SPM 17/18 Final Project Report

ORLANDO LEOMBRUNI

23/07/2018

Contents

1. Overview

This project concerns the implementation of the "parallel prefix" pattern: given a vector $[x_1, x_2, ..., x_n]$ and an operator $\oplus$ with identity I, compute the vector

$[x_1, x_1 \oplus x_2, ..., x_1 \oplus x_2 \oplus \cdots \oplus x_n]$.

The parallel prefix is thus an higher-order function, and in literature is also known as the "prefix sum" or "inclusive scan" operation. There also exists a slight variation of the operation, called "exclusive scan", which instead computes the vector

$[I, x_1, x_1 \oplus x_2, ..., x_1 \oplus x_2 \oplus \cdots \oplus x_{n-1}]$

(i.e. a "right shift" of the vector obtained from the inclusive scan, with the operator identity I as the first value). Both inclusive and exclusive scan are implemented in the project, and the terms "parallel prefix" and "parallel scan" will be used throughout this report interchangeably.

2. The parallel algorithm

The scan operation is one of the most useful primitives used in parallel programming, since it is used widely as a subroutine of other parallel algorithms. Thus, it has been extensively studied, and efficient implementations have been designed for different parallel architectures (e.g. NVidia GPGPUs); most of them, however, refer to the same parallel algorithm, designed by Blelloch in 1990 and based on the design by Ladner and Fischer for hardware parallel prefix adder modules.

Blelloch's algorithm builds a (conceptual) balanced binary tree starting from the input vector. It has two phases: the *up-sweep* or *reduce* phase from leaves (i.e. elements of the input vector) to the root, and the *down-sweep* phase from the root to the leaves. An informal description of the algorithm is the following:

1. Start with the input vector $[x_1, ..., x_n]$.
2. For each $i \in [1, \frac{n}{2}]$, let $j = 2i - 1$ and compute $z_i = x_j \oplus x_{j+1}$ in parallel.
3. Recursively compute the parallel prefix over the vector $[z_1, ..., z_{\frac{n}{2}}]$ and store the result in the vector $[w_1, ..., w_{\frac{n}{2}}]$.
4. For each $i \in [1, n]$, in parallel:
    - If i is even, put $y_i = w_{\frac{i}{2}}$;
    - If i is odd, put $y_i = x_i \oplus w_{\frac{i-1}{2}}$.
5. The vector $[y_1, ..., y_n]$ contains the parallel prefix of the input vector.

Figure 1: The upsweep phase of Blelloch's algorithm.

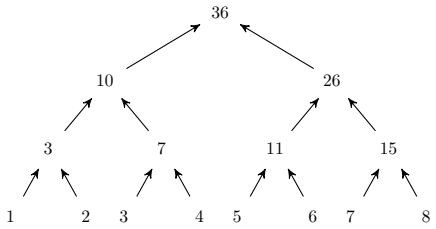
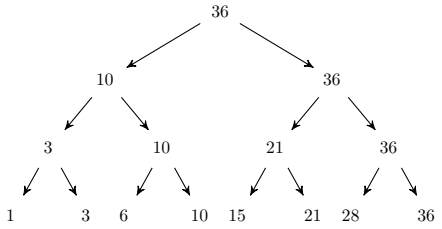


Figure 2: The downsweep phase of Blelloch's algorithm.



An example of application is shown in figures 1 and 2.

Blelloch's algorithm is **work-efficient**: the binary tree has n internal nodes and a single $\oplus$ operation is performed per internal node, so a total of n operations are performed for each sweep phase. The total work is thus $2n = O(n)$. The **span** of the algorithm, instead, is the length of the chain of tasks that must be executed one after another, and is equivalent to two traversal of the tree from leaves to root and from root to leaves. Thus, with n internal nodes, the span is $2 \log n = O(\log n)$.

2.1 Adapting Blelloch's algorithm

Blelloch's algorithm is based on the PRAM model, which assumes there are infinite processors available at any given moment; thus, its theoretical results are only applicable if there are at least n available parallel executors, which is highly unrealistic on commodity hardware even for values of n around a few thousands. Of much more interest is the case in which we have an input of size n and $p \ll n$ parallel executors.

An algorithm based on Blelloch's double-sweep algorithm appears on McCool's book, in chapter 5.4; it is based on the concept of **tiled** operations over the input vector, i.e. each executor computes a sequential operation over a portion of the input which is independent from the rest, so the executors can work in parallel. Having an input of size n, p parallel executors and $k = \frac{n}{p}$, the algorithm is the following:

1. Partition the input into p portions of size k.

2. Perform a **reduction** of each portion in parallel, i.e. if the i-th portion is $[x_1^i, ..., x_n^i]$ then compute $x_1^i \oplus \cdots \oplus x_n^i$. Parallel executor i then places the computed value into an intermediate array (of size p) in position i.
3. Perform a sequential **exclusive** scan over the p reduced values. This scan is always exclusive, even when the overall operation is an inclusive scan.
4. Perform a scan on each portion in parallel using the value computed in step 3 as the initial value, i.e. when performing the scan on the i-th portion use as initial value the i-th value of step 3. This scan will be exclusive if the overall operation is exclusive, inclusive otherwise.

The total work of this algorithm is only slightly worse than Blelloch's: it performs n operations in step 2, p operations in step 3 and n operations in step 4, for a total work of $2n + p = O(n + p)$. The span of the algorithm, however, is significantly worse: when having theoretical infinite executors, in fact, we end up with the degenerate case in which steps 2 and 4 only contribute a constant factor of 1 to the critical path, while step 3 requires a full exclusive scan of the whole input array, so the span is $2 + n = O(n)$.

This algorithm, however, achieves good results in the case we have significantly more input elements than parallel executors. The general complexity of the algorithm is $T(n) = \frac{n}{p} + p + \frac{n}{p} = O(\frac{n}{p} + p)$, so while asymptotically this quantity tends to $O(n)$ both for $p \to 1$ and $p \to n$ the algorithm performs better than the sequential case with basically any choice of p. Note that the function has minimum in $p = \sqrt{n}$, which is the ideal parallelism degree for this algorithm.

3. Parallel architecture design and performance modeling

Having discussed the parallel algorithm in the previous section, it is clear that any implementation of it should have three phases:

1. The reduction phase (parallel over p executors)
    - $\frac{n}{p}$ time
2. The exclusive scan on intermediate results (sequential)
    - p time
3. The final (inclusive/exclusive) scan phase (parallel over p executors)
    - $\frac{n}{p}$ time

The correspoding sequential algorithm just scans the whole input array one time, so fixing the input size as n we can write the expected performance parameters:

$$
\begin{aligned}
T_{seq} &= n \\
T_{par}(p) &= 2\frac{n}{p} + p \\
sp(p) &= \frac{T_{seq}}{T_{par}(p)} = \frac{n}{2\frac{n}{p} + p} = \frac{np}{2n + p^2} \\
scal(p) &= \frac{T_{par}(1)}{T_{par}(p)} = \frac{2n + 1}{2\frac{n}{p} + p} \\
\epsilon(p) &= \frac{sp(p)}{p} = \frac{n}{2n + p^2}
\end{aligned}
$$

So, in the case $p \ll n$ we can approximate these values:

$$
\begin{aligned}
T_{seq} &= n \\
T_{par}(p) &\simeq 2\frac{n}{p} \\
sp(p) &= \frac{T_{seq}}{T_{par}(p)} \simeq \frac{n}{2\frac{n}{p}} = \frac{p}{2} \\
scal(p) &= \frac{T_{par}(1)}{T_{par}(p)} \simeq \frac{2n}{2\frac{n}{p}} = p \\
\epsilon(p) &= \frac{sp(p)}{p} \simeq \frac{1}{2}
\end{aligned}
$$

Note. These performance parameters are valid in a "PRAM-like" model, i.e. where there's no communication time between processors, no contention for access to shared memory and any memory location is uniformly accessible from any processor. These are assumptions that hold sufficiently well enough on the architecture that supports the implementations presented in the next chapter.

4. Implementations

Three implementations of the abovementioned algorithm are provided. The first one uses only C++11 threads and data structures, while the second and third one are built upon the FastFlow framework with low-level constructs like `ff_node`. All three implementations are meant to be used as "header libraries" and their exposed interface is the same, making it easy for the programmer to choose another implementation without changing anything else. Also, all three implementation provide the user with

the option to use the sequential version of the scan operation by setting the parallelism degree to 0 (setting it to 1 actually builds all the necessary objects and data structures for parallel execution and then uses only one thread/node for computation).

All code is adequately commented with Doxygen annotations.

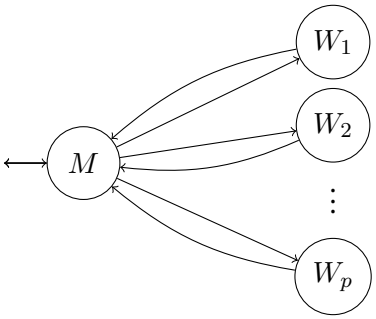
4.1 C++11 implementation (`ParallelPrefix.h`)

The C++11 implementation is built using a master-worker template (fig. 3), where the main thread assigns tasks to the workers, collects the first phase results (reduction), computes the "middle" exclusive scans, sends back the results to the workers and finally collects the second parallel phase results.

Communication between master and workers is realized through a set of synchronized queues (called "task channels" in the code), each one implementing a single producer - single consumer pattern. Moreover, since the master has to wait for the workers to finish the first phase, a very simple barrier object has been implemented. Tasks are built as objects with a "type" field to instruct workers on the job they need to do (reduction step, inclusive or exclusive scan step).

The input vector is copied into the `ParallelPrefix` object and directly accessed by all threads concurrently, exploiting the fact that the threads model is a shared memory one and that all accesses by workers are read-only (the master is the only entity that can modify the vector, and it does so only at the end of the computation when all other threads are joined).

Figure 3: The master-worker scheme used in the C++11 implementation.



4.2 FastFlow single farm version (`ff_parallel_prefix.h`)

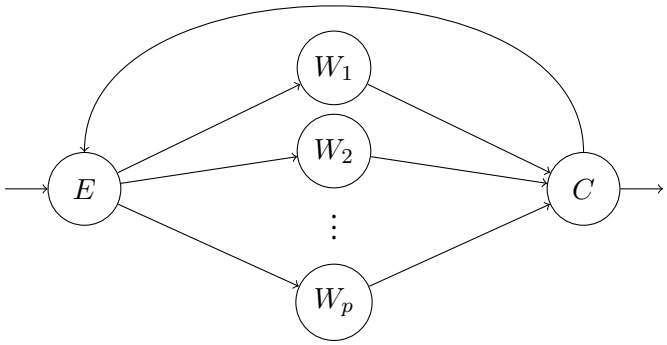
The "main" FastFlow implementation uses a single farm with wrap-around to implement the parallel algorithm (figure 4).

An Emitter node partitions the input vector and sends each portion to a different Worker, incapsulating it into a Task with type `FIRST_PHASE`. The Worker receives the Task and performs a sequential reduction over the elements of the portion enclosed in it, then sends the result to the Collector. The Collector waits until all the results (still encapsulated in `FIRST_PHASE` Tasks) are received, then performs the sequential exclusive scan (step 2 of the algorithm) over the reduced values, and finally sends the Tasks (that now contain the result of the middle exclusive scan) back to the Emitter.

When the Emitter receives a Task, it changes its type to `SECOND_PHASE` and forwards it to a Worker (that may or may not be the same Worker that processed the task before). When the Worker receives a `SECOND_PHASE` Task, it performs an (exclusive or inclusive) scan over the enclosed portion of the input using the value computed by the Collector at the previous phase as initial value for the operator of the scan. When the scan over the portion has finished, the Worker sends the Task to the Collector. When the Collector receives a `SECOND_PHASE` Task, it extracts the enclosed vector and puts it in the right place in the output vector.

Unlike the C++11 implementation, in both FastFlow implementations the input array is physically partitioned and encapsulated into Task objects.

Figure 4: The single farm model used in the first FastFlow implementation.

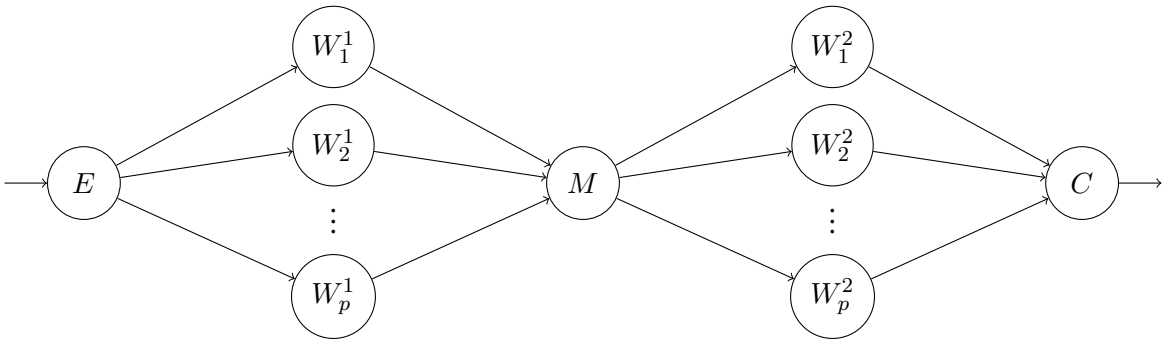


4.3 FastFlow two-farms pipeline version (`ff_parallel_prefix_v2.h`)

This variant implementation fully models the three phases outlined in chapter 3 with a (logical) three stages pipeline, as seen in figure 5 – the three stages are:

- the Emitter plus the first set of Workers implement the initial reduction phase;

Figure 5: The two-farms model used in the variant FastFlow implementation.



- the Middle node – dubbed `FirstCollector` in the code – acts as a barrier and performs the second stage sequential exclusive scan;
- the second set of Workers and the true Collector implement the third phase, the sequential inclusive or exclusive scan over a portion of the input.

Note that for simplicity this is implemented as a two-stage pipeline (the first stage being `Emitter` and `Phase1Workers`, the second stage being the rest). Since the Workers are now fully divided in two groups, there's no need to differentiate between task types; other than this peculiarity, most of the code is the same as the "main" FastFlow implementation.

This implementation was actually the first one to be devised, but it has one huge downside: due to how FastFlow works, all nodes are created and started as the same time, thus creating $2p$ threads when the user states a parallelism degree of p. This causes, for example, the execution on the PHI KNC server to fail when a parallelism degree of 256 is selected.

5. Experimental validation

A number of experiments using the provided `prefix_sum` program (see chapter 6) have been executed. The following parameters were fixed:

- input data size = $1.048.576$
- RNG seed = 0
- "waste iterations" to increase computation time = 512.

Then, for each implementation, there were performed 10 measurements with increasing parallelism degree ($p \in [0, 1, 2, 4, 8, 16, 32, 64, 128, 256]$); the whole process itself then is repeated 10 times for each implementation, leading to 100 measurements for each one of them – 10 with parallelism degree 0, 10 with parallelism degree 1, and so on and so forth. From each set, two outliers were discarded – usually the maximum and minimum measured time – and then the average was computed.

These are the resulting plots for the three main parameters, i.e. speedup, scalability and efficiency.

Figure 6: Data sets respectively for speedup, scalability and efficiency.

P	C++11	FF	FFV2	Expected
1	0.5	0.49	0.32	0.5
2	0.99	0.98	0.67	1
4	1.95	1.93	1.34	2
8	3.88	3.85	2.66	4
16	7.51	7.57	5.31	8
32	14.34	14.92	8.93	16
64	23.3	24.84	14.76	32
128	36.79	37.04	18.09	64
256	44.4	26.69	0	128

P	C++11	FF	FFV2	Expected
1	1	1	1	1
2	2	2.01	2.07	2
4	3.92	3.95	4.14	4
8	7.81	7.88	8.26	8
16	15.12	15.51	16.45	16
32	28.86	30.56	27.69	32
64	46.89	50.85	45.76	64
128	74.02	75.84	56.1	128
256	89.35	54.65	0	256

P	C++11	FF	FFV2	Expected
1	0.5	0.49	0.32	0.5
2	0.5	0.49	0.33	0.5
4	0.49	0.48	0.33	0.5
8	0.48	0.48	0.33	0.5
16	0.47	0.47	0.33	0.5
32	0.45	0.47	0.28	0.5
64	0.36	0.39	0.23	0.5
128	0.29	0.29	0.14	0.5
256	0.17	0.1	0	0.5

Figure 7: Data plot for the speedup.

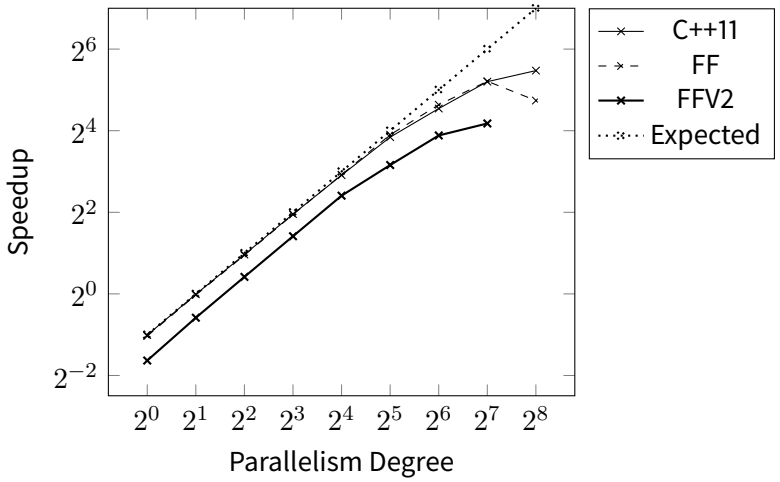


Figure 8: Data plot for the scalability.

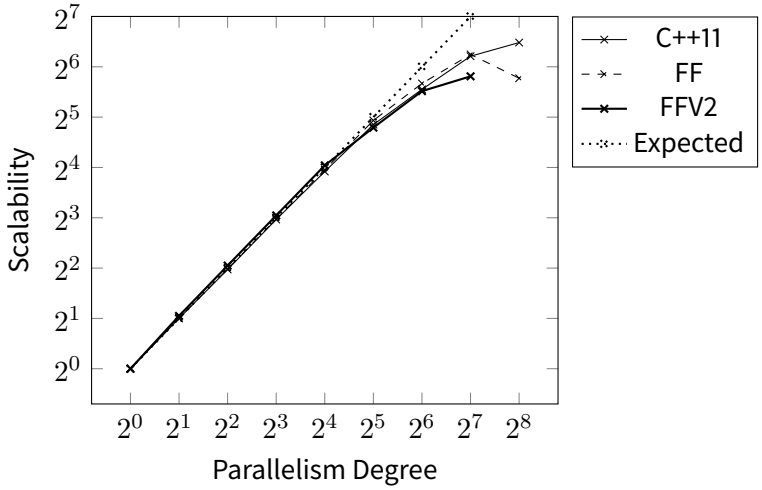
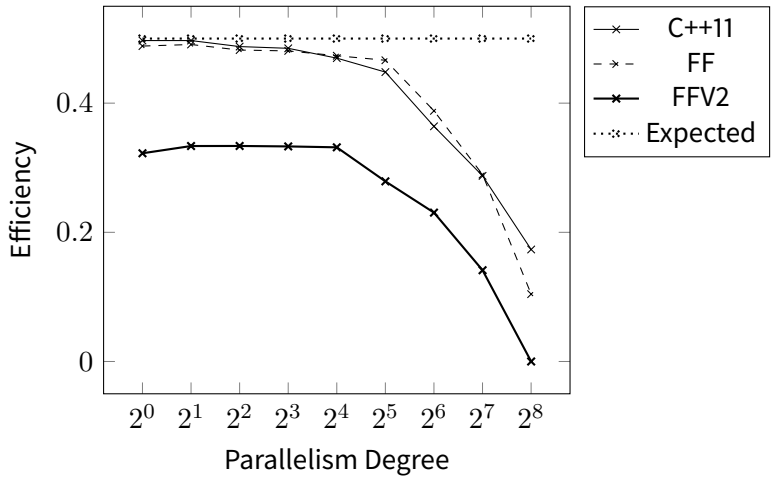


Figure 9: Data plot for the efficiency.



As we can see from the plots, the C++11 and "main" FastFlow implementations match the expected results up until a parallelism degree of 2^6, after which the performance worsens. The alternative FastFlow implementation suffers from the long setup time, then scales pretty well (w.r.t. the performance with only 1 processing element). The fact that the PHI KNC server was being utilized by other students during the tests certainly may have had a negative impact on performance, especially on higher parallelism degrees.

All the test data can be found into the `logs` folder, together with spreadsheets used to calculate averages.

6. Submission and concluding remarks

Inside the submission folder, a simple application for computing prefix sums is included in three versions (one for each implementation). Moreover, a `Makefile` is provided, together with some `bash` scripts used for generating the plot data and to check the correctness of the implementations using test data generated with a simple sequential program using the `std::inclusive_scan` operation.

The `Makefile` targets are:

- `thread` to build the prefix sum executable using the C++11 implementation;
- `ff` to build the prefix sum executable using the main FastFlow implementation;
- `ffv2` to build the prefix sum executable using the variant FastFlow implementation;
- `dataplot_thread` to generate the data used for the report's plot, C++11 impl;
- `dataplot_ff` to generate the data used for the report's plot, main FF impl;
- `dataplot_ff_v2` to generate the data used for the report's plot, variant FF impl;
- `test` to perform the correctness test.

The project was written using the CLion IDE on macOS High Sierra 10.13.5 on my machine, and the GNU `vim` and nano utilities on the PHI KNC server.