Introduction to FastFlow programming

Massimo Torquati
<massimo.torquati@unipi.it>

SPM lecture 8

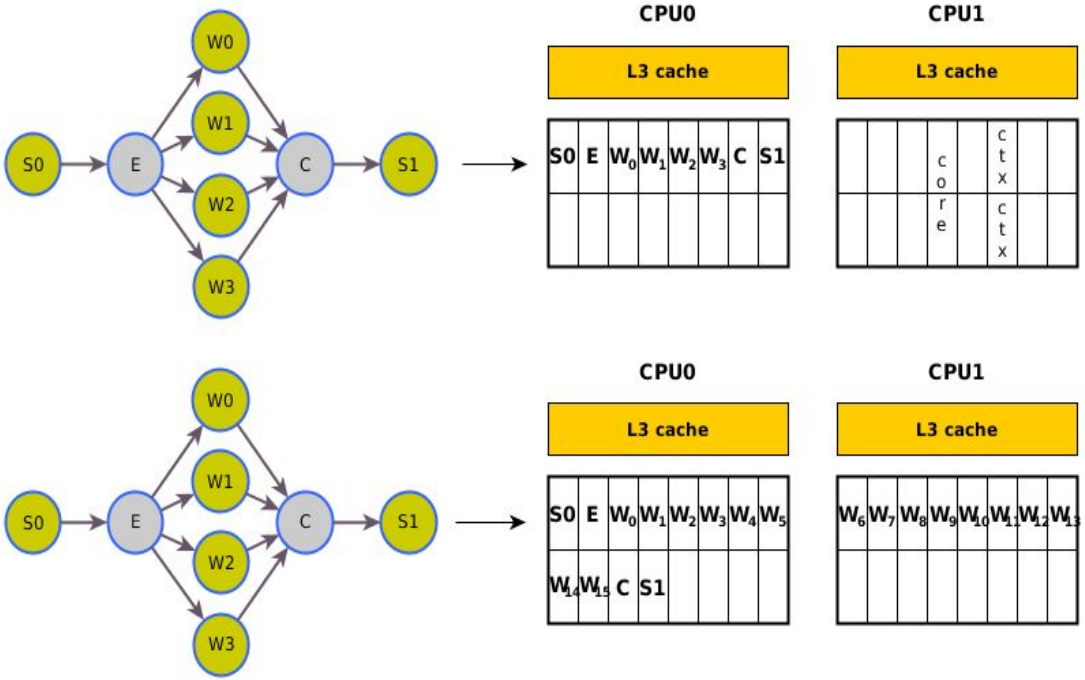
Master Degree in Computer Science
Master Degree in Computer Science & Networking
University of Pisa

Processor affinity (CPU affinity)

- It is a feature provided by several OS to control threads (processes) placement onto CPU cores
- The placement of threads onto cores is also called *thread mapping* (also *thread pinning*)
- Deciding the optimal mapping of threads to core is NP-hard
- There are several heuristics that work well in practice, which are implemented by the OS
- If the platform and the application structure is known in advance, it is possible to derive good mapping heuristics

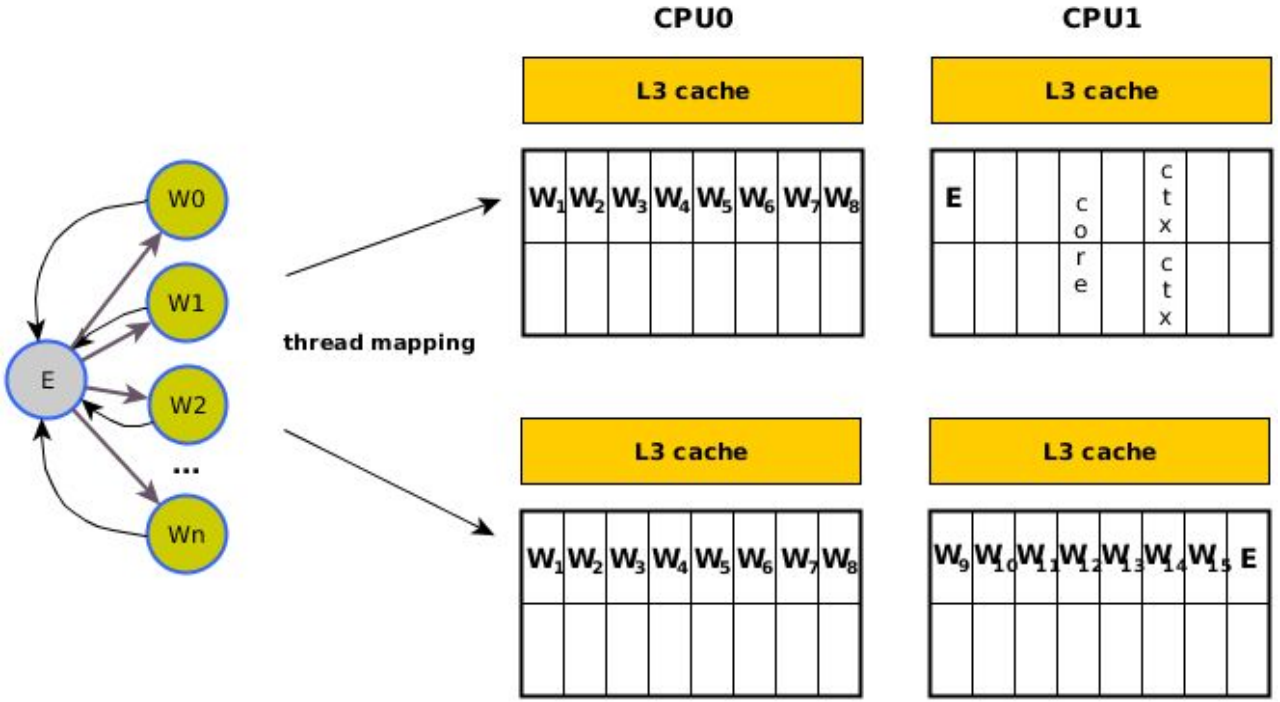
Threads mapping/pinning in FastFlow

- Default threads mapping: **linear**. Depth-first visit of the graph.
- Threads are pinned to sibling cores. Good heuristic in the Producer-Consumer model
- However, that it is not always a good choice: if the working set does not fit in the last-level cache, the performance could be worse
- The **NO_DEFAULT_MAPPING** compilation flag disables the default thread mapping
- It is possible to map threads in the svc_init() “by hand”



- The mapping could be managed by using the **optimize_static** function or by using the **no_mapping()** method offered by the top-level farm, A2A and pipeline BBs.

Controlling threads mapping in FastFlow



- Let's have a look at the example *farm_pinning.cpp* showing some mechanisms

Concurrency Control

- We already know that a communication channel connecting two FastFlow nodes is implemented by using a **non-blocking** FIFO queue (Single-Producer Single-Consumer)
  - Non-blocking is the default concurrency control mode (see SPM-lecture 1)
- If the number of threads used is greater than the number of cores, the busy-waiting loop executed by the threads might introduce too much noise for other threads
  - Suppose a collector node that “sporadically” receives data elements that share the core with a Worker (or worse with the Emitter) in a farm
- In these cases, a **blocking** concurrency control mode for accessing the queues might be more appropriate even though it is intrinsically less reactive (it has more overhead)
- In FastFlow it is possible to change to the blocking mode system-wide by compiling the code with -D**BLOCKING_MODE**

Concurrency Control

- It is also possible (as for thread mapping) to switch to blocking mode at run-time by using **optimize-static**
- Example (see, for example, *test_optimize.cpp*):

```
OptLevel opt;
opt.max_mapped_threads=ff_realNumCores();
opt.no_default_mapping=true;   // disable mapping if  #threads > max_mapped_threads
opt.max_nb_threads=ff_realNumCores();
opt.blocking_mode=true;          // enable blocking mode if  #threads > max_nb_threads
optimize_static(myPipe, opt);
```

Fast File Compressor (FFC) application example

- Problem: compress all files in a directory tree in parallel. The same of the following command:

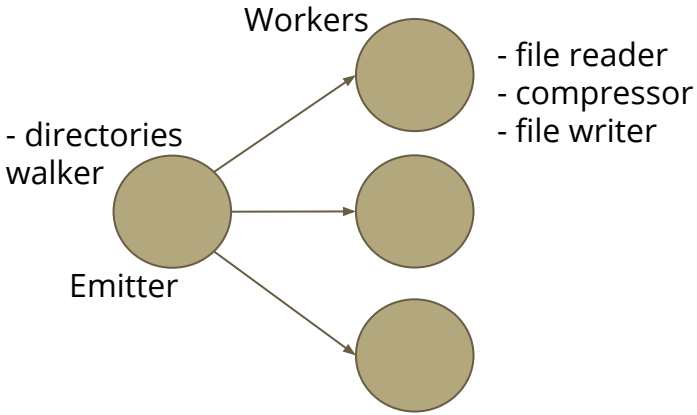
  ```
  find -type f -print0 | xargs -0 -P N -n 1 gzip
  ```

  Here ‘N’ il the number of gzip processes to spawn

- Idea: compress small file in parallel with different Worker threads, files are compressed by using miniz
- **Miniz** is a single file all-in-one compressor/decompressor
  - Miniz project: https://github.com/richgel999/miniz

FFC initial version

- Initial version

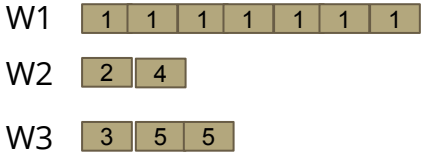


This version works well only if there are many files all having almost the same size

“Big files” problem

- The farm-based (master-worker) configuration does not help when we need to compress a few “Big files”
- Example:

Best case with 3 Workers:

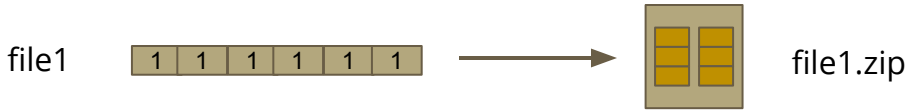


Difficult to balance the workload unless there are many files and “Small” and “Big” files are distributed uniformly in the directory tree

- What if we split “BIG files” in multiple blocks ( )?

Splitting Big files

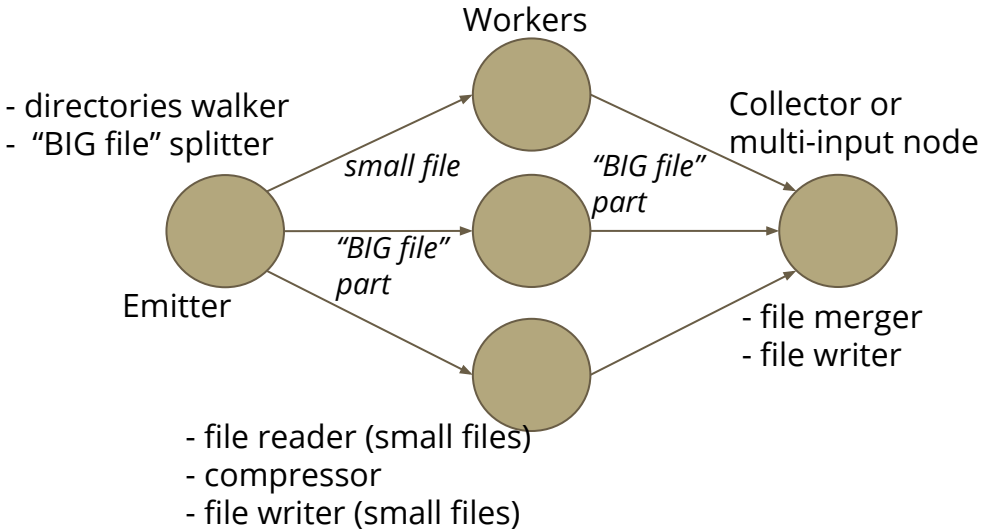
- We have to use the low-level API of Miniz
- Contiguous blocks of data are not independent (i.e., cannot be compressed in a simple way)
- … but we can split the “BIG files” in multiple files and compress them independently. Then we have to merge all compressed parts in a single (**non standard**) zip file, for example by using ***tar***.



- This means that we have to build our decompressor for such compressed “BIG files”.
- However, this approach is easier than working with Miniz streams (or gzip streams), it does not lose too much in terms of compressed size, and allows to speed-up the compression of “BIG files”.

FFC proof-of-concept solution

- We modified the base farm version so to schedule both small and “BIG files”. “BIG files” are split on the basis of a user-defined BIGFILE_LOW_THRESHOLD, compress those parts with Miniz and “merge” them in a single file
  - Naive approach for merging multiple parts (filename.part1.zip, filename.part2.zip,..., filename.partK.zip):
    tar cf filename.zip filename.part*.zip



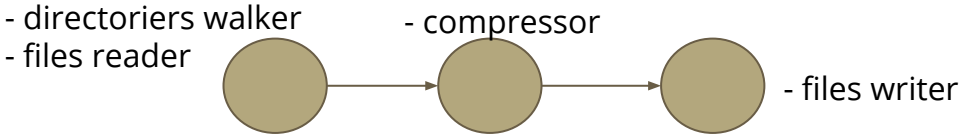
Compression of one “BIG file” 1.1GB (binary data)

Versions	Time	Compr. size
Miniz sequential	43s	261M
gzip sequential	57s	256M
gzip parallel (48)	1.8s	255M
ffc parallel (**)	1.7s	261M

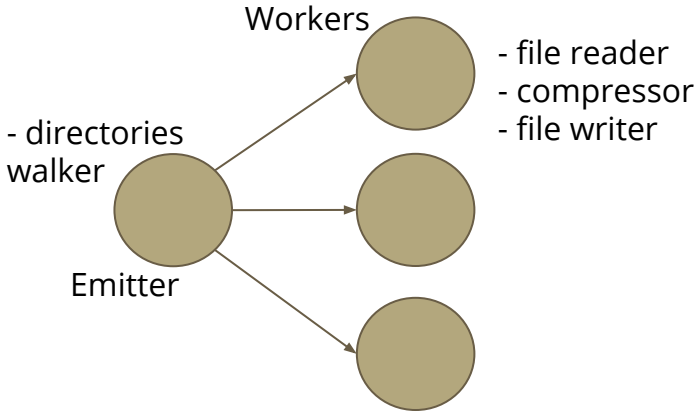
(*) dual socket Xeon E5-2695 @2.4GHz server
(**) 48 Ws, -t 10, no mapping, blocking mode

Class Work -- fast file compressor

- Implementing a fast file compressor (ffc) by using the **Miniz** single file compressor/decompressor
  - Miniz project: https://github.com/richgel999/miniz
- The sequential and the FastFlow pipeline-based versions showing how to use the Miniz features are provided as examples



- **Exercise 1:** the first step is to implement a farm-based version (easy porting of the pipeline version).

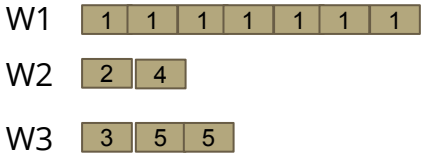


This version works well if there are many files all having almost the same size

Class Work -- fast file compressor

- The farm-based (master-worker) configuration does not help when we need to compress a few “Big files”
- Example:

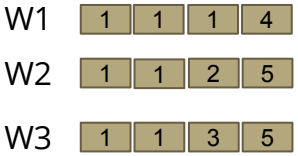
Best case with 3 Workers:



Difficult to balance the workload unless there are many files and “Small” and “Big” files are distributed uniformly

- What if we split “BIG files” in multiple blocks ( )?

The problem is that it is not easy to do this with Miniz because contiguous blocks of data are not independent ...

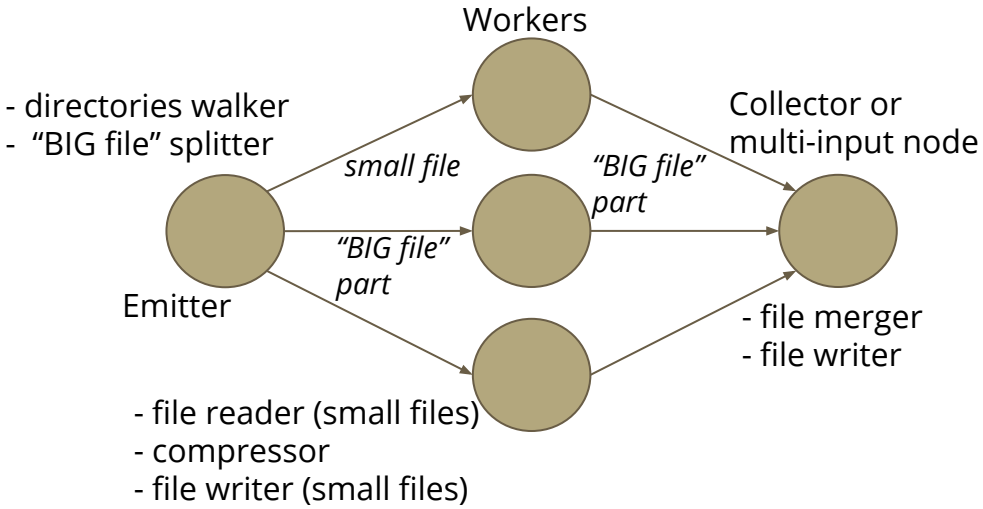


Easier to balance the workloads by using a simple on-demand policy

- … but we can split the “BIG files” in multiple files and compress them independently. Then we have to merge all compressed parts in a single (non standard) zip file, for example by using ***tar***. This means that we have to build our decompressor for such compressed “BIG files”. This approach is much easier than working with Miniz steams, it does not lose too much in terms of compressed size, and allows to speed-up the compression of “BIG files”.

Class Work -- fast file compressor

- **Exercise 2:** Modify the base farm version so to schedule both small and “BIG files”. “BIG files” are split on the basis of a user-defined BIGFILE_LOW_THRESHOLD, compress those parts with Miniz and “merge” them in a single file
  - Naive approach for merging multiple parts (filename.part1.zip, filename.part2.zip,..., filename.partK.zip):
    tar cf filename.zip filename.part*.zip



Compression of one “BIG file” 1.1GB (binary data)

Versions	Time	Compr. size
ffc sequential	43s	261M
ffc parallel (48Ws, 5M blocks) (**)	2.1s	261M

(*) dual socket Xeon E5-2695 @2.4GHz server
(**) NO_DEFAULT_MAPPING, BLOCKING_MODE