Running Computations in Parallel

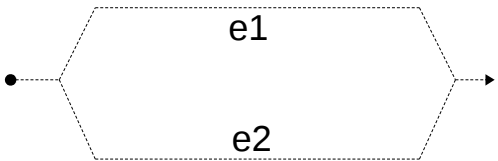
Parallel Programming in Scala

Viktor Kuncak

Basic parallel construct

Given expressions `e1` and `e2`, compute them in parallel and return the pair of results

```
parallel(e1, e2)
```



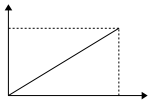
What would be the simplest way to explicitly indicate that computations should proceed in parallel? Consider one such construct, **parallel**. It takes two expressions, say, e1 and e2, computes their values in parallel, and returns the pair of these two values.

Example: computing p-norm

Given a vector as an array (of integers), compute its p-norm

A *p-norm* is a generalization of the notion of *length* from geometry

2-norm of a two-dimentional vector (a_1, a_2) is $\left(a_1^2 + a_2^2\right)^{1/2}$



The p-norm of a vector $(a_1, \ldots, a_n)$ is $\left(\sum_{i=1}^{n} |a_i|^p \right)^{1/p}$

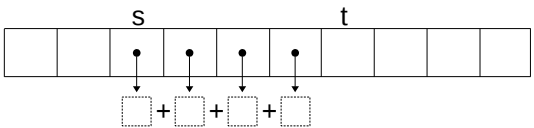
Let's see how to use this construct on a concrete example: computing a **p-norm** of a vector. What is a p-norm? Say p is equal to two. Then we get the common notion of **length** of a vector. We compute it as the sequare root of the sum of squares of the coordinates. For other values of p the definition of p-norm is that we raise the absolute value of the coordinates to the power p, sum them up, and raise the resulting sum to the power **one over p**. Let us write a Scala function that computes the p-norm.

Main step: sum of powers of array segment

First, solve *sequentially* the following `sumSegment` problem: given

- an integer array `a`, representing our vector
- a positive double floating point number `p`
- two valid indices `s` <= `t` into the array `a`

compute $\sum_{i=s}^{t-1} \lfloor |a_i|^p \rfloor$ where $\lfloor y \rfloor$ rounds down to an integer



The main step in the solution is to compute the sum of the elements the array raised to p. Let us define a slightly more general function, called **sum segment**. It should take an array a and the number p (which we represent as a double), but also the start index of the segment as well as the end boundary, an index before which we should stop summing up.

Sum of powers of array segment: solution

The main function is

```
def sumSegment(a: Array[Int], p: Double, s: Int, t: Int): Int = {
  var i= s; var sum: Int = 0
  while (i < t) {
    sum= sum + power(a(i), p)
    i= i + 1
  }
  sum
}
```

Here power computes $\lfloor |x|^p \rfloor$:

```
def power(x: Int, p: Double): Int = math.exp(p * math.log(abs(x))).toInt
```

The solution is just a traversal of the array that adds up the powers of elements. We use a while loop to do the traversal. The **power** function is a simple combination of the appropriate math library functions.

Given `sumSegment(a,p,s,t)`, how to compute p-norm?

$$||a||_p := \left(\sum_{i=0}^{N-1} \lfloor |a_i|^p \rfloor \right)^{1/p}$$

where `N = a.length`

Now that we have the sum segment function, how do we compute the p-norm?

Given `sumSegment(a,p,s,t)`, how to compute p-norm?

$$||a||_p := \left(\sum_{i=0}^{N-1} \lfloor |a_i|^p \rfloor \right)^{1/p}$$

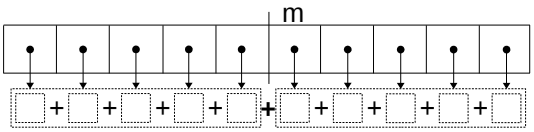
where `N = a.length`

```
def pNorm(a: Array[Int], p: Double): Int =
  power(sumSegment(a, p, 0, a.length), 1/p)
```

We give to sumSegment the entire array, from index zero to the length of the array. We then raise the result to the power one over p. This gives us a sequential version for computing the p-norm. How do we go from here to a parallel version?

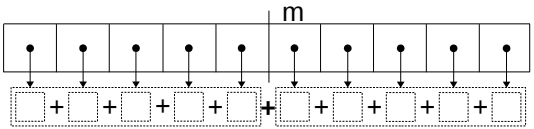
Observe that we can split this sum into two

$$||a||_p := \left(\sum_{i=0}^{N-1} \lfloor |a_i|^p \rfloor \right)^{1/p} = \left(\sum_{i=0}^{m-1} \lfloor |a_i|^p \rfloor + \sum_{i=m}^{N-1} \lfloor |a_i|^p \rfloor \right)^{1/p}$$



Now, observe that the sumation can be expressed in two parts: sum up to some middle element m, and then sum from that middle element to the end. What is a Scala expression that corresponds to using two sums?

Using sumSegment twice

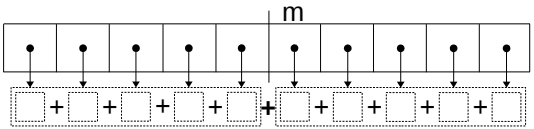


The resulting function is:

```
def pNormTwoPart(a: Array[Int], p: Double): Int = {
  val m = a.length / 2
  val (sum1, sum2) =          (sumSegment(a, p, 0, m),
                               sumSegment(a, p, m, a.length))
  power(sum1 + sum2, 1/p) }
```

All we need to do is invoke sum segment twice, then add up the two intermediate sums before raising everything to the power of one over p. This is still sequential computation. How do we make it parallel?

Making two sumSegment invocations parallel



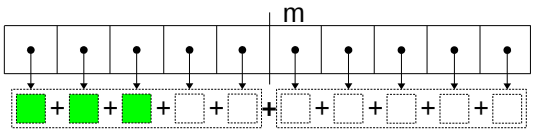
The resulting function is:

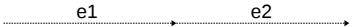
```
def pNormTwoPart(a: Array[Int], p: Double): Int = {
  val m = a.length / 2
  val (sum1, sum2) = parallel(sumSegment(a, p, 0, m),
                              sumSegment(a, p, m, a.length))
  power(sum1 + sum2, 1/p) }
```

All we need to do to do is to wrap the pair into the **parallel** construct! Given a platform that supports parallel execution, the computation **with** parallel may run up to twice as fast as the one without.

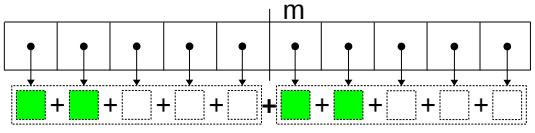
Comparing execution of two versions

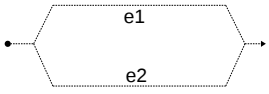
```
val (sum1, sum2) = (sumSegment(a, p, 0, m),
                    sumSegment(a, p, m, a.length))
```





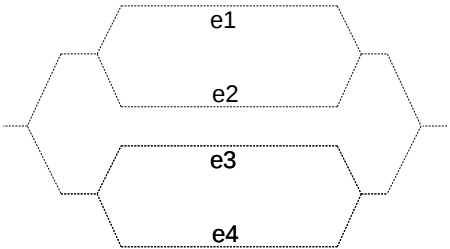

```
val (sum1, sum2) = parallel(sumSegment(a, p, 0, m),
                            sumSegment(a, p, m, a.length))
```





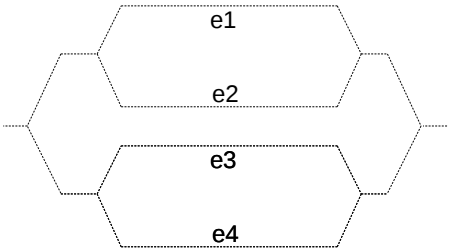
Let us compare the executions of two functions. The parallel version may take some time to set up parallel execution, but after that might make progress on processing array elements twice as fast as the sequential one.

How to process four array segments in parallel?



We have seen how to process two array segments in parallel. Suppose that we have at least four parallel hardware threads. How would we process four array segments in parallel using our parallel construct?

How to process four array segments in parallel?




```
val m1 = a.length/4; val m2 = a.length/2; val m3 = 3*a.length/4
val ((sum1, sum2),(sum3,sum4)) =
  parallel(parallel(sumSegment(a, p, 0, m1), sumSegment(a, p, m1, m2)),
           parallel(sumSegment(a, p, m2, m3), sumSegment(a, p, m3, a.length)))
```

We divide the array into four segments. We can compute the sum of first two segments in parallel, and also the sum of second two segments in parallel. Finally, these two parallel computations of pairs can themselves take place in paralllel. Once we get four partial sumes, we can add up these four.

Is there a recursive algorithm for an unbounded number of threads?

We have seen how to run summation over two or four segments in parallel. Now, suppose that we have a very long array and an essentially unbounded number of parallel hardware resources. Is there an algorithm that spawns as many parallel computations as needed?

Is there a recursive algorithm for an unbounded number of threads?

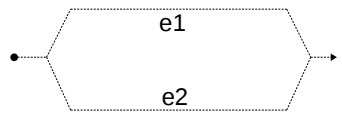
```
def pNormRec(a: Array[Int], p: Double): Int =
  power(segmentRec(a, p, 0, a.length), 1/p)

// like sumSegment but parallel
def segmentRec(a: Array[Int], p: Double, s: Int, t: Int) = {
  if (t - s < threshold)
    sumSegment(a, p, s, t)  // small segment: do it sequentially
  else {
    val m = s + (t - s)/2
    val (sum1, sum2) = parallel(segmentRec(a, p, s, m),
                                segmentRec(a, p, m, t))
    sum1 + sum2 } }
```

Here is one solution. We define segmentRec, which uses parallelism to sum up a given segment of the arrray. When the segment is small, it computes the sum sequentially. Otherwise, it divides segment in half and invokes itself recursively on two smaller segments. To compute a norm we again raise the segment sum to the power one over p. This completes our vector norm example.

Signature of parallel

```
def parallel[A, B](taskA: => A, taskB: => B): (A, B) = { ... }
```



- returns the same value as given
- benefit: `parallel(a,b)` can be faster than `(a,b)`
- it takes its arguments as *by name*, indicated with `=> A` and `=> B`

Here is the type signature of parallel. It takes the two computations as parameters, here denoted taskA and taskB. It returns a pair of A and B, storing the values of those computations. From the point of view of result, parallel behaves as an identity function. The benefit is, of course, that the computation can finish sooner than computing first taskA and then taskB. Note that the types of parameters are declared as arrow A and arrow B. Why not simply A and B?

parallel is a control structure

Suppose that both `parallel` and `parallel1` contain the same body:

```
def parallel [A, B](taskA: => A, taskB: => B): (A, B) = { ... }
def parallel1[A, B](taskA:    A, taskB:    B): (A, B) = { ... }
```

If `a` and `b` are some expressions what is the difference between

1. `val (va, vb) = parallel(a, b)`
2. `val (va, vb) = parallel1(a, b)`

To help answer this question, consider a seemingly minor variant of parallel called parallel1, which differs from parallel only in that it takes parameters **by value** instead of **by name**. How does parallel1 behave? If we have two long-running computations a and b, what is the difference between: **parallel** of a and b versus: **parallel1** of a and b?

parallel is a control structure

Suppose that both `parallel` and `parallel1` contain the same body:

```
def parallel [A, B](taskA: => A, taskB: => B): (A, B) = { ... }
def parallel1[A, B](taskA:    A, taskB:    B): (A, B) = { ... }
```

If `a` and `b` are some expressions what is the difference between

1. `val (va, vb) = parallel(a, b)`
2. `val (va, vb) = parallel1(a, b)`

The second computation evaluates sequentially, as in `val (va,vb) = (a,b)`

For parallelism, need to pass *unevaluated computations* (call by name).

In fact, the second computation does not use parallelism at all. Because the arguments to parallel1 are passed by value, they are first evaluated, one by one, and this is where the time is spent. Then their values are passed to parallel, which returns those same two values without doing any useful work. It is therefore essential that we do not evaluate the computation before giving it to a construct that runs it in parallel. For this reason it is appropriate to use call by name parameters, indicated by the arrow in the signature of parallel. Like expressions **if** and **while**, parallel is a control structure.

What happens inside a system when we use parallel?

Efficient parallelism requires support from

- language and libraries
- virtual machine
- operating system
- hardware

One implementation of `parallel` uses Java Virtual Machine threads

- those typically map to operating system threads
- operating system can schedule different threads on multiple cores

Given sufficient resources, a parallel program can run faster

We have seen how to use the parallel construct. What happens behind the scenes when we invoke computations in parallel? To support parallelism efficiently, we need support from different layers of a computing system: the language and libraries, virtual machine (such as Java Virtual Machine), the operating system, and the hardware itself. One implementation of parallel uses Java's threads. On most platforms these are mapped to threads of the underlying operating system. The operating system provides ability to run many threads and processes. When the underlying hardware has multiple processor cores, these threads can execute on different cores, which results in parallel execution. Thanks to the flexibility in different layers of the software stack, a program written with parallelism in mind will run even when there is only one processor core available (of course, without the speedup).

Underlying Hardware Architecture Affects Performance

Consider code that sums up array elements instead of their powers:

```
def sum1(a: Array[Int], p: Double, s: Int, t: Int): Int = {
  var i= s; var sum: Int = 0
  while (i < t) {
    sum= sum + a(i) // no exponentiation!
    i= i + 1
  }
sum }
val ((sum1, sum2),(sum3,sum4)) = parallel(
   parallel(sum1(a, p, 0, m1), sum1(a, p, m1, m2)),
   parallel(sum1(a, p, m2, m3), sum1(a, p, m3, a.length)))
```

As a result of this complexity, performance of parallel code is affected by some of the hardware arhitecture aspects. To illustrate this, consider the following sum1 function, which is like sumSegment, but sums up array elements themselves instead of their powers. Suppose that we try to execute four such sums in parallel on a commodity desktop with, say, four physical cores.

Underlying Hardware Architecture Affects Performance

Consider code that sums up array elements instead of their powers:

```
def sum1(a: Array[Int], p: Double, s: Int, t: Int): Int = {
  var i= s; var sum: Int = 0
  while (i < t) {
    sum= sum + a(i) // no exponentiation!
    i= i + 1
  }
sum }
val ((sum1, sum2),(sum3,sum4)) = parallel(
   parallel(sum1(a, p, 0, m1), sum1(a, p, m1, m2)),
   parallel(sum1(a, p, m2, m3), sum1(a, p, m3, a.length)))
```

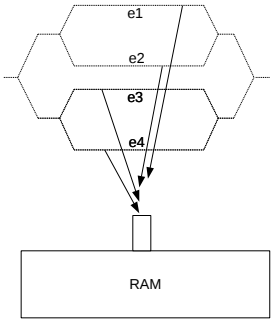
Unlike with 'sumSegment', difficult to get speedup for 'sum1'. Why?

We may find it difficult to get any notable speedup, despite the fact that we do get speedups if we run the computation that does the more expensive operation per array. What is more, this problem remains even if we make the size of the array very large. What is happening?

Underlying Hardware Architecture Affects Performance

Consider code that sums up array elements instead of their powers:

```
def sum1(a: Array[Int], p: Double, s: Int, t: Int): Int = {
  var i= s; var sum: Int = 0
  while (i < t) {
    sum= sum + a(i) // no exponentiation!
    i= i + 1
  }
sum }
val ((sum1, sum2),(sum3,sum4)) = parallel(
   parallel(sum1(a, p, 0, m1), sum1(a, p, m1, m2)),
   parallel(sum1(a, p, m2, m3), sum1(a, p, m3, a.length)))
```

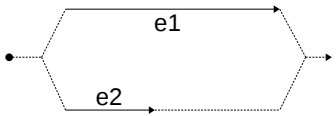


Memory is a bottleneck

It turns out that this computation is bound by the memory bandwidth. The array is stored in random access memory. Whether we have one or more cores working, the cores spend their time waiting for the elements of the array to be fetched from the random access memory. Even though computations

Combining computations of different legnth with parallel

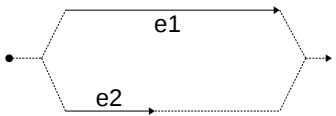
```
val (v1, v2) = parallel(e1, e2)
```



Recall that we have previously ran in parallel computations on array segments of approximately same length, resulting in similar time for different parallel threads. What happens when we invoke parallel on two computations that take very different time to execute?

Combining computations of different legnth with parallel

```
val (v1, v2) = parallel(e1, e2)
```



The running time of parallel(e1, e2) is the maximum of two running times

We need to wait for both e1 and e2 to complete execution before we can return the pair of values. Therefore, the running time, in the best case, is the maximum of the two running times.

End of Unit

End of Unit