Programming Assignment 1
CSC410 - Parallel Computing

Aaron G. Alphonsus

October 1, 2018

1 Circuit Satisfiability

1.1 Introduction

We are given a circuit and we want to find out the inputs for which it produces a true output. While this problem is NP-complete, we can solve it for small input values, and we can parallelize our solution to give us a speed-up. In the circuit we are given, there are 16 inputs. Since each input can be a 0 or 1, there are $2^{16} = 65536$ different possible inputs. The circuit we were given can be seen in figure 1.

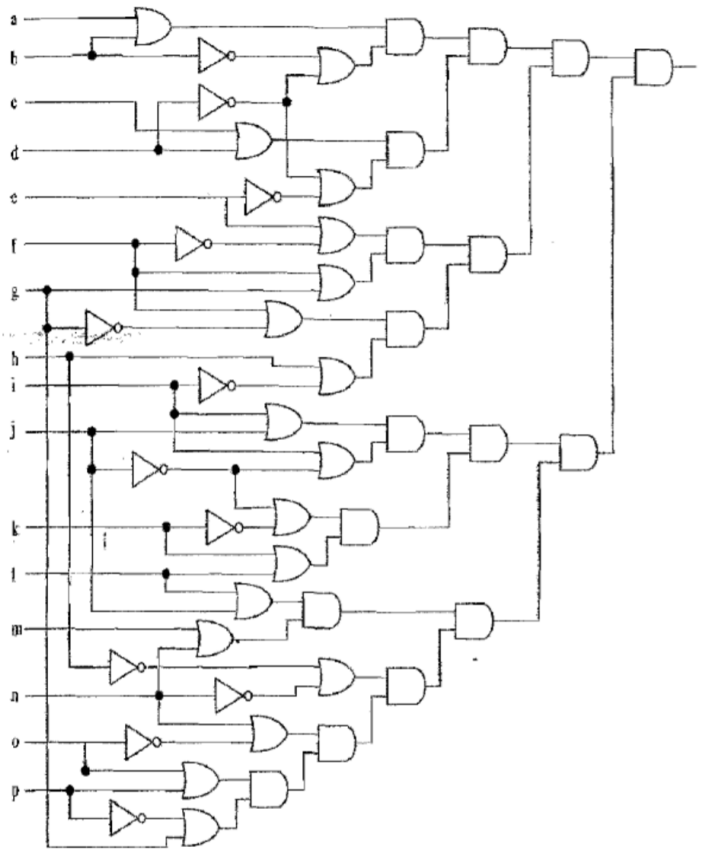


Figure 1: Circuit with 16 inputs (labeled a-p)

1.2 Algorithms and Libraries

We use a simple brute force algorithm where we test each of the 65536 possible inputs to see if any outputs 1. When we find one, we print the combination and we update our solution count. When all the tasks are done we print the number of solutions.

We parallelize the algorithm and check each of the 65536 combinations in parallel. The only dependency we need to be careful of is updating the solution counter and we use a reduction for this.

Libraries used:

- `<omp.h>`
- `<math.h>`
- `<stdlib.h>`
- `<stdio.h>`

1.3 Functions and Program Structure

The program has 5 functions:

- `main`
- `usage`
- `check_circuit`
- `time_parallel`
- `time_serial`

1.3.1 `main`

Arguments:

- `int argc`: Number of command-line arguments.
- `char* argv[]`: Pointer array storing each command-line argument.

Returns: `0` indicating normal termination.

Description:

- Takes in 2 command-line arguments `print` and `reps` and checks them for valid range. Calls usage function if invalid.
- Declares variables and calls the parallel and serial functions to run and time each method.
- Prints the time taken by each method.

1.3.2 `usage`

Arguments:

- `char* prog_name`: Character array containing the name of the program.

Returns: void

Description: Prints a message explaining how to run the program.

1.3.3 `check_circuit`

Arguments:

- `int id`: ID of the process calling the function.
- `int z`: One of the 65536 possible inputs.
- `int print`: Contains a 0 or 1 to suppress or allow printing within the function.

Returns: `1` or `0` indicating a `true` or `false` output from the circuit.

Description:

- Takes in an integer `z` representing one of the possible inputs to the circuit. Extracts each bit of `z` and stores it in an array.
- Uses an if statement to represent the circuit and test its satisfiability.
- Prints the input combination if it satisfies the circuit as well as the process id that calls the function.

1.3.4 `time_parallel`

Arguments:

- `int inputs`: Number of possible inputs (65536 in this case).
- `int thread_count`: Number of threads to parallelize the `for` loop.
- `int reps`: Number of times to repeat the algorithm.
- `int print`: Contains a 0 or 1 to suppress or allow printing within the `check_circuit` function.

Returns: Time taken to run the algorithm averaged over `reps` times.

Description:

- Runs the algorithm to check the circuit satisfiability `reps` times. Keeps a track of how long each run takes.
- Evaluates the output for each of the 65536 inputs in parallel with `thread_count` threads.
- Keeps a track of the number of inputs that satisfies the circuit and prints it out.

1.3.5 `time_serial`

Arguments:

- `int inputs`: Number of possible inputs (65536 in this case).
- `int reps`: Number of times to repeat the algorithm.
- `int print`: Contains a 0 or 1 to suppress or allow printing within the `check_circuit` function.

Returns: Time takes to run the algorithm averaged over `reps` times.

Description:

- Runs the algorithm to check the circuit satisfiability `reps` times. Keeps a track of how long each run takes.
- Evaluates the output for each of the 65536 inputs serially.
- Keeps a track of the number of inputs that satisfies the circuit and prints it out.

1.4 Compilation and Usage

Compilation: `make all`
Usage: `./prog1 <print> <reps>`

Both programs can be compiled and linked at the same time using the Makefile provided. To get rid of the executables in the folder, run the command `make clean`.

To use the program, we have a couple of command-line options to make it easier to test and experiment as we time our parallel and serial functions.

- The `print` variable expects an input of 0 or 1 corresponding to 'suppress printing' or 'allow printing' respectively.
- The `reps` variable is so that we can run both the serial and parallel algorithms multiple times and get a meaningful time comparison. Expects a value greater than or equal to 1.

To run the program once with printing of every solution:
`./prog1 1 1`
To time the program and suppress printing, bump up the number of repetitions and use 0 for the print variable:
`./prog1 0 1000`

1.5 Testing and Verification

To verify the program once we had our solution, we put our solution into Google and found that other people had published the same answer as us.

```
7296837@linux101 program1 >>./prog1 1 1
0) 1010111110011001
0) 0110111110011001
0) 1110111110011001
2) 1010111110111001
2) 0110111110111001
4) 1010111111011001
4) 0110111111011001
4) 1110111111011001
2) 1110111110111001
Number of solutions = 9

39413) 1010111110011001
39414) 0110111110011001
39415) 1110111110011001
39925) 1010111111011001
39926) 0110111111011001
39927) 1110111111011001
40437) 1010111110111001
40438) 0110111110111001
40439) 1110111110111001
Number of solutions = 9

Parallel time 18.4957 ms
Serial time = 3.2901 ms
7296837@linux101 program1 >>
```

Figure 2: Solutions to the circuit

Once we verified that we had the right answer, we moved onto timing the two versions of the program, the parallel and serial. As you can see in 6, running the program just once doesn't really give us an accurate idea of how long each implementation takes. So, we built in a way to suppress the printing and increase the number of times the algorithm is run.

```
7296837@linux101 program1 >>./prog1 0 1000
Parallel time = 0.6107 ms
Serial time = 2.3028 ms
7296837@linux101 program1 >>./prog1 0 1000
Parallel time = 0.5252 ms
Serial time = 2.2934 ms
7296837@linux101 program1 >>./prog1 0 1000
Parallel time = 0.6158 ms
Serial time = 2.2925 ms
7296837@linux101 program1 >>./prog1 0 1000
Parallel time = 0.5130 ms
Serial time = 2.2921 ms
7296837@linux101 program1 >>./prog1 0 1000
Parallel time = 0.5121 ms
Serial time = 2.2936 ms
7296837@linux101 program1 >>
```

Figure 3: Comparing times for prog1

1.5.1 Scheduling

We tried using the OpenMP loop scheduler to see if we could better our performance. While the `schedule(static, 1)` showed a marginal improvement in performance, like we saw in class, `schedule(dynamic, 1)` does not do very well. This can be seen in figure 8.

```
7296837@linux101 program1 >>./prog1 0 1000
Parallel time = 0.5315 ms
Serial time = 2.3012 ms
7296837@linux101 program1 >>./prog1 0 1000
Parallel time = 0.5212 ms
Serial time = 2.2931 ms
7296837@linux101 program1 >>./prog1 0 1000
Parallel time = 0.5104 ms
Serial time = 2.2914 ms
7296837@linux101 program1 >>./prog1 0 1000
Parallel time = 0.5189 ms
Serial time = 2.2973 ms
7296837@linux101 program1 >>./prog1 0 1000
Parallel time = 0.5168 ms
Serial time = 2.2948 ms
7296837@linux101 program1 >>
```

```
7296837@linux101 program1 >>./prog1 0 1000
Parallel time = 1.8671 ms
Serial time = 2.2963 ms
7296837@linux101 program1 >>./prog1 0 1000
Parallel time = 1.7629 ms
Serial time = 2.2949 ms
7296837@linux101 program1 >>./prog1 0 1000
Parallel time = 1.8400 ms
Serial time = 2.3460 ms
7296837@linux101 program1 >>./prog1 0 1000
Parallel time = 1.6759 ms
Serial time = 2.3469 ms
7296837@linux101 program1 >>./prog1 0 1000
Parallel time = 1.8369 ms
Serial time = 2.2953 ms
7296837@linux101 program1 >>
```

Figure 4: Comparison of the static and dynamic schedulers

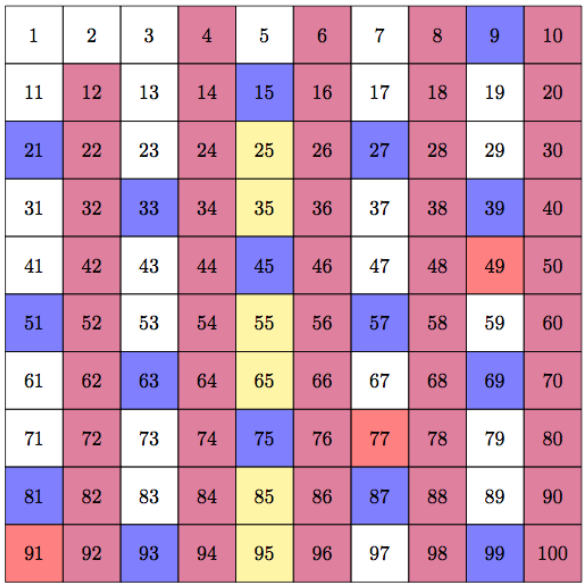
It should be noted that the dynamic scheduler does achieve a comparable performance to the static scheduler when the chunk size is increased.

• Description of the program. • Description of the algorithms and libraries used. • Description of functions and program structure. • How to compile and use the program. • Description of the testing and verification process. • Description of what you have submitted: Makefile, external functions, main, etc.

2 Sieve of Eratosthenes

2.1 Introduction

We are looking to find all the prime numbers up to an integer `n` using Eratosthenes' method.



1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Figure 5: Sieve of Eratosthenes up to 100

We would also like to parallelize the algorithm to improve our speed in finding all the primes less than or equal to `n`.

2.2 Algorithms and Libraries

The sieve of Eratosthenes algorithm is as follows:

1. Create a list of consecutive integers from 2 through `n`: `(2, 3, 4, ..., n)`.
2. Initially, let `p` equal 2, the first prime number.
3. Starting from `p`, enumerate its multiples by counting to `sqrt(n)` in increments of `p`, and mark them in the list. These will be 2`p`, 3`p`, 4`p`, etc. (`p` itself should not be marked).
4. Find the first number greater than `p` in the list that is not marked. If there was no such number, stop. Otherwise, let `p` now equal this new number (which is the next prime), and repeat from `step 3`.

When we look at this algorithm, it is clear that `step 3` can be parallelized. We could also parallelize `step 4` but we might do extra work, finding multiples of a number that isn't prime. While this won't give us an incorrect answer, it might slow us down. We have opted to only parallelize `step 3` of the algorithm.

Libraries used:

- `<math.h>`
- `<omp.h>`
- `<stdlib.h>`
- `<stdio.h>`

2.3 Functions and Program Structure

The program has 6 functions:

- `main`
- `usage`
- `time_parallel`
- `time_serial`
- `output`
- `output_testing`

2.3.1 `main`

Arguments:

- `int argc`: Number of command-line arguments.
- `char* argv[]`: Pointer array storing each command-line argument.

Returns: `0` indicating normal termination.

Description:

- Takes in 3 command-line arguments `n`, `print`, and `reps` and checks them for valid range. Calls usage function if invalid.
- Declares variables and initializes the list tracking all the primes with 1s.
- Calls the parallel and serial functions to run and time each method.
- Prints primes until `n` and the time taken by the parallel and serial methods.

2.3.2 `usage`

Arguments:

- `char* prog_name`: Character array containing the name of the program.

Returns: void

Description: Prints a message explaining how to run the program.

2.3.3 `time_parallel`

Arguments:

- `int reps`: Number of times to repeat the algorithm.
- `unsigned long long n`: Used to calculate length of primes array.
- `unsigned long long* primes`: Pointer to array containing list of numbers to be 'sieved'.

Returns: Time taken to run parallel portion of the algorithm averaged over `reps` times.

Description:

- Runs the sieve of Eratosthenes algorithm parallelizing `step 3` with `thread_count` threads. Times how long this section of the algorithm takes.
- Runs the algorithm `reps` times to get an average of the running time.

2.3.4 `time_serial`

Arguments:

- `int reps`: Number of times to repeat the algorithm.
- `unsigned long long n`: Used to calculate length of primes array.
- `unsigned long long* primes`: Pointer to array containing list of numbers to be 'sieved'.

Returns: Time taken to run `step 3` of the algorithm averaged over `reps` times.

Description:

- Runs the sieve of Eratosthenes algorithm serially. Times how long `step 3` of the algorithm takes.
- Runs the algorithm `reps` times to get an average of the running time.

2.3.5 `output`

Arguments:

- `unsigned long long n`: Used to calculate length of primes array.
- `unsigned long long* primes`: Pointer to array of prime numbers.

Returns: void

Description:

- Calculates padding value to print with correct formatting.
- Prints all the primes from 2 to `n` with 10 on each row.

2.3.6 `output_testing`

Arguments:

- `unsigned long long n`: Used to calculate length of primes array.
- `unsigned long long* primes`: Pointer to array of prime numbers.

Returns: void

Description: Prints all the primes from 2 to `n` matching the output format of the test file from `http://www.mathematical.com/primes0to1000k.html` so that we can run a diff on the two files.

2.4 Compilation and Usage

Compilation: `make all`
Usage: `./prog2 <n> <print> <reps>`

Both programs can be compiled and linked at the same time using the Makefile provided. To get rid of the executables in the folder, run the command `make clean`.

To use the program, we have a couple of command-line options to make it easier to test and experiment as we time our parallel and serial functions.

- `n` expects a value greater than or equal to 2. The program finds all the primes less than or equal to `n`.

- The `print` variable expects an input of 0 or 1 corresponding to 'suppress printing' or 'allow printing' respectively.
- The `reps` variable is so that we can run both the serial and parallel algorithms multiple times and get a meaningful time comparison. Expects a value greater than or equal to 1.

To run the program and print all primes less than 1 million:

```
./prog2 1000000 1 1
```

To time the program and suppress printing, bump up the number of repetitions and use 0 for the print variable:

```
./prog2 1000000 0 100
```

2.5 Testing and Verification

To verify the program once we had our solution, we looked for a list of primes online that didn't have a very challenging output formatting. This was so that we could adapt our print function with a few tweaks and run a diff on the two files.

We ended up picking the list on `http://www.mathematical.com/primes0to1000k.html`, creating one small file and one large file. The test files can be found in the testfiles/ directory. Files beginning with "primes*" are created from the website, "output*" are files created by our program. Running a diff on the pairs of files, the only difference should be the timing output at the end of the file.

```
7296837@linux101 program1 >>./prog1 1 1
0) 1010111110011001
0) 0110111110011001
0) 1110111110011001
2) 1010111110111001
2) 0110111110111001
4) 1010111111011001
4) 0110111111011001
4) 1110111111011001
2) 1110111110111001
Number of solutions = 9

39413) 1010111110011001
39414) 0110111110011001
39415) 1110111110011001
39925) 1010111111011001
39926) 0110111111011001
39927) 1110111111011001
40437) 1010111110111001
40438) 0110111110111001
40439) 1110111110111001
Number of solutions = 9

Parallel time 18.4957 ms
Serial time = 3.2901 ms
7296837@linux101 program1 >>
```

Figure 6: Solutions to the circuit

Once we verified that we had the right answer, we moved onto timing the two versions of the program, the parallel and serial. As you can see in 6, running the program just once doesn't really give us an accurate idea of how long each implementation takes. So, we built in a way to suppress the printing and increase the number of times the algorithm is run.

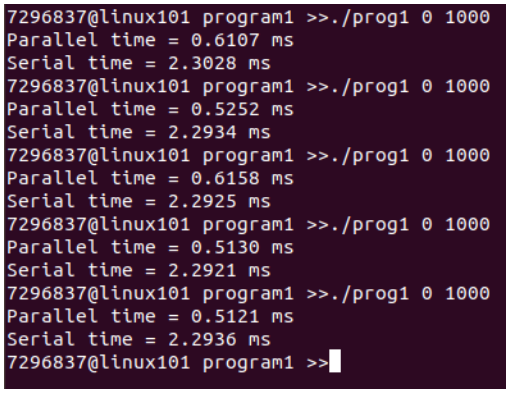


Figure 7: Comparing times for prog1

2.5.1 Scheduling

We tried using the OpenMP loop scheduler to see if we could better our performance. While the `schedule(static, 1)` showed a marginal improvement in performance, like we saw in class, `schedule(dynamic, 1)` does not do very well. This can be seen in figure 8.

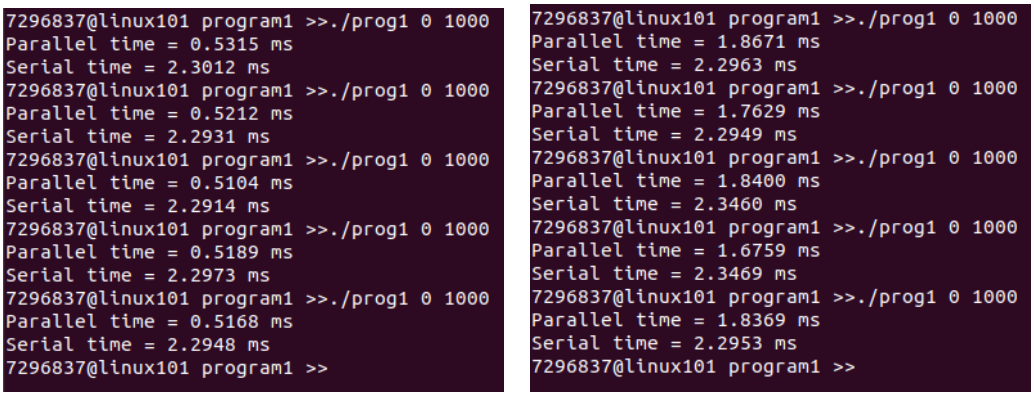


Figure 8: Comparison of the static and dynamic schedulers

It should be noted that the dynamic scheduler does achieve a comparable performance to the static scheduler when the chunk size is increased.

3 Files Submitted

- `prog1.pdf`
- `Makefile`
- `prog1.c`
- `prog2.c`
- `testfiles`