Lecture 4: NESL

This lecture will focus on NESL, which is a functional data-parallel language.

1 NESL

NESL stands for Nested Data Parallel Language. It has been developed by Blelloch et. al. at CMU. NESL is unique in that it is functional, polymorphic and type-inferencing. Polymorphic means that functions can have variable types (that is, for example, a function computing the length of a sequence of integers will work with any other type). Type-inferencing means that types need not be specified, but are deduced by the system.

Unlike other data-parallel languages, NESL supports "nested" data-parallel operations.

In the following, we shall describe the most important constructs in NESL.

1.1 Sequence

The fundamental data type in NESL is the sequence. All elements of a sequence have the same type, that can be a scalar type or another sequence (in the latter case, the sequence is said to be **nested**). For example:

Example 1
input: `[6,8,9,2]`
output: `[6,8,9,2] : [int]`

Here, "`[int]`" represents the type of the output (NESL computed it automatically), in this case a sequence of integers. Note that a sequence must contain data of the same type.

1.2 Apply-to-each

NESL provides a construct `{}` to apply operators to every element of a sequence. For example:

Example 2
input: `{ 7 - a : a in [6,8,9,2]}`
output: `[1,-1,-2, 5] : [int]`

There is also a way to specify a condition so that the operation is only applied to those elements that satisfy the condition. For example:

Example 3
input: `{ 7 - a : a in [6,8,9,2] | a < 7}`
output: `[1, 5] : [int]`

The code in Example **??** compares 7 with every element in the sequence (all comparisons are done in parallel), flags those elements which are less than 7, and then performs a *scan* to find out the correct destination of the flagged elements, and finally routes the elements to the right place and does the arithmetic.

1.3 Nesting

NESL supports nested sequences. In other words, the elements of a sequence can themselves be sequences. For example:

Example 4
input: `x=[[6, 8, 9, 2], [0, 4, 6], [2, 3, 5, 7, 11]]`
output: `[[6, 8, 9, 2], [0, 4, 6], [2, 3, 5, 7, 11]] : [[int]]`

The assigment `x = ...` can only be done at top level (use the `let` construct internally to functions).

To remove one level of nesting, use `flatten`. For example:

Example 5
input: `flatten(x)`
output: `[6,8,9,2,0,4,6,2,3,5,7,11] : [int]`

1.4 Read

In NESL, one can read the elements in a sequence at the places specified by the elements of another sequence. For example:

Example 6
input: `[6, 8, 9, 2]->[0,3]`
output: `[6,2] : [int]`

1.5 Write

We can also write to a sequence at specific places. The write operation takes a second sequence of pairs, where the first element specifies the index and the second specifies the value. For example:

Example 7 *input:* `'algorithm' <- [(0,'l'),(3, 'a), (1, 'o)]`
output: `'logarithm' : [char]`

(A pair is like a sequence of two elements, but the two elements can have different types).

2 Example of code

In this section we shall present an example NESL program and analyze its performance in terms of work and critical path. The example we choose is quicksort.

```
function qsort(a)=
   if (#a < 2) then a
               else
                  let  pivot = a[rand (#a)];
                      lesser = {e in a | a < pivot};
                       equal = {e in a | a = pivot};
                     greater = {e in a | a > pivot};
                  in qsort(lesser) ++ equal ++ qsort(greater)
```

Note: the `++` *operator concatenates two sequences.*

We shall analyze this algorithm in the best case (even if this is usually a Bad Thing to do). In this algorithm, the best case occurs when `pivot` is always in the middle of the sequence. If this is true, the critical path $C(n)$ satisfies the recurrence

$$C(n) = 2C\left(\frac{n}{2}\right) + \Theta(1) ,$$

whose solution is

$$C(n) = \Theta(n) .$$

The work $W(n)$ satisfies the recurrence:

$$W(n) = 2W\left(\frac{n}{2}\right) + \Theta(n) ,$$

whose solution is

$$W(n) = \Theta(n \lg n) .$$

Hence, the average parallelism is only:

$$\frac{W(n)}{C(n)} = \Theta(\lg n) .$$

We shall now argue that the average parallelism is poor. Recall Brent's theorem:

$$T_p \approx \frac{W}{p} + C .$$

Brent's theorem says when W/p dominates C we can get almost linear speed-up. The condition $W/p \gg C$ is equivalent to, $\frac{W}{C} \gg p$. If the average parallelism is small (like in our example) we cannot successfully exploit a big number of processors.

We shall now give an improved quicksort algorithm whose expected critical path is $O(\lg n)$ (much smaller than the $O(n)$ critical path of the previous algorithm), and therefore has a much higher average parallelism. The improvement comes from solving the subproblems in parallel: this can be done in NESL by means of a nested sequence.

```
function qsort(a)=
   if (#a <2 ) then a
               else
                 let  pivot=a[rand(#a)];
                     lesser={e in a | a < pivot};
                      equal={e in a | a = pivot};
                    greater={e in a | a > pivot};
                 [bot,top]={qsort(v) : in [lesser, greater]}
                 in bot ++ equal ++ top
```

Here, the critical path $C(n)$ in the best case satisfies:

$$C(n) = C\left(\frac{n}{2}\right) + \Theta(1) ,$$

whose solution is

$$C(n) = \Theta(\lg n) .$$

Hence, the average parallelism is

$$\frac{W(n)}{C(n)} = \Theta(n) ,$$

which is much better than the previous case.

In the following, we will analyze the expected critical path, not just the best case. The idea of the analysis is as follows: we prove that the expected size of the biggest partition decreases by at least a constant factor at each step. Therefore, after a logarithmic number of steps we expect all partitions to have size 1. However, things are not so simple and we need to use some high-probability argument.

An important tool in this kind of proofs is *Markov's inequality*.

Lemma 1 (Markov's inequality) *If z is a non-negative random variable, and $t \in \mathbf{R}^+$ then*

$$\Pr\{z \geq t\} \leq \frac{\mathrm{E}[z]}{t} .$$

Proof. $\mathrm{E}[z] \geq 0 \cdot \Pr\{z < t\} + t \cdot \Pr\{z \geq t\} = t \cdot \Pr\{z \geq t\}$ □

Theorem 2 *The expected critical path in the qsort algorithm is $\Theta(\lg n)$.*

Proof. Let n_k = random variable for number of elements in the partition in which a given element x lies at the k-th step. We want to prove that the expected value of n_k decreases exponentially in k. We have

$$n_0 = n .$$

We want now to calculate $\mathrm{E}[n_k]$, for all k. We claim that $\mathrm{E}[n_{k+1}] \leq 3\mathrm{E}[n_k]/4$. Indeed, let's look at the relative rank i of the pivot and the rank r of x. If $i < r$ then x lands in a partition of size $n_k - i$. If $i > r$ then x lands in a partition of size $i - 1$. If $i = r$ then x lands in no partition and we are done. For all given r, we shall now compute the expectation $\mathrm{E}[n_{k+1}]$ using the definition:

$$\begin{aligned}
\mathrm{E}[n_{k+1}] &= \frac{1}{n_k}\left(\sum_{i=1}^{r-1}(n_k - i) + \sum_{i=r+1}^{n_k}(i-1)\right) \\
&= \frac{(2n_k - r)(r-1) + (n_k + r + 1)(n_k - r)}{2n_k} \\
&= \frac{-r^2 + n_k r + r + n_k^2/2 - 3n_k/2}{n_k} .
\end{aligned}$$

The last term assumes its maximum value when $r = (n_k + 1)/2$, and the value is $(3/4 \cdot n_k^2 - n_k/2 + 1/4)/n_k$. We conclude therefore that

$$\mathrm{E}[n_{k+1}] \leq \frac{3}{4} n_k ,$$

as claimed. By induction,

$$\mathrm{E}[n_k] \leq \left(\frac{3}{4}\right)^k \mathrm{E}[n_0] = \left(\frac{3}{4}\right)^k \cdot n .$$

After $k = 3\log_{4/3} n$ steps, we have:

$$\mathrm{E}[n_k] \leq \left(\frac{3}{4}\right)^k = \frac{1}{n^2} .$$

From Markov's inequality, we get

$$\Pr\{n_k \geq 1\} \leq \frac{1}{n^2} ,$$

so the probablity that there is an element in a partition of size > 1 after $3\log_{4/3} n$ steps is at most $n \cdot (1/n^2) = 1/n$.

The last inequality comes from Boole's inequality (also known as "union bound") which says that the probability of the union is at most the sum of the probablities.

We conclude that the expected critical path is at most $(1 - 1/n) \cdot 3\log_{4/3} n + (1/n) \cdot n = \Theta(\lg n)$. □

Corollary 3 *The expected work in qsort is* $\Theta(n \lg n)$.