Pipeline with load balancing

Final Project for the SPM course

Francesco Balzano

Master Degree in Computer Science and Networking

A.Y. 2017-2018

Contents

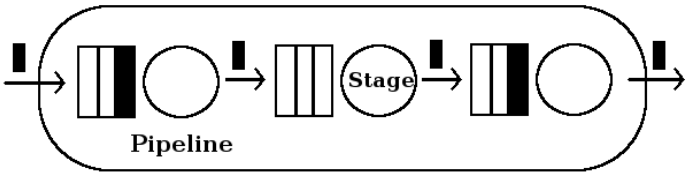

Figure 1: The pipeline is a chain of sequential stages. There is a buffer of limited size between each pair of consecutive stages.

1 Design and Implementation choices

1.1 Design

I designed a pipeline as a handler for a vector of stages. It takes care to start the execution of all the stages and to await for their termination. Each stage encapsulates a user-defined function and runs it in a separate thread of execution. Each stage takes its tasks from an input buffer and puts the results in an output buffer. Each buffer has limited size and has to guarantee that the tasks are picked with the same order in which they were put in the buffer.

1.1.1 Pipeline

A pipeline receives a list of stages and a list of tasks and:

- builds the chain, ensuring that stage i outputs the results to the input buffer of stage *i+1*;
- sends the received tasks to the first stage, actually starting the chain of computations;
- when the tasks are exhausted, sends a special task to the first stage to propagate the termination message, and awaits that all the stages terminate the execution.

In the mean time a separate thread of execution monitors the execution times of each stage of the pipeline and, for each stage, makes the average of the times. The collapsing of two consecutive stages may be triggered after a certain number of execution times has been measured for each stage. So let's suppose we are in this situation, and let i be the index of the i^{th} stage and *slowest* the index of the slowest stage: if it exists some i such that:

$$avg\_time_i + avg\_time_{i+1} < avg\_time_{slowest}$$

then the stage i and the stage $i+1$ are collapsed into a single sequential stage. I decided that collapsing actually means that:

- Stage i stops its computation until stage $i+1$ terminates and informs stage $i+1$ that it has to be collapsed;
- stage $i+1$ makes its computations on the remaining tasks (the ones already in its input buffer) and then terminates its execution;
- stage i resumes its computation: from now on it will run the computations of stage $i+1$ too.

Since in the project text it was requested only the merge of two consecutive stages, I decided that a stage can merge the adjacent stage in the pipeline, but not more than one. In other words, a stage can at most run its computation and the next stage's one.

1.1.2 Stage

A stage picks the tasks from an input buffer, applies a user-defined function to each of them and puts the results to an output buffer. It also measures the time that each computation takes. This measure does not take into account the time spent to pick an item from the input buffer or the time spent to put an item into the output buffer. The reason is that a stage could take a lot of time to pick an item because the input buffer is empty, but this may be due to a previous stage which is slow at producing tasks. So it is only measured the time that each stage takes to compute the assigned function.

1.1.3 Buffer

The data structure that implements the buffer has the following features:

- thread-safe, because it is accessed concurrently by the stage i to put a new task and by the stage $i + 1$ to pop a task
- ordered, because stage $i + 1$ must pick the tasks in the very same order in which they were put by stage i
- priority, because the "collapse" message must get the highest priority, so that is immediately picked by the receiving stage although there are other messages in the buffer

I decided that the priority of each message inserted in the buffer is explicitly assigned by the entity that puts a new message in the buffer, rather than by the buffer itself. In my opinion this gives greater flexibility, and semplifies the responsabilities of the buffer which has only to guarantee that the provided order is maintained.

1.2 Implementation

In the following paragraphs I describe, class by class, the main implementation choices. Each class is actually a struct, so their methods and members are completely visible.

1.2.1 Pipe.hpp

A Pipe is parametrized with the input type of its first Stage, `Tin`, and with the output type of its last Stage, `Tout`.
The constructor of a Pipe takes a list of Stages and builds the chain by invoking the method `add_next()` of class Stage.
To start the computation, the caller must call the method `run(list<Tin>&& input)`, providing a list of tasks as parameter. This results in the invocation of `run_manager()` in a separate thread of execution, in the repeated invocation of `monitor_times()` until a termination condition satisfied, and finally in the wait that the thread running `run_manager()` ends its execution.
The method `run_manager()` starts the execution of each Stage in a separate thread, puts all the

received tasks in the input buffer of the first Stage (possibly blocking if the buffer gets full), sends the special termination message `nullptr` to the first Stage when the tasks are exhausted and waits for the termination of each Stage. It also sets the termination condition after the termination message is sent. The reason why the termination condition is put here and not after all the stages ended their execution is that otherwise the monitoring of times could continue even after some stages ended their computations. Since the method `monitor_times()` is responsible for triggering the collapse of stages, this could end up in collapsing a stage which has already ended its computations, which has no sense.
The method `monitor_times()` takes care of monitoring the execution times and triggering the collapse of the Stages. It consists of three phases: initialization, monitoring and collapse. The last two phases are repeated until the termination condition is met.

- initialization: three vectors `count`, `avg` and `measures` are initialized. `count[i]` gives the number of execution times measured for Stage i; `avg[i]` gives the current average execution time of Stage i; `measures[i]` holds a priority queue that contains the last measures, the ones from which `avg[i]` has been computed.

- monitoring: all the Stages are scanned and their execution times are read. The same execution time is never counted twice because after it is read, it is immediately set to zero. Then for each Stage i with a fresh execution time, a measure is created and inserted in the priority queue `measures[i]`. A measure is a pair in which the first element is the execution time and the second element is the number of that execution. This number is used to order the measures: indeed if the number of measures taken from the beginning of the monitoring for Stage i exceeds the threshold `num_samples`, then the oldest measure in the priority queue is popped. Each priority queue keeps only the last measures and at steady state it keeps exactly `num_samples` measures. The measures in the priority queue `measures[i]` are the same used to compute the average execution time `avg[i]`. This average is computed incrementally every time a new measure is taken.

- collapse: this phase is run only if the execution times have been measured at least `num_samples` times for all the Stages. In this case if it exist two Stages such that the sum of their average execution times is smaller than the execution time of the slowest Stage, then they are collapsed into a single sequential stage. Two stages can be collapsed only if both of them have not been already collapsed by another Stage and did not collapse any Stage.

1.2.2 Stage.hpp

The abstract class IStage is the interface for a Stage. A Stage is parametrized with the type of its input tasks, `Tin`, and with the type of its results, `Tout`.
The constructor of a Stage takes as parameter the function that the Stage has to apply to each input task.
To run a task the caller must invoke the method `run()`, which executes `run_thread()` in a separate thread of execution and assigns the newly created thread to a member variable. This way it is possible to wait for thread completion in a second moment.
The method `run_thread()` runs the assigned function on the input tasks until a termination condition is met. Anyway, the execution of such function is stopped if the current Stage is collapsing the next Stage: in this case indeed we have to wait that the next Stage finishes all its

tasks and terminates its execution before producing new results. The execution of the function ends when `nullptr` is received in input. At this point a finalization is performed: if the Stage is terminated because it has received the last task of the stream (`collapsed` $\neq$ `-1`), then it propagates the termination message to the right Stage. If instead this Stage is terminated because it has been collapsed, it runs its function on every remaining task and then terminates without propagating the termination message.
The method `set_input()` is used to insert a new task in the input buffer of the receiving Stage. This method is also responsible for updating the count of the number of tasks received in input by the Stage. This allows to pick tasks from the input buffer in correct order.
The method `collapse_next_stage()` is responsible for collapsing the next Stage. Collapsing basically means that the next Stage executes the computation on the remaining tasks and then the corresponding thread terminates. A boolean variable `collapsing` is used to ensure that this Stage will not produce any new task in output while it is collapsing the next Stage.

1.2.3 TSOHeap.hpp

This class implements the buffer. The name stands for "Thread-Safe Ordered Heap".
It is a heap because the main data structure that it wraps is a priority queue, that is basically a heap. As already said, the heap is needed because we want both to keep the ordering of the tasks and to give the highest priority to special tasks that indicate imminent collapse. This is achieved by inserting in the heap pairs where the first element is the actual task and the second element is an integer `id` that states the order in which tasks have to be extracted from the heap. The struct "Comparator" is a functor passed to the heap to enforce the priority that we want. Since the value of `id` is increasing with the number of tasks that the heap receives, the simple ordering that we impose is that at any time the pair extracted from the heap is the one with smallest `id`. This is the reason why the heap is called ordered. Special tasks are assigned the highest priority by simply pairing them with the smallest possible integer.
Finally it is thread-safe because the two methods `pop()` and `push()` are both thread-safe. A thread that invokes the `push()` method to insert a new pair in the heap waits if the heap is full, while a thread that invoks the `pop()` method to pick the pair with smallest `id` waits if the heap is empty. Both waits are active waits, realized with the use of atomic variables, so that a thread is not descheduled from processor when waiting. For the remaining part of these methods I used a mutex to guarantee mutual exclusion in the access to the priority queue.

2 Manual

The sample program is in the file main.cpp.
A trivial makefile is provided. To compile you just need to type

- `make host`, to compile using the icc compiler without the -mmic flag and to produce the executable file `hpipe` to be run on host;
- `make mic`, to compile using the icc compiler with the -mmic flag and to produce the executable file `mpipe` to be run on mic;
- `make gnu`, to compile using the gnu compiler and to produce the executable file `gpipe`.

Let's assume we have compiled the file to run on mic. The sample program may be run in two modes:

- mode 1: `./mpipe 0`
- mode 2: `./mpipe 1`

the differences are explained in the next section. Since the goal of this program is to demonstrate that the collapsing of the stages (*i.e.* the load balancing) happens in the correct way, it may be useful to filter the output of the program adding to the previous command `| grep collapsed`.

3 Results

Since I chose to implement a pattern and not to parallelize an application, I don't have plots concerning scalability, speedup, ... to present as result. Instead, I have prepared a simple experiment (main.cpp) that should show that the pattern behaves as expected.
I have built a pipeline made of 8 stages. All the stages except the last, which prints the results, run an identity function that produces in output the same task it received in input. Anyway there are three different identity functions, which differentiate based on the time that they take to produce the result.

- Function `f` is the slowest and it is run by Stages 1, 4, 5, 6 and 7.
- Function `fast` is the fastest and it is run by Stages 2 and 3 if the program is run with the argument `0`.
- Function `fast_init` is a function that behaves as `fast` when applied on arguments smaller than 5, and behaves as `f` otherwise. Since we provide to the Stages increasing integer tasks in the range [0,100), this means that `fast_init` is fast on the first 5 tasks and slow on the remaining 95 tasks. `fast_init` is run by Stages 2 and 3 if the program is run with the argument 1.

The expected result of the experiment is the following:

- when the Stages 2 and 3 run the `fast` function, then before the end of the program at least one collapse should be triggered. The typical case is that Stage 2 collapses Stage 3, because they are the two stages running the fast function. Anyway it could also happen that Stage 1 collapses Stage 2 and Stage 3 collapses Stage 4. This last behaviour is fair too, because the only guarantee that the pipeline provides is that it collapses two stages such that the sum of their execution times is smaller then the execution time of the slowest Stage. It does not guarantee to collapse the fastest Stages first.
- when the Stages 2 and 3 run the `fast_init` function, then no collapse should be triggered because when the pipeline reaches the steady state, the fast executions have already been balanced by the following, slow executions. That's the reason why an average on the last execution times is considered.

To see whether a stage has been collapsed it is sufficient to look for a line in the standard output containing the word *collapsed*. For this reason, it could be convenient to filter the output with `grep`. A few times, a further collapse regarding Stages 7 and 8 may be discovered. The reason is that Stage 8 runs a different function to print the results, so its execution time is not perfectly aligned to the execution time of `f`.

3.1 Limitations