ARIZONA STATE UNIVERSITY

CSE 430, SLN 14814 — **Operating Systems** — Spring 2015

Instructor: Dr. Violet R. Syrotiuk

Project #1

Available 01/21/2015; milestone due 02/04/2015; full project due 02/25/2015

This project has two goals:

1. The first is to learn to use **OpenMP** directives and **Intel® Parallel Studio** tools to develop a parallel program from a serial program to verify a covering array and identify "don't care" positions.
2. The second is to get experience using `ASURE`, ASU's cluster for students, to run experiments comparing the performance of the serial and parallel programs, collect and plot data, and interpret the results. See §3 for how to register for an account.

1 Covering Arrays

Covering arrays are used in testing software, hardware, composite materials, biological networks, and others. They also form the basis for combinatorial methods to learn an unknown classification function using few evaluations — these arise in computational learning and classification, and hinge on locating the relevant attributes.

A *covering array* $CA(N; t, k, v)$ is an $N \times k$ array where *each* $N \times t$ subarray contains all ordered t-sets on v symbols *at least once*; t is called the *strength* of the covering array.

For example, the array in Table 1 is a $CA(6; 2, 5, 2)$ covering array. It has dimensions $N \times k = 6 \times 5$, strength $t = 2$, and is on $v = 2$ symbols, i.e., this particular covering array is binary.

		Columns				
		1	2	3	4	5
Rows	1	0	1	1	1	1
	2	1	0	1	0	0
	3	0	1	0	0	0
	4	1	0	0	1	1
	5	0	0	0	0	1
	6	1	1	0	1	0

Table 1: A $CA(6; 2, 5, 2)$ covering array.

1.1 Verifying that an Array is a Covering Array

In order to verify that the array in Table 1 is a covering array, we must check that every $N \times t$ subarray contains all ordered t-sets on v symbols. In this case, there are $\binom{k}{t} = \binom{5}{2} = 10$ subarrays to check. Recall that $\binom{k}{t} = \frac{k!}{t!(k-t)!}$. Thus, we must check that each of the 10, 6×2 subarrays contains (or, covers) all ordered t-sets on v. Specifically, for each 6×2 subarray defined by a pair of columns in

$$\{(1,2), (1,3), (1,4), (1,5), (2,3), (2,4), (2,5), (3,4), (3,5), (4,5)\},$$

we must check that all ordered 2-sets on v are covered. The ordered 2-sets on the $v = 2$ symbols $\{0, 1\}$ are $\{(0,0), (0,1), (1,0), (1,1)\}$. In general, there are v^t ordered t-sets to cover.

First, select the 6×2 subarray corresponding to the pair of columns $(1, 2)$. In this subarray, the 2-set $(0, 0)$ is covered in row 5. The 2-set $(0, 1)$ is covered twice, once in row 1 and another time in row 3. The

2-set $(1,0)$ is also covered twice, in rows 2 and 4. Finally, the 2-set $(1,1)$ is covered in row 6. Checking the remaining nine 6×2 subarrays in the same way, it is possible to verify that array in Table 1 is indeed a $CA(6;2,5,2)$ covering array.

This may have you wondering about how you would go about *constructing* a covering array, especially one of minimal size, i.e., minimal number of rows N. We won't address that problem in this project, but it's a very interesting question! Indeed, for certain parameters, this is an open question subject to research.

1.2 "Don't Care" Positions in a Covering Array

In any covering array $CA(N;t,k,v)$, the number of t-sets to be covered is $v^t\binom{k}{t}$, while the number actually covered is $N\binom{k}{t}$. Except possibly when $k \leq \max(v+2, t+1)$, some duplication of coverage is necessary. All of the techniques to construct covering arrays attempt to limit this duplication in an effort to minimize the number of rows N, but cannot hope to eliminate it completely. One way to eliminate some of the duplication is through discovering "don't care" positions. Every entry of a $CA(N;t,k,v)$ participates in $\binom{k-1}{t-1}$ t-way interactions. Some of these interactions may be covered elsewhere, while others may be covered only in the entry's row. In principle, a specific t-way interaction could be covered as many as $N - v^t + 1$ times or as little as once. When all of the $\binom{k-1}{t-1}$ t-way interactions involving a specific entry are covered more than once, the entry can be changed arbitrarily, or indeed omitted in the determination of coverage, and the array remains a covering array. Hence such an entry is a "don't care" position.

In the $CA(18;2,26,3)$ in Table 2, "don't care" positions are marked with a $\star$. Each $\star$ could take on any value in $\{0,1,2\}$ and the array remains a covering array.

11120211122100120202122221
00011021212221100112101122
10212221220201211010200011
01222111111121002001020002
12110110210000022022221111
21020120021102212111201120
02001022202101202000222210
10102200011011222201102102
1⋆00021211220221102011000
01002002020010001110121211
20220202100101101120012102
22202101002012110022110020
12121010202212001211002001
20111112010222011200022220
02121220121022020110010112
0⋆010022120⋆⋆0210221200202
21211000201220122222⋆211211
⋆0⋆10⋆2⋆1⋆⋆1101⋆⋆⋆⋆121⋆⋆1⋆

Table 2: A $CA(18;2,26,3)$ with "don't care" positions.

1.2.1 Finding "Don't Care" Positions

To find possible "don't care" positions, it suffices to determine the numbers of times that the $v^t\binom{k}{t}$ t-way interactions are covered. For each of the $N \times k$ entries, check whether the entry appears in any t-way interaction that is covered only once. If not, it is a possible "don't care" position. While conceptually simple, this requires space proportional to $v^t\binom{k}{t}$, which is too much in practice. Instead, initially mark each of the $N \times k$ entries as a possible "don't care." Then for each of the $\binom{k}{t}$ sets of columns in turn, use a vector of length v^t to record the number of times each of the t-way interactions arises in the t chosen columns.

Then for each that arises only once, mark all t positions in it to be no longer "don't care." This requires only $N \times k + v^t$ space, but still requires time proportional to $t \times N \times \binom{k}{t}$. At the same time, one can verify that the array is in fact a covering array, by ensuring that every t-way interaction is seen at least once. Unfortunately, if we change any one of the possible "don't care" positions to $\star$, some recomputation is then needed.

To find a set of "don't care" positions that can all be simultaneously changed to $\star$, we use the fact that rows are recorded in a specific order. For every set of t columns we consider the rows of the covering array in order; when a t-tuple is covered for the first time we mark its t positions as necessary. After every possible set of t columns is treated, all positions that are not necessary can be changed to $\star$. This can be done in the same time and space as the identification of all possible "don't care" positions.

Once done, each row may have any number of $\star$ entries from 0 to $k - t$ or may consist entirely of "don't care" positions. When the latter occurs, this row can be removed without reducing the strength of the covering array. (There is a post-optimization approach that takes as input a CA and tries to create an entire row of "don't care" positions in order to reduce the size of the CA.)

2 Program Requirements for Project #1

You are required to write **two** programs to verify a covering array and identify "don't care" positions: a serial program and a parallel (OpenMP) program.

1. Write a *serial* C/C++ program that reads a file containing an array in the following format:

   (a) The first line contains the parameters of the array separated by spaces: $N\ t\ k\ v$. Your program should be able to handle a strength $1 < t \leq 3$, and $1 < v \leq 3$ symbols. You should certainly be able to handle $k \leq 100$, but I hope that you are able to handle $k \leq 500$.

   (b) Following the line with the parameters are N lines, one for each row of the covering array. Each line has k integers from the set on v, i.e., k integers from $\{0, 1, \ldots, v-1\}$.

   The output of your serial program should be:

   (a) A decision "yes" or "no" (pass/fail; true/false) on whether the input array is a covering array.

   (b) If the array is a covering array, then print a map of the "don't care" positions similar to Table 2. Following the map, print the (r, c) coordinates of each "don't care" position, where r and c are its row and column, respectively. For example, in Table 2 the first "don't care" position has coordinates $(9, 2)$.

   Your serial program satisfying these requirements is the milestone for this project. The milestone submission deadline is before noon on Wednesday, 02/04/2015; see §4.

2. Make a copy of your serial program and parallelize it by introducing OpenMP directives. Your objective is to obtain the best speedup you can while maintaining correctness.

3. Ultimately, both your serial and parallel programs must run on `ASURE`, a Linux system. Therefore, it is strongly recommended that you develop the programs for this project on a Linux system (e.g., `general.asu.edu`), using a C/C++ compiler supported on `ASURE`.

   Your serial and parallel program, along with your results from experimentation on `ASURE`, and your analysis of them, constitute the full project. The full project deadline is before noon on Wednesday, 02/25/2015; see §4.

Sample input files will be provided on Blackboard; use them to test the correctness of your programs.

3 Experimentation on ASURE

Register for an account at ASU's Advanced Computing Center `http://a2c2.asu.edu/get-an-account/` and by clicking on the "ASU" button. From there, you should login using your ASUrite id, select the `ASURE` cluster, list me as your affiliated faculty, and CSE 430 as your "Research Area."

3.1 Requirements using ASURE

1. Copy (e.g., using `sftp`) both your serial and parallel C/C++ programs onto an `ASURE` *login node* and compile them.
2. From a *login node*, submit a series of *batch jobs* for verifying covering arrays of increasing size and strength on one of the *compute nodes* with at least 12 cores. **Do not run your programs on the login node!!!** Collect the run time of your serial and parallel program for each array. Be sure to use the same type of hardware for all of your experiments!
3. Plot the run time of your serial and parallel programs as a function of array size and strength, and analyze the results. Use your results to answer the following questions:
   (a) What speed-up, if any, is obtained by your parallel program over your serial program?
   (b) What size does the array need to be (for a given strength) before a speed-up is observed?

4 Submission Instructions

This project has two submission deadlines.

1. The milestone deadline is before noon on Wednesday, 02/04/2015.
2. The full project deadline is before noon on Wednesday, 02/25/2015.

4.1 Requirements for Milestone Deadline

Submit electronically, before class time (noon) on Wednesday, 02/04/2015 using the submission link on Blackboard for Project #1 Milestone, a zip[1] file named `yourFirstName-yourLastName.zip` containing the following items:

Design and Analysis (30%): Provide a description of the methodology you followed to produce a correct serial program. Specifically:

1. Discuss your selection of data structures for generating subarrays and t-sets.
2. Discuss the data structure you used to keep track of coverage conditions in the CA.
3. Discuss the data structure you used to keep track of "don't care" positions.

Implementation of Serial Program (50%): Your documented C/C++ source code for your serial program.

Correctness (20%): You must provide a script to run your serial program on the test input files provided; more details about the test input files will be provided in the discussion group closer to the milestone due date. Your serial program will be tested by our TA on `general.asu.edu` so it must compile and run there.

The milestone is worth 30% of the total final project.

[1] **Do not** use any other archiving program except `zip`.

4.2 Requirements for Full Deadline

Submit electronically, before class time (noon) on Wednesday, 02/25/2013 using the submission link on Blackboard for Project #1, a zip[2] file named `yourFirstName-yourLastName.zip` containing the following items:

Design and Analysis (30%): Provide a description of the methodology you followed to produce a correct parallel version of your serial program. Specifically:

1. Discuss where you introduced threads, and how you did load balancing among threads.
2. Describe any errors that arose (e.g., race conditions) and how you solved them.
3. Describe the activities you performed to improve the speed-up of your parallel program.

In addition, include the results of your analysis, i.e., the graphical results and answers to the questions posed in the third point of §3.1.

Implementation (50%): Provide your documented C/C++ source code for both your serial and parallel program. In addition, you must provide scripts to run your serial and parallel programs on the test input files provided; more details about the test input files will be provided in the discussion group closer to the due date.

You may write your code only in C or C++. You must not alter the requirements of this project in any way.

Correctness (20%) Your serial and parallel programs **must** run on `ASURE` using the scripts you provide. Our TA will test them there.

[2] **Do not** use any other archiving program except `zip`.