Memory affinity in multi-threading: the Bowtie2 case study

Claudia Misale*, Marco Aldinucci*[1],
Massimo Torquati†[2]

* *University of Torino, corso Svizzera 185, 10124 Torino, Italy*
† *University of Pisa, largo Pontecorvo 3, 56127 Pisa, Italy*

ABSTRACT

The diffusion of the Next Generation Sequencing (NGS) has increased the amount of data obtainable by genomic experiments. From a DNA sample a NGS run is able to produce millions of short sequences (called *reads*), which should be mapped into a reference genome. In this paper, we analyse the performance of Bowtie2, a fast and popular DNA mapping tool. Bowtie2 exhibits a multithreading implementation on top of pthreads, spin-locks and SSE2 SIMD extension. From parallel computing viewpoint, is a paradigmatic example of a software requiring to address three fundamental problems in shared-memory programming for cache-coherent multi-core platforms: synchronisation efficiency at very fine grain (due to short *reads*), load-balancing (due to long *reads*), and efficient usage of memory subsystem (due to SSE2 memory pressure).

We compare the original implementation against an alternative implementation on top of the FastFlow pattern-based programming framework. The proposed design exploits the high-level *farm* pattern of FastFlow, which is implemented top of nonblocking multi-threading and lockless (CAS-free) queues, and provides the programmer with high-level mechanism to tune task scheduling to achieve both load-balancing and memory affinity. The proposed design, despite the high-level design, is always faster and more scalable with respect to the original one. The design of both original and alternative version will be presented along with their experimental evaluation on real-world data sets.

KEYWORDS: Bioinformatics; Concurrency; Multiprocessors; Non-blocking synchronisation

1 Introduction

A typical DNA mapping workload is composed by millions of both short and long sequences of nucleotides (called *reads*) to be aligned against the whole reference genome. They can be processed as independent tasks. The length of a *read* is one of the key factors that determines the computational grain of the task; number of mismatches tolerated in the mapping is another one. Different biological analysis need different datasets with different distributions in the lengths of the set of *reads*. A concrete example is the concurrent, pthread-based Bowtie2, that is among the most popular and one of the fastest alignment tool running on cache-coherent multi-core platforms [CHC+13]. Bowtie2, which has been

[1]E-mail: {misale, aldinuc}@di.unito.it
[2]E-mail: torquati@di.unipi.it

deeply hand-tuned along different versions, is build as a pool of workers extracting tasks from a shared memory buffer; synchronisations are achieved by way of spin-locks. Since the alignment problem is severely memory bound and is implemented via memory hungry SSE2 primitives, a significant care have been devoted in keeping memory as local as possible within each thread. A good overall memory bandwidth is enforced by the OS that typically spreads threads onto different sockets (attached to different memory nodes).

We advocate an alternative design, where the original structure of the code has been reorganised in order to use a lock-free approach in reading data from file and in writing results. Our objective is to study which kind of improvement are achievable by using data movement and memory affinity with a lock-free implementation and to show how different algorithm implementations in a shared-memory architecture can help to improve performances. As an orthogonal aspect, it should be noted that the presented design can be easily ported onto distributed cluster of multi-cores because it is designed on top of FastFlow patterns [ADKT13], which have both a multi-core and distributed implementation. The analysis of the distributed implementation is beyond the scope of the present work.

2 Bowtie2 Implementation and its FastFlow Porting

In Bowtie2 implementation, each thread's workflow is characterized by first setting up per-thread pointers to shared global data structures and creating per-thread data structures, then cycling the following three steps: i) take a *read* (or a pair of) from the input file (global to all threads), ii) align the sequence against the genome loaded into the index file, and iii) populate global structures related to the alignment output or to general output statistics. In the first and the last of these steps, shared data should be accessed. Accesses are protected by mutexes (either pthreads locks or spin-locks). Software design is sketched in Fig. 1.

In the alternative version of Bowtie2 (*Bowtie2-FF*, from now on), we changed the original code to exhibit a high-level design using the FastFlow farm pattern. All the required code changes are very localised within a reduced chunk of code (within a single file). Basically the three main steps above mentioned. The farm pattern has specialised to behave in a master-workers fashion (see Fig. 1), where steps i) and iii) involving data sharing are mapped onto the master (Emitter E), whereas step ii) is mapped onto parallel workers (W_i). As byproduct, the Bowtie2-FF exhibits no mutexes since data dependencies are managed by the FastFlow run time using a lock-less approach.

Task balancing is automatically achieved thanks to FastFlow farm on-demand memory-affine scheduling policy, as detailed in Sec 2.2.

2.1 Data Paths and Synchronisations

The original version of Bowtie2 expects that each thread takes needed data directly from the filesystem. For this reason, it is necessary to lock the counter variable that identifies the current *read* and let each thread increment it avoiding data races. Within the FastFlow implementation, these levels of wrapping and function calls are removed and the operation of dispensing *reads* is totally deputed to a dedicated thread (Emitter). In this way, it is no more needed to lock the counter variable. The Emitter, once has taken the current sequence to be aligned, prepares the task and sends it to a worker.

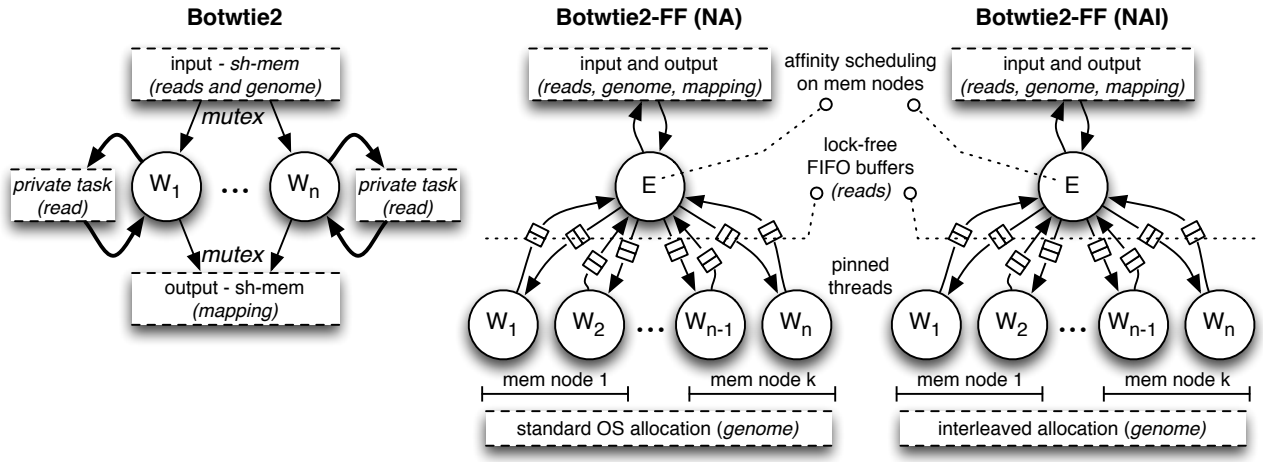

Figure 1: Architecture of Bowtie2, Bowtie2-FF with memory affinity (NA), and Bowtie2-FF with memory affinity and interleaving (NAI).

2.2 Task Scheduling and Load Balancing

As shown in figure 1, links between the Emitter (E) and each worker (W_i) and vice-versa, are implemented using lock-free FIFO queues [ADK+12]. Backward connections ($W_i \rightarrow$E), are accomplish to two main duties: 1) carry results from W_i to E, and 2) recycle exhaust tasks allocated memory in order to avoid any code change in original one, in which tasks are managed as a thread-private data. Despite FastFlow supports both bound and unbound queues, all the queues are bounded to a fixed number of elements ($QueueSize > 1$) since the scheduling policy enforces the absence of deadlocks.

The scheduling policy used is very simple: initially $QueueSize$ tasks are assigned to each worker thread by the Emitter (actually enqueued in the worker queue), then as soon as a task is get back from a worker, one new task is scheduled to the same worker (indeed, no deadlock). Tasks are allocated in the local memory of the core running the worker, which is automatically pinned to a core by the FastFlow default mapping strategy. Having fixed the size of the queues, allows to assign multiple tasks per worker obtaining the following 4 main advantages: 1) each worker has not to wait for the emitter to send him a task because tasks are immediately ready in the queue; 2) with small enough queues, the workload is dynamically partitioned among all workers thus avoiding the possible load imbalance due to long duration alignment tasks all assigned to few threads; 3) the amount of memory used during the computation is bounded to *sizeof(task)*$\times$ *N. Workers* $\times$ *QueueSize* since memory is recycled by the Emitter for the new tasks as soon as tasks come back from worker threads; 4) Workers always access the memory in their local memory node.

As further optimisation, the genome ($\approx$ 2 GB), which is shared (read-only) by all workers, can be allocated with an interleaved policy across memory nodes to exploit aggregate memory bandwidth.

3 Performance Analysis

In this section are shown two performance analysis of original Bowtie2 and Bowtie2 with numa memory pages interleaving (*Bowtie2-NI*, run with *"numactl –interleave=all"* in order to

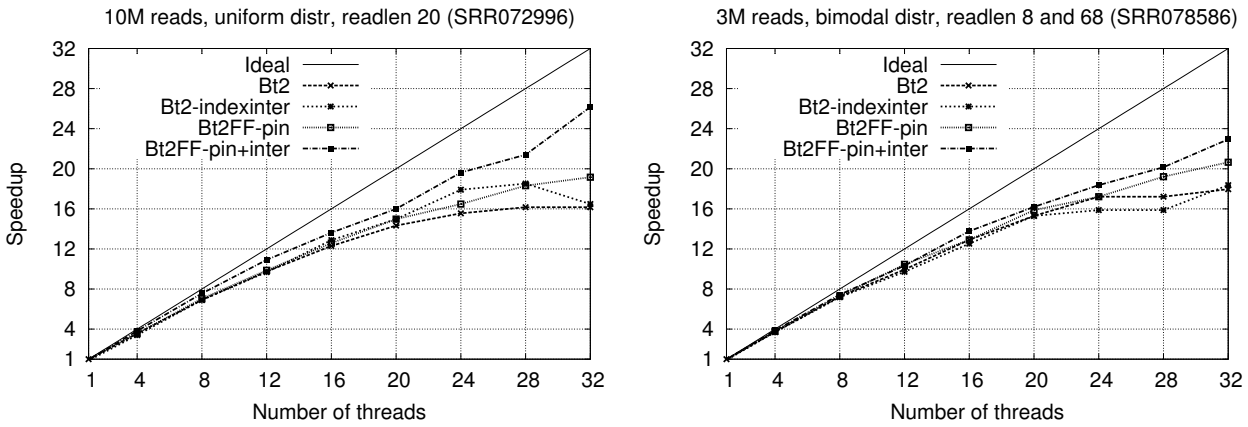

Figure 2: Speedup of Bowtie2 and Bowtie2-FF on SRR072996 (left) and SRR078586 (right) datasets (Illumina HiSeq 2000 paired end sequencing).

have as much as possible the same conditions as Bowtie 2-FF versions), Bowtie 2-FF with numa threads pinning (*Bowtie2-FF-NA*) and Bowtie2-FF with threads pinning and index interleaving (*Bowtie2-FF-NAI*). The first analysis was made by aligning a subset of a dataset of about 60 millions Reads of 20 bases length (SRR072996: Illumina HiSeq 2000 paired end sequencing). As shown in 2 (left), both Bowtie2-FF versions reach a higher speedup. Particularly we can observe that, using both pinning and interleaving, speedup can rise up from ≈ 16 to ≈ 26 using 32 threads. The second analysis was done on a dataset of about 3 millions Read of both 8 and 68 bases length (SRR078586 : Illumina HiSeq 2000 paired end sequencing). Figure 2 (right) demonstrates how a dataset with different reads' length can lead to very different performances. In fact, we can notice very different speedup curves with lower performances respect to the previous dataset. Also in this experiment, both Bowtie2-FF versions reach a better speedup respect to Bowtie2, rising up the speedup value from ≈ 18 to ≈ 23.

References

[ADK+12] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. An efficient unbounded lock-free queue for multi-core systems. In *Proc. of 18th Intl. Euro-Par 2012 Parallel Processing*, volume 7484 of *LNCS*, pages 662–673, Rhodes Island, Greece, aug 2012. Springer.

[ADKT13] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Fastflow: high-level and efficient streaming on multi-core. In Sabri Pllana and Fatos Xhafa, editors, *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing, chapter 13. Wiley, 2013.

[CHC+13] Yaoliang Chen, Ji Hong, Wanyun Cui, Jacques Zaneveld, Wei Wang, Richard Gibbs, Yanghua Xiao, and Rui Chen. Cgap-align: A high performance dna short read alignment tool. *PLoS ONE*, 8(4), 04 2013.