1 Implementation Design

1.1 Design Choices

The class `Stencil` holds both the parallel implementation using the FastFlow library and using the native C++ threads. The one using C++ threads can be called with the method `stdthread`. The operator `()` instead will use the FastFlow library. The class can also be used as a node; an example is given in the file "main.cpp", where using the function `fastflow` creates a pipe between the reader, the stencil and the writer.

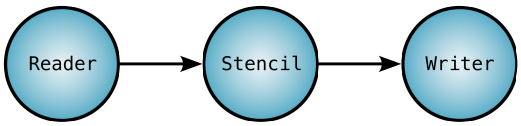


Figure 1:

The class `Reader` reads a binary file composed of 4 bytes representing the number of rows, 4 bytes representing the number of columns and then the raw matrix data. Each element is a `char`. The result is stored in the class `Task` which will be passed to the next node. If instead the operator `()` is called, only the data will be returned as a pointer.

The `Writer` instead writes to disk the task to the same folder, overwriting existing files if present.

1.2 Native C++ Threads

The structure of the implementation with native C++ threads is as follows:

The threadpool is implemented in the `threadPool.hpp` and `threadPool.cpp` files.

Since for each element the work is equivalent, the Δ used in the lambda function is simply the total number of rows divided by the number of workers, such that each worker has only one job and all jobs are roughly equal in time.

The threadpool uses a queue and once a job is pushed only one thread may execute the function. Since it is required for all jobs to finish, a condition variable is used to wake a thread that is waiting for all jobs to finish, eliminating the need for active wait.

1.3 FastFlow

The structure of the implementation using the FastFlow is similar to the one with native threads. Since the `Stencil` class is a subclass of `ff_Map`, the method used for the execution is `parallel_for`.

A custom emitter and collector would not have been faster and so the simpler approach of inheriting the methods from `ff_Map` was chosen.

2 Performance Analysis

The matrix data inside the class `Task` was both tested for performance as a vector of vectors and as a simple contiguous arena. The performance was exaclty the same so the simpler vector of vectors implementation was preferred.

In the file `main.cpp` a csv file is created from various tests on files from the `tests/` directory. The time computed is for reading the file from disk, computing the stencil with different parameters and finally writing again to disk. Reading and writing to disk are much faster than

```
1: procedure STDTHREAD(Input, Output)
2:     for result ∈ Input do
3:         arena = result
4:         while iter > 0 do
5:             for thread ∈ ThreadPool do
6:                 send a new LAMBDA with appropriate bounds to the threadpool
7:             end for
8:             swap arena with result
9:             iter = iter − 1
10:        end while
11:        wait for the threadpool to finish
12:        push result to Output
13:    end for
14: end procedure
```

```
1: procedure LAMBDA(l, Δ)  ▷ l is the thread number, Δ is the ammount of rows to process
2:     for x ∈ {l · Δ, . . . , (l + 1) · Δ − 1} do
3:         for y ∈ {0, . . . , Columns} do
4:             if then(x, y) not in the border
5:                 calculate the neighborhood of (x, y)
6:                 arena[x][y] = Stencil(neighborhood)
7:             end if
8:         end for
9:     end for
10: end procedure
```

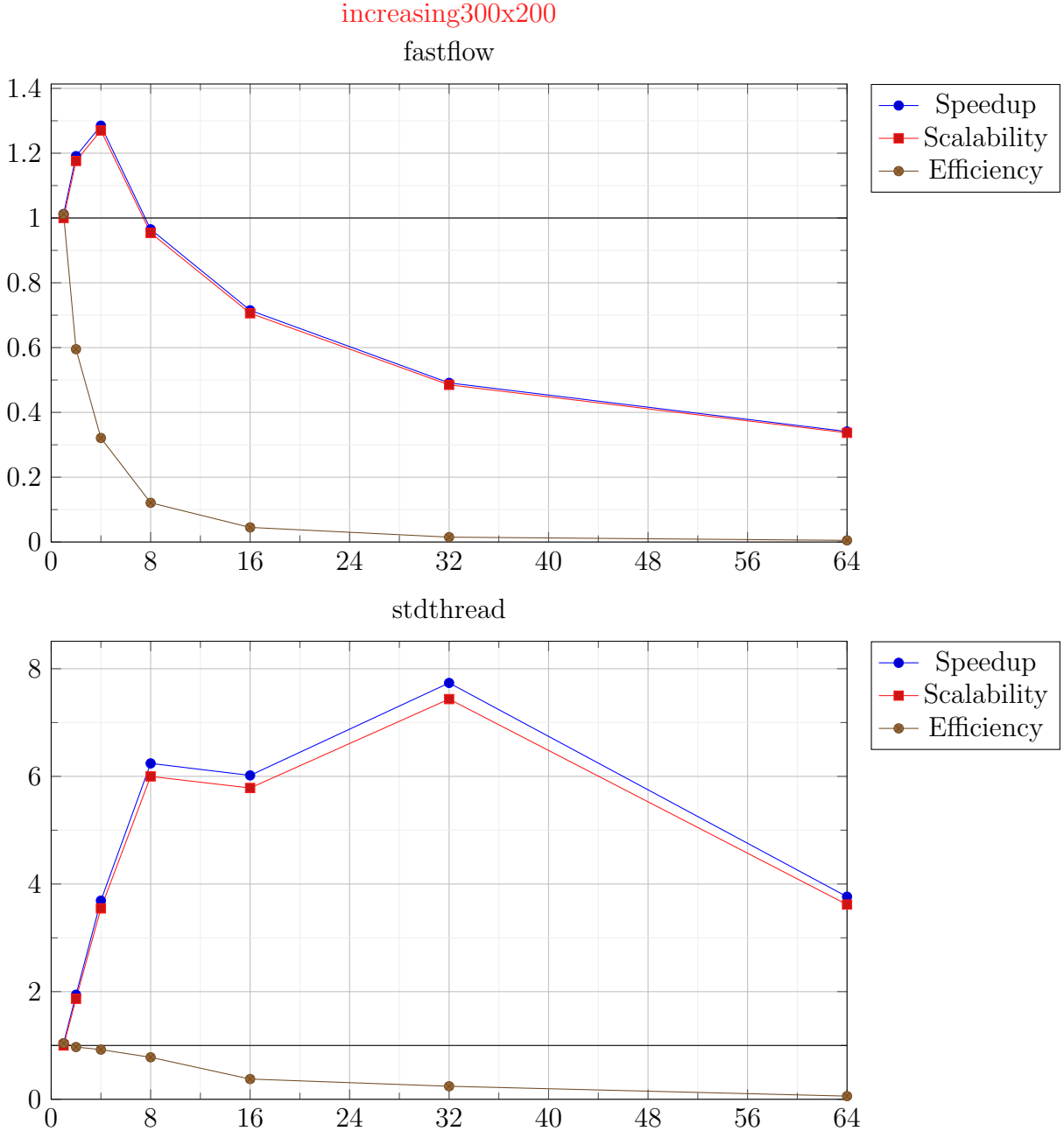
```
1: procedure FASTFLOW(Task)
2:     arena = Task
3:     while iter > 0 do
4:         parallel_for with LAMBDA as the function to execute
5:         swap arena with result
6:         iter = iter − 1
7:     end while
8:     return task
9: end procedure
```

```
1: procedure LAMBDA(x)
2:     for y ∈ {0, . . . , Columns} do
3:         if then(x, y) not in the border
4:             calculate the neighborhood of (x, y)
5:             arena[x][y] = Stencil(neighborhood)
6:         end if
7:     end for
8: end procedure
```

the computation except for the largest examples. In those cases the minimum time of reading and writing is subtracted.

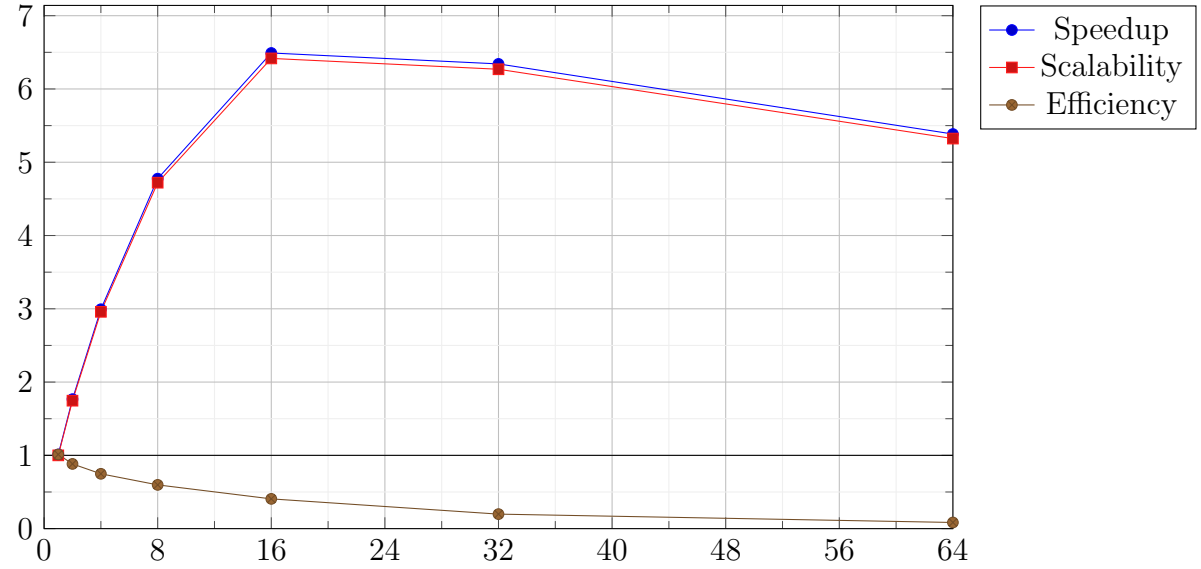
For very small matrices the efficiency, the speedup and the scalability is very poor for both versions. For larger examples instead a significant speedup is seen, but the implementation using native threads is slightly faster.

Image	Time in µs	Size in B
empty2x2	2218	12
increasing4x6	2054	32
increasing300x200	1301	60008
random400x2500	7101	1000008
equation	786324	10000008
equation2	2312927	30000008

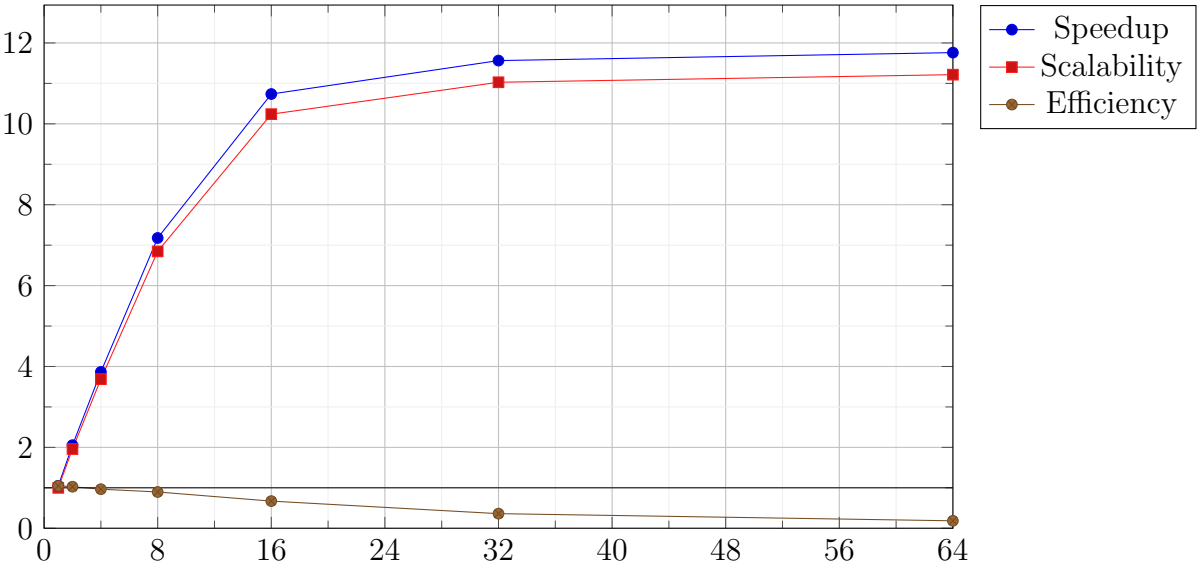


random400x2500

fastflow

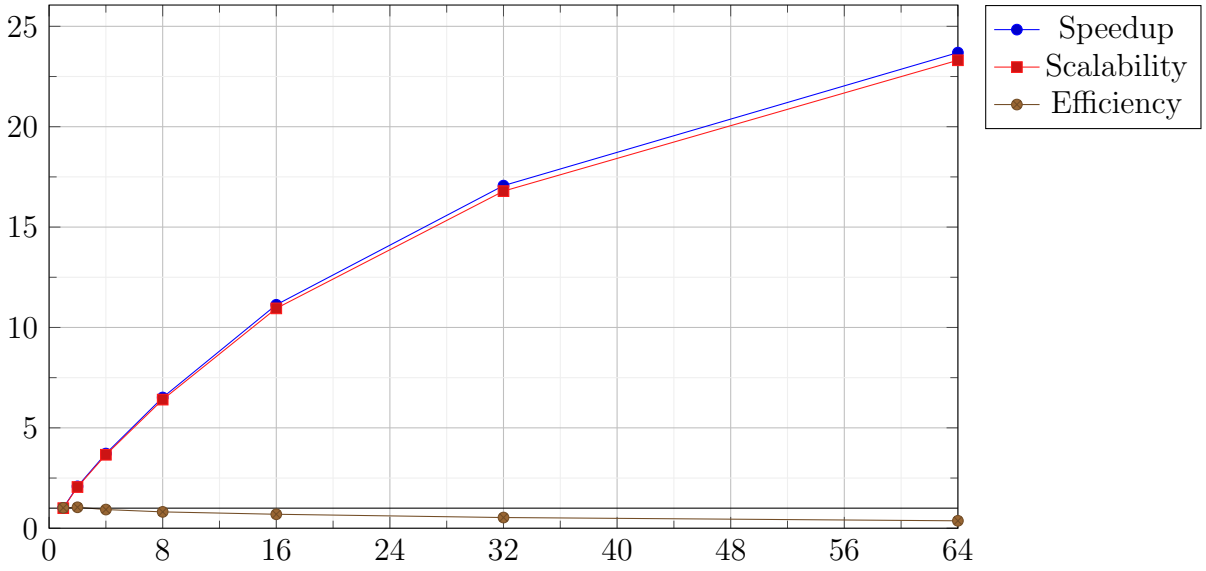


stdthread

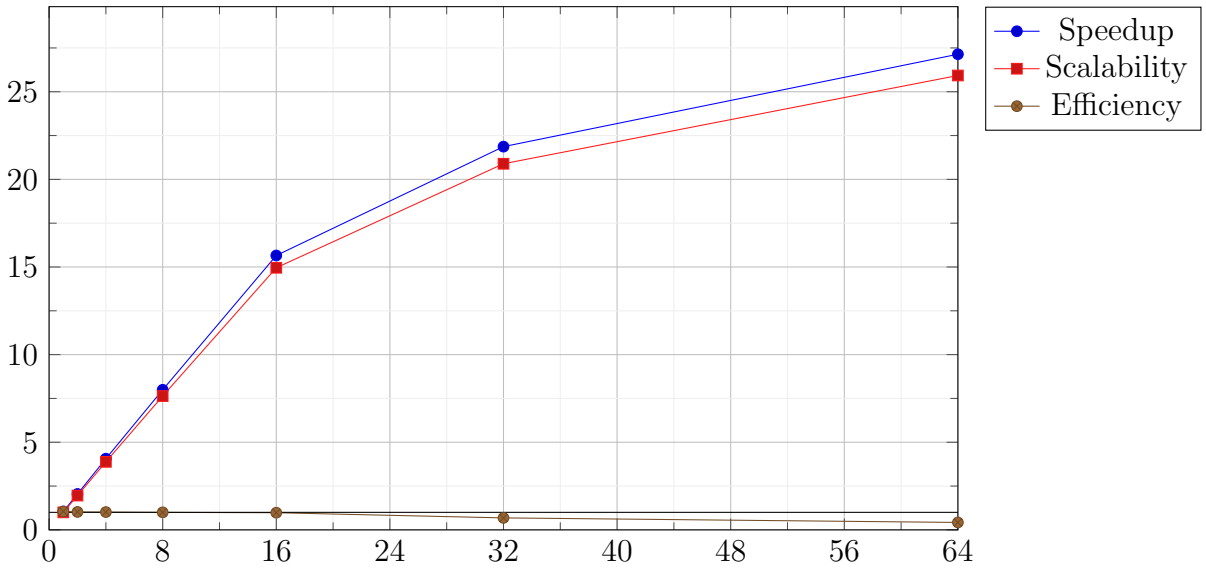


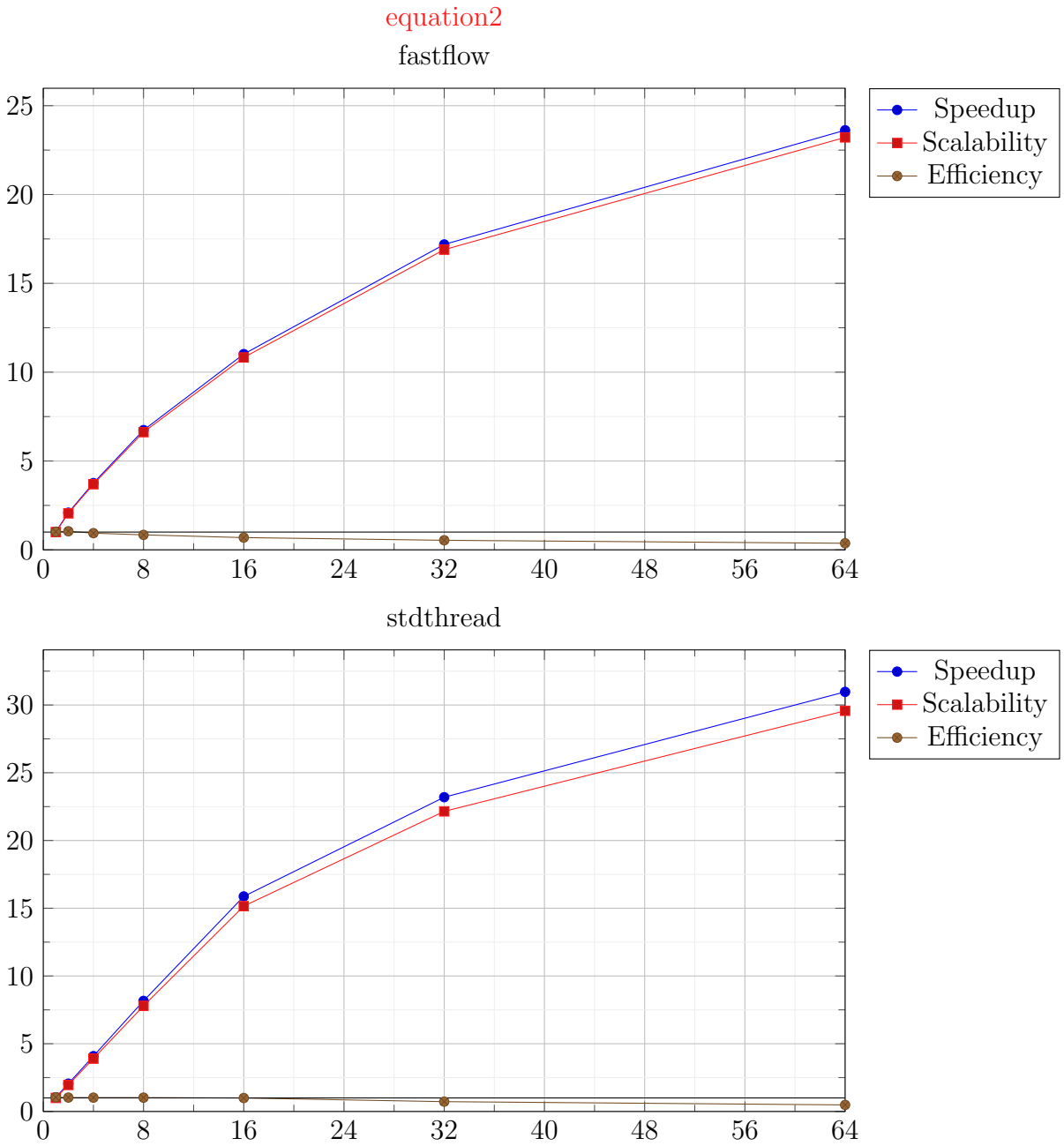
equation

fastflow



stdthread





As the size of the input increases the speedup and the scalability both follow linear trends up with a higher ammount of threds.

The scalability for both test files `equation` and `equation2` never go below 0.37, but is slightly better for the implementation with native C++ threads.