Array reduction

07/12/2016

1   Array reduction

This exercise is about parallelizing a reduction operation on all the elements of an array x of size n. Consider a binary operator $\otimes$ which is **associative** which means

$$a \otimes b \otimes c = (a \otimes b) \otimes c = a \otimes (b \otimes c)$$

In the provided sequential code, this operator is implemented by the `operator(int *a, int *b)` function which computes `*a`$\otimes$`*b` and stores the result in `a`. The provided sequential reduction code simply performs a sweep of the whole array x:

```
  for(i=1; i<n; i++)
    operator(x, x+i);
```

and thus stores in `x[0]` the result of $(((x[0]\otimes x[1])\otimes...)\otimes x[n-1]\otimes)$.

2   Package content

In the `reduction` directory you will find the following files:

- `main.c`: this file contains the main program that creates a vector x of size n and then computes its reduction using first the sequential routine `sequential_reduction` and then the parallel routine `parallel_reduction` which has to be implemented as described below. **Only this file has to be modified for this exercise**.

- `aux.c, aux.h`: these two files contain auxiliary routines and **must not be modified**.

The code can be compiled with the `make` command: just type `make` inside the `reduction` directory; this will generate a `main` program that can be run like this:

```
$ ./main n
```

where n is the size of the vector whose reduction has to be computed.

3   Assignment

- ⌨ At the beginning, the `parallel_reduction` are a copy of `sequential_reduction`. Modify these routine in order to parallelize it. Make sure that the result computed by the two routines (sequential and parallel) is consistently (that is, at every execution of the parallel code) the same.

- ✎ Is the provided sequential algorithm parallelizable? why not? Report your answer in the `responses.txt` file. Analyze the performance of the parallel code using one, two and four threads. Report the execution times in the `responses.txt` file and comment on the observed results: did you observe any speedup (reduction of the execution time) using 2 and 4 threads instead of 1? How can you interpret these results? Can you achieve a linear speedup (that is, sequential time divided by the number of threads) ?

Advice

- Think about the associativity property of the $\otimes$ operator to find ways to parallelize the reduction.

- Use a small (between 10 and 50) value for `n` when developing/debugging; use a bigger value (between 100 and 500) when you're sure that your implementation works well and you want to analyze performance.