Assignment MPI: Circuit Satisfiability

S. Varrette, V.Plugaru and P. Bouvry
<Firstname.Lastname@uni.lu>

Version 1.0

The objective of this assignment is to evaluate the satisfiability of the circuit proposed in the figure 1. The C code you can use to model this circuit is as follows:

```
#include <stdio.h>

/* Return 1 if 'i'th bit of 'n' is 1; 0 otherwise */
#define EXTRACT_BIT(n,i) ((n&(1<<i))?1:0)

int check_circuit(int input) {
   int v[16], i;
   for (i = 0; i < 16; i++) v[i] = EXTRACT_BIT(input,i);
   return ((v[0] || v[1]) && (!v[1] || !v[3]) && (v[2] || v[3]) &&
         (!v[3] || !v[4]) && (v[4] || !v[5]) && (v[5] || !v[6]) &&
         (v[5] || v[6]) && (v[6] || !v[15]) && (v[7] || !v[8]) &&
         (!v[7] || !v[13]) && (v[8] || v[9]) && (v[8] || !v[9]) &&
         (!v[9] || !v[10]) && (v[9] || v[11]) && (v[10] || v[11]) &&
         (v[12] || v[13]) && (v[13] || !v[14]) && (v[14] || v[15]));
}
```

In general term, assuming a circuit $C : 2^n \longrightarrow \{0, 1\}$ is composed by AND, OR and/or NOT binary gates, the **circuit satisfiability** problem consists in finding the combinations of input values (if any) for which the circuit C output the value 1.
This problem is important for the design and verification of logical devices. Unfortunately, it belongs to the class of NP-complete problems, which means that there is no known polynomial time algorithm able to solve general instances of this problem [1].

One way to solve this problem is to attempt a *brute-force* approach, *i.e.* try every possible combination of inputs. Since the circuit we analyse has 16 binary inputs, there are a total of $2^{16} = 65536$ input possibilities.

Exercise 1 *Sequential Program*

1. Propose a (sequential) program finding the 9 solutions to this problem, printed in hexadecimal.
   Sample execution:

```
$> ./circuit
- found solution '0x99f5' = 1010111110011001
- found solution '0x99f6' = 0110111110011001
- found solution '0x99f7' = 1110111110011001
- found solution '0x9bf5' = 1010111111011001
- found solution '0x9bf6' = 0110111111011001
- found solution '0x9bf7' = 1110111111011001
- found solution '0x9df5' = 1010111110111001
- found solution '0x9df6' = 0110111110111001
- found solution '0x9df7' = 1110111110111001
Total number of solutions: 9
```

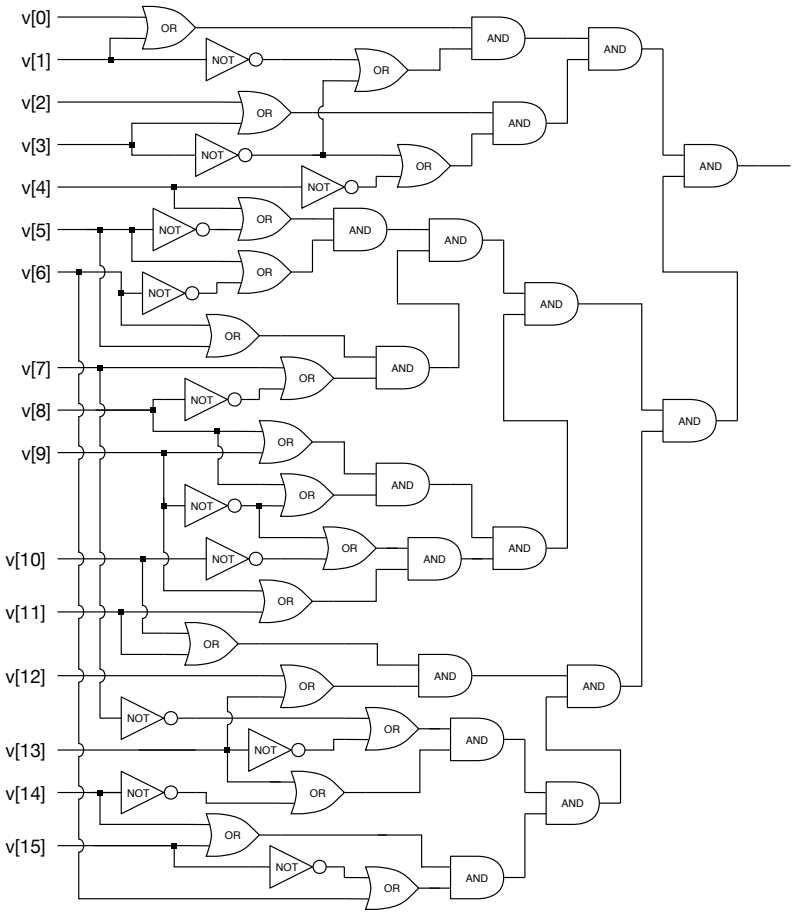


Figure 1: A circuit composed by AND, OR and NOT gates. The circuit satisfiability problem is to determine if some combination of inputs permits the output of 1.

Rely on CMake to build your program. A sample configuration file `CmakeLists.txt` is proposed in appendix A.

The Foster's design methodology for parallel program [2] is summarized in appendix B. We will apply it to this problem.

Exercise 2 *Parallel Program Design*

1. (Partitioning) A functional decomposition is natural for this application, where one task is responsible for checking a given combination of the 16 boolean inputs, and print this value **if** the circuit outputs 1. How many atomic tasks are thus required to execute?
2. (Communication) Draw the corresponding task/channel graph. Are the tasks independent? How is called this type of problem?
3. (Agglomeration and Mapping) The general strategy to apply is recalled in the Ap-

pendix C (Fig. 2).

a) There is a *fixed* number of atomic tasks. Is the communication pattern structured?

b) Is the computation time per task roughly constant or variable?

c) Conclude on the appropriate mapping strategy on p processes.

Exercise 3 *MPI Implementations*

1. Implement the MPI version of your program `mpi_circuit_1.c`. Found solutions will have to be output this time under a binary format. Sample execution:

```
$> mpirun -np 4 ./mpi_circuit_1
[Node 1] Found solution '1010111110011001'
[Node 2] Found solution '0110111110011001'
[Node 3] Found solution '1110111110011001'
[Node 1] Found solution '1010111111011001'
[Node 2] Found solution '0110111111011001'
[Node 3] Found solution '1110111111011001'
[Node 1] Found solution '1010111110111001'
[Node 2] Found solution '0110111110111001'
[Node 3] Found solution '1110111110111001'
[Node 0] done!
[Node 1] done!
[Node 3] done!
[Node 2] done!
```

   Note that a template of MPI programs is proposed in Appendix D.

2. Adapt the previous version of your program into `mpi_circuit_2.c` where a reduction operation is performed to Print the total number of solutions. Use `MPI_Reduce` to perform the global count

3. Make a final version of your program, name `mpi_circuit_3.c`, which benchmark your programs and plot the data using gnuplot. Additionally, compute through a reduction the max/min/avg time spent between processes.

A CMake Configuration File

```
cmake_minimum_required (VERSION 2.8.12)
project (MPI_CircuitSatisfiability C)

SET(CMAKE_C_STANDARD 99)
#SET(CMAKE_C_FLAGS --std=c99 )

enable_testing()

#==== MPI
find_package(MPI REQUIRED)
set_property( DIRECTORY PROPERTY COMPILE_DEFINITIONS ${MPI_C_COMPILE_FLAGS} )
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${MPI_C_COMPILE_FLAGS}")
include_directories(SYSTEM ${MPI_C_INCLUDE_PATH})
list(APPEND EXTRALIBS ${MPI_C_LIBRARIES})

#==== Let's go
add_executable(circuit circuit.c)
add_executable(mpi_template mpi_template.c)
target_link_libraries(mpi_template ${EXTRALIBS})

add_executable(mpi_circuit_1 mpi_circuit_1.c)
add_executable(mpi_circuit_2 mpi_circuit_2.c)
#add_executable(mpi_circuit_3 mpi_circuit_3.c)
target_link_libraries(mpi_circuit_1 ${EXTRALIBS})
target_link_libraries(mpi_circuit_2 ${EXTRALIBS})

#================= Automatic tests ==================
#define a macro to simplify adding tests, then use it
macro (do_test prog result)
  add_test (UniTest-${prog}-contains-${result} ${prog})
  set_tests_properties (UniTest-${prog}-contains-${result}
    PROPERTIES PASS_REGULAR_EXPRESSION ${result})
endmacro (do_test)

set(SOLUTIONS
  1010111110011001
  0110111110011001
  1110111110011001
  1010111111011001
  0110111111011001
  1110111111011001
  1010111110111001
  0110111110111001
  1110111110111001
)

foreach(solution ${SOLUTIONS})
  do_test(circuit ${solution})
endforeach(solution ${SOLUTIONS})

#=== MPI Tests
macro (do_mpi_test prog result)
  add_test (MPI_UniTest-${prog}-expect-${result}
    ${MPIEXEC} ${MPIEXEC_NUMPROC_FLAG} ${PROCS}
    ${MPIEXEC_PREFLAGS}
    ${prog}
    ${MPIEXEC_POSTFLAGS}
    )
```

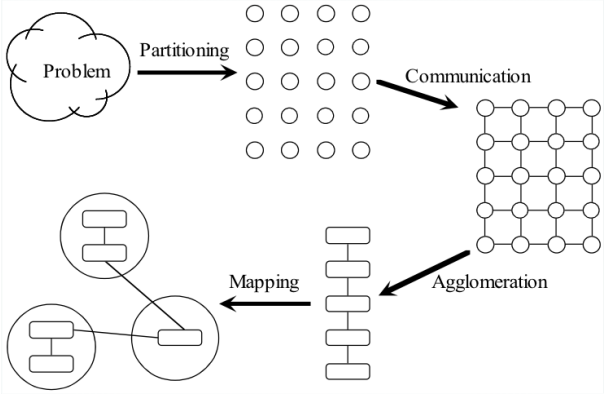
```
 set_tests_properties (MPI_UniTest-${prog}-expect-${result}
   PROPERTIES PASS_REGULAR_EXPRESSION ${result})
endmacro (do_mpi_test)

# Number of procs for MPI.
set (PROCS 4)

# TODO: find a better version that catch the command output using EXEC_PROGRAM such
# that the program is executed only once.
foreach(solution ${SOLUTIONS})
# do_mpi_test(mpi_circuit_1 ${solution})
endforeach(solution ${SOLUTIONS})

do_mpi_test(mpi_circuit_2 "9")
```

B Foster's Design Methodology for Parallel Programs: PCAM



C Mapping Strategy

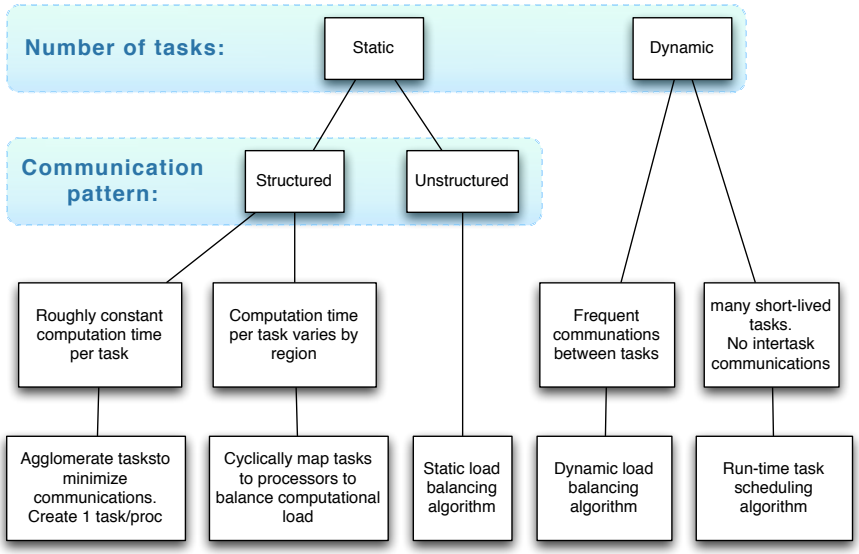


Figure 2: Classical Mapping Strategy.

D Template for MPI programs

```
#include <stdio.h> /* for printf */
#include <stdlib.h> /* for exit */
#include <stdarg.h> /* for va_{list,args... */
#include <unistd.h> /* for sleep */
#include <mpi.h>

int id = 0; // MPI id for the current process (set global to be used in xprintf)

/**
 * Redefinition of the printf to include the buffer flushing
 */
void xprintf(char *format, ...) {
   va_list args;
   va_start(args, format);
   printf("[Node %i] ", id);
   vprintf(format, args);
   fflush(stdout);
}

int main(int argc, char *argv[]) {
   int p; // MPI specific: number of processors
   unsigned int n = 0;
   double elapsed_time = 0.0;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&p);
   MPI_Comm_rank(MPI_COMM_WORLD,&id);

   if (id == 0) {
            if (argc < 2) {
                  xprintf("Total Number of processes : %i\n",p);
                  xprintf("Input n = ");
                  scanf("%u", &n);
            } else
                  n = atoi(argv[1]);
   }
   MPI_Barrier(MPI_COMM_WORLD);
   elapsed_time = -MPI_Wtime();
   // send n to the other processes
   MPI_Bcast((void *)&n, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
      if (id != 0)
            xprintf("(received) n = %i\n",n);
   // Do something useful...
   sleep(n);
   // at the end, compute elapsed time
   elapsed_time += MPI_Wtime();
   if (id == 0)
      xprintf("Elapsed time: %2f s\n", elapsed_time);
   MPI_Finalize();
   return 0;
}
```

References

[1] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

[2] I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.