Data-Parallel Programming

Parallel Programming and Data Analysis

Aleksandar Prokopec
Everybody has to work. Work tasks are diverse.
def startup[A, B, C](a: =>A, b: =>B, c: =>C): (A, B, C) = {
  val taskB = task { b }
  val taskC = task { c }
  (a, taskB.join(), taskC.join())
}
Factory
Everybody has to work. Everybody does the same work.
Everybody has to work. Everybody does the same work.

*How can we express parallelism here?*
Everybody has to work. Everybody does the same work.

How can we express parallelism here?

def factory[A, B](items: Seq[A])
 Everybody has to work. Everybody does the same work.

*How can we express parallelism here?*

```python
def factory[A, B](items: Seq[A])(f: A => B): Seq[B]
```
Previously, we learned about task-parallel programming.

* A form of parallelization that distributes execution processes across computing nodes.

We know how to express parallel programs with task and parallel constructs.
Data-Parallelism

Previously, we learned about task-parallel programming.

A form of parallelization that distributes execution processes across computing nodes.

We know how to express parallel programs with task and parallel constructs.

Next, we learn about the data-parallel programming.

A form of parallelization that distributes data across computing nodes.
Data-Parallel Programming Model

The simplest form of data-parallel programming is the parallel for loop.

Example: initializing the array values.
Data-Parallel Programming Model

The simplest form of data-parallel programming is the parallel for loop. Example: initializing the array values.

``` scala
def initializeArray(xs: Array[Int])(value: Int): Unit
```
Data-Parallel Programming Model

The simplest form of data-parallel programming is the parallel for loop. Example: initializing the array values.

```scala
def initializeArray(xs: Array[Int])(value: Int): Unit = {
  for (i <- (0 until xs.length).par) {
  }
}
```
The simplest form of data-parallel programming is the parallel for loop.

Example: initializing the array values.

```scala
def initializeArray(xs: Array[Int])(value: Int): Unit = {
  for (i <- (0 until xs.length).par) {
    xs(i) = value
  }
}
```
The simplest form of data-parallel programming is the parallel for loop.

Example: initializing the array values.

```scala
def initializeArray(xs: Array[Int])(value: Int): Unit = {
  for (i <- (0 until xs.length).par) {
    xs(i) = value
  }
}
```

The parallel for loop is not very functional – it can only affect the program through side-effects.
The simplest form of data-parallel programming is the parallel for loop.

Example: initializing the array values.

```scala
def initializeArray(xs: Array[Int])(value: Int): Unit = {
  for (i <- (0 until xs.length).par) {
    xs(i) = value
  }
}
```

The parallel for loop is not very functional – it can only affect the program through side-effects.

As long as iterations of the parallel loop write to separate memory locations, the program is correct.
Example: Mandelbrot Set

Although simple, parallel for loop allows writing interesting programs.

Render a set of complex numbers in the plane for which the sequence $z_{n+1} = z_n^2 + c$ does not approach infinity.
We approximate the definition of the Mandelbrot set – as long as the absolute value of \( z_n \) is less than 2, we compute \( z_{n+1} \) until we do maxIterations.

```scala
private def computePixel(xc: Double, yc: Double, maxIterations: Int): Int = {
  var i = 0
  var x, y = 0.0
  while (x * x + y * y < 4 && i < maxIterations) {
    val xt = x * x - y * y + xc
    val yt = 2 * x * y + yc
    x = xt; y = yt
    i += 1
  }
  color(i)
}
```
How do we render the set using data-parallel programming?

```scala
def render(): Unit = {
  for (idx <- 0 until image.length) {
    val (xc, yc) = coordinatesFor(idx)
    image(idx) = computePixel(xc, yc, maxIterations)
  }
}

def parRender(): Unit = {
  for (idx <- (0 until image.length).par) {
    val (xc, yc) = coordinatesFor(idx)
    image(idx) = computePixel(xc, yc, maxIterations)
  }
}
```
Rendering the Mandelbrot Set: Demo

Time for a demo!
Time for a demo!

Summary:

- task-parallel implementation – the slowest.
- data-parallel implementation – about $2\times$ faster.
Workload

Different data-parallel programs have different workloads.

*Workload* is a function that maps each input element to the amount of work required to process it.
Uniform Workload

Defined by a constant function: $w(i) = \text{const}$
Uniform Workload

Defined by a constant function: \( w(i) = const \)

Easy to parallelize.
Irregular Workload

Defined by an arbitrary function: $w(i) = f(i)$
Irregular Workload

Defined by an arbitrary function: \( w(i) = f(i) \)

In the Mandelbrot case: \( w(i) = \#\text{iterations} \)

The workload depends on the problem instance.
Irregular Workload

Defined by an arbitrary function: \( w(i) = f(i) \)

In the Mandelbrot case: \( w(i) = \# \text{iterations} \)

The workload depends on the problem instance.

Goal of the data-parallel scheduler: efficiently balance the workload across processors without any knowledge about the \( w(i) \).