Monitoring Proximity and Coverage Graphs of a ZigBee-Based Geo-Referenced Mobile Wireless Sensor Network

Mehmet Ozgan
Matr.Nr: 0526530
mozgan@gmail.com

Lilly Maria Treml
Matr.Nr: 1528458
e1528458@student.tuwien.ac.at

Abstract—In this project we simulate the motion of nodes in a ZigBee network using GPU. To do so we compute several graphs describing the spatial attributes and communication between those nodes. This includes a Voronoi diagram of a given set of seeds in a 2D grid by applying jump flooding algorithm (JFA) on GPU-based many-core architecture. The solution were implemented in CUDA C programming language.

Index Terms—GPU, Voronoi diagram, proximity graph, connectivity graph, CUDA, parallel computing, network simulation

1 Introduction

Traditionally, a lot of computer programs have been written for serial computation. In serial computation, there is only one processor to compute of result. This processor has access into the cells of RAM (Read Only Machine) to read and to write the datas. Each memory access takes a single time. Basically, this mode is a *single-core* computation.

On the other hand, a parallel computer is a set of processors that are able to work cooperatively to compute a large problem. There are two main trajectories for designing microprocessor: *multicore* and *many-core*.

The *multicore* trajectory seeks to maintain the execution speed of sequential programs while moving into multiple cores. Each processor in this set can run its own program for computing of results in the same time. In this case, the memory is shared. A current exemplar is Intel®Core™ i7-7920HQ Processor which has 4 cores.

The *many-core* trajectory focuses more on the execution throughput of parallel applications. In this case, each thread has a private register set and local memory, and each block has a shared memory which can be read from/written to by all threads in the same block, and finally each grid has a global memory which is accessible from all the blocks inside. On a higher level each processor has its own cache and the whole device (GPU) has a DRAM (Dynamic Random Access Memory). The Fig. 1 shows the architecture difference between CPU and GPU.

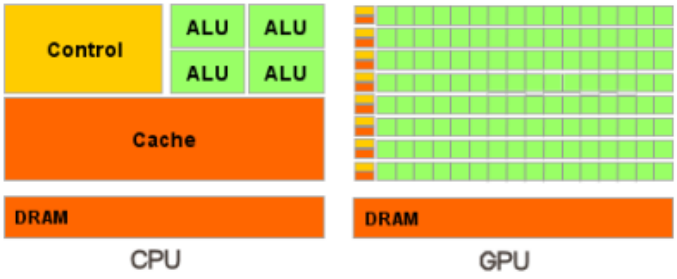

Fig. 1. Basic structure of a typical CPU and GPU

A graphics processing unit (GPU) is a chip that handles any functions relating to what displays on computer's screen. GPUs are used in many different devices for example personal computers, embedded systems, mobile phones etc.

2 Problem Description

In a ZigBee Network, nodes can only communicate if the following two conditions are satisfied:

- Nodes are in the communication range
- Type (router, end-device, coordinator) of the nodes are compatible

Therefore, to monitor the change in connections and coverage, a simulation using paralell algorithms should be implemented. The simulation includes the motion of the nodes according to different mobility models, as described in [6] and generating a proximity-graph, a voroni-diagram and a connectivity-graph at each timestep to gain information about the connectivity and spacial attributes of the network. However, the nodes should not collide with each other or other obsticles, therefore a collision avoidance using GPU needs also to be implemented.

3 Parallelism in GPU

A CUDA program consists of one or more phases that are executed on either the host (CPU) or a CUDA-capable device (GPU). The host code can be written in ANSI C, C++ or both of them. The device code is written in ANSI C extended with keywords for labeling data-parallel functions, called *kernels*, and their associated data structures. A simple kernel call looks like the following code:

```
kernel<<< gridDim, blockDim >>>(args);
```

`gridDim` is the dimensions of the grid in blocks and `blockDim` is the dimensions of the block in threads.

4 Voronoi Diagram

Definition 4.1. Let $S = \{p_1, p_2, \ldots, p_n\}$ be a set of n different points in the $\mathbb{R}^2$ plane. A *Voronoi diagram* divides the plane into n polygonal regions called *Voronoi regions*, and i^{th} Voronoi region denoted by $\mathcal{V}(p_i)$ of point p_i.
For $p, q \in S$ and $p \neq q$, let

- $L(p,q) = \{z \in \mathbb{R}^2; |p-z| = |q-z|\}$,
- $A(p,q) = \{z \in \mathbb{R}^2; |p-z| < |q-z|\}$.

$L(p,q)$ is the line segment of p and q. For each element r in the set $L(p,q)$, the distance between r and p is equal to between r and q. $A(p,q)$ gives the set of points in the given plane which are nearest to p than q. The set

$$A(p,S) = \bigcap_{\substack{q \in S \\ p \neq q}} A(p,q) \tag{1}$$

of all points z that are closer to p than to any other element of S. In other words, $A(p,S)$ is the area of Voronoi region $\mathcal{V}(p)$.

5 Mobility Model

The mobility modles describe the motion of the nodes in the simulated network. In [6] several models are described. In our simulation we use the *Random Walk* Model and the *Random Waypoint* Model, which can be chosen by the user. Further, for usability reasons we fixed the step-size of the simulation, as we do not have a GUI.

5.1 Random Waypoint

At the start of the simulation each node gets a random destination point assigned. With continous velocity each node moves towards the destination, if it reaches the destination, it gets assigend a new random destination and velocity. [6] To do so, each destination is saved in a map with the node-id as key, after each step of the simulation it is checked if the node arrived at the destination, depending on this condition the next move will be executed. Further, to consider the different velocities of the nodes, they get assigend a random velocity, which can be either *xmax* or *ymax*, depending which is the smaller one, divided by *step number* at max. In case of a possible collision,the nodes get a new destination assigend, therefore they should automatically move away from each other.

5.2 Random Walk

Similar to the Random Waypoint, the Random Walk the movement of the nodes depends on randomness. The idea behind this movement, is the emulation of particle movement in physics [6]. As there is a uniformly possibilty of a particle being at a specific position, the nodes change their direction at each simulation step [6]. Therefore, the coordinates in the map will be reassigened after every step, which results in a random movement of the nodes. As we want to present a significant difference in the position of the nodes, the nodes "jump" randomly up, down, left, or right. To do so, wie randomly assign new coordinates to each node, which gives an equal distributed possibility of where the node will appear. There is also a slight possibility, that the node does not move or just appears at the other side,if the node is too close to the boundry. In case of a collision, the node moves away in the field, till there is no more collision.

6 Connectivity Graph

This directed graph describes the communication between the nodes. As mentioned before, nodes

in the ZigBee protocol can only communicate with each other if they are in each others range and compatible. At each step, those conditions are validated and according to the result, the graph generates the communication representation. To visualize those connections, we provide a .png file and an array, which represents the edges between the nodes.

7 Proximity Graph

This undirected graph describes the spatial distance between nodes. If the nodes satisfy predefined geometric conditions. In this project the geomteric conditions are predefined by the Voronoi-Diagramm. Therfore the proximity graph gives the edges of the nodes using the vornoi cells.

8 Jump Flooding Algorithm

In this project we use a Jump Flooding Algorithm (JFA) to generate the Voronoi diagram. The reason for this is, as [7] states, the speed of JFA is nearly independent to the number of input sites. However, there is a possibility of pixel errors in the output, as some pixels incorrectly record the nearest sites. [7]. The principle of the JFA is quite simple (see 2. At the beginning we have a $n \times m$ grid. The pixel in the lower left corner of the grid is our start point, therefore we suppose it has some attributes/information we want to propagate over the grid. First, we propagate the information of the start (seed) to its (maximum) eight neighbors at positions (x+i,y+j) where i,j1,0,1 [7]. In every step, we propagate the information using the neighbors to the next pixel in the grid, therefore the whole grid is "filled" after *n-1* steps. [1]

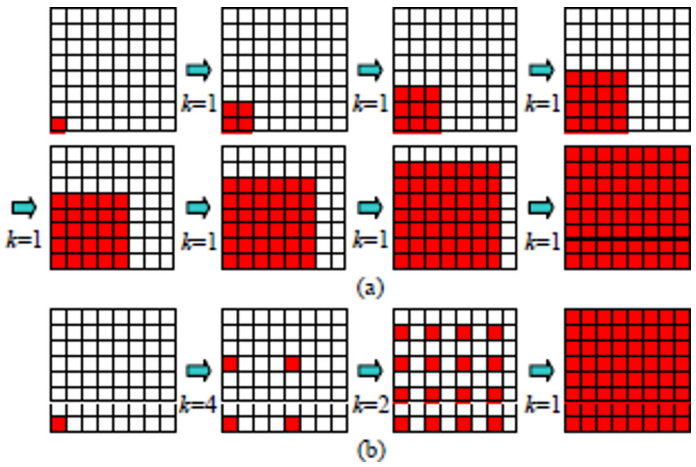

Fig. 2. Schematic representation of the JFA according to [7]

9 Collision Avoidance

To detect collisions we create an array *gpu_collision*, which represent a collision between two nodes, if the value is set to 1. This array is created while computing the Voronoi diagram, by doing so we can use the cells to limit the possible collisions. The array is then checked during the motion and subsequently, as mentioned in section 5.2 and section 5.1, the collisions are handled accordingly.

10 Benchmarking

As our algroithms are based on randomness, we used fixed values for sites and destination, to provide statistical relevant information. This information then can be used to benchmark and evaluate the efficiency and differences of the algorithms depending on the number of nodes. The fixed values are stated below:

- *max x-coordinate* 2048
- *max y-coordinate* 2048
- *number of sites* 8

10.1 Results

Results of the benchmarking are provided in a result file, which will be generated after execution. The file gives infomration about the used GPU and running time of the different algorithms in ms and the spatial attributes of the graphs. For example see listing 1. Further a more specific documentation and explanation of the algorithms can be found in the comments of the code.

```
The used GPU
        Device name: GeForce GT 740M
        Memory Clock Rate (KHz):
           900000
        Memory Bus Width (Bits): 64
        Peak Memory Bandwidth (GB/s):
           14.400000


Results for 2048x2048 coordinates
   system
with 4 routers and 4 end devices
--- --- --- --- --- --- --- --- ---
   --- --- --- --- --- --- --- --- ---
    ---
The elapsed time in GPU for
   Connectivity algorithm: 2.00 ms
The elapsed time in GPU for JFA
   Voronoi algorithm: 258.16 ms
```

```
The elapsed time in GPU for Proximity
   algorithm: 18.26 ms

Vertices of proximity graph: (1, 0)
   (1, 2) (1, 6) (2, 3) (2, 7) (3, 4)
   (3, 7) (4, 5) (5, 0) (5, 6) (5, 7)
   (6, 0) (7, 0) (7, 4)
Vertices of connectivity graph: (0, 7)
    (2, 7) (6, 1) (7, 0) (7, 3) (7, 4)
Collisions: (3, 4) (4, 3)
```

Listing 1. Example for Benchmark Result File

11 Simulation

11.1 Usage

For usability reasons we use a make-file to create the executables, depending on what the user wants. By executing *make help* in the command line, the user can see the options. When choosing *BENCH* the benchmarking is called and the user does not have to provide any parameters. Otherwise, the user has to choose the mobility model additionally to the parameters specified in the project description. If the input is incorrect, we provide a usage-message as assistance.

11.2 Output

We provide several output files. This includes the Voronoi Diagram, the Connectivity Graph and the Connectivity Graph as *png*, additionally we represent the graphs as list of edges in the result file. The edge-list representation is to ensure an overview of the graphs, as for example, the lines which represent the edges in the connectivity graphs, are partly hard to see in the *png*.

12 Conclusion

As many references state, the JFA algorithm is prone to errors in the pixels. Therefore we tried to optimize the algorithm and reduce its errors, but it is still far from perfect. Further due to the lack of randomness in *curand()*, it was a challenge to correctly implement the random based mobility models and initialization. Additionally the benchmarking results will be distorted by using random based algorithms, therefore we can only provide the resources and time consumed by the algorithms.

In summary, *CUDA* is a great option to code in GPU, unfortunately it seems to be error-prone, as for example *curand()* mentioned above. Further, the smaller number of available datatypes complicate the implementation of simple algorithms. However, the higher efficiency of using GPU instead of CPU in simulation is mentioned in several references and CUDA is a great tool to get started.

References

[1] Guodong Rong, *Jump Flooding Algorithm On Graphics Hardware And Its Applications*, 2007

[2] Guodong Rong, Tiow-Seng Tan, *Jump Flooding in GPU with Applications to Voronoi Diagram and Distance Transform*, 2006

[3] David B. Kirk, Wen-mei W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 2010.

[4] Rolf Klein, *Concrete and Abstract Voronoi Diagrams*, Springer-Verlag Berlin Heidelberg, 1989.

[5] V. Thambawita, R. Ragel, D. Elkaduwe, *To Use or Not to Use: Graphics Processing Units (GPUs) for Pattern Matching Algorithms*.

[6] S.M. Mousavi, H. R. Rabiee, M. Moshref, A. Dabirmoghaddam, *Mobility Pattern Recognition in Mobile Ad-Hoc Networks*, ACM International Conference on Mobile Technology, Applica- tions and Systems (ACM Mobility Conference 2007), Singapore, 10-12 September 2007

[7] Rong, Guodong, and Tiow-Seng Tan. *Variants of jump flooding algorithm for computing discrete Voronoi diagrams.*, Voronoi Diagrams in Science and Engineering, 2007. ISVD'07. 4th International Symposium on, pp. 176-181. IEEE, 2007