Efficient Smith-Waterman on multi-core with FastFlow

Marco Aldinucci
Computer Science Department
University of Torino, Italy
Email: aldinuc@di.unito.it

Massimiliano Meneghin, Massimo Torquati
Computer Science Department
University of Pisa, Italy
Email: {meneghin, torquati}@di.unipi.it

***Abstract*—Shared memory multiprocessors have returned to popularity thanks to rapid spreading of commodity multi-core architectures. However, little attention has been paid to supporting effective streaming applications on these architectures. In this paper we describe FastFlow, a low-level programming framework based on lock-free queues explicitly designed to support high-level languages for streaming applications. We compare FastFlow with state-of-the-art programming frameworks such as Cilk, OpenMP, and Intel TBB. We experimentally demonstrate that FastFlow is always more efficient than them on a given real world application: the speedup of FastFlow over other solutions may be substantial for fine grain tasks, for example +35% over OpenMP, +226% over Cilk, +96% over TBB for the alignment of protein P01111 against UniProt DB using the Smith-Waterman algorithm.**

I. Introduction

The recent trend of increasing core count in commodity processors has led to a renewed interest in the design of both methodologies and mechanisms for effective parallel programming of shared memory computer architectures.

Typically, low-level approaches provide programmers only with primitives for the management of *flows of control* (creation, destruction), their synchronisation and data sharing, which are usually accomplished via critical regions accessed through mutual exclusion (mutex). For example, the POSIX thread library can be used for this purpose. Programming complex parallel applications is this way is certainly hard; tuning them for performance is often even harder due to the non-trivial effects induced by memory fences (used to implement mutex) on data replicated in cores' caches, which constitute one of the key sources of performance degradation in communication intensive streaming parallel applications. Avoiding memory fences means not only avoiding locks but also avoiding any kind of atomic operation in memory (e.g. Compare-And-Swap, Fetch-and-Add). Although there exist several assessed fence-free solutions for *asynchronous symmetric* communication implemented via Single-Producer-Single-Consumer (SPSC) queues [1], these results cannot be easily extended to *asynchronous asymmetric* communications implemented via Multiple-Producer-Multiple-Consumer queues (MPMC) (necessary to support arbitrary streaming networks), without using advanced atomic primitives.

This work was partially funded by the project BioBITs founded by "Regione Piemonte - Italy", and by the WG Ercim CoreGrid topic "Advanced Programming Models".

A way to ease the programmer's task and improve program efficiency is to raise the level of abstraction of concurrency management primitives in order to enable code optimisation. For example, threads might be abstracted out in higher-level entities that can be pooled and scheduled in user space, possibly according to specific strategies to minimise cache flushing or maximise load balancing of cores. Synchronisation primitives can be also abstracted out and associated with semantically meaningful points of the code, such as function calls and returns, loops, etc. Intel *Threading Building Block* (TBB) [2], *OpenMP* [3], and *Cilk* [4] all provide this kind of abstraction, each in its own way. They significantly simplify the development of applications for multi-core architectures. However, the above-mentioned programming frameworks are not designed to effectively support streaming applications. The only pattern that fits this usage is TBB's *pipeline* construct, which can be used to describe only a linear chain of filters; none of them natively supports any kind of *task farming* on stream items, despite it being a quite common paradigm.

In [5], we introduced FastFlow a low-level methodology supporting lock-free (fence-free) Multiple-Producer-Multiple-Consumer (MPMC)[1] queues able to support low-overhead communication in multi-core architectures. In this paper, we briefly sketch the implementation of the farm streaming network using FastFlow *and* the most popular programming frameworks for multi-core architectures (i.e. TBB, OpenMP, Cilk), and we show that the FastFlow farm is generally faster than the other solutions for a real-world application: the Smith-Waterman local sequence alignment algorithm (SW). This latter comparison will be performed using the same "sequential" code in all implementations, i.e. the x86/SSE2 vectorised code derived from Farrar's high-performance implementation [6], [7].

II. Related Works

The stream programming paradigm offers a promising approach for programming multi-core systems. Stream languages are motivated by the application style used in image processing, networking, and other media processing domains. Several languages and libraries are available for programming stream applications, but many of them are oriented to coarse grain computations. Examples include *StreamIt*, *Brook*, NVidia *CUDA* and *OpenCL*. Some languages, such as TBB, provide explicit mechanisms for

[1] Also generalising Single-Producer-Multiple-Consumer (SPMC) and Multiple-Producer-Single-Consumer (MPSC) queues.

both streaming and other parallel paradigms, whereas some others, such as *OpenMP* and *Cilk* offer mainly mechanisms for Data Parallelism and Divide&Conquer computations.

StreamIt [8] is an explicitly parallel programming language based on the Synchronous Data Flow model. A program is represented as a set of filters, i.e. autonomous actors (containing Java-like code) that communicate through first-in first-out (FIFO) data channels. Filters can be assembled in *pipeline*, possibly with a *FeedbackLoop*, or according to a *SplitJoin* data-parallel schema.

Brook [9] provides extensions to the C language with single program multiple data (SPMD) operations on streams. User defined functions operating on stream elements are called *kernels* and can be executed in parallel. Brook kernels feature blocking behaviour: the execution of a kernel must complete before the next kernel can execute. A similar execution model is available on graphics processing units (GPUs) via the OpenCL framework [10].

Streaming applications are also targeted by TBB [2] through the *pipeline* construct. FastFlow – by intent – is methodologically similar to TBB, since it aims to provide a library of explicitly parallel constructs (a.k.a. skeletons [11]) that extend the base language (e.g. C, C++). However, TBB does not support any kind of non-linear streaming networks, which therefore has to be embedded in a pipeline. This has non-trivial programming and performance drawbacks since pipeline stages must bypass data that they are not interested in.

OpenMP [3] and *Cilk* [4] are two other very popular thread-based frameworks for multi-core architectures. *OpenMP* and *Cilk* mostly target Data Parallel and Divide&Conquer programming paradigms, respectively.

Cilk extends C/C++ to provide programmers with mechanisms to spawn independent flows of controls *(cilk-threads)* according to the *fork/join* model. The scheduling of the computation of flows is managed by an efficient work-stealing scheduler. Control flows can synchronize in shared memory under a quite relaxed memory consistency (DAG consistency) [12].

Intel *Threading Building Blocks* (TBB) is a C++ template library consisting of *containers* and *algorithms* that abstract the usage of native threading packages (e.g. POSIX threads). In particular, the library abstracts access to the multiple processors by allowing the operations to be treated as *tasks*, which are allocated to individual cores dynamically by the library's run-time engine. The tasks and synchronisations among them are extracted from language constructs such as `parallel_for`, `parallel_reduce`, `parallel_scan`, and `pipeline`. Tasks might also cooperate via shared memory through concurrent containers (e.g. `concurrent_queue`), several flavours of mutex (lock, and atomic operations (e.g. Compare_And_Swap) [2]. This approach groups TBB in a family of solutions for parallel programming aimed at enabling programmers to explicitly define parallel behaviour via parametric exploitation patterns (*skeletons*, actually) that have been widely explored over the last two decades for parallel computing [11], [13], [14], [15].

At the level of communication and synchronisation mechanisms, Giacomini et al. [16] highlight the fact that traditional locking queues feature a high overhead on today's multi-cores. Revisiting Lamport's work [1], which proves the correctness of wait-free mechanisms for concurrent SPSC queues on systems with memory sequential consistency commitment, they proposed a set of wait-free and cache-optimised protocols. They prove the performance benefit of those mechanisms on pipeline applications on today's multi-core architectures. A direct design of lock-free MPMC queues is more complex and requires at least an atomic operation [17]. As we shall see in Sec. IV, FastFlow follows an alternative approach: a MPMC queue is designed by coupling several lock-free SPSC under the control of a sequencer thread. Therefore FastFlow extends Giacomini's work [16] from simple pipelines to *any streaming network*.

III. FastFlow

FastFlow is a template library that offers a set of low-level mechanisms to support low-latency and high-bandwidth data flows in a network of threads running on a cache-coherent multi-core. On these architectures, the key performance issues concern memory fences, which are required to keep the various caches coherent. FastFlow provides the programmer with two basic mechanisms: efficient communication channels and a memory allocator. Communication channels, as typical is in streaming applications, are unidirectional and asynchronous. They are implemented via fence-free MPMC queues. The memory allocator is built on top of these queues, thus taking advantage of their efficiency.

Traditionally, MPMC queues are built as passive entities: threads concurrently synchronise to access data; these synchronisations are usually supported by one or more atomic operations (e.g. Compare-And-Swap) that behave as memory fences. FastFlow design follows a different approach: in order to avoid any memory fence, MPMC queues are build by assembling fence-free SPSC queues by means of an arbiter thread, which gathers or multicast/unicast data items from input queues to output queues (see also Fig. 1). We call these entities *Emitter* (E) or *Collector* (C) according to their role; they read an item from one or more fence-free SPSC queues and write onto one or more lock-free SPSC queues. This requires a memory copy but no atomic operations (this is a trivial corollary of lock-free SPSC correctness [16]).

The big performance advantage of this solution stems from the higher speed of the copy with respect to the memory fence. This advantage is further increased by avoiding cache invalidation triggered by fences, which also depends on the size and the memory layout of copied data.

The memory allocator, which is not discussed in this paper, can be replaced either with an OS standard allocator (paying a performance penalty) or a third-party allocator (e.g. TBB scalable allocator [2]). More details on FastFlow design and memory allocator basics can be found in [18].

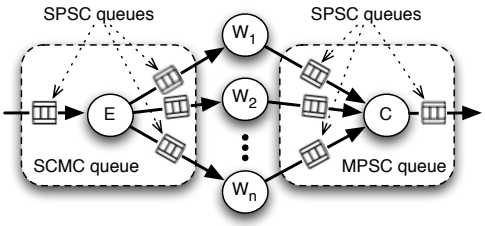

Figure 1. FastFlow farm.

IV. The Stream Parallel Farm Paradigm

The parallelism exploitation patterns are usually categorised into three main classes: Task, Data, and Stream Parallelism. These classes are often encoded in high-level paradigms (a.k.a. *skeletons*) to be encoded by programming language constructs [19]. While many of the programming frameworks mentioned in Sec. II offer Data and Task Parallel skeletons, only few offer Stream Parallel skeletons (such as TBB's pipeline). None of them offers the farm skeleton, which exploits functional replication and abstracts out the parallel filtering of successive independent items of the stream under the control of a scheduler, as a first-class concept.

The farm skeleton can be implemented using a master-worker schema. In the streaming settings, the master can be conveniently split into two functionally distinct parts: the *Emitter* (E) that schedules the stream items to a pool of *Workers* (W_i) and the *Collector* (C) that gathers the results[14]. We use this general schema to implement the farm skeleton using FastFlow, OpenMP, TBB, and Cilk. In all these programming models but FastFlow, the Collector is not run as separate thread but its code is spread out onto the Workers, and is run in mutual exclusion. The performance of these implementations will be compared in Sec. V. In the following we briefly discuss the peculiarity of each implementation. We refer back to [18], [20] for a detailed description and code listings.

As shown in Fig. 1), the FastFlow farm is particularly simple since it can be implemented by surrounding a pool of Workers with an Emitter and a Collector, all realised using POSIX threads that communicate via FastFlow fence-free SPMC and MPSC queues.

The OpenMP farm is driven by the Emitter, which is started in a single copy using a `single nowait` directive, and spawns a parallel `task` for each stream item in `untied` fashion[2]. Each task, once started, runs asynchronously with respect to the Emitter and eventually writes the result into the output stream. Since many tasks can write concurrently, the output gathering is defined within a `critical` region (Collector).

The Cilk farm is defined as a function that can behave as Emitter, Worker, and Collector. The first invocation of the farm function behaves as Emitter, which iteratively picks a stream item from the input stream and spawns a `cilk_thread` using the item as a parameter of the invocation. All farm function invocations after the first behave as a Worker followed by a Collector. As in OpenMP, the collection can happen concurrently and it has been realised by exploiting a Cilk `inlet` function, which atomically handles the result of the function.

Intel TBB provides programmers with the `parallel_for` construct that can be used to realise the farm skeleton. The `parallel_for` is able to dispatch a bunch of data items to a set of workers, but these tasks must be dispatched together in a Data Parallel fashion. This restriction may be avoided exploiting a `pipeline` of two stages: Emitter and Worker. The Emitter receives the stream items from the input stream and packs them into an array. The array is passed to the second stage that applies a `parallel_for` function to each element (worker function). Each worker function eventually writes the result of the computation into a `concurrent_queue`.

V. Smith-Waterman: A Comparative Analysis

In bioinformatics, sequence database searches are used to find the similarity between a query sequence and subject sequences in a database in order to determine similar regions between two nucleotide or protein sequences, which are encoded as a string of characters. The sequence similarities can be determined by computing their optimal local alignments using the Smith-Waterman (SW) algorithm [21], which is a dynamic programming algorithm that is guaranteed to find the optimal local alignment with respect to the scoring system being used. Instead of looking at the total sequence, it compares segments of all possible lengths and optimises the similarity measure.

This approach is expensive in terms of computing time and memory space due to the rapid growth of biological sequence databases (UniProtKB/Swiss-Prot database Release 57.5 of 07-Jul-09 contains 471472 sequence, comprising 167326533 amino acids) [22]. Therefore, recent works in this area focus on implementing the SW algorithm on different architectures, like GPUs [10] and Cell/BE [7] and on Intel multi-cores exploiting vector instruction sets [6]. Among these implementations, we selected the SWPS3 [7], an optimised extension of Farrar's work [6] on the Strip Waterman-Algorithm for the Cell/BE and on x86/64 CPUs.The original SWPS3 is designed as a master-worker computation: the master process reads the query sequence, initialises the data structures for vector computations, and then forks all the worker processes; each worker has its own copy of the data. All the sequences in the reference database are read and sent to a worker process over POSIX pipes. Each worker computes the alignment score according to the SW algorithm, and sends the resulting score back over a pipe.

The computational time of the algorithm is sensitive with respect to the query length used for the matching, the scoring matrix (in our case BLOWSUM50) and the gap penalty. Due to the very different lengths of the subject sequences in the database, there is a high variance in the single query matching service time. For the computation of

[2]Run of independent tasks is available in OpenMP starting from version 3.0.

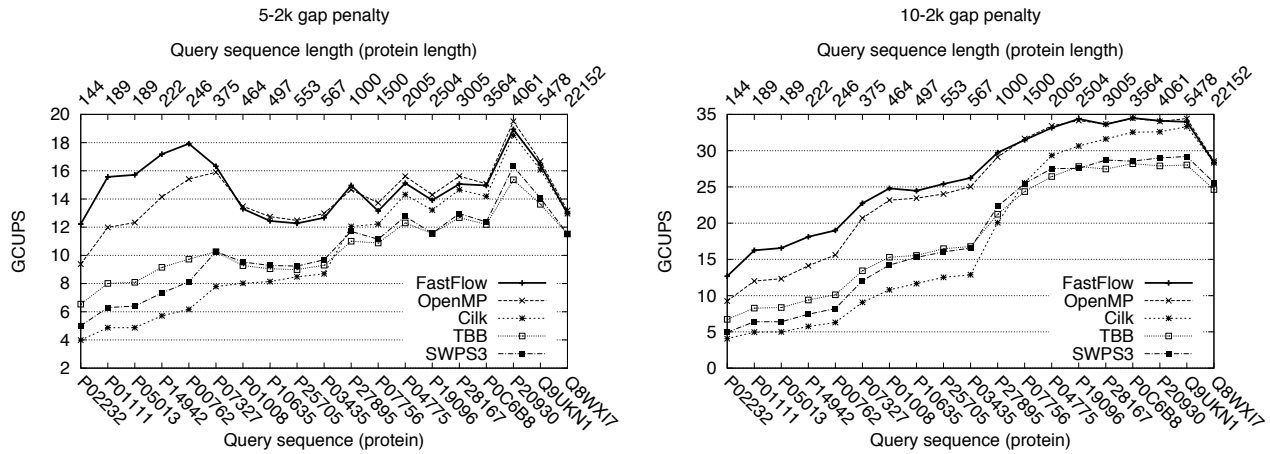

Figure 2. Smith-Waterman sequence alignment algorithm: comparison between FastFlow, OpenMP, TBB, and Cilk implementations. The SWPS3, which is based on POSIX primitives, is the original version from which the other has been derived. All the implementations share exactly the same sequential (x86/SSE2 vectorised) code.

protein P04775, we measured, for all the single matches, an average service time of 375μS with 0.690μS and 21837.1μS the minimum and maximum values. To give an idea of the different lengths, we report some statistics of the UniProtKB/Swiss-Prot where the shortest sequence is formed by 2 amino acids and the longest by 35213, while the average sequence length is 352 amino acids. Furthermore, the higher the gap open and gap extension penalties, the fewer iterations are needed for the calculation of the single cell of the similarity score matrix. In our tests we used the scoring matrix BLOSUM50 with two gap penalty range: 10-2k and 5-2k.

We rewrote the original SWPS3 code in OpenMP, Cilk, TBB and FastFlow following the schemata presented before, while keeping untouched the sequential kernel code to achieve a fair comparison.

The Emitter entity reads the sequence database and produces a stream of pairs: $\langle$*query sequence, subject sequence*$\rangle$. The query sequence remains the same for all the subject database sequences. The Worker entity computes the striped Smith-Waterman algorithm on the input pairs using the SSE2 instruction set. The Collector gets the resulting score and produces the output string containing score and sequence name.

For performance reasons, it is important to provide a copy of the data structures needed for the SSE2 computation to each worker thread. Notwithstanding that this data is read-only, the third-party SSE somehow seems to trigger the cache invalidation while accessing the data, which seriously affects the performance. To overcome this problem we employ a trick: we use the POSIX *Thread-Specific-Storage* (TSS) support, setting in the TSS the required data for each thread of the application. This is not a critical aspect for the Cilk and TBB algorithm implementation, which do not natively support any kind of TSS, because the TSS data is read-only and each thread has its own copy. In OpenMP this is not a particular problem because we have the possibility to identify a specific worker thread with the library call `omp_get_thread_num()`. The same possibility to identify a thread is offered by the FastFlow framework as each parallel entity is mapped to one and only one thread.

To remove the dependency on the query sequences and the databases used for the tests, Cell-Updates-Per-Second (CUPS) is a commonly used performance measure in bioinformatics. A CUPS is the time for the computation of one cell in the matrix of the similarity score, including all memory operations. Given a query sequence of length Q and a database of size D, the GCUPS (billion Cell Updates Per Second) value is calculated by: $GCUPS = |Q| \cdot |D|/10^9 \cdot T$, where T is the total execution time in seconds. The performance of the different SW algorithm implementations has been benchmarked and analysed by searching for 19 sequences of length from 144 (P02232 sequence) to 22,142 (Q8WXI7 sequence) against the Swiss-Prot release 57.5 database. All experiments are executed on a shared memory Intel platform with 2 quad-core Xeon E5420 Harpertown @2.5GHz with 6MB L2 cache and 8 GBytes of main memory. We have used, under the Linux 2.6.18 kernel, version 4.4.0 of the GCC compiler (which features the OpenMP v3.0 support), version 5.4.6 of Cilk and version 2.1 of TBB library.

Figure 2 reports the performance comparison between FastFlow, OpenMP, Cilk, TBB and SWPS3 version of SW algorithm for x86/SSE2.

As can be seen from the figures, the FastFlow implementation outperforms the other implementations for short query sequences. The smaller the query sequences are, the bigger the performance gain is. This is mainly due to the lower overhead of FastFlow communication channels with respect to the other implementations; short sequences require a smaller service time.

Cilk gives a lower performance value with respect to the original SWPS3 version with small sequences while performing very well with longer ones. OpenMP offers the best performance after FastFlow.

VI. Conclusions

In this work we have introduced FastFlow, a low-level template library based on lock-free communication channels explicitly designed to support low-overhead high-throughput streaming applications on commodity cache-coherent multi-core architectures. We have shown that FastFlow can be directly used to implement complex streaming applications exhibiting cutting-edge performance on a commodity multi-core.

Also, we have demonstrated that FastFlow makes possible the efficient parallelisation of third-party legacy code, such as the x86/SSE vectorised Smith-Waterman code. In the short term, we envisage FastFlow as the middleware tier of a "skeletal" high-level programming framework that will discipline the usage of efficient network patterns, possibly extending an existing programming framework (e.g. TBB) with the stream-specific constructs. To this end, we studied how a farm construct can be realised using several state-of-the-art programming frameworks for multi-core, and we have experimentally demonstrated that the FastFlow farm is faster than other farm implementations, at least for the Smith-Waterman application.

As expected, the performance advantage of FastFlow over the other frameworks is significant for fine-grained computations. This makes FastFlow suitable for implementation of a fast macro data-flow executor (actually wrapping around the order preserving farm), thus achieving the automatic parallelisation of many classes of algorithms. FastFlow is open source software under GPL available at http://mc-fastflow.sourceforge.net/.

References

[1] L. Lamport, "Specifying concurrent program modules," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 2, pp. 190–222, 1983.

[2] *Threading Building Blocks*, Intel Corp., 2009, http://www.threadingbuildingblocks.org/.

[3] I. Park, M. J. Voss, S. W. Kim, and R. Eigenmann, "Parallel programming environment for openmp," *Scientific Programming*, vol. 9, pp. 143–161, 2001.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55–69, Aug. 1996.

[5] M. Aldinucci, M. Danelutto, M. Meneghin, P. Kilpatrick, and M. Torquati, "Efficient streaming applications on multi-core with FastFlow: the biosequence alignment test-bed," in *Proc. of Intl. Parallel Computing (PARCO)*, Lyon, France, Sep. 2009, to appear.

[6] M. Farrar, "Striped Smith-Waterman speeds database searches six times over other simd implementations," *Bioinformatics*, vol. 23, no. 2, pp. 156–161, 2007.

[7] A. Szalkowski, C. Ledergerber, P. Kraehenbuehl, and C. Dessimoz, "SWPS3 - fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2," *BMC Research Notes*, vol. 1, no. 1, 2008.

[8] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A language for streaming applications," in *Proc. of the 11th Intl. Conference on Compiler Construction (CC)*. London, UK: Springer-Verlag, 2002, pp. 179–196.

[9] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for gpus: stream computing on graphics hardware," in *ACM SIGGRAPH '04 Papers*. New York, NY, USA: ACM Press, 2004, pp. 777–786.

[10] *OpenCL*, Khronos Compute Working Group, Nov. 2009. [Online]. Available: http://www.khronos.org/opencl/

[11] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computations*, ser. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[12] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall, "Dag-consistent distributed shared memory," in *Proc. of the 10th Intl. Parallel Processing Symposium*, Honolulu, Hawaii, USA, Apr. 1996, pp. 132–141.

[13] M. Danelutto, R. D. Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi, "A methodology for the development and the support of massively parallel programs," *Future Generation Compututer Systems*, vol. 8, no. 1-3, pp. 205–220, 1992.

[14] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, S. Magini, P. Pesciullesi, L. Potiti, R. Ravazzolo, M. Torquati, M. Vanneschi, and C. Zoccolo, "The implementation of ASSIST, an environment for parallel and distributed programming," in *Proc. of 9th Intl Euro-Par 2003 Parallel Processing*, ser. LNCS, vol. 2790. Klagenfurt, Austria: Springer, Aug. 2003, pp. 712–721.

[15] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Usenix OSDI '04*, Dec. 2004, pp. 137–150.

[16] J. Giacomoni, T. Moseley, and M. Vachharajani, "Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue," in *Proc. of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP)*. New York, NY, USA: ACM, 2008, pp. 43–52.

[17] S. Prakash, Y. H. Lee, and T. Johnson, "A nonblocking algorithm for shared queues using compare-and-swap," *IEEE Trans. Comput.*, vol. 43, no. 5, pp. 548–559, 1994.

[18] M. Aldinucci, M. Torquati, and M. Meneghin, "FastFlow: Efficient parallel streaming applications on multi-core," Università di Pisa, Dipartimento di Informatica, Italy, Tech. Rep. TR-09-12, Sep. 2009.

[19] M. Cole, *Skeletal Parallelism home page*, 2009. [Online]. Available: http://homepages.inf.ed.ac.uk/mic/Skeletons/

[20] M. Aldinucci and M. Torquati, *FastFlow website*, 2009. [Online]. Available: http://mc-fastflow.sourceforge.net/

[21] M. S. Waterman and T. F. Smith, "Identification of common molecular subsequences." *J. Mol. Biol.*, vol. 147, pp. 195–197, 1981.

[22] *UniProt web site*, UniProt Consortium, Jul. 2009 (last accessed). [Online]. Available: http://www.uniprot.org/