfProcs-1, MPI_COMM_WORLD);
```

3.2 Komunikacja Procesów – ostatnia iteracja

By uniknąć wysłania wiadomości, które nigdy nie zostaną odebrane należy zadbać o to by w ostatniej iteracji MPI_Send nie zostało wykonane. W tym przypadku zostaje to uzyskane prostym sprawdzeniem czy nie jesteśmy w ostatniej iteracji. Dzięki temu unikamy wycieków REQUEST'ów.

3. Diagram Blokowy





Schemat blokowy

Cały, nieucięty diagram został załączony do projektu w formacie .png

4. Działanie programu – kompilacja i uruchomienie

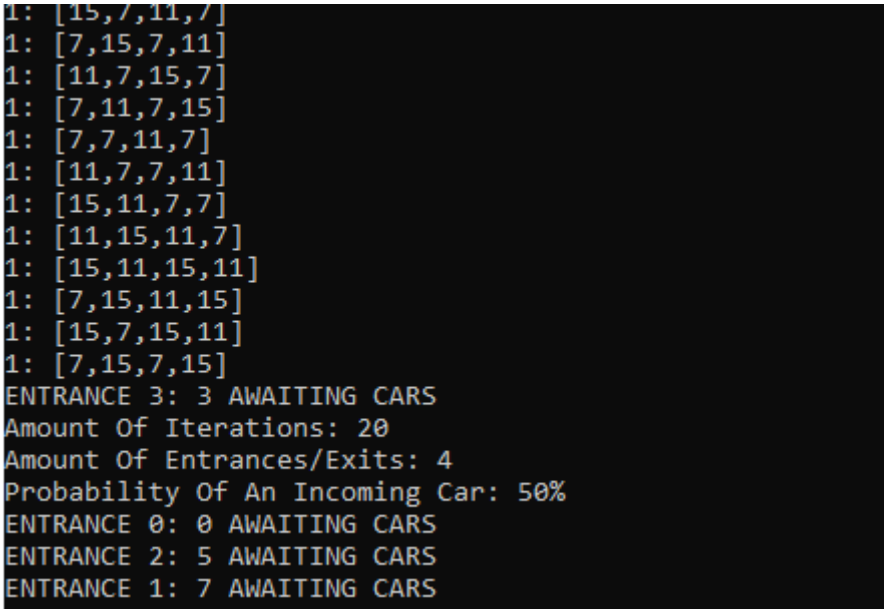
Kompilacja: *make*

Uruchomienie Programu: *make run procs=<ilość procesów>*

Czyszczenie plików obiektowych: *make clean*

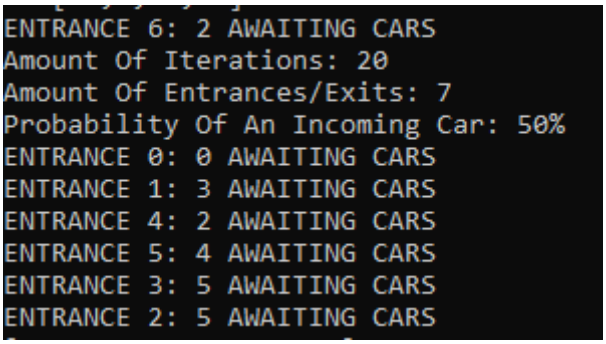
5. Przykładowe wyniki

Przykład dla ronda z czteroma wjazdami z prawdopodobieństwem pojawienia się nowego samochodu na wjeździe = 50% oraz 20 iteracjami. Niestety kolejnośc wypisywanych komunikatów jest inna za każdym uruchomieniem ze względu na naturę obsługi wyjścia standardowego w programach równoległych.



Przykład ronda z 4 wjazdami/wyjazdami dla 20 iteracji oraz 50% szansy na pojawienie się nowego samochodu na wjeździe

Jak widzimy po 20 iteracjach na trzech wjazdach ronda czekają samochody. Powyżej wyników widzimy poszczególne wyniki w każdej iteracji dla procesu.



Przykład ronda z 7 wjazdami/wyjazdami dla 20 iteracji oraz 50% szansy na pojawienie się nowego samochodu na wjeździe

Efekt uruchomienia programu dla 20 iteracji dla 2 wjazdów/wyjazdów na rondzie:

```
0: [-1,-1,-1,-1]
1: [-1,-1,-1,-1]
1: [3,-1,-1,-1]
0: [3,-1,-1,-1]
0: [3,3,-1,-1]
0: [3,3,3,-1]
0: [7,3,3,3]
0: [-1,7,3,3]
0: [-1,-1,7,3]
0: [-1,-1,-1,7]
0: [-1,-1,-1,-1]
0: [-1,-1,-1,-1]
0: [3,-1,-1,-1]
0: [-1,3,-1,-1]
0: [7,-1,3,-1]
0: [3,7,-1,3]
0: [-1,3,7,-1]
0: [7,-1,3,7]
0: [3,7,-1,3]
0: [3,3,7,-1]
0: [-1,3,3,7]
0: [3,-1,3,3]
1: [3,3,-1,-1]
1: [3,3,3,-1]
1: [7,3,3,3]
1: [-1,7,3,3]
1: [-1,-1,7,3]
1: [-1,-1,-1,7]
1: [7,-1,-1,-1]
1: [-1,7,-1,-1]
1: [3,-1,7,-1]
1: [-1,3,-1,7]
1: [7,-1,3,-1]
1: [3,7,-1,3]
1: [-1,3,7,-1]
1: [7,-1,3,7]
1: [7,7,-1,3]
1: [3,7,7,-1]
1: [3,3,7,7]
1: [7,3,3,7]
Amount Of Iterations: 20
Amount Of Entrances/Exits: 2
Probability Of An Incoming Car: 50%
ENTRANCE 0: 0 AWAITING CARS
ENTRANCE 1: 1 AWAITING CARS
```

Przykład ronda z 2 wjazdami/wyjazdami dla 20 iteracji oraz 50% szansy na pojawienie się nowego samochodu na wjeździe

Pozostałe parametry mogą zostać zmienione w pliku roundabout.cpp na samym początku pliku:

```
constexpr int ROUNDABOUT_PART_SIZE = 4;
// best value for ITERATIONS is ROUNDABOUT_PART_SIZE * numberOfProcesses (eg. 20 for 5 procs.)
constexpr int ITERATIONS = 20;
constexpr double CAR_PROBABILITY = 0.5;
```