Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Generating Integral Graphs Using PRACE Research Infrastructure

Krzysztof T. Zwierzyński*[a],

[a] Poznan Supercomputing and Networking Center,
ul. Z. Noskowskiego 12/14, 61-704 Poznan, Poland

Abstract

In this white paper we report the work that was done on the problem of generation combinatorial structures with some rare invariant properties. These combinatorial structures are connected integral graphs. All (588) of such graphs of order $1 \le n \le 12$ are known. The main goal of this work was to reduce the time of generation by distributing graph generators over hosts in PRACE-RI, and to reduce the time of sieving integral graphs by applying eigenvalue calculation in GPGPU device using the OpenCL technique. This work is also a study of how to minimize the overhead connected with using OpenCL kernels.

Introduction

A simple graph $G = G(V, E)$ is called *integral* if all eigenvalues $Sp(G) = \{\lambda_1, \lambda_2, ..., \lambda_n\}$ of its adjacency matrix are integral. Every complete graph of order $n = |V|$ is integral with the spectrum $Sp(K_n) = \{n - 1, (-1)^{(n-1)}\}$. If the maximum degree Δ of graph G is bounded, then the class of connected integral graphs with this bound is finite [1]. For $\Delta = 1$ there is only one such a graph K_2. For $\Delta = 2$ the only such graphs are: $K_3 = C_3$, C_4 and C_6 (shown in Figure 1). The numbers $c_1(n)$ of connected integral graphs ($1 \le n \le 12$) are shown in Table 1 [8].

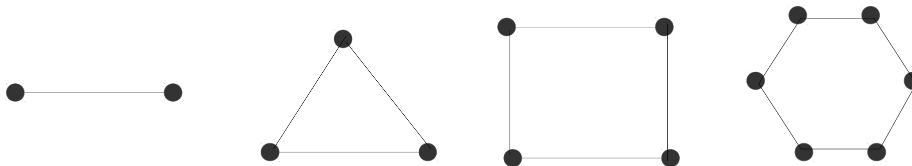

Figure 1: The example of small integral graphs and their spectrum:
$Sp(K_2) = \{1, -1\}$, $Sp(K_3) = \{2, (-1)^{(2)}\}$, $Sp(C_4) = \{2, 0^{(2)}, -2\}$, $Sp(C_6) = \{2, 1^{(2)}, (-1)^{(2)}, -2\}$

Integral graphs can have possible applications in quantum physics as a model of perfect state transfer (PST) [7]. It has been proven that circulant graphs which are integral have a property PST (i.e. graphs: K_2, C_4 and C_6). But the question which other classes of graphs can have this property still remains open. Good candidates for this classes are integral graphs.

There is the ***geng*** algorithm written by B.D. McKay [5] which solves the problem of generation of non-isomorphic graphs of a given order n. The process of generation can be split into many (but limited by some constant) independent subtasks, and run parallel in the grid. This gives an opportunity to reduce the computation time by the factor of nodes m. But, there are two main disadvantages of this solution. The first, a direct partition into m parts gives us a different runtime of ***geng*** for each subtask, and this provides to the bad load balancing between nodes. The second, each partition introduces some overhead connected with using a queuing system and input and output maintenance. To improve the load balancing we can increase the number of subtask, but this also introduces some extra overhead. It is non-trivial to find the compromise in the number of subtask due to the different time of its executions.

Table 1: The number of connected integral graphs $c_1(n)$, $1 \le n \le 12$

n	1	2	3	4	5	6	7	8	9	10	11	12	Sum
$c_1(n)$	1	1	1	2	3	6	7	22	24	83	113	325	588

* Corresponding author. E-mail address: zwierzak@man.poznan.pl

If we are interested in generating only regular graphs, then we can use the algorithm ***genreg*** written by M. Meringer [6]. For generating only the random set of graphs of given order *n* the algorithm ***genrang*** by B.D. McKay can be used.

Description of combinatorial problem

In the case of dealing with the generation of combinatorial objects (i.e. graphs of some order *n*) and their selection due to some criterion for each combinatorial algorithms, there is always a limit of its applicability. This is particularly evident in the case of problems for which the size of the search space grows exponentially with *n*. Then, even if we have an algorithm that solves the problem in an acceptable time for the selected *n*, the solution of the problem for *n* greater only by one may require additional treatments in programming, computing and storage resources. Generating connected integral graphs belongs to such class a problems, and size of combinatorial space is growing as 2^(binomial{*n*}{2}) / *n*!.

This can be done in an acceptable time for the calculation of *n* and is highly dependent on the method of combinatorial search space and its size. If the method let us divides the space into separate subclasses, then the appropriate calculation speed can be achieved by using a sufficiently large number of computing machines (for example, if the number of nodes is equal to the number of subclasses). If we can obtain this acceleration depends on the overhead associated with the time of distribution of tasks to compute nodes, the time required to collect the partial results, and whether the duration of the calculations for the subclasses is relatively homogeneous. The total calculation time is also influenced by the number of available machines and their current and planned load. In this embodiment there is also a problem with the reliability of the calculation, since as the number of computing nodes grows the probability of a failure or communication problems increases considerably with the number of elements. If any part of the calculation for any reason will not be achieved then it is necessary to repeat it, which adversely affects the computation time.

For the problem of generating connected integral graphs, which is part of the calculation scheme presented previously, it is required to prepare an algorithm that breaks down the task into subclasses, will oversee the execution of subtasks and aggregated results. Such an algorithm can be achieved by the shell script wrapper which delegates the task to the scheduling system. What may be controlled in such a case is the selection of the number of classes which are divided into the entire task. Dividing the task in to subclasses will usually reduce the time of execution, however it increases the time overhead associated with their service. Usually, however, the strategy division of tasks for more than a computing machine, leads to a better load balancing of nodes. This solution for the tasks that are shorter minimize also the risk for some reasons, this task will not be performed (for example, if a node fails or is planned to off).

The first step of calculation is getting statistical information i.e. minimal, maximal, and average time of execution. For the algorithm ***geng*** also the number of output graphs per subtask can be important. If we predict that computation can take many hours then we can devide the hole process of generation, run it into parts divided into cluster. When we extend the problem to finding some graphs with given properties (i.e. with integral spectrum) it can be done in two ways: generate graphs, save them to file, and then make some calculations on this data. The second way is checking properties before saving graphs. Adding sieving directly into the generation algorithm we increase the time per graph, but if many of them are eliminated, then we improve the necessary time for saving data.

Using OpenCL technique in combinatorial problems

To take full advantage of the possibilities offered by technology OpenCL usually it is necessary to write a dedicated code for kernels. But it is a very time consuming and requires a detailed knowledge of the specific GPGPU device on which the calculations are performed. The detection of errors in the OpenCL code is also difficult because it requires special techniques, especially when there is a synchronization and communication between threads or only part of threads working wrong. Usually we already have a working version of some code, designed and tested for the CPU, which carries out the task of computing in a single-threaded and sequential. Then, if the code does not contain parts unacceptable by the OpenCL design, we can only adjust and move calculations to GPGPU. Such a procedure, however, does not lead to the acceleration of the calculation for two main reasons: first, a transfer data to/from GPGPU device and GPGPU computing often introduces some overhead and second a single thread in GPGPU device is usually slower than the corresponding thread in the CPU. When we use OpenCL, the kernel can be also prepared and run by CPU. The decision which architecture of calculation to choose (CPU, GPGPU or both) can be controled by autotunnig method.

GPGPU force calculation stems from the possibility of multiple threads that are independent. If we generate integral graphs the ideal solution would be to transfer a generation algorithm to a range of GPGPU. Such a solution is possible, but there is a problem with the transmission and preservation of the results of the calculations, since the number of solutions is not known. It also would require modification of the code ***geng***, that it can be run in GPGPU device. The case is complicated by the fact that the tested version of OpenCL cannot directly save the calculation results to a file or output stream.

The approved and tested solution space separated generates and calculates the spectrum of graph. This results from the analysis show that the eigenvalues calculation time is longer than time of the graph generation. The calculation of the spectrum is transferred to GPGPU. Generated graphs are stored into buffer, and when it is full, the information is transmitted to the GPGPU, and there is a spectrum calculated for each graph in the buffer in concurrent, and then verified that the graph is integral. Not to modify the software code used ***geng*** pipelining. So we run a generator in the creek and in with the program selector to the parameters accordingly.

Important for the generation is a process for storing information on a graph. The methods of calculation of the spectrum of graphs is not only restricted to zero-one, but in general, to real symmetric matrices. This requires a transformation from a binary to a more memory consuming representation. Due to the fact that the time of data transfer from the CPU to the GPGPU is significant, it was decided to sub-binary. Then the decoding graph is implemented in the GPGPU. This solution can also speed up the calculation because of the speed in accessing global memory as an array version can be stored locally.

OpenCL overhead reducing tips

When we would use OpenCL technique for calculation we need to prepare code of kernel, compile it, and distribute it to all devices (GPGPU or CPU). In case only one type of devices are used and hardware configuration is not changing during the process of calculation then we can make some work in advance, to not repeat the same initial job for every node, for any program execution. There are two possible path of using kernel. The most flexible case: we have a kernel code written in C99, we compile it to PTX format before the execution binaries are build for every devices. Such a solution gives us the worst overhead, because we need to load compilator into the system and run it, and it is a very resource consuming operation. The one way to avoid this is using a precompiled version of the kernel source. But even in that case the OpenCL compilator is loaded into memory and used for PTX input to produce binaries. The other case is when we already have binaries for a given GPGPU device, and we just load it. In such a case using compilation is not needed. The only overhead here is connected with I/O operation when we load binaries stored into files. To reduce the size of kernel binaries some selection of the code can be used before the compilation, then we remove parts that will not be used. The time of transfer of data from the CPU memory to the GPGPU device memory is also significant. Before data transfer we can encode (zip) data and decode (unzip) data into GPGPU. This trick take some extra time of calculation, but if speed of transmission is slow it can be useful.

The next step in the optimization of this process is the preparation of the calculation of the minimum code and binaries for the kernel GPGPU. Introducing the fly compilation of the kernels, guarantee maximum portability solutions for all machines that we use. However, where it is a collection of machines of the same type – is to prepare the code for only one node and pass it to other computing nodes. We should avoid building the kernel code fragments that are not used, if binaries are stored on disk.

Sieving integral graphs

In the case of the method used for calculating spectrum, it is possible to count eigenvalues independent, because the algorithm first converts an adjacency graph matrix into a tridiagonal matrix and performs the calculation for the subsequent eigenvalue. At this point it would be possible to further reduce the time of execution. The problem can also be considered to use some number of tricks due to the nature of this combinatorial problem. For example, there is no point in calculating the last eigenvalue of the spectrum as we can get it from the sum of previous calculations also continued when one of the components is already established as an non-integral does not change anything. We can also minimize the space by half analyzing the graph and its complement at once. Here we gain on the transfer of data between the CPU and the GPGPU device.

The main idea is to use GPGPU in such a way, that they calculate spectra of many small matrices at the same time (see Figure 2). It is the simplest way to reduce time complexity. Also small number of changes in the code of the numerical method is needed. But this method changes the way of processing graphs - we need first to collect some graphs in memory and then use sieving method for this group. The size of the group can be choose arbitrary. The code of a numerical method for calculating eigenvalues is based on [4].

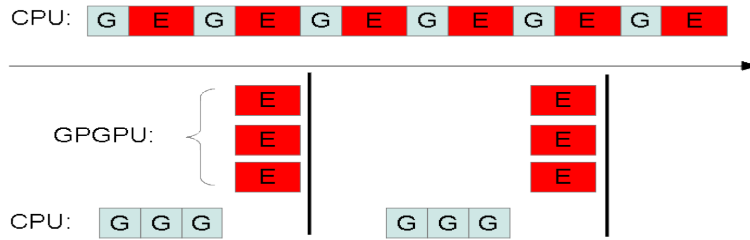

Figure 2: The time of graph generation (G) and eigenvalue calculation and sieving (E) in sequential and in parallel

In the next section some tests performed on cluster PC based on AMD Interlagos architecture are shown. Each node of the cluster contains two CPU (24 cores): AMD Opteron™ 6234 2.4 GHz and two GPGPU: NVIDIA TeslaM2050.

```
#!/bin/bash
n1=4
n2=16
while [ $n1 -le $n2 ]; do
 echo n1=$n1
 ./b1.sh $n1
 let n1=n1+1
done
```

Listing 1: The meta bash script (file: mbat.sh) used for tests

```
#!/bin/bash
# (-P2 the edge probability 1/2 = 0.5)
# (-g for graph6 format, compatible with geng)
# -S0 the deterministic seed of random generator
rep=100000
n=$1
time ../nauty24r2/genrang -P2 -g $n $rep -S0 | wc -l
```

Listing 2: The bash script (file: b1.sh) used for tests ("*Time of generation*")

```
#!/bin/bash
n=$1
time ../nauty24r2/geng $n -c | ./meigenCPU $n 1 | wc -l
```

Listing 3: The bash script (file: b2.sh) used for tests ("*CPU time*(*n*)")

```
#!/bin/bash
n=$1
block=448
time ../nauty24r2/geng $n -c | ./meigenGPU $n $block | wc -l
```

Listing 4: The bash script (file: b3.sh) used for tests ("*GPGPU time*(*n, block_size*)")

Test 1. The random generator ***genreg*** of graphs is used. The sample of graphs of size rep = 100 000 is generated for each n = 4, 5, ..., 16. We compare the execution time for two version of algorithm that sieves integral graphs on the single host: CPU and GPGPU. The time of execution is shown in the Table 2. For the calculation on GPGPU graphs where group into the block of size = 448, 896, 1792 (the multiplicity of the number of GPGPU threads). This experiment shows that if size of work is not sufficiently large, then the solution with CPU wins. But when the order of the graphs is bigger, then it is better to use the GPGPU version.

Table 2: The time of execution algorithm for sieving graphs of order $4 \le n \le 16$ (*rep* = 100 000)

n	4	5	6	7	8	9	10	11	12	13	14	15	16
Time of generation [s]	0.033	0.049	0.067	0.091	0.117	0.147	0.184	0.221	0.269	0.313	0.360	0.411	0.463
CPU time(*n*) [s]	0.745	0.922	1.098	1.329	1.596	1.873	2.215	2.585	2.984	3.420	3.898	4.438	4.997
GPGPU time(*n,448*) [s]	1.782	1.877	1.860	1.745	1.600	1.567	1.584	1.620	1.663	1.713	1.763	1.802	1.872
GPGPU time(*n,896*) [s]	1.512	1.595	1.597	1.543	1.471	1.434	1.437	1.477	1.506	1.563	1.602	1.658	1.712
GPGPU time(*n,1792*) [s]	1.396	1.441	1.449	1.428	1.395	1.410	1.429	1.473	1.516	1.565	1.615	1.661	1.714

Test 2. If we use the bigger sample of random graphs *rep* = 1 000 000 (see Table 3) then the solution with GPCPU is always better and speedup is close to 10x for *n* = 16.

Table 3: The time of execution algorithm for sieving graphs of order $4 \le n \le 16$ (*rep* = 1 000 000)

n	4	5	6	7	8	9	10	11	12	13	14	15	16
CPU time(*n*) [s]	7.474	8.793	10.34	32.72	14.78	17.47	20.42	44.75	29.91	34.27	38.99	44.17	50.19
GPGPU time(*n,1792*) [s]	2.629	2.985	3.049	2.849	2.548	2.686	3.062	3.430	3.883	4.340	4.900	5.328	5.845

Test 3. The generator ***`geng`*** of graphs and ***`meigenCPU`*** and ***`meigenGPU`*** for sieving graphs is used. All connected graphs of a given order are generated. This experiments shows that the overhead connected with using GPGPU is close to 1.3 [s]. Thus, we can have the speedup only when the number of the graphs to sieve is sufficiently large (i.e. for *n* = 10 the speedup is 15x, see Table 4).

Table 4: The time of execution algorithm for generation and sieving graphs of order $4 \le n \le 10$ (*n* = 11 only for GPGPU)

n	4	5	6	7	8	9	10	11
$g_1(n)$	6	21	112	853	11117	261 080	11 716 571	1 006 700 565
$c_1(n)$	2	3	6	7	22	24	83	113
CPU time(*n*) [s]	0.027	0.037	0.038	0.039	0.189	4.526	3m 57.027s	------
GPGPU time(*n,1792*) [s]	1.267	1.269	1.266	1.273	1.281	1.570	15.939	30m59.839s

Case *n* = 10. The Listing 5 describes the partition of job into *mod* subtasks. The size of search space: 11 716 571 graphs. The execution time 36 [s]. (`time ../nauty24r2/geng $n -c $res/$mod | ./meigenGPU $n 1792`)

```
#!/bin/bash
n=10
res=0
mod=10
while [ $res -lt $mod ]; do
 ./b3.sh $n $res $mod
 let res=res+1
done
```

Listing 5: The meta bash script used for tests

On Listing 6 the method of using the SLURM job scheduling system is presented. The total time of calculations can be obtain as a difference between the last time of modification of **.out* file and the time of creation of *init.txt* file. Using this approach and *mod* = 10, we reduce the time to 3 [s].

```
#!/bin/bash
n=$1
res=0
mod=$2
echo "begin" > n$n-$mod-init.txt
while [ $res -lt $mod ]; do
 sbatch -n1 --output="n$n($res-$mod)-%j.out" ./b3.sh $n $res $mod
 let res=res+1
done
echo "end" > n$n-$mod-finish.txt
```

Listing 6: The bash script used for tests

Case *n* = 11. There are 113 integral connected graphs of order $n = 11$. The calculation on single node with GPGPU support takes 30 minutes. The size of the search space: 1 006 700 565 connected graphs. Using the approach showed on Listing 6 and *mod* = 10, for $n = 11$ we reduce the time to 3 minutes. Using more subtask results in error during getting access to GPGPU device to some of them. Thus, all this subtask exit with error should be identify and run again. This problem can be also considered as a subject to autotunnig algorithm (the strategy can be suspending to send next jobs if some subtask ends with GPGPU errors).

Case *n* = 12. There are 325 integral connected graphs of order $n = 12$. The size of the search space: 164 059 830 476 connected graphs. If we use $n = 12$ and *mod* = 100 then the time for a unit of calculation takes approximatively 40÷80 minutes.

Case *n* = 13. The size of the search space: 50 335 907 869 219 connected graphs. The exact number of integral connected graphs of order $n = 13$ is not know. It is grather or equal to 526. If we use $n = 13$ and *mod* = 500 000 then the time for a unit of calculation takes approximatively 2÷6 minutes.

Test 4. The generator **`geng`** of graphs **`meigenGPU`** for sieving graphs is used. All connected graphs of a given order and bounded degree $\Delta \leq 4$ (see Table 5) was generated ($block = 1792).
(`#time ../nauty24r2/geng $n -c -D4 | ./meigenGPU $n $block | wc -l`).

Table 5: The time of execution algorithm for generation and sieving graphs of order $5 \leq n \leq 13$

n	5	6	7	8	9	10	11	12	13
$g_1(n,\Delta{=}4)$	21	78	353	1 991	12 207	89 402	739 335	6 800 637	68 531 618
$c_1(n,\Delta{=}4)$	3	5	4	6	5	10	3	14	1
GPGPU time(*n,1792*) [s]	1.3	1.3	1.3	1.3	1.3	1.6	4.2	32.1	300.3

Overhead and autotunnig

In the case n increases, the acceleration associated with the use GPGPU may at some point be locked. This limit is connected with the maximum size of the data buffer. Then we need to reduce the number of graphs simultaneously provided to calculate the spectrum, and some of threads remain idle. When the number of idle threads is large, then we can duplicate the calculations for the graph, and the separate calculation of eigenvalues of the spectrum. If we are single users of the system we have the possibility to plan and predict the expected time of computation. When we use job scheduling system (i.e. SLURM) we have some extra constrains on the process of the computation.

Autotunnig in this solution on the lowest level is an automatic choice of size of buffer of graphs (block size). It should be adapted to the number of threads that can run simultaneously and GPGPU device memory limitations. A possible improvement is the doubling of the data buffer, so that when the calculation of spectrum

is carried out for graph from one buffer, it is possible to generate graphs to the second buffer. Such behavior should be prepared as a ready-made patterns of behavior that can be used in the system, if necessary, provided that the implementation of the new scheme will not be possible to rectify overhead introduced a profit from their use.

The problem of autotunnig is how to adopt code and workflow to the current configuration of available hardware, software and load. Also it is very important to recognize changes in topology, hardware, version of libraries. Flexible code with many possible execution path can be big, this is not so important if we use it locally, but if the same code is transfered to another nodes this also introduce some time overhead. Another problem can appear when GPGPU devices are not heterogeneous in the grid, then we need to manage many binaries. The control on the calculation should be sensitive on error detection and time of job execution.

Conclusions

As a conclusion, the experiments indicates that it is necessary to support the design of algorithms that allow for a self-steering algorithm and its adaptation to the hardware capabilities. As far as possible, the algorithm should be designed as flow control, which can be isolated in blocks, which can be replaced by equivalent codes, in particular the OpenCL kernel code.

In particular, the programmer should specify the method of distribution of tasks and aggregation of results. In an autotunnig strategy is possible and the calculations are carried out for too long, the calculation of concurrent should be replaced or distributed solution, if necessary. Mainly this is a situation where the parameter n is growing then even small improvements in the code can lead to significant savings of the time of combinatorial structures generation.

For each calculation scheme computational strategy can be different. At this point would be necessary to add the ability to learn on the basis of benchmarks, or simulation calculations for different hardware configurations, even those that are not currently available. Another task is designing the scalable computation and evaluation when the slowdown associated with the distribution and division calculations are acceptable.

We should also keep in mind the purpose of the calculation, sometimes it is not necessary to find all solutions, and getting even some solutions can be interesting. For integral graphs the search space can also be splitted into bipartite graphs and non-bipartite graph, regular and non-regular. Those graphs whose automorphism group is one and those that has a matrix representation with the biaxial symmetry. To obtain the graph of a certain class also some graph operations can be used. There are also known some families of integral graphs. The division of the search space and the information about already known objects can be also used to reduce the time. If we already know some integral graphs we can find similar objects both for the same n and different, combining, for example, smaller graphs or using a local exchange edges between vertices. There it is also possible to use in parallel some heuristic search methods (i.e. the evolutionary technique) to obtain almost all or some sample set of integral graphs of a given order [2, 3]. On the other hand, an excessive adjustment of the code leads to the fact that it is difficult to use it for other problems.

In the future it would be interesting to move also the graph generation algorithm to GPGPU. This will reduce the overhead introduced by data transfer between the generation algorithm and the sieving algorithm. We can consider the case when in one workgroup one of the thread generates graphs, and other calculates the spectrum, sieving integral graphs, and save them to output.

This problem of generation connected integral graphs can be considered as a benchmark for testing the computational power of a grid. The number of solutions is not so big, thus there is no problem with the output data storage. For $n = 13$ and the current stage of technology the time of generation on a single computer node can take approximately 2 years. For that reason only well tested and tuned methods can complete this task in parallel in a reasonable time even for future exascale systems.

Acknowledgements

This work was financially supported by PRACE – Second Implementation Phase Project, funded in part by the 7th EU Framework Programme under grant agreement no. 283493.

References

[1] K.T. Balińska, D. Cvetković, Z. Radosavljević, S. Simić, D. Stevanović, A survey on integral graphs, Univ. Beograd. Publ. Elektrotehn. Fak. Ser. Mat. 13, 2002, pp. 42–65.

[2] K.T. Balińska, M. Kupczyk, K.T. Zwierzyński, Algorithms for generating integral graphs, Miscellanea Algebraicae, WZiA AŚ Kielce Rok 5, No.1 2001, pp. 7–21.

[3] K.T. Balińska, M.Kupczyk, S.K. Simić, K.T. Zwierzyński, On generating all integral graphs on 12 vertices, Computer Science Center Report No. 482, The Technical University of Poznań, 2001, pp. 1–36.

[4] A. Marciniak, D. Gregulec, J. Kaczmarek, Podstawowe procedury numeryczne w języku Turbo Pascal, Wydawnictwo Nakom, 2000.

[5] B.D. McKay, http://cs.anu.edu.au/~bdm/nauty/

[6] M. Meringer, Fast Generation of Regular Graphs and Construction of Cages, Journal of Graph Theory 30, 1999, pp.137–146.

[7] M.D. Petković, M. Bašić, Further results on the perfect state transfer in integral circulant graphs, Journal Computers & Mathematics with Applications archive, Volume 61 Issue 2, January, 2011, pp. 300–312.

[8] G.F. Royle, http://oeis.org/A064731