Università di Pisa

Master Degree in Computer Science

Report for Parallel and Distributed Systems: paradigms and models

"Stencil" parallel pattern

Teachers:
Prof. Marco Danelutto
Prof. Patrizio Dazzi

Student:
Elvis Rossi
ID: 561394

Academic year 2022—2023

Contents

1 Building and Executing the project

The project uses `cmake` to create the native makefiles. The flag `CMAKE_BUILD_TYPE` can be used to specify the type of build; two options are supported: `Debug` and `Release`. The main file creates a `.csv` file with the execution time of different test cases with input files located in `./tests`. On MacOS, thread pinning for the Fastflow library is disabled since is not supported by the operating system.

To compile and run the project:

```
cmake -DCMAKE_BUILD_TYPE=Release -S . -B build/
cd build/
make
./main
```

2 Implementation Design

2.1 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.

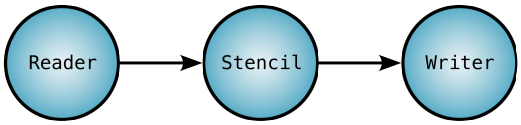


Figure 1:

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` in all the test cases. 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 `Task` class can support matrices of different element type rather than `char`.

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

The `Stencil` class divides the matrix in roughly equal parts and distributes them to other workers.

2.2 Native C++ Threads

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

```
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:        append result to Output
13:    end for
14: end procedure
```

```
1: procedure LAMBDA(l, Δ)        ▷ l is the index of block of rows, Δ is the number of rows
2:     for x ∈ {l · Δ, ..., (l + 1) · Δ − 1} do
3:         for y ∈ {0, ..., Columns} do
4:             if (x, y) not in the border then
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
```

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.

2.3 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.

```
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 Task
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 (x, y) not in the border then
4:             calculate the neighborhood of (x, y)
5:             arena[x][y] = Stencil(neighborhood)
6:         end if
7:     end for
8: end procedure
```

3 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 exactly 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. Instead of averaging the times of different runs, the minimum of the runs is chosen since outliers skew the mean greatly. Reading and writing to disk are much faster than the computation except for the largest examples. In those cases the minimum time of reading and writing is subtracted.

Since

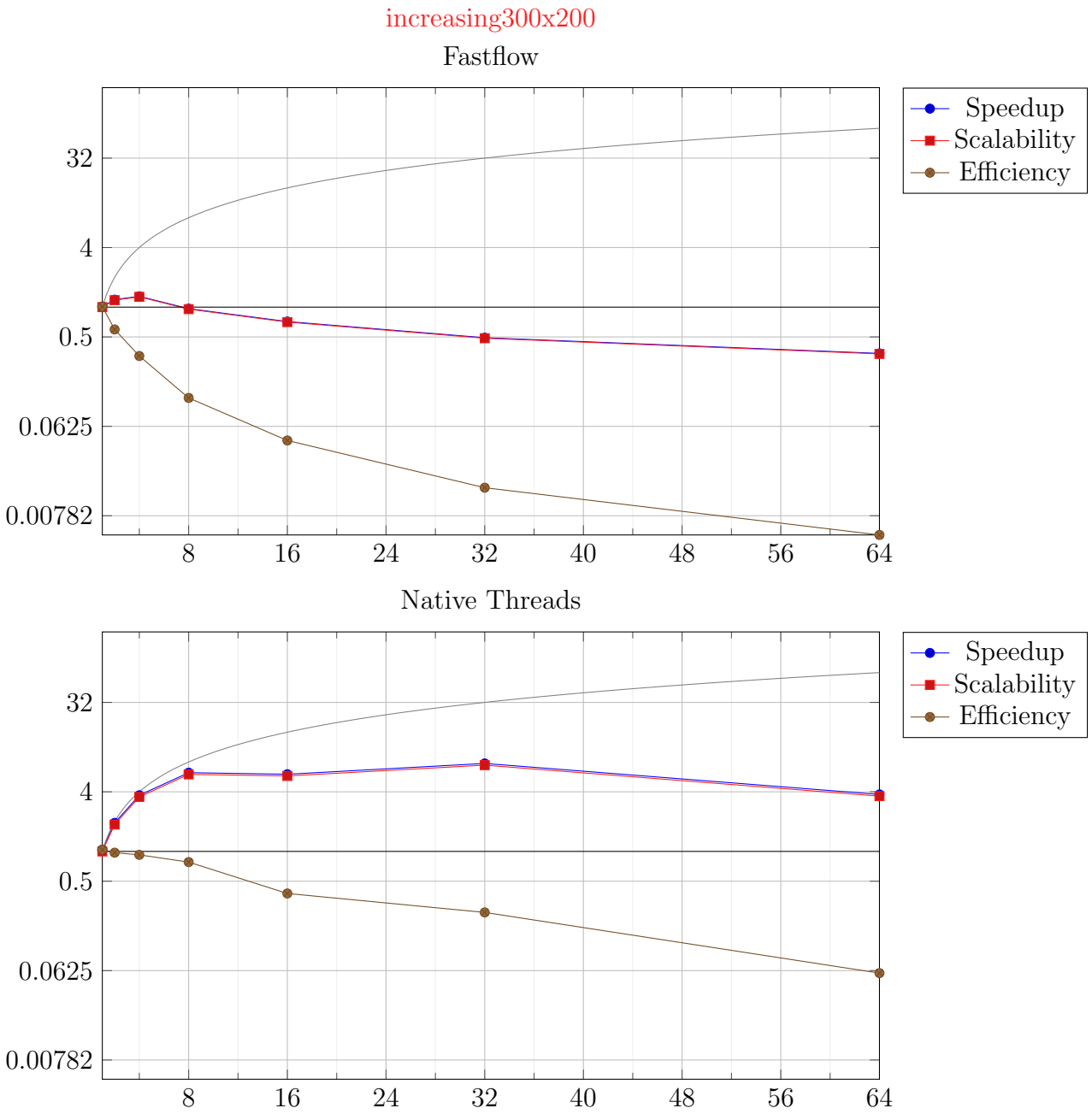
$$T_{\text{total}} = T_{\texttt{Reader}} + T_{\texttt{Stencil}} + T_{\texttt{Writer}}$$

and the value of $T_{\texttt{Reader}} + T_{\texttt{Writer}}$ is known on average then the values speedup, scalability and efficiency are calculated as follows

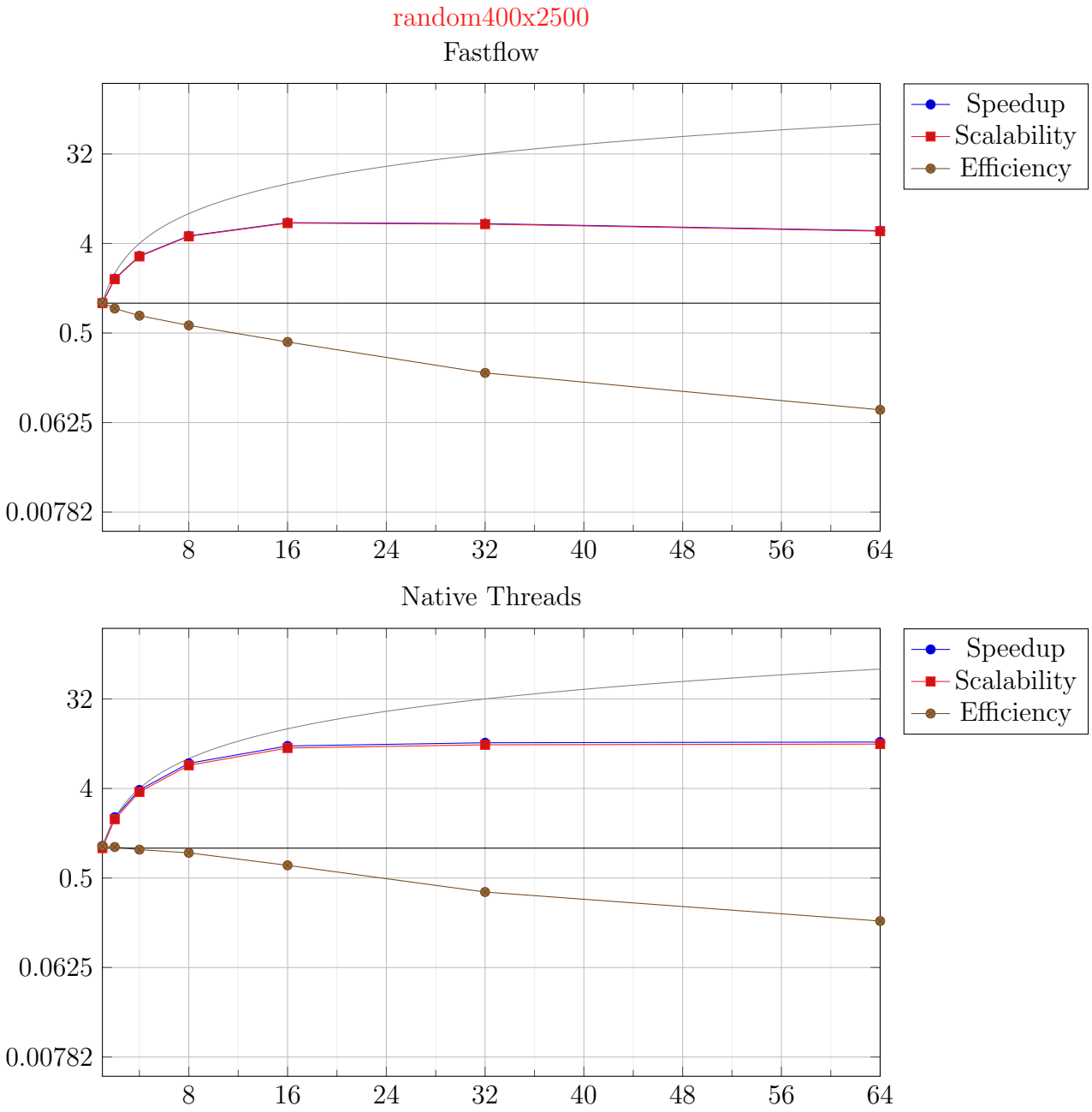
$$\begin{aligned}
\text{Speedup}(n) &= \frac{T_{\text{seq}}}{T_{\text{par}}(n) - (T_{\texttt{Reader}} + T_{\texttt{Writer}})} \\
\text{Scalability}(n) &= \frac{T_{\text{par}}(1) - (T_{\texttt{Reader}} + T_{\texttt{Writer}})}{T_{\text{par}}(n) - (T_{\texttt{Reader}} + T_{\texttt{Writer}})} \\
\text{Efficiency}(n) &= \frac{\text{Speedup}(n)}{n}
\end{aligned}$$

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	$T_{\texttt{Reader}} + T_{\texttt{Writer}}$ in µs	Size in B
empty2x2	2218	12
increasing4x6	2054	32
increasing300x200	1301	60008
random400x2500	7101	1000008
equation	786324	10000008
equation2	2312927	30000008

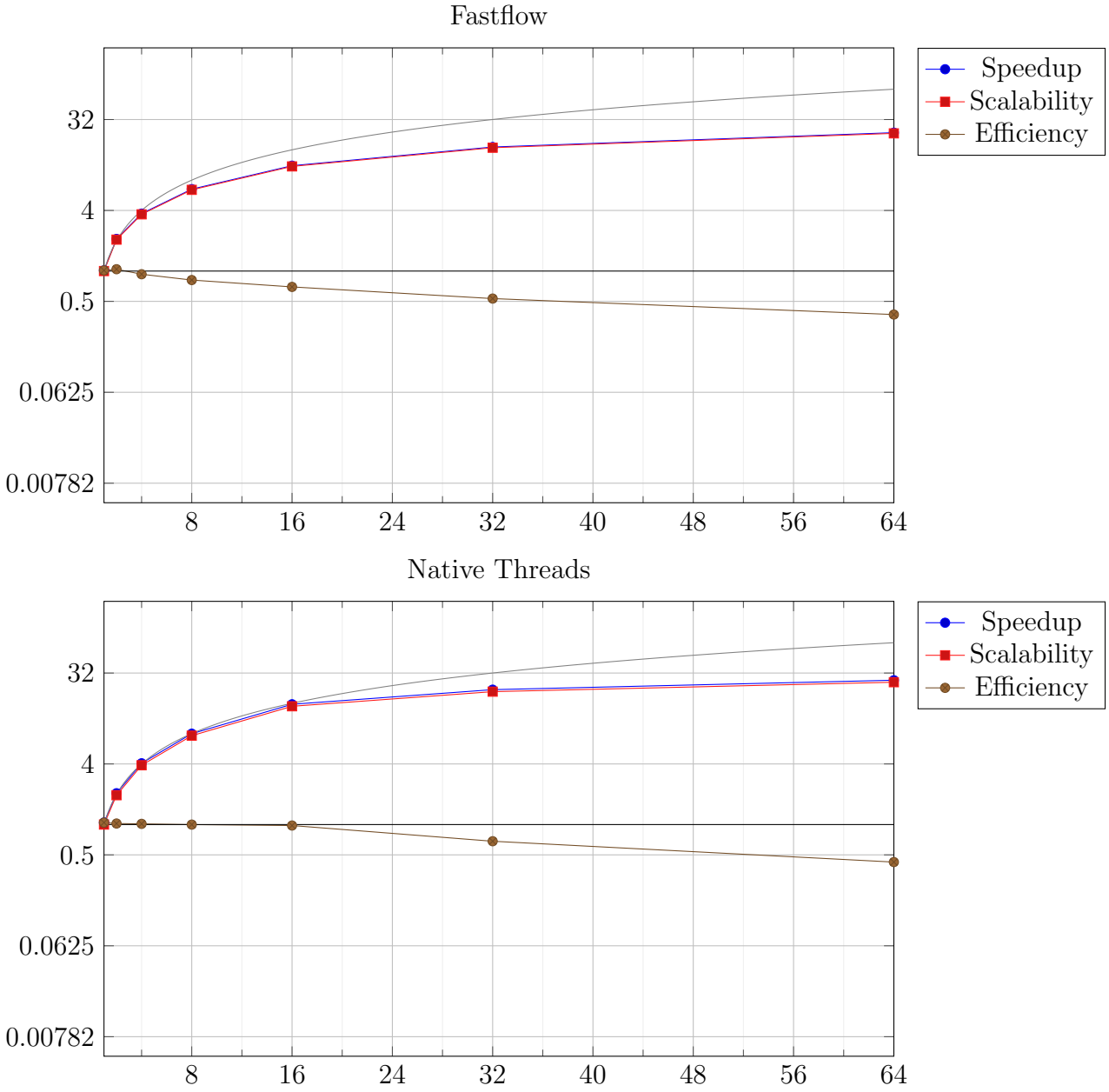


For the file `increasing300x200` the fastflow has a peek of speedup and scalability when using 4 workers in the stencil stage but quickly looses performance due to the small size of the input. For the native thread version instead the speedup and the scalability always stays above 1 but has a peek at 32 workers.

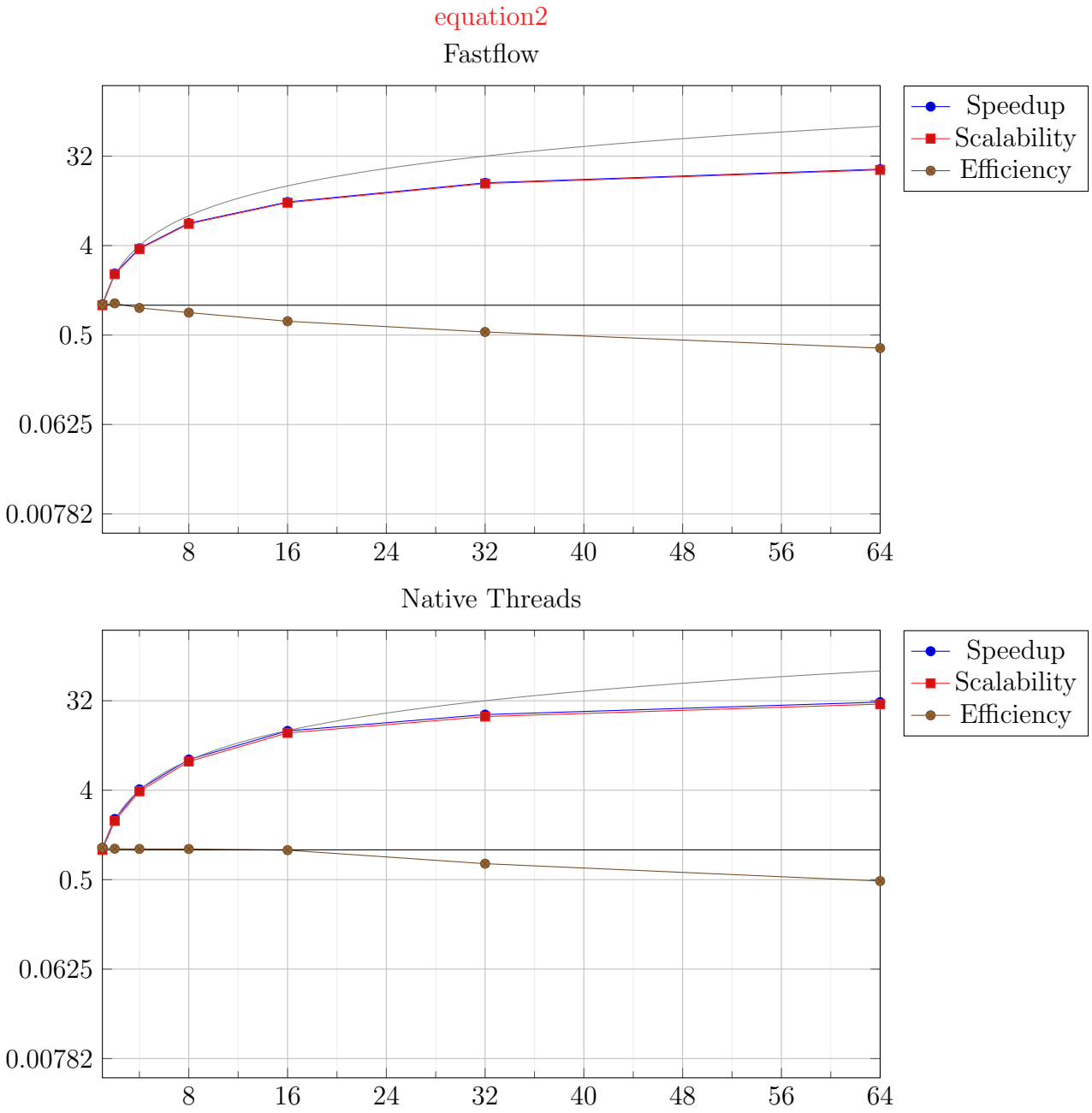


The file `random400x2500` performs best with 16 workers in the Fastflow implementation and slightly better at 64 workers compared to 32 workers in terms of speedup and scalability but has a significant drop in efficiency from 0.361 to 0.184. The relationship between number of workers and speedup is close to linear up to 8 workers.

equation



The file `equation` more closely follows a linear relationship between speedup or scalability and number of workers for both versions.



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

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.

The difference in the three quantities between the test with file `equation` and the test with file `equation1` is much smaller for the Fastflow version. In the native thread version instead there is a small improvement especially with a higher number of workers.