Prefix Scan

1 Sequential and parallel Prefix Scan

Prefix Scan is an operation that computes all the partial sums of a vector of values. Given an array, the result of this operation is an array where each element is the sum of the preceding elements in the original vector up to the corresponding position. There are two versions of prefix scan: inclusive and exclusive. For an inclusive scan, the result is the sum of all preceding values, as well as the value of the element in the position under consideration. Therefore, the value of coefficient i in the result vector is equal to the sum of all the values $1, 2, ..., i$ in the original vector. The exclusive version does not include the value of the vector element at the position of interest. Therefore, the value of coefficient i in the result vector is equal to the sum of all the values $1, 2, ..., i-1$ in the original vector. Here is an example of these two variants:

Original vector	5	3	8	1	2	6	4	7	2	9	4	1	3	8	6	7

Inclusive Prefix Scan	5	8	16	17	19	25	29	36	38	47	51	52	55	63	69	76

Exclusive Prefix Scan	0	5	8	16	17	19	25	29	36	38	47	51	52	55	63	69

Both variants are commonly implemented *in-place* which mean that the result is stored in the input array. **The objective of this exercise is to implement a parallel version of the inclusive prefix scan using OpenMP.**

The sequential version of the inclusive prefix scan is extremely simple and consists of the following lines of code

```
for(i=1; i<l; i++)
  pref[i] = pref[i]+pref[i-1];
```

As you can see, the above loop cannot be parallelized with a simple `#pragma omp parallel for` directive because of a loop-carried dependency (iteration `i` uses the result of iteration `i-1`). For this reason a more complicated algorithm has to be used; this methods consists of the following 5 steps:

1. the initial vector is split into four chunks (parts) of roughly the same size and each chunk is assigned to one thread;

2. each thread computes the inclusive prefix scan of the chunk it was assigned;

3. each thread copies the last element of its local result into the corresponding location of a temporary array `loc_last` (i.e., thread number 0 will copy the value in `loc_last[0]`, thread number 1 in `loc_last[1]` etc.);

4. **one** thread (it can be the master as well as any other thread, but only one) computes the **exclusive** prefix scan of the `loc_last` array;

5. each thread sums the corresponding element of the resulting `loc_last` array into its own chunk.

Here is an example of how this algorithm is applied to the vector in the previous figure when the number of threads is equal to 4:
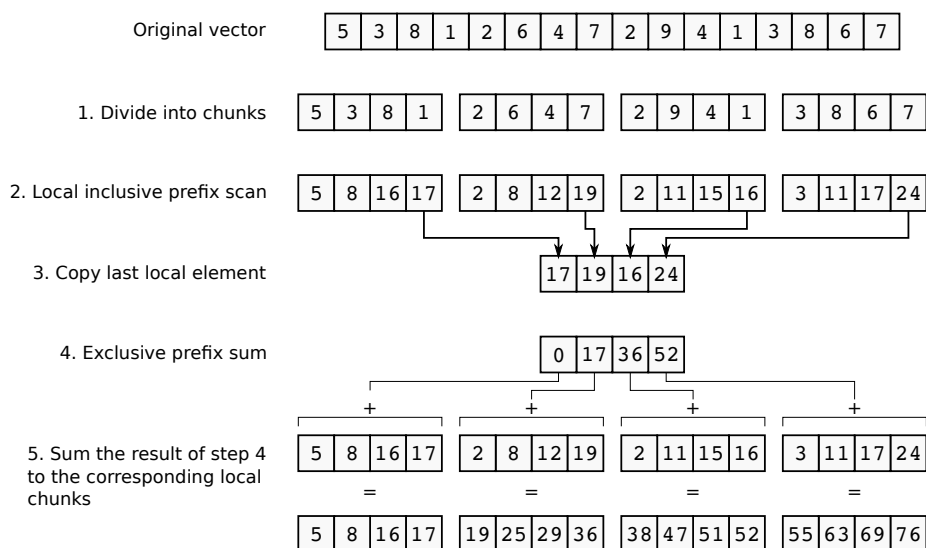
2   Package content

The `PrefixScan` directory contains a single file named `prefixscan.c`. This file contains:

- a sequential inclusive prefix scan routine `prefix_inc_seq` that takes two arguments:

  1. an integer `l` containing the length of the array whose prefix scan has to be computed;
  2. a reference `*pref` of the array whose prefix scan has to be computed;

- a sequential exclusive prefix scan routine `prefix_exc_seq` with the same interface as the inclusive routine;

- the declaration of the parallel inclusive prefix scan routine `prefix_inc_par` (at the beginning the routine is empty and the objective of the exercise is to write it);

- a main program that creates a vector, fills it with random numbers, computes the inclusive prefix scan with both the sequential and the parallel version and finally checks that the result of the parallel version is the same as that of the sequential one.

The code can be compiled with the `make` command: simply type `make` int the `PrefixScan` directory. This will generate an executable `main` file that can be run like this

```
$ ./main l
```

where l is the length of the array whose prefix scan has to be computed. Obviously, at the beginning the execution of the main program will report an error because the parallel version is not implemented.

3 Assignment

- ⌨ Implement the parallel inclusive prefix scan algorithm described above using OpenMP inside. Compile and run the code with 1, 2 and 4 threads to check the correctness of your parallel implementation.

- ✎ Did you observe any speedup (reduction of the execution time) when using 2 or 4 threads instead of 1? If not, how can you explain this? report your comments in the `responses.txt` file.

Advice.
- Note that inside the empty initial version of the parallel routine, the `loc_last` array is already declared of size `MAXTHREADS` (equal to 16). Clearly you only have to use the first `nth` elements where `nth` is the number of threads in your parallel region.

- Pay attention to the synchronization between threads.

- Note that the original vector is only logically partitioned into chunks: this means threads don't have to allocate memory for storing the local chunks but that each thread simply has to identify the first element and the length of its own local chunk.

- Use the `prefix_inc_seq` sequential routine to compute the inclusive prefix scan on the local chunks as described in step 2 of the parallel algorithm. Use the `prefix_exc_seq` routine to compute the exclusive prefix scan of the `loc_last` array as described in step 4 of the parallel algorithm.

- Compile with `make main_dbg` to get some messages useful for debugging.