PARALLEL - PROJECT 1

STAVRINOS KOSTOPOULOS, PARASKEUAS KOSTA

01/12/21

—

PARALLEL X DISTRIBUTED COMPUTING

—

Harald Gjermundrød

- Project 1 (Multiple Parallel and Concurrent Programs)
- Team
    - Stavrinos Kostopoulos
    - Paraskeuas Kosta
- Development Environment
    - For Program 1 and 3 -> USED ECLIPSE IDE / JAVA
    - For Program 2 -> USED VISUAL STUDIO IDE / C++
- Libraries
    - SHOWN IN THE SCREENSHOTS BELOW, PROVIDED WITH THE CODES AND COMMENTS!
- Work Divided
    - Stavrinos Kostopoulos -> Programs 1 and 2
        - Includes Comments and html files (JavaDoc for java, Doxygen for c++)
    - Paraskeuas Kosta -> Program 3
        - Includes Comments and html files (JavaDoc for java)
    - Used GitHub as a cloud service for collaboration/feedback among team

Program 1(a) - (Parallel Fork/Join Mechanism)

Snipped Code

- **Compute()**

Exe1_Fork_Join_Add.java

```java
    /**
     * Function <code>compute</code> is the computing function
     * equivalent to the run() of divide and conquer,
     * which is be responsible for solving the problem directly or
     * by executing the task in parallel
     * @return Returns sum of first and last tasks (meaning two threads)
     */
    protected Long compute()
    {

        int length_l = ending - starting;

        if (length_l <= threshold_t)
        {
            return add();
        }

        //like thread A
        Exe1_Fork_Join_Add firstTask = new Exe1_Fork_Join_Add(numbers_n, starting, starting + length_l / 2);

        //starts asynchronously
        firstTask.fork();

        //like thread B
        Exe1_Fork_Join_Add secondTask = new Exe1_Fork_Join_Add(numbers_n, starting + length_l / 2, ending);

        Long secondTaskResult = secondTask.compute();
        Long firstTaskResult = firstTask.join();

        return firstTaskResult + secondTaskResult;

    }
```

- **Main Driver()**

```java
    /**
     * Function <code>main</code> is the driver which jump-starts the
     * application
     * @param args The default command line arguments
     */
    public static void main(String[] args)
    {
         System.out.println("The summation of the numbers from 1 to 3 million IN PARALLEL is: ");
         System.out.println(Exe1_Fork_Join_Add.startForkJoinSum(3_000_000));

    }

}
```

- **Output**

```
The summation of the numbers from 1 to 3 million IN PARALLEL is:
4500001500000
```

- **Packets / Libraries Used**

```
13 //imports packets
14 import java.util.concurrent.ForkJoinPool;
15 import java.util.concurrent.ForkJoinTask;
16 import java.util.concurrent.RecursiveTask;
17 import java.util.stream.LongStream;
```

Code Explanation:

- **Summary:**
  It is a Parallel program using fork / join, that sums all the numbers from 1 to 3 million.

- **In short, I firstly,**
  - **start forking thread A (first task) asynchronously,**
  - **THEN I compute thread B (second task) by initialising it afterwards as result of second task**
  - **and finally joining the first task by also initialising it as a result of first task**
  - **AND at the very end I add them together (the two task results).**

- **Explanation ALSO written in DETAIL AS COMMENTS on source codes provided with**

Program 1(b) - (Parallel Fork/Join Mechanism) – *EXTRA*

Snipped Code

- **Compute()**

Exe1_Fork_Join_Add.java | Exe1_Fork_Join_Fibonacci.java

```
50      /**
51       * Function <code>compute</code> is the computing function
52       * equivalent to the run() of divide and conquer,
53       * which is be responsible for solving the problem directly or
54       * by executing the task in parallel
55       * @return Returns number_n (n) as a sum of f1 and f2 numbers (meaning the two threads)
56       */
57      protected void compute()
58      {
59          // n number
60          long n = number_n;
61
62          if (n <= threshold_t)
63          {
64              //fibonacci function kicks in to do its job
65              number_n = fibbonacci(n);
66          }
67
68          else
69
70          {
71              //like thread A
72              Exe1_Fork_Join_Fibonacci f1 = new Exe1_Fork_Join_Fibonacci(n - 1);
73
74              //like thread B
75              Exe1_Fork_Join_Fibonacci f2 = new Exe1_Fork_Join_Fibonacci(n - 2);
76
77              ForkJoinTask.invokeAll(f1, f2);
78              number_n = f1.number_n + f2.number_n;
79          }
80      }
```

- **Main Driver()**

```
104     /**
105      * Function <code>main</code> is the driver which jump-starts the
106      * application
107      * @param args The default command line arguments
108      */
109     public static void main(String[] args)
110     {
111
112         Exe1_Fork_Join_Fibonacci test = new Exe1_Fork_Join_Fibonacci(30);
113         new ForkJoinPool().invoke(test);
114         System.out.println("The value of the Fibonacci nth number provided, which is " + test.number_n + ", is indeed:");
115         System.out.println(test.get_Number());
116
117     }
118
119 }
```

- **Output**

```
The value of the Fibonacci nth number provided, which is 832040, is indeed:
832040
```

- **Packets / Libraries Used**

```
13 //import Packets
14 import java.util.concurrent.ForkJoinPool;
15 import java.util.concurrent.ForkJoinTask;
16 import java.util.concurrent.RecursiveAction;
17
```

Code Explanation:

- **Summary:**
  It is a Parallel program using fork / join, that finds the value of the nth Fibonacci number (user can change it in the main driver)

- **In short, I firstly,**
  - **create two threads representing the two Fibonacci numbers (f1, f2), usually used**
  - **Then I initialise the n input (number given by the user) with the fibonacci function that takes the n input and does the following -> once number (n) provided, will sum the values of the -1 AND -2 positions of it (of the number provided)**
  - **Finally, add the two threads created together**

- **Explanation ALSO written in DETAIL AS COMMENTS on source codes provided with**

Program 2 - (Concurrency with need of 2 condition vars)

Snipped Code

- **The function where all the work is being done**

```
/**
 * Function <code>consumer_Function</code> is the consumers function.
 * <BR>
 */
void consumer_Function()
{
    //Function that consumes data forever unless you manually kill it
    while (true)
    {
        int number_To_Process = 0;

        /*Only need to lock mutex for the time it takes us to pop item out.
        Adding this scope, it releases the lock right after item is popped*/
        {
            /*Condition variables need unique_lock rather  a lock_guard, cause
            mutex might be locked and unlocked numerous times. By default, this
            line will lock the mutex*/
            std::unique_lock<std::mutex> g(queue_Mutex);
            std::unique_lock<std::mutex> g2(queue_Mutex2);

            /*This call to `wait` will firstly check if contions are met.
            For instance, If queue is not empty. Now, If queue not empty, the execution of code will continue onwards
            If queue is empty, it will unlock the mutex and wait until a signal is sent to the condition variables.
            When signal sent, it will acquire the lock and check the conditions again!
            Also, WE DO NEED BOTH OF THEM, OTHERWISE IT WILL CRASH*/
            queue_Condition_Variable.wait(g, [] { return !dataQueue.empty(); });
            queue_Condition_Variable2.wait(g2, [] { return !dataQueue.empty(); });

            /*Don't need to check if queue is empty anymore,
            because the Condition Variables do that for us*/
            number_To_Process = dataQueue.front();
            dataQueue.pop();
        }

        //Proceeds onwards ONLY if numbers are available
        if (number_To_Process)
        {
            std::cout << "Now Processing Number: " << number_To_Process << std::endl;
        }
    }

    //We don't have to test the post condition because it's a void funtion, which is not returning something
}
```

- **Main Driver**

```
109     //Main Driver
110     /**
111      * Function <code>main</code> is the driver function that starts the programme.
112      * <BR>
113      * @return Returns <code>0</code> on success, any other value otherwise.
114      */
115     int main()
116     {
117         std::thread producer_1(producer_Function);
118         std::thread consumer_1(consumer_Function);
119
120         producer_1.join();
121         consumer_1.join();
122
123         system("pause");
124         return 0;
125     }
```

- **Output – Need human interaction to be stopped**
- **Needs to be manually killed by the user**

```
--------------------------------------------------------------------------------
PARALLEL COMPUTING - PROJECT 1 / PROGRAM 2 - CONCURRENT PROGRAMMING USING CONDITION VARIABLES
--------------------------------------------------------------------------------

Added Numbers To Queue Are The Following Ones: 68, 35
Now Processing Number: 68
Now Processing Number: 35

Added Numbers To Queue Are The Following Ones: 70, 25
Now Processing Number: 70
Now Processing Number: 25

Added Numbers To Queue Are The Following Ones: 59, 63
Now Processing Number: 59
Now Processing Number: 63

Added Numbers To Queue Are The Following Ones: 6, 46
Now Processing Number: 6
Now Processing Number: 46

Added Numbers To Queue Are The Following Ones: 28, 62
Now Processing Number: 28
Now Processing Number: 62

Added Numbers To Queue Are The Following Ones: 96, 43
Now Processing Number: 96
Now Processing Number: 43

Added Numbers To Queue Are The Following Ones: 37, 92
Now Processing Number: 37
Now Processing Number: 92

Added Numbers To Queue Are The Following Ones: 3, 54
Now Processing Number: 3
Now Processing Number: 54

Added Numbers To Queue Are The Following Ones: 83, 22
Now Processing Number: 83
Now Processing Number: 22

Added Numbers To Queue Are The Following Ones: 19, 96
Now Processing Number: 19
Now Processing Number: 96

Added Numbers To Queue Are The Following Ones: 27, 72
Now Processing Number: 27
Now Processing Number: 72
```

- **Packets / Libraries / Initialisations Used**

```
11  #include <queue>
12  #include <thread>
13  #include <iostream>
14  #include <mutex>
15  #include <condition_variable>
16
17  using namespace std;
18
19  std::queue<int> dataQueue;
20  std::mutex queue_Mutex;
21  std::mutex queue_Mutex2;
22  std::condition_variable queue_Condition_Variable;
23  std::condition_variable queue_Condition_Variable2;
24
```

Code Explanation (Used A Queue):

- **Summary: It's a concurrent program that uses two condition variables, that keeps generating data forever, unless you manually kill it through the terminal of yours**

- **In short, I firstly,**
  - **Initialise the num to process = 0**
  - **THEN, create a unique lock for my two mutexes, meaning locking them and which will be held by my 2 condition variables later**
  - **Then I call 'wait' to check if conditions are met (which "wait" includes the corresponding mutex, array and returns a not empty dataQueue), by doing it for both conditions variables**
  - **Afterwards, I initialise the number to process with the front element of the queue and then popping!**

- **Explanation ALSO written in DETAIL AS COMMENTS on source codes provided with**

Program 3 - (Sudoku Solver Checker) – Paraskeuas Kosta

Snipped Code

- **Codes Used to Implement Problem (5 Total)**
- **Provided IN ZIP (WHOLE)**
- **Uses Hardcode Solution / User can change it in the IDE**

Rows_Columns_Object.java | SudokuTest.java | SudokuValid.java | ValidCols.java | ValidRows.java

```java
/** \file      Rows_Columns_Object.java
*   \brief     Sudoku Solver
*   \details   Program that checks if the Sudoku answer is correct using threads
*   \author    Stavrinos Kostopoulos, Paraskeuas Kosta
*   \version   0.3
*   \date      2021-2022
*   \bug       No bugs so far
*   \copyright Stavrinos Kostopoulos, Paraskeuas Kosta (University of Nicosia) - 4th Year Course Project
*/

/**
 * Class <code>Rows_Columns_Object</code>
 * Description -> initiasizing our columns and rows
 * @author Stavrinos Kostopoulos
 * @author Paraskeuas Kosta
 */
public class Rows_Columns_Object extends SudokuTest
{

    int row_r;
    int col_c;

   /**
    * Default constructor
    * general object that will be extended by worker thread objects, only contains
    * @param rows_r row that is relevant to the thread
    * @param cols_c column that is relevant to the thread
    */
    Rows_Columns_Object(int rows_r, int cols_c)
    {
        this.row_r = rows_r;
        this.col_c = cols_c;
    }
```

- **Output – Valid Input Given by User**

Rows_Columns_Object.java | SudokuTest.java | SudokuValid.java | ValidCols.java | ValidRows.java

```java
/** \file      SudokuTest.java
 *
 */

/**
 * Class <code>SudokuTest</code> is the driver for our program
 * Description -> driver for a Sudoku Solver using threads to check for a correct (valid) answer
 * @author Stavrinos Kostopoulos
 * @author Paraskeuas Kosta
 */
public class SudokuTest
{

    // global constant for the total threads used
    private static final int THREAD_NUM = 27;

    //create the sudoku table inputs
    public static final int[][] sudokuTable =
        {
                { 6, 2, 4, 5, 3, 9, 1, 8, 7 },
                { 5, 1, 9, 7, 2, 8, 6, 3, 4 },
                { 8, 3, 7, 6, 1, 4, 2, 9, 5 },
                { 1, 4, 3, 8, 6, 5, 7, 2, 9 },
                { 9, 5, 8, 2, 4, 7, 3, 6, 1 },
                { 7, 6, 2, 3, 9, 1, 4, 5, 8 },
                { 3, 7, 1, 9, 5, 6, 8, 4, 2 },
                { 4, 9, 6, 1, 8, 2, 5, 7, 3 },
                { 2, 8, 5, 4, 7, 3, 9, 1, 6 }
        };

    //array where threads will update at run time
    public static boolean[] validation;

```

```
The Sudoku Provided By The User It Is INDEED Valid!
```

- **Output – Invalid Input Given by User**

Rows_Columns_Object.java | SudokuTest.java | SudokuValid.java | ValidCols.java | ValidRows.java

```java
/** \file      SudokuTest.java

/**
 * Class <code>SudokuTest</code> is the driver for our program
 * Description -> driver for a Sudoku Solver using threads to check for a correct (valid) answer
 * @author Stavrinos Kostopoulos
 * @author Paraskeuas Kosta
 */
public class SudokuTest
{

    // global constant for the total threads used
    private static final int THREAD_NUM = 27;

    //create the sudoku table inputs
    public static final int[][] sudokuTable =
        {
                { 6, 2, 4, 5, 3, 9, 1, 8, 7 },
                { 5, 1, 9, 7, 2, 8, 6, 3, 4 },
                { 8, 3, 7, 6, 1, 4, 2, 9, 5 },
                { 1, 4, 3, 8, 6, 5, 7, 2, 9 },
                { 9, 5, 8, 2, 9, 7, 3, 6, 1 },
                { 7, 6, 2, 3, 9, 1, 4, 5, 8 },
                { 3, 7, 1, 9, 5, 6, 8, 4, 2 },
                { 4, 9, 6, 1, 8, 2, 5, 7, 3 },
                { 2, 8, 5, 4, 7, 3, 9, 1, 6 }
        };

```

```
The Sudoku Provided By The User It Is INDEED Invalid!
```

- **Checking Validity (Same notion for rows and columns java source code, provided with)**

Rows_Columns_Object.java | SudokuTest.java | SudokuValid.java | ValidCols.java | ValidRows.java

```java
    public void run()
    {

        if (row_r > 6 || row_r % 3 != 0 || col_c > 6 || col_c % 3 != 0)
        {
            System.out.println("Invalid row or column for subsection!");
            return;
        }

        //check for the numbers 1 - 9, if they only appear once in 3x3 (sudoku) subsection
        boolean[] validArray = new boolean[9];

        for (int i = row_r; i < row_r + 3; i++)
        {
            for (int j = col_c; j < col_c + 3; j++)
            {
                int num = sudokuTable[i][j];

                if (num < 1 || num > 9 || validArray[num - 1])
                {
                    return;
                }

                else

                {
                    validArray[num - 1] = true;
                }
            }
        }

        //if we reach here then 3x3 (sudoku) is indeed valid
        validation[row_r + col_c / 3] = true;
    }
```

Code Explanation:

- **Summary:**
  It is program that checks whether the sudoku solution provided by the user is valid on not

- **Explanation ALSO written in DETAIL AS COMMENTS on source codes provided with**

- **In short (Explanation through code):**

```
47    public static void main(String[] args)
48    {
49        validation = new boolean[THREAD_NUM];
50        Thread[] thread_t = new Thread[THREAD_NUM];
51        int threadNum = 0;
52
53        //create 9 threads for 9 3x3, 9 threads for 9 columns and 9 threads for 9 rows
54        for (int i = 0; i < 9; i++)
55        {
56            for (int j = 0; j < 9; j++)
57            {
58                if (i % 3 == 0 && j % 3 == 0)
59                {
60                    thread_t[threadNum++] = new Thread(new SudokuValid(i, j));
61                }
62
63                if (i == 0)
64                {
65                    thread_t[threadNum++] = new Thread(new ValidCols(i, j));
66                }
67
68                if (j == 0)
69                {
70                    thread_t[threadNum++] = new Thread(new ValidRows(i, j));
71                }
72            }
73        }
74
75        //start all threads
76        for (int i = 0; i < thread_t.length; i++)
77        {
78            thread_t[i].start();
79        }
80
```

```
81          //wait for all threads to finish
82          for (int i = 0; i < thread_t.length; i++)
83          {
84              try
85              {
86                  thread_t[i].join();
87              }
88
89              catch (InterruptedException e)
90
91              {
92                  e.printStackTrace();
93              }
94          }
95
96          for (int i = 0; i < validation.length; i++)
97          {
98              if (!validation[i])
99              {
100                 //System.out.println(sudokuTable);
101                 System.out.println("The Sudoku Provided By The User It Is INDEED Invalid!");
102                 return;
103             }
104         }
105
106         //System.out.println(sudokuTable);
107         System.out.println("The Sudoku Provided By The User It Is INDEED Valid!");
108     }
109 }
```

- **OVERALL CONCLUSION:**
  - **Fork / join parallelism does make things easier**
  - **Concurrency its very helpful when it comes to situations where codes need to be broken into multiple pieces (parts)**
  - **Sudoku solver uses the start / join mechanism related to the “divide and conquer” one, shown in chapter 3 - 4**

- **ALL CODES X HTMLS ARE PROVIDED IN ZIP FILE UPLOADED ON MOODLE**

THE END