Computer Science

UDC 004.02

Oliynyk Olena
Senior Lecturer of Software Engineering Department
Kharkiv National University of Radioelectronics

Mykhnevych Dmytro
Student of the
Kharkiv National University of Radioelectronics

PARALLELISM AND CONCURRENCY IN GOLANG. COMPARISON WITH OTHER PROGRAMMING LANGUAGES (C#, JAVA, C)

Summary. *Comparison of computational complexity of parallel programs written on Go, C#, Java, C.*

Key words: *parallel programming, Go, goroutines.*

The amount of data that needs to be processed is increasing every day. It is almost impossible to imagine software today that does not use some form of parallelism or concurrency. The article will talk about the comparison of the efficiency of executing parallel programs in the Go programming language, which was created for the development of multi-threaded programs, with other programming languages such as C #, Java, C.

Go is an open-source programming language developed by the Google team. Go tries to combine the simplicity of dynamically typed languages such as Python, fast compilation, and garbage collection.

This article will describe several common practical problems that will be solved using multithreading or asynchrony. Installation guide and link to the full code are also included.

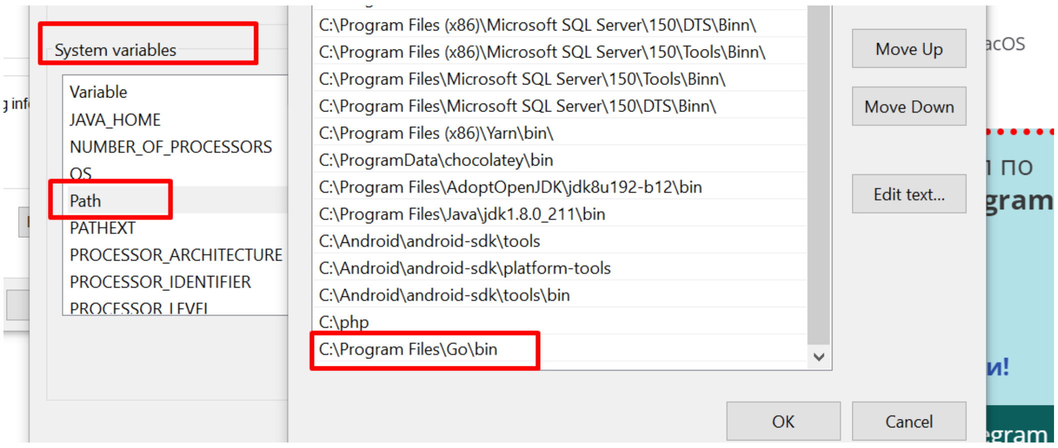
1. SETTING UP THE ENVIRONMENT

To run the code in the Go programming language, you need to do the following:

1. Since Go often uses open source repositories, you need to install Git on your computer (https://git-scm.com/download/win).

2. Download the Go compiler by selecting the appropriate version (https://golang.org/doc/install). To verify the successful compiler installation, you can open a command prompt and enter the "`go version`" command.

3. Check or add the appropriate system environment variable (see Fig. 1.):


System variables

Variable
JAVA_HOME
NUMBER_OF_PROCESSORS
OS
Path
PATHEXT
PROCESSOR_ARCHITECTURE
PROCESSOR_IDENTIFIER
PROCESSOR_LEVEL

```
C:\Program Files (x86)\Microsoft SQL Server\150\DTS\Binn\
C:\Program Files (x86)\Microsoft SQL Server\150\Tools\Binn\
C:\Program Files\Microsoft SQL Server\150\Tools\Binn\
C:\Program Files\Microsoft SQL Server\150\DTS\Binn\
C:\Program Files (x86)\Yarn\bin\
C:\ProgramData\chocolatey\bin
C:\Program Files\AdoptOpenJDK\jdk8u192-b12\bin
C:\Program Files\Java\jdk1.8.0_211\bin
C:\Android\android-sdk\tools
C:\Android\android-sdk\platform-tools
C:\Android\android-sdk\tools\bin
C:\php
C:\Program Files\Go\bin
```

Fig. 1. Setting the environment variable

4. Download IDE for Go programming language. JetBrains' GoLand development environment is the best option for comfortable Go programming. You can download the IDE at https://www.jetbrains.com/ru-ru/go/download/#section=windows.

2. COMPARISONS OF PARALLEL PROGRAMS

2.1 Finding the maximum number in a matrix

Formulation of the problem: find the maximum number in the matrix of size 100 * 1000000. To compare the performance of the Go programming language, implement the solution of the corresponding problem in the Java and C programming languages. Both solutions must use parallelism, concurrency, or asynchrony.

2.1.1 Go implementation

First, we create a 100 * 1000000 matrix and fill it with random numbers. Then the generated matrix is passed to the findMaxValue function, which in turn does the following:

1. Creates a buffered channel into which the maximum elements of each row of the matrix will be written. Creates a sync.WaitGroup structure and adds matrixSearcher goroutines there, the number of which is equal to the number of rows in the matrix. At the same time, it launches these goroutines, passing the current matrix row and a channel for recording the results into them.

2. Launches another goroutine that waits for the previous goroutines to complete and closes the channel for results so that there is no deadlock when reading from the channel in the main function.

3. From the obtained results chooses the maximum and returns it.

The Go implementation is shown below:

```
func findMaxValue(matrix [][]int) int {
rowsAmount := len(matrix)
wg := new(sync.WaitGroup)
results := make(chan int, rowsAmount)
for i := 0; i < rowsAmount; i++ {
	wg.Add(1)
	go matrixSearcher(matrix[i], wg, results)
}
go func() {
	wg.Wait()
```

```
            close(results)
      }()
      max := math.MinInt32
      for val := range results {
            if val > max {
                  max = val
            }
      }
      return max
      }

      func matrixSearcher(row []int, wg *sync.WaitGroup, resultChannel chan
int) {
      defer wg.Done()
      max := math.MinInt32
      for i := range row {
            if row[i] > max {
                  max = row[i]
            }
      }
      resultChannel <- max
      }
```

2.1.2 Java implementation

Java implementation is similar to Go implementation but uses a different means of parallel execution of the program, namely the ExecutorService, which creates a specified number of futures, which will then execute the given list of tasks. The task is to find the maximum element in the current row of the matrix. The largest one is selected from the obtained results.

The Java implementation is shown below:

```
      public static int matrixMaxValueParallel(int[][] matrix) {
              try {
                  ExecutorService                 executor                =
Executors.newFixedThreadPool(8);
                  List<Callable<Integer>> tasks = new ArrayList<>();

                  for (int[] row : matrix) {
```

```
                Callable<Integer> task = () -> getRowMaxValue(row);
                tasks.add(task);
            }
            List<Future<Integer>> futures = executor.invokeAll(tasks);
            List<Integer> integers = new ArrayList<>();
            for (Future<Integer> future : futures) {
                integers.add(future.get());
            }
            executor.shutdown();
            return Collections.max(integers);
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
        return Integer.MIN_VALUE;
    }

    private static int getRowMaxValue(int[] row) {
        int max = Integer.MIN_VALUE;
        for (int number : row) {
            if (number > max) {
                max = number;
            }
        }
        return max;
    }
```

2.1.3 C implementation

The C implementation uses the same algorithm as the Java and Go implementation. To run the program in parallel, we used the pthread library, which can be downloaded using the package manager vcpkg: https://vcpkg.io/en/packages.html.

The C implementation is shown below:

```
void* find_max_elem(void* thread_info) {
int max = INT_MIN;
Thread_Args* args = (Thread_Args*)thread_info;
int to_row = args->from_row + args->batch_size;
for (int i = args->from_row; i < to_row; i++) {
      for (int j = 0; j < args->column; j++) {
```

```
                if (matrix[i][j] > max) {
                      max = matrix[i][j];
                }
          }
    }
    int index = args->from_row / args->batch_size;
    result[index] = max;
    return 0;
    }

    void example1(int row, int column) {
    result = (int*)malloc(NUM_THREADS_FIRST_EXAMPLE * sizeof(int));
    pthread_t threads[NUM_THREADS_FIRST_EXAMPLE];
    for (int i = 0; i < NUM_THREADS_FIRST_EXAMPLE; i++) {
          Thread_Args* ta = new Thread_Args();
          ta->batch_size = row / NUM_THREADS_FIRST_EXAMPLE;
          ta->column = column;
          ta->from_row = i * ta->batch_size;
          pthread_create(&threads[i], NULL, find_max_elem, (void*)ta);
    }
    for (int i = 0; i < NUM_THREADS_FIRST_EXAMPLE; i++) {
          pthread_join(threads[i], NULL);
    }
    int         max_element         =         get_max_arr_element(result,
NUM_THREADS_FIRST_EXAMPLE);
    free(result);
    }
```

2.1.4 Results

After the above-described algorithm was implemented in all three programming languages, the program execution time was measured.

The following results were obtained:

Table 1

Maximum number search execution time

Go	Java	C
29.9 milliseconds	34.2 milliseconds	51 milliseconds

It can be seen from the table that the execution on Go took the least time.

2.2 Sudoku validation

Formulation of the problem: the input is sudoku (9 * 9 matrix), which can be filled completely or incompletely. It is necessary to make sure that there are no errors in the current sudoku (i.e., there are no repetitions in rows, columns, and 3 * 3 squares).

2.2.1 Go implementation

The input to the isSudokuValid function is a 9 * 9 matrix, which is a field for playing Sudoku. The matrix can be filled completely, or it can be incomplete. The isSudokuValid function creates 5 goroutines that will work in parallel: the first goroutine checks for duplicates in rows, the second for columns, and the next three for duplicates in squares (3 squares for each goroutine). All results are recorded in the appropriate channel. The sync.WaitGroup structure is used for synchronization.

Below is the Go implementation of validating rows of a sudoku:

```
func sudokuLineValidator(matrix [9][9]int, sudokuRowNum int, wg *sync.WaitGroup, resultChannel chan bool) {
	defer wg.Done()
	for i := sudokuRowNum * 3; i < (sudokuRowNum + 1)  * 3; i += 3 {
		for j := 0; j < len(matrix); j += 3 {
			squareDictionary := map[int]int{}
			for sqi := i; sqi < i + 3; sqi++ {
				for sqj := j; sqj < j + 3; sqj++ {
					squareDictionary[matrix[sqi][sqj]] += 1
				}
			}
			if !checkElementsDistinct(squareDictionary) {
				resultChannel <- false
				return
			}
		}
	}
	resultChannel <- true
	}
	func    columnsValidator(matrix    [9][9]int,    wg    *sync.WaitGroup,
resultChannel chan bool) {
	defer wg.Done()
	columnDictionary := map[int]int{}
	for i := 0; i < len(matrix); i++ {
		for j := 0; j < len(matrix[i]); j++ {
			columnDictionary[matrix[j][i]] += 1
		}
		if !checkElementsDistinct(columnDictionary) {
			resultChannel <- false
			return
		}
		columnDictionary = map[int]int{}
	}
	resultChannel <- true
	}
```

2.2.2 C# implementation

The C # implementation uses the same algorithm as the Go implementation, however asynchronous execution will be achieved using Tasks objects.

Below is the C# implementation of validating rows of a sudoku:

```
        private bool RowsValidator(int[,] sudoku)
        {
            for (int i = 0; i < sudoku.GetLength(0); i++)
            {
                Dictionary<int, int> rowDictionary = new();

                for (int j = 0; j < sudoku.GetLength(1); j++)
                {
                    int currentElement = sudoku[i, j];
                    if (rowDictionary.ContainsKey(currentElement))
                    {
                        rowDictionary[currentElement]++;
                    }
                    else
                    {
                        rowDictionary[currentElement] = 1;
                    }
                    }
                    if (!IsDictionaryUniq(rowDictionary))
                    {
                        return false;
                    }
                }
                return true;
            }

            private static bool IsDictionaryUniq(Dictionary<int, int>
dictionady)
            {
                return !dictionady.Select(el => new
                {
                    key = el.Key,
                    value = el.Value
                }).Where(n => n.key != 0 && n.value > 1).Any();
            }
        }
    }
```

2.2.3 C implementation

The C implementation uses the same algorithm as the Go and C# implementation, however parallel execution will be achieved using pthread library.

Below is the C implementation of validating rows of a sudoku:

```
void* rows_validator(void* index_for_result_pointer) {
int index_for_result = (int)index_for_result_pointer;
for (int i = 0; i < SUDOKU_SIZE; i++)
{
    int* row_elements = (int*)malloc(SUDOKU_SIZE * sizeof(int));
    for (int j = 0; j < SUDOKU_SIZE; j++)
    {
        int currentElement = sudoku[i][j];
        row_elements[j] = currentElement;
    }
    if (!all_elements_uniq(row_elements))
    {
        free(row_elements);
```

```
                result[index_for_result] = 0;
                return NULL;
        }
}
result[index_for_result] = 1;
return NULL;
}

bool all_elements_uniq(int* arr)
{
int origVal = 0, newVal = 0;
for (int i = 0; i < SUDOKU_SIZE; i++) {
        origVal = arr[i];
        for (int k = i + 1; k < SUDOKU_SIZE; k++) {
                if (origVal != 0 && origVal == arr[k])
                {
                        return false;
                }
        }
}
return true;
}
```

2.2.4 Results

After the above-described algorithm was implemented in all three programming languages, the program execution time was measured.

The following results were obtained:

Table 2

Sudoku validation execution time

Go	C#	C
536800 nanoseconds	789 microseconds	3 milliseconds

It can be seen from the table that the execution on Go took the least time and the implementation on C is the slowest.

Conclusions. We can see from the results that the C program is slower in two examples. This can be explained by the fact that when writing code in the C #, Java and Go programming languages, not threads were created, but add-on objects: for C # - Tasks objects, for Java - Future objects, for Go - goroutines. There are no such add-ons in the C programming language, so threads must be created directly.

The implementation of parallelism and concurrency in the Go programming language is very different from the implementation in languages such as C # and Java, so it would be good to study this technology for a better understanding of concurrency and multithreading.

Full code from the examples is accessible via link https://github.com/DimaMykhnevych/ParallelismGolang.

In conclusion, Go is fast, so if you focus on speed when developing a product, Go can be one of the candidates.

References

1. Go documentation. URL: https://go.dev/doc/ (date of the application 12.12.2021).