ARIZONA STATE UNIVERSITY

CSE 330, SLN 20748 — **Operating Systems** — Spring 2019

Instructor: Dr. Violet R. Syrotiuk

Project #1

Available 01/18/2019; milestone due 02/01/2019; full project due 02/18/2019

This project has two goals:

1. To apply **OpenMP** directives to develop a parallel program from a serial program implementing a utility that searches for a pattern described by a regular expression in one or more text files.
2. To use Agave, one of ASU's cluster machines, to run experiments to investigate the scalability of your parallel program.

This project is to be completed individually. It is expected that you will maintain a *private* repository of your code as your develop it for this project, using *e.g.*, `github`. You may be required to show us the commits you have made to your repository. You must write your code only in C or C++; it **must** compile and run on `general.asu.edu`. You may not alter the requirements of this project in any way.

1 Regular Expressions

Regular expressions have an important role in computer science applications. In applications involving text, users may want to search for strings that satisfy certain patterns. Regular expressions provide a method of describing such patterns. For example, there are utilities in operating systems, in programming languages, and in text editors that provide mechanisms for the description of patterns by using regular expressions. In this project you will write a utility to search for a pattern described by a regular expression one or more text files.

In arithmetic, we use the operations of addition ($+$) and multiplication ($\times$) to build up expressions such as $(5 + 3) \times 4$. The value of this arithmetic expression is the integer 32.

Similarly, we can use operations to build expressions describing languages. These are called regular expressions. An example is $(0 \cup 1)0^*$.

The value of a regular expression is a language. In this example, the value is the language consisting of all strings starting with a zero or a one followed by any number of zeros. The symbols 0 and 1 are short for the sets $\{0\}$ and $\{1\}$. Therefore the first part of the expression $(0 \cup 1)$ means $(\{0\} \cup \{1\})$. The value of this part of the language is $\{0, 1\}$. The second part of the expression, 0^* means $\{0\}^*$ and its value is the language consisting of all strings containing any number of zeros.

Concatenation is often implicit in regular expressions. If $\circ$ is the concatenation symbol, then $(0 \cup 1)0^*$ is short for $(0 \cup 1) \circ 0^*$. Concatenation attaches the strings from the two parts of the expression to obtain the value of the entire expression.

In arithmetic, $\times$ has precedence over $+$ meaning that when there is a choice, the $\times$ operation is done first. Thus in $2 + 3 \times 4$, the 3×4 is done before the addition. To have the addition done first, parentheses must be added as in $(2 + 3) \times 4$. In regular expressions, the star $(*)$ operation has highest precedence, followed by concatenation, and finally union, unless parentheses change the usual order.

1.1 Formal Definition of a Regular Expression

Here, we provide a formal inductive definition of a regular expression, *i.e.*, it defines regular expressions in terms of smaller regular expressions.

R is a *regular expression* if R is:

1. a for some a in the alphabet Σ,
2. ϵ,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. $(R_1{}^*)$, where R_1 is a regular expression.

In this definition, the regular expressions a and ϵ in items 1 and 2 represent the languages $\{a\}$ and $\{\epsilon\}$, respectively. In item 3, the regular expression $\emptyset$ represents the empty language. Do not to confuse the regular expressions ϵ and $\emptyset$. The expression ϵ represents the language containing a single string, namely the empty string. The expression $\emptyset$ represents the empty language, *i.e.*, the language that does not contain any strings.

In items 4 and 5 of the definition, the expressions represent the languages obtained by taking the union or concatenation of the languages R_1 and R_2, respectively. Union and concatenation are binary operations. Union takes all the strings in both R_1 and R_2 and puts them together in one language. Concatenation attaches a string from R_1 in front of a string from R_2 in all possible ways to get strings in the new language.

Finally, in item 6, the expression represents the star of the language R_1. Star is a unary operation. It works by attaching any number of strings in R_1 to get a string in the new language. Because "any number" includes zero as a possibility, the empty string ϵ is always a member of $R_1{}^*$ no matter what R_1 is.

Parentheses in an expression may be omitted. If they are, then evaluation follows precedence order: star, then concatenation, and then union.

When we want to distinguish between a regular expression R and the language it describes, we write $L(R)$ to denote the language of R. A language is regular if and only if some regular expression describes it.

2 Program Requirements for Project #1

In this project, you are required to write **two** programs to search for a pattern described by a regular expression in one or more text files:

1. The first is a serial program; this is required by the milestone deadline of 02/01/2019. That is, you are to write a *serial* C/C++ program named `search` with the following command line arguments:

   `search` ⟨`pattern`⟩ ⟨`list of text file names`⟩

   where ⟨`pattern`⟩ is a regular expression in double quotes in the format described in §2.1, and where ⟨`list of text file names`⟩ is a space separated list of text file names. For each file in this list, the file contents are to be searched for all occurrences matching the ⟨`pattern`⟩ specified. See §2.3 for the required output format.

2. A parallel (OpenMP) program, developed from your serial `search` program by the full project deadline of 02/18/2019. That is, you are to write a *parallel* C/C++ program named `psearch` taking the same command line arguments and producing the same output as `search`.

You may assume that the regular expression given for the ⟨`pattern`⟩ is syntactically correct, and that the files in the ⟨`list of text file names`⟩ to search exist, and are text files.

Both programs **must** compile and run on `general.asu.edu`, a Linux machine. ASU's Agave cluster machine is also a Linux machine with the same compilers as `general.asu.edu`. While you may develop your programs in another environment, you are responsible for them compiling and running on `general.asu.edu` because that is where your programs will be compiled and tested.

2.1 Regular Expressions in Practice

For this project, we use $\Sigma = \{\texttt{a}, \texttt{b}, \ldots, \texttt{z}, \texttt{A}, \texttt{B}, \ldots, \texttt{Z}, \texttt{1}, \texttt{2}, \ldots, \texttt{9}, \texttt{.}\}$. That is, the alphabet consists of lower and upper case alphabetic characters, the digits one through nine, and a period. Because the operators are not available on a typical keyboard, we'll use `+` for the binary union ($\cup$) operator, `*` for the unary star operator (*), and assume concatenation is implicit, *i.e.*, rather than $R_1 \circ R_2$ we'll write $R_1 R_2$. Again if parentheses are omitted, then evaluation follows precedence order: star, then concatenation, and then union.

For example, the regular expression *abc* is represented by the pattern `"abc"`. In this case, the corresponding language contains a single string, namely the concatenation of the characters `a`, `b`, and `c` into the string `abc`. The regular expression $abc \cup def$ is represented by the pattern `"abc+def"`. Because concatenation has higher precedence than union, the corresponding language contains two strings, namely {`abc`, `def`}. Finally, the regular expression $(a \cup b)^* aba$ is represented by the pattern `"(a+b)*aba"`. Fully parenthesized, this would be given as `((a+b)*)(((a)(b))(a))`, so you can see why we omit parentheses unless needed to express the language of interest. The language corresponding to this regular expression is infinite; it contains any sequence of `a`'s or `b`'s ending with `aba`. That is, the language contains the strings

$$\{\texttt{aba}, \texttt{aaba}, \texttt{baba}, \texttt{aaaba}, \texttt{ababa}, \texttt{bbaba}, \texttt{aaaaba}, \texttt{aababa}, \texttt{abaaba}, \texttt{abbaba}, \ldots\}.$$

In practice, our search utility is not concerned with regular expressions that include ϵ or $\emptyset$. Whenever a regular expression is provided, *do not report matches on the empty string.*

2.2 Implementation Approach

If you are comfortable with the definition of a regular expression and understand how to convert one to a finite automaton, you can jump directly to step 3 below. Otherwise, I suggest that you work on the project incrementally, as follows.

1. First, work on recognizing regular expressions consisting only of concatenation, *i.e.*, the ⟨`pattern`⟩ is a single string. In this case, your program reports matches of all occurrences to the string in the language in each file; see §2.3 for the output format required.
2. Then incorporate the binary union operator. That is, extend your project to support a ⟨`pattern`⟩ expressing the union of two or more strings. In this case, your program reports matches of all occurrences of any string in the (finite) set of strings in the language in each file; see §2.3 for the output format required.

3. Finally, incorporate the unary star operator. It is known that regular expressions and finite automata are equivalent in their descriptive power. Any regular expression can be converted into a finite automaton that recognizes the language it describes. You can convert the regular expression in two steps, first to a *non-deterministic finite automaton* (NFA), and then to a *deterministic finite automaton* (DFA). The algorithms for these steps are well known; see [1, 2, 3], as examples.

To handle regular expressions that contain parentheses, you will need to parse the expression. This can be done easily using simple data structures such as a queue and stack, or using recursion; one of Dijkstra's algorithms can be adapted for this purpose; see, *e.g.*, [4].

2.3 Output Format

For each file in the list, your programs must produce output indicating the start and end positions on each line of the file that match the ⟨`pattern`⟩ *on standard output. Do not write your output to a file!* The nice thing about writing to `stdout` is that it can be redirected to a file.

Specifically, your program must produce output of the form:

⟨`file name`⟩, ⟨`line number`⟩, ⟨`start column`⟩, ⟨`end column`⟩: ⟨`string matched`⟩

For example, suppose your program is invoked as follows:

```
search "re" haiku.txt
```

and the contents of the file `haiku.txt` has the poem:

```
Chaos reigns within.
Reflect, repent, and reboot.
Order shall return.
```

Then `search` should produce the output:

```
haiku.txt, 1, 7, 8: re
haiku.txt, 2, 10, 11: re
haiku.txt, 2, 22, 23: re
haiku.txt, 3, 13, 14: re
```

If the pattern were instead `"re+er"` then the output produced would also include:

```
haiku.txt, 3, 4, 5: er
```

Finally, if the pattern were `"o*"` then the output produced should be:

```
haiku.txt, 1, 4, 4: o
haiku.txt, 2, 25, 26: oo
```

While your serial program will naturally produce output in a deterministic manner, be aware that the output of the threads of your parallel program may be interleaved in a non-deterministic manner.

3 Experimentation on Agave, an ASU Cluster

An account on Agave, one of ASU's cluster machines for students has been created for you. Detailed instructions on using the cluster will be covered in a recitation.

1. Copy (*e.g.*, using `sftp`) both your serial and parallel C/C++ programs onto an Agave *login node*; compile them on a *login node*.
2. Copy a selection of text files to use as input to your search onto the cluster.
3. Submit a series of jobs to verify that patterns of increasing complexity and numbers of files are matched on one of the *compute nodes*. Collect the run time of your serial and parallel program for increasing numbers of cores. Be sure to use the same type of hardware for serial and parallel programs on the same parameters!
4. Plot the run time of your serial and parallel programs against least two different independent variables, *e.g.*, you could consider plotting run time as a function of the number of text files searched, or as a function of the number of cores.
5. 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) Can you determine circumstances needed before a speed-up is observed?

4 Submission Instructions

This project has two submission deadlines.

1. The milestone deadline is 11:59pm on Friday, 02/01/2019.
2. The full project deadline is 11:59pm on Monday, 02/18/2019.

NO LATE SUBMISSIONS ARE ACCEPTED. An unlimited number of submissions is allowed so submit early, and submit often. (Your last submission is the one that will be graded.)

4.1 Submission Instructions for Milestone Deadline

Submit electronically, before 11:59pm on Friday, 02/01/2019 using the submission link on Blackboard for the Project #1 Milestone, a zip[1] file named `yourFirstName-yourLastName.zip` containing your milestone solution.

Your zip must must unzip into two folders: `Documents` and `Serial` that contain the following items and nothing more:

Design (10%): In the folder named `Documents`, provide a PDF document in which you:

1. Discuss your selection of data structures and algorithms for matching a regular expression within a text file.
2. Discuss the status of your program, *e.g.*, which operators can you support, do you know of any bugs?
3. While this project is to be completed individually, describe any significant interactions with anyone (peers or otherwise) that may have occurred.
4. Cite any external books, and/or websites used or referenced.

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

Implementation of Serial Program (40%): In a folder named `Serial`, provide:

1. The files making up your documented C/C++ source code for your serial program.
2. A `makefile` to compile your program and generate an executable named `search` in the `Serial` folder. It is expected that your `search` executable can be called in a script to test the correctness of your program.

Correctness (50%): Our TA will evaluate the correctness of your serial program on `general.asu.edu`, so your program must compile and execute there. If you implement concatenation only you can earn up to 70% of these marks, if you implement concatenation and union you can earn up to 80% of these marks, and if you implement all operators you can earn up to 100% of the marks.

Sample input files, `makefile`s, and scripts will be provided in recitations.
The milestone is worth 30% of the total Project #1 grade.

4.2 Submission Instructions for Full Project Deadline

Submit electronically, before 11:59pm on Monday, 02/18/2019 using the submission link on Blackboard for the full Project #1, a zip file named `yourFirstName-yourLastName.zip` containing your solution to the full project; see footnote 1.

Your zip must must unzip into three folders: `Documents`, `Serial`, and `Parallel` that contain the following items and nothing more:

Design and Analysis (25%): In the folder named `Documents`, provide a PDF document that describes of the methodology you followed to produce a correct parallel version of your serial program. Specifically:

1. Discuss your strategy for introducing OpenMP directives to parallelize your serial program.
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, such as profiling. Did you change any data structures or program structure as a result?
4. Describe your strategy of experimentation and provide graphical results plotting the run time of your serial and parallel programs for *e.g.*, patterns of increasing complexity and numbers of input files. Be sure to label the axes of your graphs, and provide the parameters used in experimentation (*e.g.*, cluster used, number of cores, etc.).
5. Answer the questions posed in Step 5 of §3.
6. While this project is to be completed individually, describe any significant interactions with anyone (peers or otherwise) that may have occurred.
7. Cite any external books, and/or websites used or referenced.

Implementation (25%): Provide the following:

1. In a folder named `Serial`, provide the files making up your documented C/C++ source code for your serial program. In addition, provide a `makefile` to compile your serial program, storing the executable in a file named `search` in this directory. (This serial program may be the same as or different from your milestone submission depending on the status of your milestone submission.)

2. In a folder named `Parallel` provide the files making up your documented C/C++ source code for the parallel program you evolved from your serial program. In addition, provide a `makefile` to compile your parallel program, storing the executable in a file named *psearch* in this directory.

Correctness (50%) Your serial and parallel programs **must** compile and run on `general.asu.edu` and on the Agave cluster. Our TA will test them there.

5 Honours Project Credit

Assuming there are some students who want to do an honours project in this course, each project in this course will have a section entitled "Honours Project Credit." If you complete all three, I will give you honours project credit, assuming an honours contract exists.

For Project #1, the honours project is to allow the list of files to be searched to be specified as a regular expression. That is, your serial and parallel programs should take the following command line arguments:

`search` ⟨`search-string pattern`⟩ ⟨`file-name pattern`⟩
`psearch` ⟨`search-string pattern`⟩ ⟨`file-name pattern`⟩

where both the ⟨`search-string pattern`⟩ and the ⟨`file-name pattern`⟩ are specified by the regular expression for ⟨`pattern`⟩ in double quotes as described in §2.1.

To determine the list of files to search, you should first obtain a list of files in the current directory. Then, search each file that matches the ⟨`file-name pattern`⟩ for strings that match the ⟨`search-string pattern`⟩.

For example, if you invoke `search` with two patterns:

```
search "*.h" "test*.cc"
```

then your program should search the contents of files that match any file name in the current directory that have a prefix of `test` concatenated with any number of characters in Σ appended by `.cc`, for include files with the search pattern.

Prior to producing your search output, you should produce a list of the files that match the ⟨`file-name pattern`⟩.

References

[1] Michael Sipser, *Introduction to the Theory of Computation*, Third Edition, Cengage Learning, 2012, (ISBN-13: 978-1133187790).

[2] James Power, *Converting a regular expression to a NFA - Thompson's Algorithm*, http://www.cs.may.ie/staff/jpower/Courses/Previous/parsing/node5.html.

[3] James Power, *Constructing a DFA from an NFA ("Subset Construction")*, http://www.cs.may.ie/staff/jpower/Courses/Previous/parsing/node9.html.

[4] *Shunting-yard Algorithm*, https://en.wikipedia.org/wiki/Shunting-yard_algorithm.