System and Device Programming Projects

Device Programming Part (Prof. Quer)

A.Y. 2020-2021

Projects will undergo the following rules and restrictions:

- Projects are optional and the exam is passed only if the written test (both parts) is passed.
- Each student can take a project only once in his curricula path. The project must be selected before taking any written examination tests. Each project must be selected by groups of 2 or (at most) 3 students.
- Projects will be assigned to group of students on a first-come-first-served basis.
- All projects taken in 2020-2021 must be completed and delivered before or at the latest during the examination session of February 2022. The beginning of a course in the A.Y. 2021-2022 will automatically erase all pending projects.
- The evaluation of each project will follow a short presentation given by the group of candidates during one examination session. To enroll for the presentation, the project must be uploaded on the portal web page before the written test of that examination section. The student must enroll for the exam in that session (even if it only discussing the project and not taking any written test).
- Marks may differ for different students within the same group, depending on the effort they put into the project and on the final oral presentation. Projects may amend the final mark adding to it a value ranging from -2.0 to +6.0 marks. Project taken (assigned by the instructors) and **not completed** by the group will be evaluated (for all students within the group) with a final value of **-2.0**. Once the final mark for a project has been assigned, there is no time limit to its validity.

Projects will be assigned with the following protocol:

- Phase 1 (selection phase)
    - By **Friday the 21st of May** each group of students willing to take a project must indicate two group works (first and second choice)

        - Before that deadline there will be the opportunity to discuss with the instructors characteristics and details of all projects either in real-time or off-line
    - The choices must be stated by uploading on the portal, at the page "Elaborati", a **single** file with **name**, **content** and **description** equal to the following **unique** string

      rn1_rn2_rn3_p1_p2

      where
        - **rn1**, **rn2** and **rn3** are the register numbers of the 2 or (at most) 3 students forming the group.
        - **p1** and **p2** are the works selected by the group (namely, first choice **p1,** and second choice **p2**), i.e., q1, q2, etc., q5 for Quer's projects and c1, c2 or c3 for the Cabodi's projects.
    - Two correct examples of this file name, content, and comment to insert on the "Elaborati" web page are the following:
        - 134567_146789_189345_q1_c2 which means that the students of register numbers 134567, 146789 and 189345 selected the Quer's project number 1 as a first choice and the Cabodi's project number 2 as second choice.
        - 176789_191352_c3_q4 which means that the students of register numbers 176789 and 191352 selected the Cabodi's project number 3 as a first choice and the Quer's project number 4 as second choice.
- Phase 2 (tentative assignment phase)
    - By **Friday the 28th of May** the instructors will publish on the portal web page a list of assignments, i.e., a list in which each group will be assigned a single work following a first-come-first-served algorithm and trying to maximize the student's preferences
- Phase 3 (final assignment phase)
    - By **Friday the 4th of June** each group will have the possibility to accept or reject the final assignment and eventually, **but only in very specific cases**, to select a different one:
        - Accept the project will be the default, no action will be necessary.
        - Reject (or change) the project will be possible during a further discussion (either on or off-line) with the instructors.

Project 1: Parallel Graph Coloring

Project's summary

In graph theory, graph coloring is a special case of graph labeling; in which particular labels, traditionally called "colors", are assigned to elements of a graph subject to certain constraints. The convention of using colors originates from coloring the countries of a map. In its simplest form, it is a way of coloring the vertices of a graph such that **no** two adjacent vertices are of the same color. This version of the problem is called "vertex coloring".
Graph coloring enjoys many practical applications as well as theoretical challenges. Many graph applications are based on vertex coloring and many graph properties are based on graph coloring. Moreover, vertex coloring is the most famous version of coloring since other coloring problems can be transformed into a vertex coloring instance.

Problem Definition

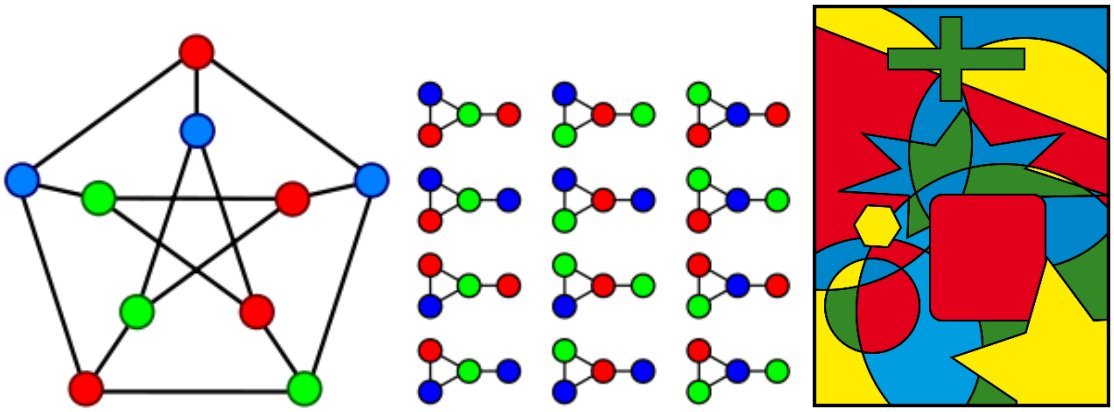
A graph $G=(V,E)$ includes a set of vertices V and a set of edges E, where $E \subseteq V \times V$.
Graph coloring

$$C: V \rightarrow N$$

is a function that assigns a color to each vertex that satisfies the property such that:

$$C(v) \neq C(u) \quad \forall (u,v) \in E.$$

In mathematical and computer representations, it is typical to use the first few positive or non-negative integers as the "colors". In general, one can use any finite set as the "color set". The nature of the coloring problem depends on the number of colors but not on what they are. The color representation can be optimized for memory space (e.g., using bit-fields in C++).
The following pictures shows a few examples of graph coloring (left-hand side and center) and an example of application (right-hand side, i.e., map coloring).



The target of the project is to implement (at least) two different parallel graph coloring algorithms on a multi-core architecture (a CPU) in C or C++. A possible extension includes the use of the C++ boost library to take advantage of the availability of many-core architectures (i.e., OpenCL-based GPUs). The target is to be as fast as possible to color large graphs using the smallest possible number of colors (the smaller the number of colors, the better the coloring). Memory usage must be kept under control, especially to deal with large graph (millions of nodes).
The starting point of the project is a well-known sequential and greedy algorithm. This algorithm is represented by the following pseudo-code.

```
Given G=(V,E)
n = |V|
choose a random permutation of the vertex {v1, v2, …, vn}
U = V
for i=1 to n do
   select vi
   C = {colors of all colored neighbors of v}
   c = smallest color not in C
   color vi with color c
   U = U – {vi}
endfor
```

Given the graph G, this implementation orders its vertices, and it proceeds sequentially on them (the "for" cycle). At each iteration, it selects the current vertex v_i and it builds the set C including the colors used by all vertices adjacent to v_i. Once done that, it selects the

smallest color not in C, namely c, and it assigns c to v_i. It finally eliminates v_i from the set of vertex U, and it cycles on the remaining vertices.
The parallel versions will be based on the works reported in the reference section, which, in turn, include other references with several possible parallel implementations.

Required Background

Background aspects are all covered within the following courses:

- Advanced problem solving and programming classes (e.g., "Algorithm and Programming")
- "System and Device Programming" course, i.e., concurrent programming.

Working Environment

Students can work on their laptop or desktop. Each group can decide whether to develop the application under a Unix-like or a Windows system. The implementation can be done in both C, C++, or C++ using the boost library (for GPU optimizations). Several graphs to experiment on are available in two similar but not identical formats. Parallel algorithms are described on the papers reported in the reference section; these works are available in the reference directory.

Constraints

Before starting the project, and by all means, before the 15th of July, each group **must** contact the tutors to receive further instructions and agree on the implementation details. All projects must be delivered including the following material:

- The source C/C++ files.
- A README text file (written in plain ASCII) including a short "user manual", i.e., a document describing how to compile and run the program, under which system, which API, etc.
- A DOCUMENTATION text file (written in Word, Latex, Mark-down, etc.) including a short "designer manual", i.e., a document including:
    - All main design choices (how the reading part has been performed, how the data structure has been organized, how the parallelism has been designed, etc.)
    - The experimental evaluation of the tools, i.e., tables or graphics reporting a reasonable set of experimental results. The experimental evidence should

include a comparison (in terms of memory and of elapsed time) between the original sequential version of the tool and the 2-3 parallel versions with different parallelization levels (i.e., with 1, 2, 4, 8, etc., threads) and different size of the input graph (up to millions of nodes).

- Optionally, an OVERHEAD set (organized in PowerPoint) to be used during the project discussion in the project evaluation phase.

Contacts

Prof. Stefano Quer (stefano.quer@polito.it).

References

J. R. Allwright, R. Bordawekar, P. Coddington, K. Dinçer, Christine Martin, "A Comparison of Parallel Graph Coloring Algorithms", 1995

M. Luby, "A Simple Parallel Algorithm for the Maximal Independent Set Problem", Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing (STOC'85), pp. 1-10, 1985, Providence, Rhode Island, USA

Project 2: Analysis of Simulation Dumps through Parallelization on Multicore Architectures

Project's summary

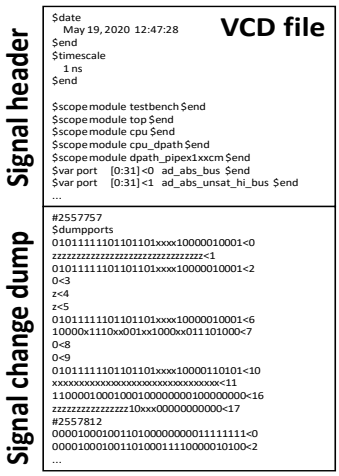
With the explosion of the size of the chips in terms of the number of gates and the advent of novel failure mode strategies, a major concern for testing and reliability of hardware devices is constituted by the elaboration times of the simulation and the fault simulation phases. Nowadays, when complex failure modes and fault metrics are considered on Systems-on-Chips integrating tens of millions of gates, the capabilities of standard ATPGs and fault simulators may be insufficient to reach the desired result accuracy. In these cases, it may be useful to adopt a post-processing phase to analyze simulation results. Thus, we first save into a file all values and timings for the set of signals (i.e., possibly all signals of the circuit) which are the target of the simulation. Then, we analyze the simulation dump off-line.
The project focus on the analysis of simulation files (of size up to 1Tbytes) adopting modern language features (such as fast direct access table, optimized memory allocation, minimized and asynchronous I/O waiting times, multiple-buffering techniques, etc.) and parallellization techniques.

Problem Definition

When a simulator must generate a trace which must be analyzed off-line, a commonly used file format is the so-called Value Change Dump (VCD). Its layout is composed of two logical sections:

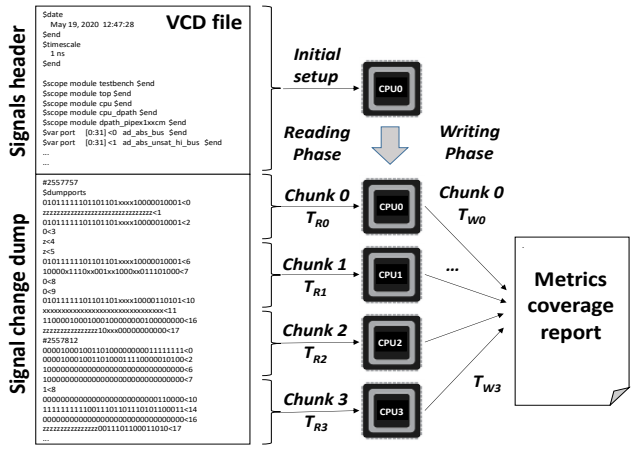
- The first one includes a header and a declaration part. The former defines generic information on the simulation performed. The latter locates all signals into a hierarchical description, it assigns an integer identifier to each signal, and it specifies an initial value for each signal of the design.
- The second section reports all values assumed by the previously defined signals over subsequent time frames. Transitions are reported by increasing time values, such that, for each time unit in the future, only the set of all transient signals is indicated with the corresponding new value. A subset of these signals (when not all of them) is the target of our analysis.

A diagrammatical representation of such a format, and its high-level layout, is reported in the following figure.



The manipulation of a VCD file is theoretically straightforward, as it implies the transformation of its information from a time-based to a signal-based organization. However, practically speaking, the huge size of current VCD files (up to 1Tbytes) implies enormous processing resources, in terms of both memory and computation time.

To improve the efficiency of all I/O operations, we re-designed the original single-core application as a concurrent multi-core divide-and-conquer approach. From a very high-level point of view, we thus adopt the scheme represented in the next figure which shows how different working threads concurrently manipulate different sections of the source VCD file with smart low-level system-calls for all critical (in memory or time) phases.



Although this logic scheme is apparently very simple, to reach the desired efficiency and scalability, the application must be organized very carefully. Moreover, the parallel implementation has strong influence on the methodology used to evaluate the testing measure. Testing results are essentially time-based, whereas in the parallel implementation we proceed in parallel on different file partitions, corresponding to different time frames.

Required Background

Background aspects are all covered within the following classes

- Programming classes (e.g., "Algorithm and Programming")
- "System and Device Programming" course, i.e., concurrent programming especiallly I n C++.

Working Environment

Students can work on their laptop or desktop. We have already developed several parallel versions of the original sequential tool, dealing with different testing measures (full, statistic, statical, etc.). The group will have to re-write, optimize, or extend or one or more of these tools. The development phase has to be performed in a Unix-like POSIX system and in C/C++ language.

Constraints

Before starting the project, and by all means, before the 15th of July, each group **must** contact the tutors to receive further instructions and agree on the implementation details. All projects must be delivered including the following material:

- The source C/C++ files.
- A README text file (written in plain ASCII) including a short "user manual", i.e., a document describing how to compile and run the program, under which system, which API, etc.
- A DOCUMENTATION text file (written in Word, Latex, Mark-down, etc.) including a short "designer manual", i.e., a document including:
    - All main design choices (how the reading part has been performed, how the data structure has been organized, how the parallelism has been designed, etc.)
    - The experimental evaluation of the tools, i.e., tables or graphics reporting a reasonable set of experimental results. The experimental evidence should include a comparison (in terms of memory and of elapsed time) between the original sequential version of the tool and the 2-3 parallel versions with different parallelization levels (i.e., with 1, 2, 4, 8, etc., threads) and different size of the input graph (up to millions of nodes).
- Optionally, an OVERHEAD set (organized in PowerPoint) to be used during the project discussion in the project evaluation phase.

Contacts

Prof. Stefano Quer (stefano.quer@polito.it), Prof. Paolo Bernardi (paolo.bernardi@polito.it), Ing. Andrea Calabrese (andrea.calbrese@polito.it).

References

D. Appello, P. Bernardi, A. Calabrese, S. Littardi, G. Pollaccia, S. Quer, V. Tancorre, R. Ugioli, "Accelerated Analysis of Simulation Dumps through Parallelization on Multicore Architectures", In Proceedings of the 24th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS), Vienna, April 7-9, 2021

Project 3: Real-Time Operating System Simulator and Fault Injector

Problem Definition

A real-time operating system (RTOS) is an operating system where each executed task comes with time constraints to define its run. The RTOS usually comes as a library to be compiled together with all applications to run, obtaining one executable file to implement this feature. This file includes all HW-related necessary features (boot, scheduler, mutexes, etc.) and the applications. This project aims to develop an RTOS Simulator (based on the FreeRTOS operating system) with Fault Injection capabilities.

Goals

FreeRTOS is an open RTOS developed in C.
The first step is to study the operating system, i.e., its organization, the configuration strategy, and how to customize it, reporting all data structures involved in the OS lifetime. Once the knowledge of the OS is sufficient, students must prepare a building environment based on CMake (hence supporting Linux/OSX and Window OS) to generate a FreeRTOS instance as a Simulation on the Host machine.
The second step of the project requires to include a Fault Injection process. It can inject a Single Event Upset (SEU) faults within the memory data structure of the RTOS. SEU faults are a single bit-flip of a memory cell. A proper Fault Injector asynchronously modifies the memory cell and collects the results. It can also gather all the information about the application/OS behavior when no faults are injected (it is commonly called "golden execution").
The Fault Injector must be completely programmable in the sense that it must be possible to define:

1. The number of injections to test.
2. The list of structures where to inject faults.
3. The time window of the execution that constrains the random time of injection (thus the minimum and maximum times).

Moreover, the Fault Injector must control the OS because the injection may interfere with the OS behavior in different ways:

1. There is no error at all (classified with the label Masked): If the system's output shows no alteration concerning the golden execution.
2. A Silence Data Corruption (SDC): If the system runs smoothly, but the output shows a difference with respect to the golden execution.
3. A Delay: If the system provides the same golden execution results, and they are delivered later than expected. To properly profile this scenario, refer to the way

FreeRTOS defines task deadlines to associate this misbehavior with a missing deadline concept.

4. A Hang: If the delay is so long that it prevents the tasks from ending, but the OS is still up and running, the system may face an endless loop, i.e., deadlocks, spinlocks, starvations, etc.
5. A Crash: If the OS stops working.

The cases above represent the classification of every single injection. Design a logging system to track them.
Eventually, if possible, try to deliver a solution that parallelizes the injections: The parallelization must run in parallel several executions, each one with a single fault injection (i.e., without injecting multiple bit-flip during the same run).

Constraints

The maximum number of groups that can access this project is limited to two.
Before starting the project, and by all means, before the 15th of July, each group **must** contact the tutors to receive further instructions and agree on the implementation details.
All projects must be delivered including the following material:

- The source C/C++ files.
- A README text file (written in plain ASCII) including a short "user manual", i.e., a document describing how to compile and run the program, under which system, which API, etc.
- A DOCUMENTATION text file (written in Word, Latex, Mark-down, etc.) including a short "designer manual", i.e., a document including:
    - All main design choices (how the reading part has been performed, how the data structure has been organized, how the parallelism has been designed, etc.)
    - The experimental evaluation of the tools, i.e., tables or graphics reporting a reasonable set of experimental results. The experimental evidence should include a comparison (in terms of memory and of elapsed time) between the original sequential version of the tool and the 2-3 parallel versions with different parallelization levels (i.e., with 1, 2, 4, 8, etc., threads) and different size of the input graph (up to millions of nodes).
- Optionally, an OVERHEAD set (organized in PowerPoint) to be used during the project discussion in the project evaluation phase.

Contacts

Prof. Stefano Quer (stefano.quer@polito.it), Prof. Alessandro Savino (alessandro.savino@polito.it).

References

- https://www.freertos.org/FreeRTOS-simulator-for-Linux.html
- https://www.freertos.org/FreeRTOS-Windows-Simulator-Emulator-for-Visual-Studio-and-Eclipse-MingW.html
- https://github.com/megakilo/FreeRTOS-Sim
- https://github.com/alxhoff/FreeRTOS-Emulator

Project 4: Entity Component System Parallel Computing for real-time Computer Graphics and Virtual Environments

Problem Definition

The popular game engine Unity has recently introduced the Data Oriented Technology Stack (DOTS), an entirely new way of approaching virtual environment simulations. This new Data-Oriented approach is optimized to take full advantage of today's multi-core processors, allowing to develop highly multithreaded and optimized C# code that can facilitate the real-time management of highly complex systems.

Objectives

For this project, we ask students to understand the ins and outs of DOTS and develop a testing scenario that can stress this technology.

Basic knowledge of Unity is needed to execute DOTS code, as the main focus of the project is developing an efficient multithreaded system based on DOTS without the necessity to create or manage the graphical representation of any virtual element.
For example, in the case of a vehicular traffic system, the contribution of the work is to manage the behavior of a vast number of cars (e.g., 1M), focusing on the scenario rules (i.e., when and how a car should stop, when it should change lanes, when it must stop at traffic lights, etc.) and not on providing a "nice" graphical representation of the environment. A basic version for visual debugging will suffice the requirements.

Students are required to pick one of the following suggested complex systems and simulate it in their project:

- Traffic simulation of a city, including personal vehicles and public transportations, thus simulating streets, intersections, traffic lights, parkings, bus stops, etc.
- Steering behaviour based crowd simulation, i.e., simulating crowds in the context of public events like concerts, pedestrian movements, emergency evacuations and similar by implementing steering behaviours (https://en.wikipedia.org/wiki/Boids) into Unity's DOTS / ECS. Implementations of DOTS steering behaviours are available online. Students are either required to create their own version, or thoroughly compare different solutions available on a common testbed, and analyze their advantages and disadvantages.

The students involved in the project are expected to develop a complete application within the selected test scenario and to write a report detailing the implementation work and the results obtained (in terms of computational speed/number of agents/elements managed, and so on).

Required Background

As far as the "System and Device Programming" course is concerned, the project will give specific emphasis to the following topics:

- Visual Studio advanced programming and debugging.
- Parallel programming.

Moreover, each student will need his/her own personal insight on the following topics and tools:

- C#, i.e., an extension of C++
- A minimal knowledge of Unity3D, the development environment (assistance on the use of the tool will be provided by the external tutors).

Working Environment

Students can work on their laptop or desktop. The tools required for the development are Unity3D and VisualStudio (with the C# development environment) and both of them can be downloaded for free.

Constraints

The maximum number of groups that can access this project is limited to four.
Each group can be formed by a maximum of three candidates.
Before starting the project, and by all means, before the 15th of July, each group **must** contact the tutors to receive further instructions and agree on the implementation details.

Contacts

Prof. Stefano Quer (stefano.quer@polito.it), Prof. Andrea Bottino (andrea.bottino@polito.it), Ing. Edoardo Battegazzorre (edoardo.battegazzorre@polito.it), Ing. Francesco Strada, (francesco.strada@polito.it).