



**Софийски университет “Св. Климент Охридски”,  
Факултет по математика и информатика**

**Курсов проект**  
по Системи за паралелна обработка

**Тема: Граф - обхождане в ширина (BFS)**

Изготвил:

Стилиян Емилов Горанов

фн: 81554, Компютърни науки, курс 3, поток 1, група 1

Научен ръководител:

ас. Христо Христов

## 1. Цел на проекта

Целта на проекта е реализация на паралелен алгоритъм за обхождане на даден граф в ширина, като алгоритъмът генерира различните обхождания, започвайки от всеки един връх на графа.

## 2. Изисквания към проекта

- Програмата трябва да използва паралелни процеси (нишки, задачи) за да разпредели работата по обхождането от всеки връх на повече от един процесор.

- Програмата трябва да позволява команден параметър, чрез който да разбира каква е размерността на графа (броят на върховете му) - например “-n 10240”; Ребрата на графа програмата трябва да генерира произволно.

- Програмата трябва да позволява команден параметър указващ входен текстов файл, съдържащ графа, който ще обхождаме – например „-i graph-data.in“. Параметрите „-n“ и „-i“ са взаимно-изключващи се; Ако все пак бъдат зададени и двата решението как да реагира програмата е Ваше. Форматът на файла graph-data.in е следният:

=== цитат ===

n

0 1 1 0 1 1 0 ... 0

1 0 1 0 1 0 0 ... 1

...

1 0 0 0 0 0 0 ... 1

=== цитат ===

Тоест:

1вият ред съдържа единствено число, указващо размерността на графа;

На оставащите n реда във файла са разположени редовете от матрицата на съседство описваща нашият граф. Елементите на всеки ред от матрицата са разделени със интервали.

- Програмата да позволява команден параметър, указващ изходен файл, съдържащ резултата от обхождането – например „-o graph-data.out“.

Форматът на изходният файл можете да определите сами, стига

файлът да е текстов; При липса на този команден параметър не се записва във файл резултата от обхождането на графа.

- Програмата трябва да има задължителен команден параметър, който задава максималния брой нишки (задачи), на които разделяме работата по обхождането на графа – например “-t 1” или “-tasks 3”.

- Програмата трябва да извежда подходящи съобщения на различните етапи от работата си, както и времето отделено за изчисление и резултата от изчислението

- Да се осигури възможност за „quiet“ режим на работа на програмата, при който се извежда само времето отделено за обхождане на графа, отново чрез подходящо избран друг команден параметър – например “-q”.

### **3. Описание на алгоритъма**

#### **1. Обхождане в ширина (BFS) на граф**

Обхождането на граф означава посещаването на всеки връх и всяко ребро точно веднъж и по точно дефиниран ред. Програмата използва един от най-известните алгоритми за обхождане на граф - в ширина. Накратко алгоритъмът може да се опише така:

Избираме произволен връх на графа, откъдето да започнем обхождането и оттам изследваме абсолютно всички негови съседни, преди да преминем към изследването на съседите от следващия слой, т.е. съседите на съседите на началния връх. И така нататък, докато не обходим целия граф.

Това става по следния начин в проекта:

```
private[graph] def bfsTraversalFrom(start: Vertex): BFSTraversal = {
  @tailrec
  def bfs(toVisit: Queue[Vertex], reached: Set[Vertex], path: BFSTraversal): BFSTraversal = {
    if (toVisit.isEmpty) path
    else {
      val current = toVisit.head
      val newNeighbours = getNeighbours(current).right.get.filter(!reached(_))

      bfs(
        toVisit.dequeue._2.enqueue(newNeighbours),
        reached ++ newNeighbours,
        current :: path
      )
    }
  }

  bfs(Queue(start), Set(start), List.empty).reverse
}
```

## 2. Въведение в Scala Futures

### Какво представлява монадът Future?

Future-ите ни дават възможност да изпълняваме много задачи в паралел по ефективен и асинхронен начин. Самият Future е нещо като ‘контейнер’ (placeholder object) за стойност, която все още може и да не съществува (оттук идва и името). Идеята е, че стойността, която е във Future обекта, бива предоставена конкурентно в даден момент от времето и съответно може да бъде използвана впоследствие.

Ползата от всичко това (композиране на конкурентни задачи по описания начин) е, че получаваме бърз, асинхронен, не-блокиращ паралелен код.

Идиоматичен пример за използване на Future в Scala:

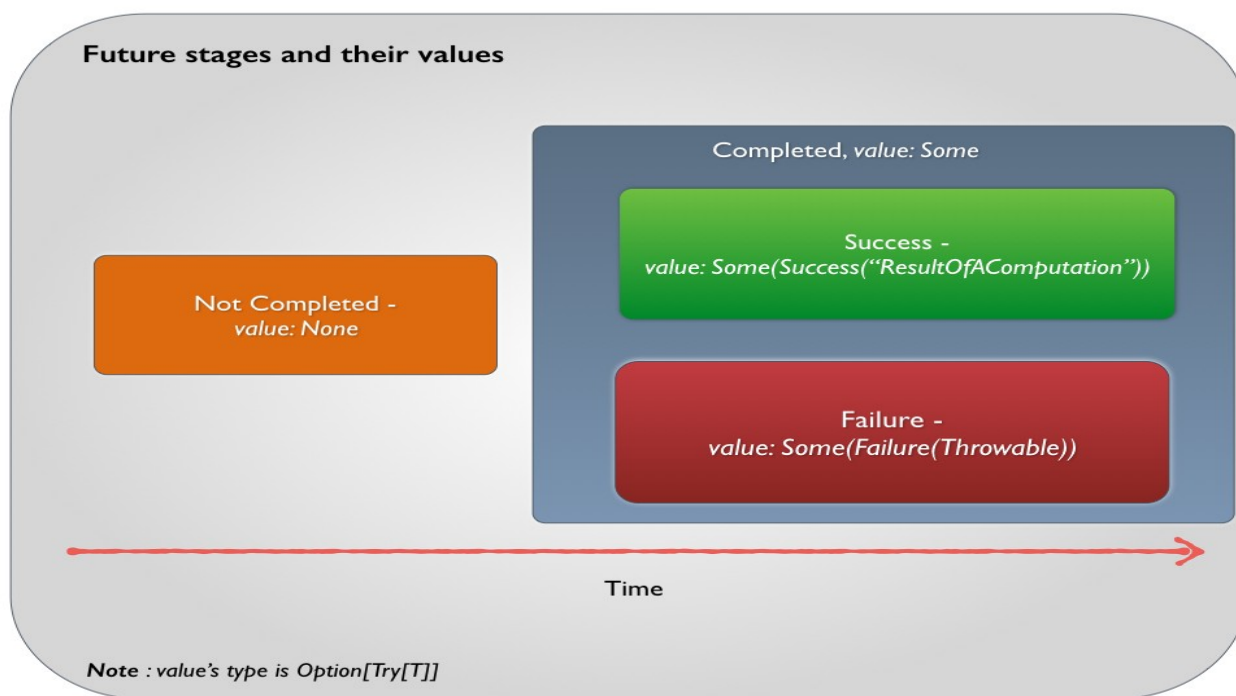
```
implicit val ec: ExecutionContext = ...
val inverseFuture : Future[Matrix] = Future {
  fatMatrix.inverse()
} // ec is implicitly passed
```

Идеята в примера е, че изпълнението на функцията `fatMatrix.inverse()` е делегирана към `ExecutionContext`. А пък обектът `inverseFuture` един вид ‘олицетворява’ резултатът от изчислението.

## Execution Context

Накратко `ExecutionContext`-а е отговорен за изпълнението на дадени задачи (`Future` обектите разчитат на такъв, за да работят). Той изпълнява задачите в нова нишка, в `thread pool` (може би най-често ползваният начин), или в текущата нишка (въпреки че това в доста случаи е нежелателно).

Резултатът от изпълнението на `Future` съответно може и да върне грешка – както се вижда на следващата картинка.



## 3. Паралелен алгоритъм

За написване на програмата е използван езикът `Scala 2.12.8`. Използван е паралелизъм по данни – `SPMD`. Работата на всички `Future`-и (задачи) е асинхронна. Архитектурата на програмата е по модела `Master-Slaves`.

Main нишката е Master, като тя започва изпълнението на абсолютно всички task-ове, по възможност върху отделни нишки (посредством Future). Конкретно идеята за BFS е следната:

- В зависимост от параметъра за броя на задачите (-t n), се създава thread pool с n на брой работници (нишки), който ще играе ролята на ExecutionContext, нужен за създаването на Future-и.
- Main нишката създава нов Future за всеки един връх от графа, като задачата на Future-а е да върне резултат от асинхронното изпълнение на функцията bfsTraversalFrom (снимката на функцията е по-горе).
- Резултатите от всички Future-и се връщат в колекция накрая, като ако е подаден параметър за записване на резултати във файл, резултатите се записват в подходящ формат.

Основната функция, която създава ExecutionContext и задава задачите:

```
def bfsTraversalStartingFromAllVertices(numberOfTasks: Int): BFSTraversalFromAllVerticesResult = {
  logger.debug("Starting BFS traversal from all vertices (" + getNumVertices + ") with number of tasks: " + numberOfTasks)

  val threadPool = Executors.newFixedThreadPool(numberOfTasks)
  implicit val ec = ExecutionContext.fromExecutor(threadPool)

  if (numberOfTasks < 1) {
    throw new IllegalArgumentException("Number of tasks for bfs traversal cannot be less than 1!")
  }

  val sortedListOfVertices = getVertices.toList.sorted

  val result = time {
    sortedListOfVertices.map(start_BFS_task_from_vertex(_)).map(Await.result(_, Duration.Inf))
  }

  logger.debug("Total number of threads used in current run: " + result._1.map(_.threadID).distinct.size)
  logger.debug("Total time elapsed (milliseconds) in current run: " + result._2 + "\n-----\n")

  threadPool.shutdown

  BFSTraversalFromAllVerticesResult(
    allResults = result._1,
    timeForCompletionInMilliseconds = result._2,
    numberOfThreads = numberOfTasks)
}
```

Функцията за стартиране на нова задача (съответно връщаща Future обект, съдържащ бъдещия резултат от изпълнението на BFS), викана от основната функция:

```
private def start_BFS_task_from_vertex(startingVertex: Vertex)(implicit ec: ExecutionContext)
    : Future[BFSTraversalFromSingleVertexResult] = Future {
    logger.debug("Start BFS from vertex " + startingVertex)
    val result = time {
        bfsTraversalFrom(startingVertex)
    }

    logger.debug("Finish BFS started from vertex " + startingVertex
        + ". Time elapsed in milliseconds: " + result._2)

    BFSTraversalFromSingleVertexResult(result._1, result._2, Thread.currentThread.getName.split("-").last.toLong)
}
```