Parallelization of random graph algorithms

Fabrizio Corriera

27/11/2023

Contents

1 Introduction

The scope of this work is to explore and analyse the parallelization of different random graph algorithms, using different techniques, and comparing the results.

Random graphs are mathematical models used to represent a wide range of complex systems, such as social networks, biological networks, and communication networks. They provide a way to study and understand the emergent properties of large, interconnected systems by introducing randomness into the structure of the graphs.

The general idea behind random graphs is to generate graphs with certain probabilistic properties, rather than specifying the connections between nodes explicitly. This randomness allows researchers to study the average behavior and characteristics of graphs within a certain probability distribution. Random graph models are particularly useful for capturing the inherent unpredictability and complexity of real-world networks.

Some common models and algorithms for generating random graphs include:

- **Erdős-Rényi (ER) Model:** In the Erdős-Rényi model, edges are added between pairs of nodes with a fixed probability p. The resulting graphs often exhibit a phase transition, where below a certain probability, the graph is likely to consist of isolated components, and above that probability, a giant connected component emerges.
- **Watts-Strogatz Model:** The Watts-Strogatz model starts with a regular lattice and rewires edges with a certain probability. This small amount of randomness helps capture both the regularity found in some networks and the small-world phenomenon observed in many real-world networks.
- **Barabási-Albert Model:** The Barabási-Albert model is a preferential attachment model where nodes are added to the graph one at a time. Each new node connects to existing nodes with a probability that is proportional to their degree. This model leads to scale-free networks, where a small number of nodes have a disproportionately large number of connections.
- **Random Geometric Graph (RGG):** A Random Geometric Graph (RGG) is a type of random graph where nodes are distributed randomly in a metric space, and edges are formed based on geometric proximity. In other words, nodes are points in a space, and an edge is established between two nodes if the distance between them is below a certain threshold. This model is particularly relevant in scenarios where the spatial arrangement of nodes plays a crucial role, such as wireless sensor networks, communication networks, or applications involving proximity-based interactions.

While random graph models offer valuable insights into the structural properties of networks, it is important to specify that they often fall short in fully capturing the complexity and nuanced characteristics of real-life networks, like heterogeneity, temporal dynamic, communities structure and many more.

Parallelization of this kind of algorithms is not a much discussed subject, nor it is considered a relevant matter when talking about parallelization. Some studies have been published mainly regarding the distributed version of these algorithms, but very few results may be found regarding local parallelization.

2 Functional requirements

The functional requirements for this project are:

- Develop at least a functional parallelized version of a known random graph algorithm.
- Test the perfermance with different parameters with different orders of magnitude.
- Compare the performance of the linear version with the parallelized one.

The best result would be to be able to use different parallelization methods on different algorithms to truly be able to appreciate the differences in performance.

3 Tools used

For this work we have used the following tools:

- **OpenCL:** OpenCL (Open Computing Language) is an open standard for parallel programming across heterogeneous platforms. It allows developers to write programs that can execute on various devices, including CPUs, GPUs, and accelerators, enabling efficient parallel processing for tasks like scientific simulations, image processing, and machine learning. OpenCL provides a framework for exploiting the computational power of diverse hardware architectures in a unified manner.
- **C programming language:** C, a general-purpose programming language, is widely used for its efficiency and versatility. OpenCL C is a subset of C designed for parallel programming within the OpenCL framework. Developers use the OpenCL API to create programs that leverage parallel computing across diverse devices, such as CPUs and GPUs, making it a powerful tool for high-performance computing and parallel processing tasks.

4 OpenCL

4.1 A gentle introduction

OpenCL, or Open Computing Language, is an open standard developed by the Khronos Group, designed to enable heterogeneous computing across various

platforms. It provides a framework for parallel programming, allowing developers to harness the computational power of diverse devices, including CPUs, GPUs, and accelerators.

It introduces a parallel programming model where tasks are divided into work items, organized into work groups. These work items are executed concurrently, exploiting the parallel processing capabilities of the underlying hardware. OpenCL operates with a host-device model: the host, typically a CPU, manages the overall execution and launches tasks on the device, such as a GPU. The device executes the parallelized tasks specified by the OpenCL C language, that is a language tailored for parallel programming. It includes constructs for data parallelism, work item synchronization, and memory management specific to OpenCL.

OpenCL provides a hierarchical memory model, including global, local, and private memory spaces. Developers must carefully manage data movement between these spaces to optimize performance. OpenCL's execution model allows developers to specify how work items are divided into work groups, facilitating efficient parallel execution. This model is especially powerful for applications requiring massive parallelism, like image processing and scientific simulations.

The general steps for OpenCL workflow are:

1. Platform and device discovery: we begin by discovering available OpenCL platforms and devices on a system. A platform represents a particular vendor's OpenCL implementation, while a device refers to a specific hardware unit, like a GPU or CPU.

2. Kernel programming: we write OpenCL kernels, which are functions written in OpenCL C that define the behavior of a single work item. These kernels can be executed in parallel on multiple devices.

3. Compilation and execution: OpenCL programs consist of both host code (typically written in C or another high-level language) and kernel code (written in OpenCL C). The host code manages the execution flow, while the kernel code is compiled and executed on the devices.

4. Data transfer: efficient data transfer between the host and device is crucial. OpenCL provides functions to move data between the host and device memories, ensuring that the right data is available for computation.

5. Synchronization and control: OpenCL provides mechanisms for synchronizing work items within a work group and controlling the overall execution flow. Events and barriers help manage dependencies and synchronization between different parts of the program.

4.2 Setting up an OpenCL program

We will now illustrate technically the general steps needed to implement an OpenCL algorithm.

```
//Configure the OpenCL environment
cl_platform_id platform = 0;
clGetPlatformIDs(1, &platform, NULL);
cl_device_id device = 0;
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
cl_context context = clCreateContext(NULL, 1, &device, NULL,
    NULL, NULL);
cl_command_queue queue = clCreateCommandQueue(context, device,
    0, NULL);
```

Here, in this example we first define and inizialize a variable of type "cl_platform_id" and then use a function "clGetPlatformIDs" to assign it the value of the id of the first platform available. The function signature is:

- clGetPlatformIDs(cl_uint num_entries, cl_platform_id *platforms, cl_uint *num_platforms)
  - num_entries: this parameter specifies the max number of entries we want to be appended to *platforms*.
  - platforms: this parameter specifies the variable where we want the IDs of the platforms to be appended. This variable will contain at most *num_entries* IDs or all the platforms available in the system, whichever is the minimum.
  - num_platforms: the function will return the total number of platforms available to this variable if specified.

So, now we have the IDs for the platforms we want our kernel to be executed on. We now need to find the id of the devices. In order to do this, very similarly we define a variable of type "cl_device_id" and then use a function "clGetDeviceIDs" to pass it the IDs for the desired devices. The function signature is:

- clGetDeviceIDs(cl_platform_id platform, cl_device_type device_type, cl_uint num_entries, cl_device_id *devices, cl_uint *num_devices)
  - platform: we pass to the function the ID of the platform we are querying for devices.
  - device_type: a bitfield that identifies the type of OpenCL device. The *device_type* can be used to query specific OpenCL devices or all OpenCL devices available.
  - num_entries: this parameter specifies the max number of entries we want to be appended to *devices*.
  - devices: this parameter specifies the variable where we want the IDs of the devices to be appended. This variable will contain at most *num_entries* IDs or all the devices whose type matches *device_type*, whichever is the minimum.
  - num_devices: the function will return the total number of devices available to this variable if specified.

Now we are ready to define our context. An OpenCL context is created with one or more devices. Contexts are used by the OpenCL runtime for managing objects such as command-queues, memory, program and kernel objects and for executing kernels on one or more devices specified in the context. To define the context we use the "clCreateContext" function:

- clCreateContext(cl_context_properties *properties, cl_uint num_devices, const cl_device_id *devices, void *pfn_notify(const char *errinfo, const void *private_info, size_t cb, void *user_data), void *user_data, cl_int *errcode_ret)
  - properties: specifies a list of context property names and their corresponding values.*properties* can be NULL in which case the platform that is selected is implementation-defined.
  - num_devices: this variable contains the number of devices specified in *devices*.
  - devices: a pointer to a list of unique devices returned by *clGetDeviceIDs* for a platform.
  - pfn_notify: a callback function that can be registered by the application. This callback function will be used by the OpenCL implementation to report information on errors that occur in this context.
  - user_data: passed as the *user_data* argument when *pfn_notify* is called. *user_data* can be NULL.
  - errcode_ret: returns an appropriate error code. If *errcode_ret* is NULL, no error code is returned.

After defining the context we have to define a command queue. An OpenCL command queue is used by the host application to send kernels and data transfer functions to a device for execution. By enqueueing commands into a command queue, kernels and data transfer functions may execute asynchronously and in parallel with application host code. We use the "clCreateCommandQueue" function in order to do this.

- clCreateCommandQueue(cl_context context, cl_device_id device, cl_command_queue_properties properties, cl_int *errcode_ret)
  - context: we must pass an OpenCL context in order to create a command queue.
  - device: must be a device associated with *context*.
  - properties: specifies a list of properties for the command-queue.
  - errcode_ret: returns an appropriate error code. If *errcode_ret* is NULL, no error code is returned.

After the configuration of our OpenCL context, we can go on and talk about compiling the kernel.

```
//Compile the kernel
cl_program program = clCreateProgramWithSource(context, 1, &
    kernelstring, NULL, NULL);
clBuildProgram(program, 0, NULL, "", NULL, NULL);
```

This is one of the main features that highlights the flexibility of OpenCL. What we are doing with these lines of code is dinamically compiling code (that can either be defined as a string in the same file containing the main function or can be written in another file).

The function *clCreateProgramWithSource* creates a program object for a context, and loads the source code specified by the text strings in the strings array into the program object.

- clCreateProgramWithSource(cl_context context, cl_uint count, const char **strings, const size_t *lengths, cl_int *errcode_ret)
  - context: the OpenCL context on which we want to run the program.
  - count: this parameter specifies the number of pointers contained in *strings*.
  - strings: an array of pointers to optionally null-terminated character strings that make up the source code.
  - lengths: an array with the number of chars in each string (the string length). If an element in *lengths* is zero, its accompanying string is null-terminated. If *lengths* is NULL, all strings in the *strings* argument are considered null-terminated. Any length value passed in that is greater than zero excludes the null terminator in its count.
  - errcode_ret: returns an appropriate error code. If *errcode_ret* is NULL, no error code is returned.

The clBuildProgram instead builds (compiles and links) a program executable from the program source or binary.

- clBuildProgram(cl_program program, cl_uint num_devices, const cl_device_id *device_list, const char *options, void (CL_CALLBACK *pfn_notify)(cl_program program, void *user_data), void *user_data)
  - program: the variable containing the program object.
  - num_devices: The number of devices listed in *device_list*.
  - device_list: a pointer to a list of devices associated with program. If *device_list* is a NULL value, the program executable is built for all devices associated with *program* for which a source or binary has been loaded.
  - options: a pointer to a null-terminated string of characters that describes the build options to be used for building the program executable.

  - pfn_notify: a function pointer to a notification routine. The notification routine is a callback function that an application can register and which will be called when the program executable has been built (successfully or unsuccessfully). If *pfn_notify* is not NULL, clBuildProgram does not need to wait for the build to complete and can return immediately once the build operation can begin.
  - user_data: passed as an argument when *pfn_notify* is called. *user_data* can be NULL.

```
//Configure the kernel
cl_kernel kernel = clCreateKernel(program, "ER", NULL);
```

This function creates a kernel object, that is basically a function declared in a program.

- cl_kernel clCreateKernel(cl_program program, const char *kernel_name, cl_int *errcode_ret)
  - program: a program object with a successfully built executable.
  - kernel_name: a function name in the program declared with the __kernel qualifier.
  - errorcode_ret: returns an appropriate error code. If *errcode_ret* is NULL, no error code is returned.

4.3 Memory management

Before executing the program we just built we should make some considerations regarding the memory: the program we built will not run on the same device on which we are executing the host code, so it will not be able to access the memory registers where we instancieted the variables declared on the host program. Therefore it is necessary to copy the variables needed for the program execution on the device memory.

```
//Prepare OpenCL memory objects
cl_mem bufG = clCreateBuffer(context, CL_MEM_READ_WRITE,
    N*N*sizeof(int), NULL, NULL);
```

clCreateBuffer, as its name suggestes, creates a buffer object.

- clCreateBuffer(cl_context context, cl_mem_flags flags, size_t size, void *host_ptr, cl_int *errcode_ret)
  - context: a valid OpenCL context used to create the buffer object.
  - flags: a bit-field that is used to specify allocation and usage information such as the memory arena that should be used to allocate the buffer object and how it will be used.
  - size: the size in bytes of the buffer memory object to be allocated.

- host_ptr: a pointer to the buffer data that may already be allocated by the application.
- errcode_ret: returns an appropriate error code. If *errcode_ret* is NULL, no error code is returned.

```
//Copy the data to the device
clEnqueueWriteBuffer(queue, bufG, CL_TRUE, 0,
    N*N*sizeof(int), G, 0, NULL, NULL);
```

After allocating the memory we must copy the data in the buffer.

- clEnqueueWriteBuffer(cl_command_queue command_queue, cl_mem buffer, cl_bool blocking_write, size_t offset, size_t size, const void *ptr, cl_uint num_events_in_wait_list, const cl_event *event_wait_list, cl_event *event)
  - command_queue: is a valid host command-queue in which the write command will be queued. *command_queue* and buffer must be created with the same OpenCL *context*.
  - buffer: the buffer object we are writing on.
  - blocking_write: indicates if the write operations are blocking (CL_TRUE) or nonblocking(CL_FALSE).
  - offset: the offset in bytes in the buffer object to write to.
  - size: the size in bytes of data being written.
  - ptr: the pointer to buffer in host memory where data is to be written from.
  - num_events_in_wait_list: this parameter specifies the size of the array *event_wait_list*. If *event_wait_list* is NULL, *num_events_in_wait_list* must be 0.
  - event_wait_list: this parameter specifies events that need to complete before this particular command can be executed. If it is NULL, then this particular command does not wait on any event to complete.
  - event: returns an event object that identifies this particular write command and can be used to query or queue a wait for this particular command to complete. *event* can be NULL in which case it will not be possible for the application to query the status of this command or queue a wait for this command to complete.

```
//Set kernel arguments
clSetKernelArg(kernel, 0, sizeof(unsigned int), (void*)&N);
clSetKernelArg(kernel, 1, sizeof(double), (void*)&p);
clSetKernelArg(kernel, 2, sizeof(unsigned int), (void*)&seed);
clSetKernelArg(kernel, 3, sizeof(unsigned int), (void*)&base);
clSetKernelArg(kernel, 4, sizeof(cl_mem), (void*)&bufG);
```

After we create the kernel and before we run it we must pass it its arguments, in the same order we defined in the __kernel program function.

- cl_int clSetKernelArg(cl_kernel kernel, cl_uint arg_index, size_t arg_size, const void *arg_value)
  - kernel: a valid kernel object.
  - arg_index: the index of the argument we are passing to the kernel. Arguments to the kernel are referred by indices that go from 0 for the leftmost argument to $n-1$, where n is the total number of arguments declared by a kernel.
  - arg_size: the size of the argument value. If the argument is a memory object, the size is the size of the memory object.
  - arg_value: the pointer to data that should be used as the argument value for argument specified by *arg_index*. The argument data pointed to by *arg_value* is copied and the *arg_value* pointer can therefore be reused by the application after clSetKernelArg returns. The argument value specified is the value used by all API calls that enqueue *kernel* (clEnqueueNDRangeKernel) until the argument value is changed by a call to clSetKernelArg for *kernel*.

```
// Copy the output back to CPU memory
clEnqueueReadBuffer(queue, bufG, CL_TRUE, 0, N*N*sizeof(int),
    G, 0, NULL, NULL);
```

As we can write on a buffer we can also read from it.

- clEnqueueReadBuffer(cl_command_queue command_queue, cl_mem buffer, cl_bool blocking_read, size_t offset, size_t size, const void *ptr, cl_uint num_events_in_wait_list, const cl_event *event_wait_list, cl_event *event)
  - command_queue: is a valid host command-queue in which the write command will be queued. *command_queue* and buffer must be created with the same OpenCL *context*.
  - buffer: the buffer object we are writing on.
  - blocking_read: indicates if the read operations are blocking (CL_TRUE) or nonblocking(CL_FALSE).
  - offset: the offset in bytes in the buffer object to write to.
  - size: the size in bytes of data being read.
  - ptr: the pointer to buffer in host memory where data is to be read into.
  - num_events_in_wait_list: this parameter specifies the size of the array *event_wait_list*. If *event_wait_list* is NULL, *num_events_in_wait_list* must be 0.
  - event_wait_list: this parameter specifies events that need to complete before this particular command can be executed. If it is NULL, then this particular command does not wait on any event to complete.

– event: returns an event object that identifies this particular write command and can be used to query or queue a wait for this particular command to complete. *event* can be NULL in which case it will not be possible for the application to query the status of this command or queue a wait for this command to complete.

4.4 Executing the kernel

```
// Run the kernel
const size_t local[2] = { TS, TS };
const size_t global[2] = { N, N };
clEnqueueNDRangeKernel(queue, kernel, 2, NULL, global,
    local, 0, NULL, &event);

// Wait for calculations to be finished
clWaitForEvents(1, &event);
```

The main function of this snippet of code is clEnqueueNDRangeKernel, that, as the name suggests, enqueues a command to execute a kernel on a device.

- clEnqueueNDRangeKernel(cl_command_queue command_queue, cl_kernel kernel, cl_uint work_dim, const size_t *global_work_offset, const size_t *global_work_size, const size_t *local_work_size, cl_uint num_events_in_wait_list, const cl_event *event_wait_list, cl_event *event)
  - command_queue: a valid command-queue. The kernel will be queued for execution on the device associated with *command_queue*.
  - kernel: a valid kernel object. The OpenCL context associated with *kernel* and *command_queue* must be the same.
  - work_dim: the number of dimensions used to specify the global work-items and work-items in the work-group. *work_dim* must be greater than zero and less than or equal to three.
  - global_work_offset: this parameter can be used to specify an array of *work_dim* unsigned values that describe the offset used to calculate the global ID of a work-item instead of having the global IDs always start at offset (0, 0,... 0). In the version of OpenCL we used (1.0) it must be set to NULL.
  - global_work_size: this parameter is a pointer to an array of *work_dim* unsigned values that describe the number of global work-items in *work_dim* dimensions that will execute the kernel function. The total number of global work-items is computed as the product of all the elements of the array pointed by *global_work_size*
  - local_work_size: this parameter is a pointer to an array of *work_dim* unsigned values that describe the number of work-items that make up a work-group (also referred to as the size of the work-group) that will execute the kernel specified by *kernel*. The total number of work-items in the work-group must be less than or equal to the

CL_DEVICE_MAX_WORK_GROUP_SIZE value specified in table of OpenCL Device Queries for clGetDeviceInfo. *local_work_size* can also be a NULL value in which case the OpenCL implementation will determine how to be break the global work-items into appropriate work-group instances.

- num_events_in_wait_list: specify the size of the array pointed by *event_wait_list*.
- event_wait_list: specify events that need to complete before this particular command can be executed. If *event_wait_list* is NULL, then this particular command does not wait on any event to complete.
- event: returns an event object that identifies this particular kernel execution instance. Event objects are unique and can be used to identify a particular kernel execution instance later on. If *event* is NULL, no event will be created for this kernel execution instance.

5 Development

We will now analyse and understand how the random graph algorithms we briefly described earlier can be parallelized.

5.1 Erdős-Rényi Model

The Erdős-Rényi model, as we said earlier is a totally random model, so the only parameters in this model are:

- N = the number of nodes in the graph.
- p = the probability that two nodes will form an edge between them.

```
#define DIM 100

#define P 0.3

#define SEED 349872935

int main(int argc, char *argv[]) {
    unsigned int N = DIM; // Number of nodes
    double p = P; // Probability of edge creation
    double r; // Random number
    int i, j; // Iterators
    int* graph = (int*)malloc(N*N*sizeof(int*));
    clock_t start, end;
    double cpu_time_used;

    // Set the random seed
    srand(SEED);
```

The variable "graph" as we can see is an array of int that has length of N^2. So we are using an adjacency matrix to represent our graph, and we decided to

use an array instead of a matrix because of the explicit memory allocation, that gives us more control on how the memory is used.

The algorithm then just iterates through every node couple (so we have a for loop that iterates N^2 times), sample a random number, checks if that random number is lower than the threshold we declared, and if it the case than an edge is created between the two nodes.

```
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        if (i == j) {
            graph[i*N+j] = 0;   // No self-loops
        }
        else {
            // Generate a random number between 0 and 1
            r = (double)rand() / RAND_MAX;
            // Check if an edge should be created
            graph[i*N+j] = (r < p) ? 1 : 0;
        }
    }
}
```

As we can see, the general idea of the algorithm is very simple. Here we are showing the implementation for a directed graph; if we would have liked to have an implementation for an undirected graph, the only difference would have been to se the value to 1 not only to the position (i, j), but to the position (j, i) as well.

It is easy to see that the whole for loop is highly parallelizable, and since we don't have any read access to the memory and we must do N^2 write operations in any case, we can easily manage this problem as a 2-D problem with N^2 threads (one for each couple of nodes).

We then proceed writing the following kernel code for OpenCL:

```
const char *kernelstring =
    "__kernel void ER(const unsigned int N, const double p, const
        uint seed, const uint base,"
    "                 __global int* G){"
    "   const int globalNode_i = get_global_id(0);"
    "   const int globalNode_j = get_global_id(1);"
    "   uint seed_x = (seed + globalNode_i + globalNode_j) * (
        globalNode_i + 1) << globalNode_i;"
    "   uint t = seed_x ^ (seed_x << 11);"
    "   uint result = base ^ (base >> 19) ^ (t ^ (t >> 8));"
    "   double rand = (double) (result%56891279) / 56891279;"
    "   if ((rand<p) && (globalNode_i!=globalNode_j)){"
    "       G[globalNode_i*N+globalNode_j] = 1;"
    "   }"
    "   else G[globalNode_i*N+globalNode_j] = 0;"
    "}";
```

As we said earlier we can define the kernel code as a string, but it has the same structure as any other function declared in C. We can see the modifier *__kernel* written before the type of the function (void) and then the name of the function.

The arguments for the kernel are:

1. N: the number of nodes of the graph.
2. p: the probability to create a new edge.
3. seed and base: these two values are necessary in order to create a working rng in OpenCL, since, at the best of our knowledge there is no random number generator defined natively in OpenCL C language.
4. G: this parameter with the *__global* modifier is the buffer we declared earlier in the host code, that refers to the adjacency matrix.

The first thing we do is to call the *get_global_id* function, specifying the dimension we are refering to (0 or 1 in this case since we are working with a 2-D problem). The two numbers we get, *globalNode_i* and *globalNode_j* will both be the identifiers of the work-group we are currently in and the indices of the nodes (just like i and j in the for loop we saw int the sequential algorithm).

We then proceed to obtain our random number, using a version we defined of the xorshift algorithm.

```
uint seed_x = (seed + globalNode_i + globalNode_j) *
    (globalNode_i + 1) << globalNode_i;
uint t = seed_x ^ (seed_x << 11);
uint result = base ^ (base >> 19) ^ (t ^ (t >> 8));
double rand = (double) (result%56891279) / 56891279;
```

In each workgroup we must make sure to obtain a different random number, and that's why we modify the seed using the global IDs. We then proceed to do some hash operations using xor and shifts, enough to obtain a number from which we cannot obtain the original seed and base. We then cast the random int we obtained to double, do a module operation for a user defined number, and divide by that same number. This way we will obtain a number in range $[0, 1[$.

In the end, exactly like the sequential ER algorithm, we compare the random number with our probability value, and if it's the lesser number, we will create the edge, otherwise we will not.

```
if ((rand<p) && (globalNode_i!=globalNode_j)){
    G[globalNode_i*N+globalNode_j] = 1;
}
else G[globalNode_i*N+globalNode_j] = 0;
```

It is important to specify that we decided to implement the ER random graph algorithm for a directed graph, instead of an undirected one (which is the most common version), because it is more efficient to parallelize. In an undirected ER random graph generator, we would not need N^2 loops to complete the algorithm, because this way we would explore each couple of nodes exactly two times. Below would be the most efficient way to implement the sequential algorithm for ER random graph generation.

```
for (i = 0; i < N; i++) {
    for (j = i+1; j < N; j++) {
        // Generate a random number between 0 and 1
        double r = (double)rand() / RAND_MAX;
        // Check if an edge should be created
        graph[i][j] = (r < PROB) ? 1 : 0;
        graph[j][i] = graph[i][j]; // Undirected graph, so mirror
            the edge
    }
}
```

It is not possible to parallelize efficiently this kind of behavior with an OpenCL kernel.

5.2 Watts-Strogatz Model

The Watts-Strogatz model, similarly to the ER model we need to generate a random number that will shape the randomness of the graph. In this model, however, we start with a given structure, that is a ring lattice, that we reshape randomly.

The parameters we need for this algorithm are:

- N = the number of nodes in the graph.
- p = the probability that an edge will be rewired.
- k = the initial number of edges for each node.

```
#define DIM 100

#define P 0.3

#define INIT_CONN 4

#define SEED 349872935

int main() {
    unsigned int N = DIM;           // Number of nodes
    double PROB = P;    // Probability of edge creation
    int K = INIT_CONN;
    double r;
    int i, j, l;
    int* G = (int*)malloc(N*N*sizeof(int*));
    clock_t start, end;
    double cpu_time_used;

    // Set the random seed
    srand(SEED);
```

One important detail to keep in mind is that k must always be an even number, because each node n must have the same number of edges with its $n+i \pmod N$ neighbors and its $n-i \pmod N$ neighbors, $\forall i \in [1, k/2]$.

And so, the algorithm is written this way:

```
for (i=0; i<N*N; i++) { G[i] = 0; }
for (i = 0; i < N; i++) {
    for (j = 1; j <= K/2; j++){
        r = (double)rand() / RAND_MAX;
        if (r<PROB) {
            do{
                l = (rand() % (N-K-2)) + (K/2) + 1;
            } while (G[i*N+((i+l)%N)] == 1);
            G[i*N+((i+l)%N)] = 1;
            G[((i+l)%N)*N+i] = 1;
        }
        else {
            G[i*N+((i+j)%N)] = 1;
            G[((i+j)%N)*N+i] = 1;
        }
    }
}
```

Basically, after initializing the whole adjacency matrix to 0, we loop through the nodes, and for every node in the graph we explore its next $k/2$ neighbors, and, based on the result of the rng we decide if we will create a shortcircuit in the graph or if we will connect the two nodes like we are expected to.

This apporach is sligthly more efficient than the one where we first create the lattice ring and then proceed to check for the generation of shortcircuits, but the result is identical, since, if we were to set $p = 1$ we would still get a lattice ring.

The computation complexity for this algorithm is not deterministic because of its random nature: once we are in the case where we have to create a shortcircuit we must select a new random node for the new edge, and doing so we can't connect two already connected edges; that's why we use a do-while loop to choose the node to which we will connect and that can take a variable amount of time. For this reason we wrote also a modified version of the algorithm were in the end the number of edges will not be exactly $N * (k/2)$ but it will be $|e| \leq N * (k/2)$.

```
for (i=0; i<N*N; i++) { G[i] = 0; }
for (i = 0; i < N; i++) {
    for (j = 1; j <= K/2; j++){
        r = (double)rand() / RAND_MAX;
        if (r<PROB) {
            l = (rand() % (N-K-2)) + (K/2) + 1;
            G[i*N+((i+l)%N)] = 1;
            G[((i+l)%N)*N+i] = 1;
        }
        else {
            G[i*N+((i+j)%N)] = 1;
            G[((i+j)%N)*N+i] = 1;
        }
    }
}
```

And our parallelized version of the algorithm will be based on this one, in order to avoid concurrential accesses to the memory. At first sight, this seems

like a 2-D problem, like the ER one, but actually, we can parallelize it in a 1-D way, having N workgroups instead of N^2, and then loop inside them for $k/2$ times, reducing sensibly the computation time.

```
const char *kernelstring =
    "__kernel void WS(const unsigned int N, const double p, const
        int k, const uint seed, const uint base,"
    "   __global int* G){"
    "   const int globalNode = get_global_id(0);"
    "   for (int i = 1; i <= k/2; i++){"
    "       uint seed_x = (seed + globalNode + i) * (globalNode +
        1) << globalNode;"
    "       uint t = seed_x ^ (seed_x << 11);"
    "       uint result = base ^ (base >> 19) ^ (t ^ (t >> 8));"
    "       double rand = (double) (result%56891279) / 56891279;"
    "       if ((rand<p)){"
    "           uint seed_x = (seed + globalNode + i) * (globalNode
         + 1) << globalNode;"
    "           uint t = seed_x ^ (seed_x << 11);"
    "           uint result = base ^ (base >> 19) ^ (t ^ (t >> 8));
        "
    "           int rand = (result%(N-k-2)) + (k/2) + 1;"
    "           G[globalNode*N+((globalNode+rand)%N)] = 1;"
    "           G[((globalNode+rand)%N)*N+globalNode] = 1;"
    "       }"
    "       else {"
    "           G[globalNode*N+((globalNode+i)%N)] = 1;"
    "           G[((globalNode+i)%N)*N+globalNode] = 1;"
    "       }"
    "   }"
    "}";
```

As we can see in this kernel we call the *get_global_id* function just one time, because the problem is a 1-D problem. In the kernel we use a for loop to iterate through the $k/2$ connections. Again here we use the xorshift algorithm for the rng, and we must use it a first time for the probability to create a shortcircuit, and a second time to find a new node to connect to.

5.3 Barabási-Albert Model

The BA model is once more a model that has a probabilistic behaviour, but it does not rely on a uniform distribution statistically speaking, so the approach is a bit more complicated. Each node will prefer to connect to the nodes in the network that already are well connected, and this is a behaviour commonly saw in the real life networks.

The parameters for the BA algorithm are:

- N: the number of nodes of the graph.
- M: the number of edges that each node will have when added to the graph.

From parameter M we also obtain the starting number of nodes in the graph ($M_0 > M$).

```
#define DIM 100

#define M 2

#define SEED 349872935

#define BASE 569872384

#define TS 10 // Threadblock sizes

int main(int argc, char *argv[]) {
    int i, j, k, m, a, r;
    unsigned int N = DIM; // Number of nodes
    int* deg = (int*)malloc(N*sizeof(int*)); // Array with the
        degrees of the nodes
    int* graph = (int*)malloc(N*N*sizeof(int*));
    unsigned int edges; // We just need the number of edges because
         we know that the sum of the degrees
    //in an undirected graph is 2|E|
    clock_t start, end;
    double cpu_time_used;

    // Set the random seed
    srand(SEED);
```

As we can see we need some additional variables to implement this algorithm:

- deg: this is an array of size N where we store the degree score of every node in the graph (the degree is the number of edges it is connected to).
- edges: this value stores the total number of edges in the network and this value is extremely important because in an undirected graph, the total degree of the network is $D_{Tot} = 2|E|$. And we need the total degree of the graph in order to compute the probabilities for the connection to a certain node, that, in literature is defined as $p_i = \frac{k_i}{\sum_j k_j}$, where k is the degree of the node.

```
for (i=0; i<N; i++) { deg[i] = 0; } // Initialize the degree array
for (i=0; i<N*N; i++) { graph[i] = 0; } // Initialize the matrix
edges = 0;

// Set the starting M+1 nodes with M starting edges
for (i = 1; i <= M; i++){
    graph[i] = 1;
    graph[i*N] = 1;
    deg[i] += 1;
    edges += 1;
}
deg[0] = M;
```

In the first phase of the algorithm we have $n > M$ nodes, specifically in our implementation we have $M + 1$ nodes, with M connections. As default we connect all the M nodes after the node 0 to said node 0, creating this way the first hub, and preferential attachment, of the graph.

After this first step we implemented the algorithm this way:

```
for (i = M+1; i < N; i++) { // For every node
    m = 0;
    while (deg[i] < M) { // For M times
        r = rand()%(edges*2); // We find our random value in a
            natural range of numbers instead of
        //[0,1]. This way we will need much less operations
        for (j = 0; j<N; j++) { // For every node we can connect to
            a = 0; // Initialize our probability
            for (k = 0; k<=j; k++) {
                a += deg[k]; //Compute the probability
            }
            if (r < a) {
                if (graph[i*N+j] == 0 ){
                    graph[i*N+j] = 1;
                    graph[j*N+i] = 1;
                    deg[i] += 1;
                    deg[j] += 1;
                    edges += 1;
                    m += 1;
                    break;
                }
            }
        }
    }
}
```

This algorithm requires us to update the probabilities to connect to a certain node after the creation of each edge; that's why for each node we must do M loops and inside each one of them we need to do N more loops to compute the new probability distribution.

As we can see, instead of generating a random number in range $[0, 1[$, we create a random number in range $[0, 2|E|[$, and we also compute the probability without normalizing it. This way we can avoid 2 floating point operations that would be executed every iteration of the inner loop structure.

Analyzing this algorithm it is easy to notice how it is an algorithm that is intrinsically sequential, because every edge creation is statistically dependant from the previous one, then it would be impossible to parallelize it. The best we could do would be to parallelize the random number generation, but as we saw already OpenCL is not very suitable for this and we would not have a boost in perfomance such to justify the OpenCL implementation.

5.4 Random Geometric Graph Model

This model is probably the one, from the ones we analyzed in this work that would most gain an advantage from its parallelization. We do not have any random behaviour in this model (we randomly generate points in space, but it is not directly related to the random graph algorithm), and we might parallelize both write and read operations, besides floating point operations (the distance calculation between the points).

The parameters for this algorithm are:

- N: the number of nodes in the graph.
- R: the max distance between two nodes in order to create an edge between them.

```
#define DIM 100

#define R 0.3

#define SEED 349872935

int main(int argc, char *argv[]) {
    int i, j;
    double r, x, y;
    double dist = R;
    unsigned int N = DIM; // Number of nodes
    double* X = (double*)malloc(N*sizeof(double*)); // Array with
        the nodes abscissa
    double* Y = (double*)malloc(N*sizeof(double*)); // Array with
        the nodes ordinate
    int* graph = (int*)malloc(N*N*sizeof(int*));
```

We need a seed too in this algorithm, because, as we said earlier, we have to generate the coordinates for the nodes. X and Y are the two arrays containing the 2-D coordinates for the nodes.

```
// Create random N points in a 2-D space
for (i=0; i<N; i++){
    X[i] = (double)rand()/RAND_MAX; // We get a random number in
        interval [0,1[
    Y[i] = (double)rand()/RAND_MAX;
}
```

This is how we generated the random coordinates (we chose the interval $[0, 1[$ because it is the densest numerical interval for the ALU), and it will be the same way we will generate the coordinates for the parallelized version of the algorithm.

```
for (i=0; i<N*N; i++) { graph[i] = 0; } // Initialize the matrix
for (i=0; i<N; i++){
    for (j=i+1; j<N; j++){
        // We prefer to use two more variables (x and y), instead
            of doing just
        //one operation ( r = sqrt(pow((X[i]-X[j]), 2) + pow((Y[i]-
            Y[j]), 2)) ) because this is less
        //computationally heavy
        x = X[i]-X[j]; // Distance along the x axis
        y = Y[i]-Y[j]; // Distance along the y axis
        r = x*x + y*y;
        r = sqrt(r);
        if (r<dist){ // If the distance between the two nodes is
            less than R then we will connect them
            graph[i*N+j] = 1;
            graph[j*N+i] = 1;
        }
    }
}
```

The general idea of the algorithm is pretty simple: we compute the euclidean distance between the nodes and then compare it with our R. If the distance is less then R, then we create an edge between the nodes.

The most naive and easy way to parallelize it is to create N^2 workgroups that explore every nodes couple (there are sveral limitations to this implementation that we will discuss later). So we can write the kernel object this way.

```
const char *kernelstring =
    "__kernel void RGG(const unsigned int N, const double dist,"
    "   const __global double* X, const __global double* Y,"
    "   __global int* G){"
    "   const int globalNode_i = get_global_id(0);"
    "   const int globalNode_j = get_global_id(1);"
    "   if (globalNode_i != globalNode_j) {"
    "       double x = X[globalNode_i] - X[globalNode_j];"
    "       double y = Y[globalNode_i] - Y[globalNode_j];"
    "       double r = x*x + y*y;"
    "       r = sqrt(r);"
    "       if (r < dist) {"
    "           G[globalNode_i*N+globalNode_j] = 1;"
    "           G[globalNode_j*N+globalNode_i] = 1;"
    "       }"
    "   }"
    "}";
```

With the *get_global_id* function we get the nodes on wich we will work in the workgroup, then, if they are not equal (we don't want self loops), we compute their euclidean distance and, after checking if it is less then R, we create the edge.

This approach has a good level of optimization, but we can see some problems about it:

1. We are iterating two times through each node couple.
2. We are executing way too many read operations from the GPU memory, making each workgroup way more slow than it needs to be.

We could probably do better with another approach within OpenCL or using another, more optimized paradigm (like CUDA).

6 System specifics

The hardware specifics on which we ran our experiments are:

- RAM: 32 GB memory DDR5 6200MHz
- CPU: Intel Core i5 13600KF (3.5 GHz / 5.1 GHz)
- GPU: NVIDIA GeForce RTX 4070 Ti (12 GB GDDR6X - 2310MHz / 2610MHz)

The OS on which we developed and ran the code is:

- Ubuntu 22.04.3 LTS 64-bit

7 Results and analysis

Down below we can see the results of our experiments; the experiments have all been run 1000 times, and the times here presented are the mean of those runs.

Graph size	Type	Mean ER random graph generation times (s)				
		p=0.0	p=0.25	p=0.5	p=0.75	p=1.0
10 nodes	Sequential	9.0599e-07	9.1299e-07	9.2500e-07	9.1500e-07	1.2440e-06
	Parallel	5.0559e-06	5.7489e-06	6.1939e-06	6.2319e-06	6.0120e-06
100 nodes	Sequential	9.8419e-05	9.8585e-05	9.8447e-05	9.8461e-05	9.8590e-05
	Parallel	1.6815e-05	1.7218e-05	2.3836e-05	2.4379e-05	1.7968e-05
1000 nodes	Sequential	1.0166e-02	1.0157e-02	1.0157e-02	1.0167e-02	1.0154e-02
	Parallel	1.0821e-03	1.1289e-03	1.0877e-03	1.1264e-03	1.1022e-03

Table 1: Times of random graph generation for the ER algorithm

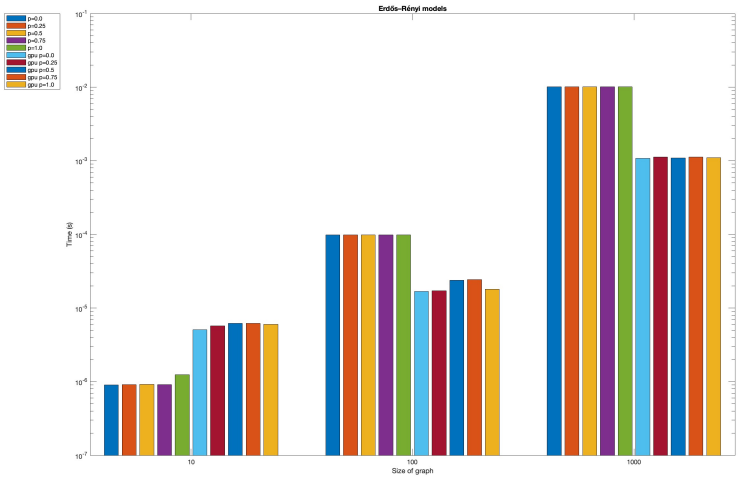

Figure 1: Times for generation of a random graph with the ER model

k	Type	Mean WS random graph generation times for size 10 (s)				
		p=0.0	p=0.25	p=0.5	p=0.75	p=1.0
2	Sequential	2.3599e-07	2.7200e-07	2.5800e-07	2.9400e-07	3.3200e-07
	Parallel	5.8250e-06	5.1080e-06	6.0549e-06	5.9129e-06	5.1710e-06
4	Sequential	2.7399e-07	3.6900e-07	4.2099e-07	4.8599e-07	4.9299e-07
	Parallel	5.2909e-06	5.3970e-06	6.4969e-06	6.2769e-06	5.4079e-06
6	Sequential	3.9700e-07	5.0900e-07	6.3100e-07	6.3100e-07	8.4500e-07
	Parallel	5.6150e-06	6.3270e-06	6.6980e-06	5.9599e-06	1.1776e-05

Table 2: Times of random graph generation of size 10 for the WS algorithm

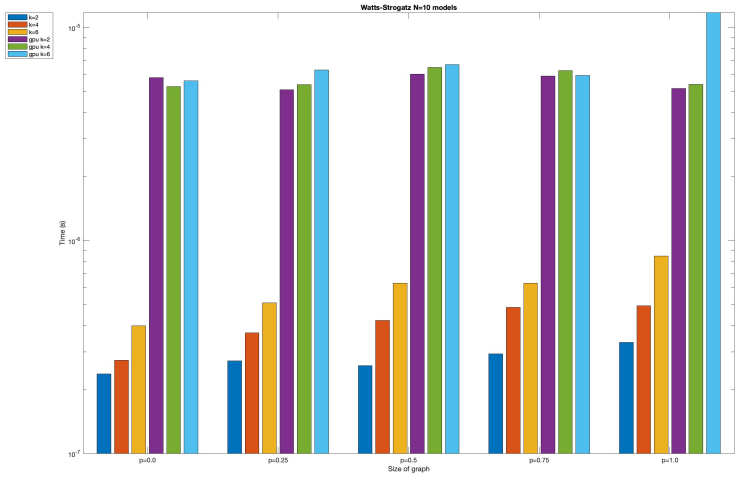

Figure 2: Times for generation of a random graph with the WS model (graph size = 10)

k	Type	Mean WS random graph generation times for size 100 (s)				
		p=0.0	p=0.25	p=0.5	p=0.75	p=1.0
2	Sequential	8.1290e-06	1.1407e-05	8.0050e-06	1.1844e-05	1.0751e-05
	Parallel	7.0380e-06	5.8659e-06	7.4170e-06	5.1590e-06	5.1819e-06
4	Sequential	7.9710e-06	9.1730e-06	9.1710e-06	1.4361e-05	1.2217e-05
	Parallel	5.4060e-06	6.5799e-06	7.8619e-06	5.5099e-06	7.5549e-06
6	Sequential	1.5258e-05	1.4800e-05	1.5965e-05	1.6379e-05	1.3159e-05
	Parallel	6.3899e-06	5.7869e-06	9.2949e-06	6.3090e-06	6.5039e-06

Table 3: Times of random graph generation of size 100 for the WS algorithm

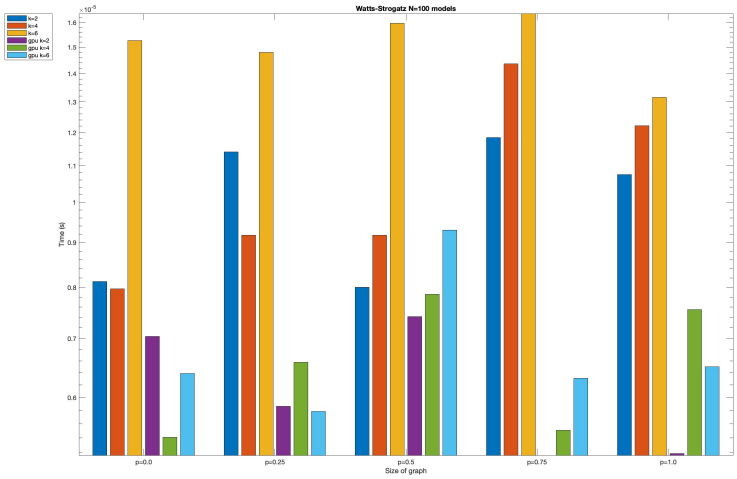

Figure 3: Times for generation of a random graph with the WS model (graph size = 100)

k	Type	Mean WS random graph generation times for size 1000 (s)				
		p=0.0	p=0.25	p=0.5	p=0.75	p=1.0
2	Sequential	8.5567e-04	8.6808e-04	8.2603e-04	8.2840e-04	8.7826e-04
	Parallel	7.7850e-06	7.2760e-06	7.5220e-06	8.3159e-06	7.9590e-06
4	Sequential	8.4732e-04	9.3006e-04	8.8170e-04	9.3210e-04	8.5639e-04
	Parallel	8.2900e-06	1.0778e-05	9.0240e-06	8.8009e-06	9.0489e-06
6	Sequential	8.6740e-04	1.0109e-03	8.4060e-04	8.6691e-04	9.5498e-04
	Parallel	8.6189e-06	8.5099e-06	9.8530e-06	1.2326e-05	1.3004e-05

Table 4: Times of random graph generation of size 1000 for the WS algorithm

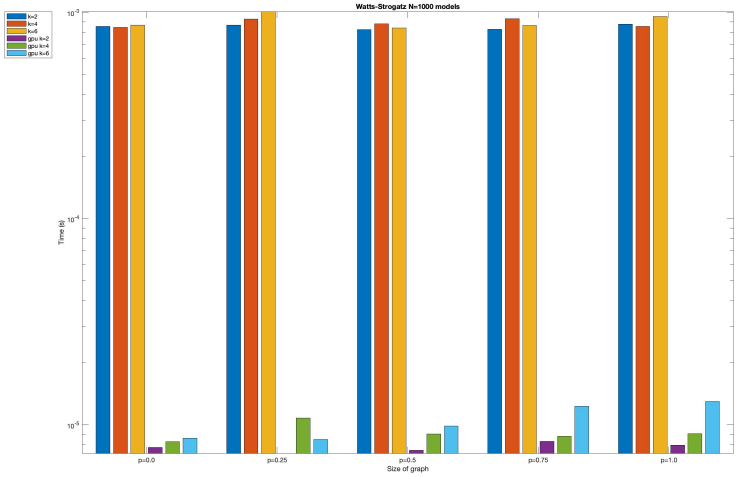

Figure 4: Times for generation of a random graph with the WS model (graph size = 1000)

Graph size	Type	Mean RGG random graph generation times (s)				
		r=0.0	r=0.25	r=0.5	r=0.75	r=1.0
10 nodes	Sequential	2.6200e-07	3.0400e-07	3.0600e-07	3.3399e-07	3.3700e-07
	Parallel	1.4087e-05	6.6610e-06	5.9139e-06	6.6969e-06	6.7710e-06
100 nodes	Sequential	1.6940e-05	2.4759e-05	2.9933e-05	2.3700e-05	2.2212e-05
	Parallel	2.2761e-05	2.1811e-05	2.1145e-05	2.3099e-05	2.2315e-05
1000 nodes	Sequential	1.8851e-03	3.7473e-03	5.3220e-03	3.8771e-03	2.4333e-03
	Parallel	1.4671e-03	1.5237e-03	1.4779e-03	1.5075e-03	1.4681e-03

Table 5: Times of random graph generation for the RGG algorithm

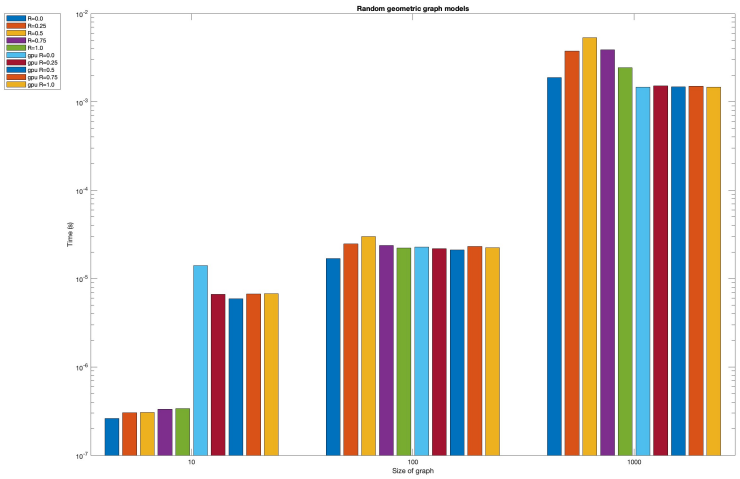

Figure 5: Times for generation of a random graph with the RGG model

As we can see fropm the results shown above we can make some considerations regarding the parallelization efficiency at lower graph dimensions. In every plot we can see how, for small graphs, the parallel solution is slower than the sequential one; this is probably due to the slower memory access on the gpu as compared to the cpu. Then, considering this difference in memory speed, the parallelization can't give us a speed-up greater than 1, but in the worst case we presented (RGG for 10 nodes and $r = 0.0$) it reaches the value of 0.018598708028678926.

Yet, we can also see that as the graph size grows, the parallel solution becomes faster than the sequential one, givinge us, in opur best case (WS for 1000 nodes, $k = 2$ and $p = 0.25$), a speed-up equal to 119.30827377680043. However we can see that not all parallel solutions are equally as good. Especially in the RGG algorithm case we can see that the advantage of parallelization is almost negligible.

8 Final thoughts

Regarding the idea of parallelization for random graph algorithms, we can infer that it is not a convenient practice: most of the time when implementing an algorithm for the generation of a ranodm graph we are interested in models with peculiar characteristics, that are very computationally heavy to implement, and that most of the time, like in the Barabási-Albert model we saw, they depend on their own previous state, making the process of task or even data parallelization very challenging. Of course in this work we merely scratched the tip of the iceberg, focusing just on OpenCL, without exploring other solutions like CUDA

or a possible distributed solution, so we have just a narrow vision of the whole problem.

As a future devolpment for this kind of study might be interesting to compare both different parallel OpenCL solutions and solutions obtained with different tools.