Università di Pisa

Dipartimento di Informatica

Technical Report: TR-09-12

FastFlow: Efficient Parallel Streaming Applications on Multi-core

Marco Aldinucci Massimo Torquati
Massimiliano Meneghin

September 2, 2009

Address: Largo B. Pontecorvo 3, 56127 Pisa, Italy. Tel: +39 050 2212700 Fax: +39 050 2212726

FastFlow: Efficient Parallel Streaming Applications on Multi-core

Marco Aldinucci[*] Massimo Torquati
Massimiliano Meneghin

September 2, 2009

Abstract

Shared memory multiprocessors come back to popularity thanks to rapid spreading of commodity multi-core architectures. As ever, shared memory programs are fairly easy to write and quite hard to optimise; providing multi-core programmers with optimising tools and programming frameworks is a nowadays challenge. Few efforts have been done to support effective streaming applications on these architectures. In this paper we introduce 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 all of them in a set of micro-benchmarks and on a real world application; the speedup edge of FastFlow over other solutions might be bold for fine grain tasks, as an example +35% on OpenMP, +226% on Cilk, +96% on TBB for the alignment of protein P01111 against UniProt DB using Smith-Waterman algorithm.

1 Introduction

The recent trend to increase core count in commodity processors has led to a renewed interest in the design of both methodologies and mechanisms for the effective parallel programming of shared memory computer architectures. Those methodologies are largely based on traditional approaches of parallel programming.

Typically, low-level approaches provides the programmers only with primitives for flows-of-control management (creation, destruction), their synchronisation and data sharing, which are usually accomplished in critical regions accessed in mutual exclusion (mutex). As an example, POSIX thread library can be used to this purpose. Programming parallel complex applications is this

[*]Computer Science Department, University of Torino, Italy. Email: adinuc@di.unito.it

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 core's caches.

Indeed, memory fences are one of the key sources of performance degradation in communication intensive (e.g. 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). While there exists several assessed fence-free solutions for *asynchronous symmetric* communications[1], these results cannot be easily extended to *asynchronous asymmetric* communications[2], which are necessary to support arbitrary streaming networks.

A first way to ease programmer's task and improve program efficiency consist in to raise the level of abstraction of concurrency management primitives. As an 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 to semantically meaningful points of the code, such as function calls and returns, loops, etc. Intel *Threading Building Block* (TBB) [25], *OpenMP* [33], and *Cilk* [16] all provide this kind of abstraction (even if each of them in its own way).

This kind of abstraction significantly simplify the hand-coding of applications but it is still too low-level to effectively automatise the optimisation of the parallel code: here the major weakness lies in the lack of information concerning the *intent* of the code (idiom recognition [35]); inter-procedural/component optimisation further exacerbates the problem. The generative approach focuses on synthesising implementations from higher-level specifications rather than transforming them. From this approach, programmers' intent is captured by the specification. In addition, technologies for code generation are well developed (staging, partial evaluation, automatic programming, generative programming). Both TBB and OpenMP follow this approach. The programmer is required to explicitly define parallel behaviour by using proper constructs [5], which clearly delimit the interactions among flows-of-control, the read-only data, the associativity of accumulation operations, the concurrent access to shared data structures.

However, the above-mentioned programming framework for multi-core architectures are not specifically designed to 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 those natively support any kind of *task farming* on stream items (despite it is a quite common pattern).

The objective of this paper is threefold:

- To introduce FastFlow, i.e. low-level methodology supporting lock-free (fence-free) Multiple-Producer-Multiple-Consumer (MPMC) queues able to support low-overhead high-bandwidth multi-party communications in

[1]Single-Producer-Single-Consumer (SPSC) queues [29].
[2]Multiple-Producer-Multiple-Consumer queues (MPMC).

multi-core architectures, i.e. any *streaming network*, including cyclic graphs of threads.

- To study the implementation of the farm streaming network using FastFlow *and* the most popular programming frameworks for multi-core architectures (i.e. TBB, OpenMP, Cilk).
- To show that FastFlow farm is generally faster than the other solutions on both a synthetic micro-benchmark and a real-world application, i.e. 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 [22]. We will also show that the FastFlow implementation is faster than the state-of-the-art, hand-tuned parallel version of the Farrar's code (SWPS3 [23]).

In the longer term, we envision FastFlow as the part of a run-time support of a set of high-level streaming skeletons for multi- and many-core, either in insulation or as extension of the TBB programming framework.

2 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. Example are *StreamIt* [41], *Brook* [15], and *CUDA* [27]. Some other languages, as TBB, provide explicit mechanisms for both streaming and other parallel paradigm, while some others, as *OpenMP* [33] and *Cilk* mainly offers mechanisms for Data Parallelism and Divide&Conquer computations. These mechanisms can be also exploited to implement streaming applications, as we shall show in Sec. 3, but this requires a greater programming effort with respect to the other cited languages.

StreamIt is an explicitly parallel programming language based on the Synchronous Data Flow (SDF) programming model. A StreamIt program is represented as a set of autonomous actors that communicate through first-in first-out (FIFO) data channels. StreamIt contains syntactic constructs for defining programs structured as task graphs, where each tasks contain Java-like sequential code. The interconnection types provided by are: *Pipeline* for straight task combinations, *SplitJoin* for nesting data parallelism and *FeedbackLoop* for connections from consumers back to producers. The communications are implemented either as shared circular buffers or message passing for small amounts of control information.

Brook [15] provides extensions to C language with single program multiple data (SPMD) operations that work on streams. User defined functions operating on stream elements are called *kernels* and can be executed in parallel. Brook kernels feature a blocking behaviour: the execution of a kernel must complete before the next kernel can execute. This is the same execution model that is available on graphics processing units (GPUs), which are indeed the main target of this programming framework. In the same class can be enumerated CUDA [27], which is an infrastructure from NVIDIA. In addition, CUDA programmers are required to use low-level mechanisms to explicitly manage the various level of the memory hierarchy.

Streaming applications are also targeted by TBB [25] through the *pipeline* construct. FastFlow – as intent – is methodologically similar to TBB, since it aims to provide a library of explicitly parallel constructs (a.k.a. parallel programming paradigms or skeletons) that extends the base language (e.g. C, C++, Java). However, TBB does not support any kind of non-linear streaming network, which therefore has to be embedded in a pipeline. This has a non-trivial programming and performance drawbacks since pipeline stages should bypass data that are not interested with.

OpenMP [33] and *Cilk* [14] are other two very popular thread-based frameworks for multi-core architectures (a in deep language descriptions is reported in 3.2 and 3.3 sections). *OpenMP* and *Cilk* mostly target Data Parallel and Divide&Conquer programming paradigms, respectively. OpenMP for example has only recently extended (3.0 version) with a *task* construct to manage the execution of a set of independent tasks. The fact that the two languages do not provide first class mechanisms for streaming applications is reflected in their characteristic of well performing only with coarse- and medium-grained computations, as we see in Sec. 4.

At the level of communication and synchronisation mechanisms, Giacomini et al. [24] highlight that traditional locking queues feature a high overhead on today multi-core. Revisiting Lamport work [29], which proves the correctness of wait-free mechanisms for concurrent Single-Producer-Single-Consumer (SPSC) queues on system with memory sequential consistency commitment, they proposed a set of wait-free and cache-optimised protocols. They also prove the performance benefit of those mechanisms on pipeline applications on top of today multi-core architectures. Wait-free protocols are a subclass of lock-free protocols exhibiting even stronger properties: roughly speaking lock-free protocols are based on retries while wait-free protocols guarantee termination in a finite number of steps.

Along with SPSC queues, also MPMC queues are required to provide a complete support for streaming networks. Those kind of data structures represent a more general problem than SPSC one, and various works has been presented in literature [28, 31, 36, 42]. Thanks to the structure of streaming applications, we avoid the problem of managing directly MPMC queue: we exploit multiple SPSC queues to implement MPSC, SCMP and MPMC ones.

Therefore exploiting a wait-free SPSC also for implementing more complex shared queues, FastFlow widely extend the work of Giacomini et al., from simple pipelines to *any streaming networks*. We show effective benefits of our approach with respect to the other languages TBB, OpenMP and Cilk.

3 Stream Parallel Paradigm: the Farm Case

Traditionally types of parallelisms are categorised in three main classes:

- *Task Parallelism.* Parallelism is explicit in the algorithm and consists of running the same or different code on different executors (cores, processors, etc.). Different flows-of-control (threads, processes, etc.) communicate with one another as they work. Communication takes place usually to pass data from one thread to the next as part of a graph.
- *Data Parallelism* is a method for parallelising a single task by processing independent data elements of this task in parallel. The flexibility of the technique relies upon stateless processing routines implying that the data elements must be fully independent. Data Parallelism also support *Loop-level Parallelism* where successive iterations of a loop working on independent or read-only data are parallelised in different flows-of-control (according to the model *co-begin/co-end*) and concurrently executed.
- *Stream Parallelism* is method for parallelising the execution (a.k.a. filtering) of a stream of tasks by segmenting the task into a series of *sequential*[3] or *parallel* stages. This method can be also applied when there exists a *total* or *partial* order, respectively, in a computation preventing the use of data or task parallelism. This might also come from the successive availability of input data along time (e.g. data flowing from a device). By processing data elements in order, local state may be either maintained in each stage or distributed (replicated, scattered, etc.) along streams. Parallelism is achieved by running each stage simultaneously on *subsequent* or *independent* data elements.

These basic form of parallelism are often encoded in high-level paradigms (a.k.a. *skeletons*) to be encoded in programming language construct. Many skeletons appeared in literature in the last two decades covering many different usage schema of the three classes of parallelism, on top of both the message passing [18, 20, 38, 2, 9, 12, 34, 7, 21, 3] and shared memory [1, 25] programming models.

As an example, the farm skeleton models the functional replication and consists of running multiple independent tasks in parallel or filtering many successive tasks of a stream in parallel. It typically consists of two main entities: a master (or scheduler) and multiple workers (farm is also known as *Master-Workers*). The scheduler is responsible for distributing the input task (in case by decomposing the input task into small tasks) toward the worker pool, as well as for gathering the partial results in order to produce the final result of the computation. The worker entity get the input task, process the task, and send the result back to the scheduler entity. Usually, in order to have pipeline parallelism between the scheduling phase and the gathering phase, the master entity is split in two main entities: respectively the *Emitter* and the *Collector*.

[3]In the case of total sequential stages, the method is also known as *Pipeline Parallelism*.

Farm can be declined in many variants, as for example with stateless workers, stateful workers (local or shared state, read-only or read/write), etc.

The farm skeleton is quite useful since it can exploited in many streaming applications. In particular, it can be used in any *pipeline* to boost the service time of slow stages, then to boost the whole pipeline [4].

As mentioned in previous section, several programming framework for multi-core offer Data Parallel and Task Parallel skeletons, only few of them offer Stream Parallel skeletons (such as TBB's pipeline), none of them offers the farm. In the following we study the implementation of the farm for multi-core architectures. In Sec. 3.1 we introduce a very efficient implementation of the farm construct in FastFlow, and we propose our implementation using other well-known frameworks such as OpenMP, Cilk, and TBB. The performance are compared in Sec. 4.

3.1 FastFlow Farm

FastFlow aims to provide a set of low-level mechanisms able to support low-latency and high-bandwidth data flows in a network of threads running on a SCM. These flows, as typical in streaming applications, are supposed to be mostly unidirectional and asynchronous. On these architectures, the key issues regard memory fences, which are required to keep the various caches coherent.

FastFlow currently provides the programmer with two basic mechanisms: MPMC queues and a memory allocator. The memory allocator, is actually build on top of MPMC queues and can be substituted either with OS standard allocator (paying a performance penalty) or a third-party allocator (e.g. Intel TBB scalable allocator [25]).

The key intuition underneath FastFlow is to provide the programmer with lock-free MP queues and MC queues (that can be used in pipeline to build MPMC queues) to support fast streaming networks. Traditionally, MPMC queues are build as passive entities: threads concurrently synchronise (according to some protocol) 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, the synchronisations among queue readers or writers are arbitrated by an active entity (e.g. a thread), as shown in Fig. 1. We call these entities *Emitter* (E) or *Collector* (C) according to their role; they actually read an item from one or more lock-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 [24]). Notice that, FastFlow networks do not suffer from the ABA problem [32] since MPMC queues are build explicitly linearising correct SPSC queues using Emitters and Collectors.

The performance advantage of this solution descend from the higher speed of the copy with respect to the memory fence, that advantage is further increased by avoiding cache invalidation triggered by fences. This also depends on the size and the memory layout of copied data. The former point is addressed using data pointers instead of data, and enforcing that the data is not concurrently

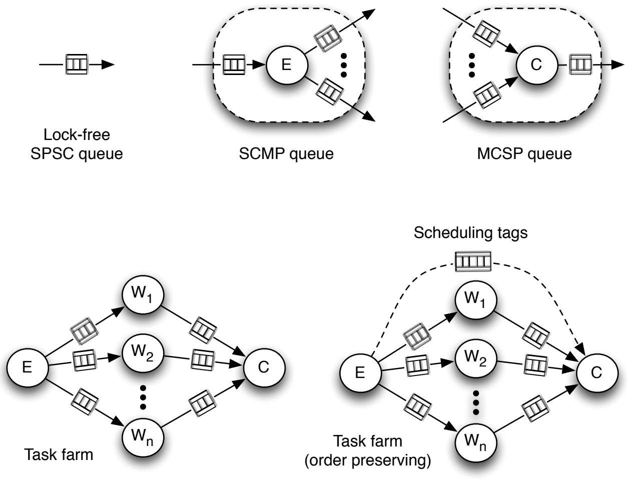


Figure 1: FastFlow concepts: Lock-free SPSC queue, SPMC queue, MCSP queue, task farm, and order preserving task farm.

written: in many cases this can be derived by the semantics of the skeleton that has been implemented using MPMC queues (as an example this is guaranteed in a stateless farm and many other cases).

When using dynamically allocated memory, the memory allocator plays an important role in term of performance. Dynamic memory allocators (`malloc`/`free`) rely on mutual exclusion locks for protecting the consistency of their shared data structures under multi-threading. Therefore, the use of memory allocator may subtly reintroduce the locks in the lock-free application. For this reason, we decided to use our own custom memory allocator, which has specifically optimised for SPMC pattern. The basic assumption is that, in streaming application, typically, one thread allocate memory and one or many other threads free memory. This assumption permits to develop a multi-threaded memory allocator that use SPSC channels between the allocator thread and the generic thread that performs the free, avoiding the use of costly lock based protocols for maintaining the memory consistency of the internal structures. Notice however, the FastFlow allocator is not a general purpose allocator and it currently exhibits several limitations, such as a sub-optimal space usage. The further development of FastFlow allocator is among future works.

3.1.1 Pseudo-code

The structure of the farm paradigm in FastFlow is sketched in Fig. 2. The `ff_TaskFarm` is a C++ class interface that implements the parallel farm construct composed by an Emitter and an optional Collector (also see Fig. 1). The number of workers should be fixed at the farm object creation time.

```
class Emitter: public ff::ff_node {
public:
   void * svc(void *) {
     while(∃ newtask){
       newtask = create_task();
       return newtask;
     }
     return NULL; //EOS
   }
};

class Collector: public ff::ff_node {
public:
   void * svc(void * task) {
     collect_task(task);
     return task;
   }
};

class Worker: public ff::ff_node {
public:
   void * svc(void * task) {
     compute_task(task);
     return task;
   }
};

int main (int argc, char *argv[]) {
  Emitter  E;
  Worker   W;
  Collector C;
  ff::ff_TaskFarm farm(nworkers);
  farm.add_emitter(E);
  farm.add_worker(W);
  farm.add_collector(C);

  farm.run();
}
```

Figure 2: Farm structure in FastFlow.

The usage of the interface is straightforward: Firstly, Emitter, Worker and Collector classes are defined deriving the `ff_node` class. Secondly, for each of the three classes the abstract method `svc` (i.e. *service* method) should be implemented. The method contains the sequential code of the worker entity. Finally, the three objects are registered with the object of the class `ff_TaskFarm`.

In Fig. 2, the Emitter produces a new task each time the `svc` method is called. The run-time support is in charge to schedule the tasks to one of the available workers. The scheduling can be performed according to several policies, from simple round-robin to an user-defined stateful policy [2]. Observe that ordering of tasks flowing through the farm, in general, is not preserved. However, task ordering can be ensured either using the same deterministic policy for task scheduling and collection, or by dynamically tracking scheduling choices and performing the collection accordingly. This latter solution, schematised in Fig. 1 (order preserving task farm), is actually derived from tagged-token macro data-flow architecture [10, 9, 38].

3.2 OpenMP Farm

The OpenMP (*Open Multi-Processing*) [33, 11] is a set of standardised API developed to extend C, C++ and Fortran sequential languages in order to support shared memory multiprocessing programming. OpenMP is based on compiler directives, library routines, and environment variables that can be used to transform a sequential program into a thread-based parallel program.

A key concept is that a well written OpenMP program should result in a completely correct sequential program when it is compiled without any OpenMP supports. Therefore, all the OpenMP directives are implemented as *pragmas* into the target languages: they are exploited by those compilers featuring OpenMP support in order to produce a thread-based parallel programs and discarded by the others. OpenMP (standard) features three different classes of mechanisms to express and manage various aspects of parallelism, respectively to:

- identify and distribute parallel computations among different resources;
- manage scope and ownership of the program data;
- introduce synchronisation to avoid race conditions.

Program flows-of-control (e.g. threads, processes and any similar entity) are not directly managed by language directives. Programmers highlight program sections that should be parallelised. The OpenMP support automatically defines threads and synchronisations in order to produce an equivalent parallel program implementation. In the scale introduced in Sec. 1, it exhibits a medium/high abstraction level.

The main OpenMP mechanism is the `parallel` pragma, which is used to bound pieces of sequential code which are going to be computed in parallel by a team of threads. Inside a `parallel` section, in order to specify a particular parallel paradigm that the team thread have to implement, specific directives can be inserted. The `parallel for` directive expresses data parallelism while the `section` functional parallelism. Because of the limitation of the `section` mechanism, which provides only static functional partition, from the version 3.0 OpenMP provides a new construct called `task` to model independent units of work which are automatically scheduled without programmers intervention. As suggested in [11] `task` construct can be used to build a web server, and since web servers exhibits a typical streaming behaviour, we will use the `task` construct to build our farm schema.

```
int main (int argc, char *argv[])
{
  #pragma omp parallel private(newtask)
  {
    /* EMITTER */
    #pragma omp single nowait
    {
      while(∃ newtask){
        newtask = create_task();
        /* WORKER */
        #pragma omp task untied
        {
          compute_task(newtask);
          /* COLLECTOR */
          #pragma omp critic
          {
            collect_task(newtask);
          }
        }
      }
    }
  }
```

Figure 3: Farm structure in OpenMP.

3.2.1 Pseudo-code

OpenMP do not natively include a farm skeleton, which should be realised using lower-level features, such as the `task` construct. Our OpenMP farm schema is shown in Fig. 3.2.1. The schema is quite simple; a `single` section is exploited to highlight the Emitter behaviour. The new independent tasks, defined by the Emitter, are marked with the `task` directive in order to leave their computation scheduling to the OpenMP run time support.

The Collector is implemented in a different way. Instead of implementing it with a `single` section (as for the Emitter), and therefore introducing an explicit locking mechanism for synchronisation between workers and Collector, we realise Collector functionality by means of workers cooperative behaviour: they simply output tasks using an OpenMP `critic` section. This mechanism enable us to output tasks from the stream without introducing any global synchronisation (barrier).

3.3 Cilk Farm

Cilk is a language for multi-threaded parallel programming that extends the C language. Cilk provides 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 a efficient work-stealing scheduler [14].

Cilk controls flows are supported by a share memory featured by a DAG consistency [13], which is a quite relaxed consistency model. Cilk-threads synchronise according to the DAG consistency at the join (*sync* construct), and optionally, atomically execute a sort of call-back function (*inlet* procedure).

Cilk lock variables are provided to define atomic chunks of code, enabling programmers to address synchronisation patterns that cannot be expressed using DAG consistency. As matter of fact, Cilk lock variables represent an escape in a programming model which has been designed for avoiding critical regions.

3.3.1 Pseudo-code

Our reference code structure for a farm implemented in Cilk is introduced in Fig. 3.3.1. A thread is spawn at the beginning of the program to implement the emitter behaviour and remain active until the end of the computation. The emitter thread defines new tasks and spawn new threads for their computation.

To avoid explicit lock mechanism we target `inlet` constructs. Ordinarily, a spawned Cilk thread can return its results only to the parent thread, putting those results in a variable in the parent's frame. The alternative is to exploit an `inlet`, which is a function internal to a Cilk procedure to handle the results of a spawned thread call as it returns. One major reason to use inlets is that all the inlets of a procedure are guaranteed to operate atomically with regards to each other and to the parent procedure, thus avoiding race conditions that can come out when the multiple returning threads try to update the same variables in the parent frame.

The `inlet`, which can be compared with OpenMP `critic` sections, can be easily exploited to implement the Collector behaviour as presented in the definition of the emitter function in Fig. 3.3.1.

Because `inlet` feature the limitation that the function have to be called from the cilk procedure that hosts the function, our emitter procedure, and our worker procedure have to be the same to use `inlet`. We differentiate the two behaviour exploiting a tag parameter and switching on its value.

3.4 TBB Farm

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 which individual threads of execution are created, synchronised, and terminated manually. Instead 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, and by automating efficient use of the cache. 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) [26, 37].

```
cilk int * emitter(int * newtask, int tag) {
  inlet void collector(int * newtask) {
    collect_task(newtask);
  }
  switch(tag) {
    case WORKER: {
      compute_task(newtask);
    }break;
    case EMITTER: {
      while(∃ newtask){
        newtask = create_task();
        collector(spawn emitter(newtask, WORKER));
      }
    }break;
    default: ;
  }
  return newtask;
}

cilk int main(int argc, char *argv[]) {
  null = spawn emitter(NULL, EMITTER);
  sync;
}
```

Figure 4: Farm structure in Cilk.

This approach groups TBB in a family of solutions for parallel programming aiming to enable programmers to explicitly define parallel behaviour via parametric exploitation patterns (*skeletons*, actually) that have been widely explored the last two decades both for distributed memory [17, 19, 9, 38, 21, 3] and shared memory [1] programming models.

3.4.1 Pseudo-code

The structure of the farm paradigm using the TBB library is sketched in Fig. 5. The implementation is based on the `pipeline` construct. The pipeline is composed of three stages: Emitter, Worker, Collector. The corresponding three objects are registered with the `pipeline` object in order to instantiate the correct communication network. The Emitter stage produces a pointer to arrays of basic tasks, referred as `Task` in the pseudo-code, each one of length `PIPE_GRAIN` (for our experiments we set the `PIPE_GRAIN` to 1024). The Worker stage is actually a filter that allows the execution of the `parallel_for` on input tasks. The `parallel_for` is executed using the `auto_partitioner` algorithm provided by the TBB library, this way the correct splitting of the `Task` array in chunks of basic tasks which are assigned to the executor threads, is left to the run-time support.

```
class Emitter:public tbb::filter {
public:
   Emitter(const int grain):tbb::filter(tbb::serial_in_order),grain(grain) {}
   void * operator()(void*) {
     newtask_t * Task[grain];
     while(∃ newtask){
       Task = create_task();
       return Task;
     }
     return NULL; //EOS
   }
};

class Compute {
  task_t ** Task;
public:
  Compute(task_t ** Task):Task(Task){}
  void operator() (const tbb::blocked_range<int>& r) const {
    for (int i=r.begin();i<r.end();++i)
       compute_task(Task[i]);
  }
};

class Worker:public tbb::filter {
 task_t ** Task;
public:
   Worker(const int grain):tbb::filter(tbb::serial_in_order),grain(grain) {}
   void * operator()(void * T) {
     Task = static_cast<task_t**>(T);
     tbb::parallel_for(tbb::blocked_range<int>(0,grain),
                      Compute(Task),
                      tbb::auto_partitioner());
     return Task;
   }
};

int main (int argc, char *argv[]) {
  Emitter  E(PIPE_GRAIN);
  Worker   W(PIPE_GRAIN);
  tbb::task_scheduler_init init(NUMTHREADS);
  tbb::pipeline pipeline;
  pipeline.add_filter(E);
  pipeline.add_filter(W);
  pipeline.run(1);
}
```

Figure 5: Farm structure in TBB.

4 Experiments and Results

We compare the performance of FastFlow farm implementation against OpenMP, Cilk and TBB using two families of applications: a synthetic micro-benchmark and the Smith-Waterman local sequence alignment algorithm. All experiments reported in the following sections 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.

4.1 Farm Communication Overhead

In order to test the overhead of the communication infrastructure for the different implementations of the farm construct, we developed a simple micro-benchmark application that emulate a typical parallel filtering application via a farm skeleton. The stream is composed by a ordered sequence of tasks which have a synthetic computational load associated. Varying this load is possible to evaluate the speedup of the paradigm for different computation grains. Each task is dynamically allocated by the emitter entity and freed by the Collector one. It consists of an array of 10 memory words that the worker reads and updates before passing the task to the Collector entity. Furthermore, each worker spend a fixed amount of time that correspond to the synthetic workload.

When possible, we have used the best parallel allocator (i.e. allocator which implementation is optimised to be used in parallel) available for the dynamic memory allocation. For example in the case of TBB, we used the TBB `scalable_allocator`.

In the OpenMP and Cilk cases, where no optimised parallel allocator are provided, we exploited the standard libc allocator. Anyway, with respect to the previously presented micro-benchmark, the performance degradation associated to an allocator, which is not optimised for parallel utilisation, is not determining. In fact the maximum parallelism required by the allocator is two: only one thread performs `malloc` operations (i.e. the emitter) and only one thread that performs `free` ones (i.e. the Collector).

As it is evident from the comparison of the trend of the curves in Fig. 6, the FastFlow implementation exhibits the best speedup in all cases. Is it interesting to notice that, for very fine grain computations (e.g. 0.5 μS), the OpenMP and Cilk implementations feature, with the increasing of the parallelism degree, a speedup factor lower than one: the addition of more workers just introduces only overhead, therefore leading to performances that are worst than the sequential ones.

In streaming computation with this type of computation grain, the communication overhead between successive stages in the farm construct is the most important limiting factor. Therefore we can assert that FastFlow is effective for streaming networks because it has been developed and implemented in order to provide communications with extremely low overhead as proved by the collected results.

4.2 Smith-Waterman Algorithm

In bioinformatics, sequence database searches are used to find the similarity between a query sequence and subject sequences in the database in order to determining similar regions between two nucleotide or protein sequences, encoded as a string of characters. The sequence similarities can be determined by

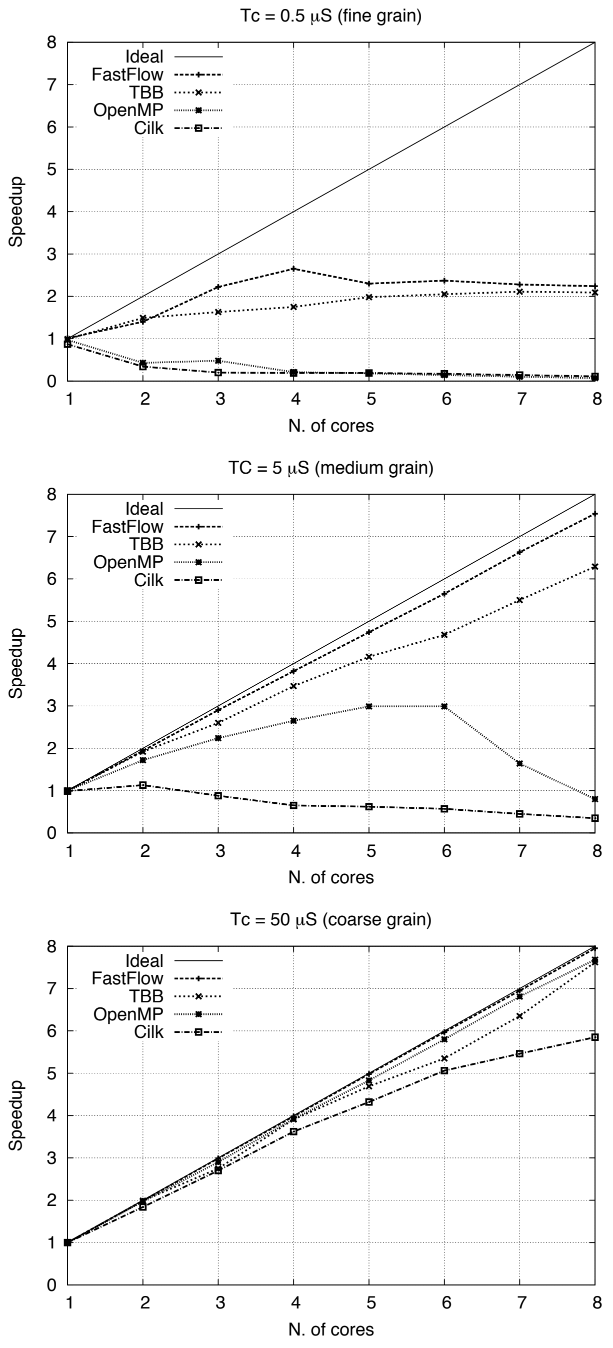


Figure 6: The speedup of different implementations of the farm paradigm for different computational grains, where Tc is the Computation Time per task: FastFlow vs OpenMP vs TBB vs Cilk.

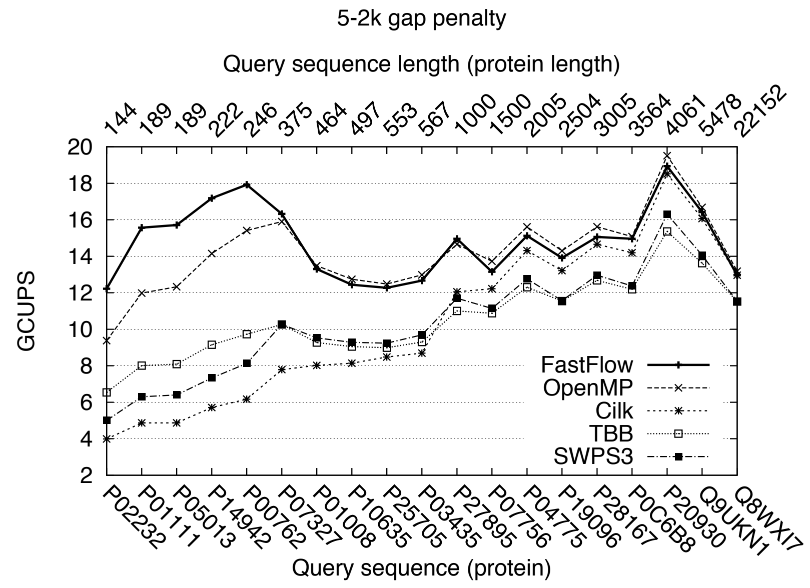
		Stream Task Time		
Query	Query len	Min (μS)	Max (μS)	Avg (μS)
P02232	144	0.333	2264.9	25.0
P10635	497	0.573	15257.6	108.0
P27895	1000	0.645	16011.9	197.0
P04775	2005	0.690	21837.1	375.0
P04775	5478	3.891	117725.0	938.5

Table 1: Minimum, maximum and average computing time for a selection of query sequences tested using a penalty gap 5-2 k.

computing their optimal local alignments using the Smith-Waterman (SW) algorithm [44]. SW is a dynamic programming algorithm that 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. The costs of this approach is expensive in terms of computing time and memory space used due to the rapid growth of biological sequence databases (the UniProtKB/Swiss-Prot database Release 57.5 of 07-Jul-09 contains 471472 sequence entries, comprising 167326533 amino acids) [43].

The recent emergence of multi- and many-core architectures provides the opportunity to significantly reduce the computation time for many costly algorithms like the Smith-Waterman one. Recent works in this area focus on the implementation of the SW algorithm on many-core architectures like GPUs [30] and Cell/BE [22] and on multi-core architectures exploiting the SSE2 instruction set [23, 39]. Among these implementations, we selected the SWPS3 [40] an optimised extension of the Farrar's work presented in [23] of the Strip Waterman-Algorithm for the Cell/BE and on x86/64 CPUs with SSE2 instructions. The original SWPS3 version is designed as a master-worker computation where the master process distribute the workload toward a set of worker processes. The master process read the query sequence, initialise the data structures needed for the SSE2 computation, and then fork all the worker processes so that each worker has its own copy of the data. All the sequences in the reference database are read and sent to the worker processes over POSIX pipes. The worker computes the alignment score of the query with the database sequence provided by the master process, and sent back over a pipe the resulting score.

The computational time is sensitive with respect to the query length used for the matching, the scoring matrix (in our case BLOWSUM50) and the gap penalty. As can be seen from Table 1 very small sequences require a smaller service time with respect to the longest one. Notice the high variance in the task service time reported in the table, this is due to the very different length of the subject sequence in the reference database (the average sequence length in UniProtKB/Swiss-Prot is 352 amino acids, the shortest sequence comprise 2 amino acids whereas the longest one 35213 amino acids). Furthermore, the higher the gap open and gap extension penalties, the fewer iterations are needed



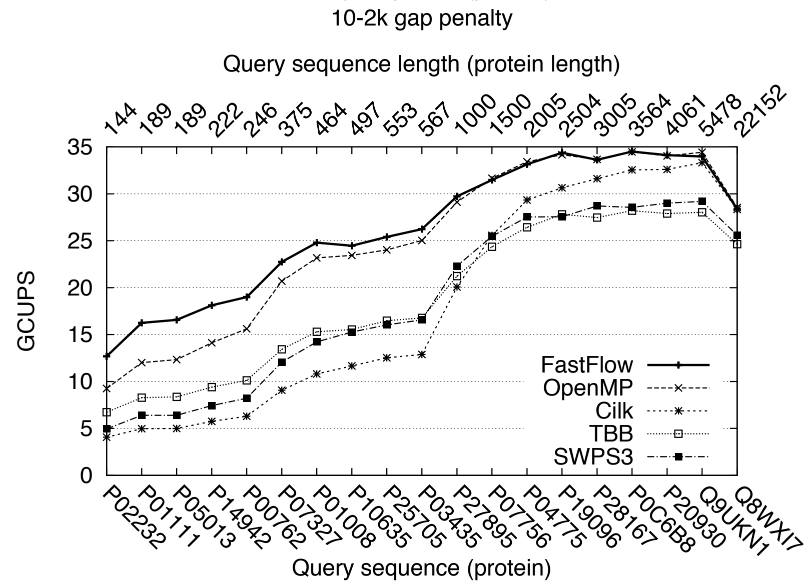


Figure 7: Smith-Waterman sequence alignment algorithm: comparison between FastFlow, OpenMP, TBB, and Cilk implementations. The SWPS3, which is based on POSIX primitives, is the original version from which the other has been derived. All the implementations share exactly the same sequential (x86/SSE2 vectorised) code.

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. In this, we did not modify the sequential code at all to achieve a fair comparison. For performance reasons, it is important to provide each worker threads with a copy of the data structures needed for the SSE2 computation. This is a critical aspects especially for implementations in Cilk and TBB, which do not natively support any kind of *Thread-Specific-Storage* (TSS). Notwithstanding this data is read-only, the third-party SSE somehow seems triggering the cache invalidation accessing the data, which seriously affect the performance. To overcome this problem we exploit a tricky solution: we use TSS exploiting a lower-level with respect to the programming model. In OpenMP this is not a problem because we have the possibility to identify the worker thread with the library call `omp_get_thread_num()`. The same possibility to identify a thread is offered by FastFlow framework as each parallel entity is mapped on one thread.

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

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 represents the time for a complete 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:

$$GCUPS = \frac{|Q|\ |D|}{T\ 10^9}$$

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 (the P02232 sequence) to 22,142 (the Q8WXI7 sequence) against Swiss-Prot release 57.5 database. The tests has been carried out on a dual quad-core Intel Xeon @2.50GHz running the Linux OS (kernel 2.6.x).

Figure 7 reports the performance comparison between FastFlow, OpenMP, Cilk, TBB and SWPS3 version of SW algorithm for x86/SSE2 executed on the test platform described above.

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

Cilk obtains lower performance value with respect to the original SWPS3 version with small sequences while performs very well with longer ones. OpenMP offers the best performance after FastFlow. Quite surprisingly TBB does not obtain the same good speedup that has been obtained with the micro-benchmark. It is still not clear which are the reasons, further investigation is required to find out the overhead source in the TBB version.

5 Conclusions

In this work we have introduced FastFlow, a low-level template library based on lock-free communication channel 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 it possible the efficient parallelisation of third-party legacy code, as the x86/SSE vectorised Smith-Waterman code. In the short term, we envision FastFlow as 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 stream-specific constructs. As this end, we studied how a streaming farm can be realised using several state-of-the-art programming frameworks for multi-core, and we have experimentally demonstrated that FastFlow farm is faster than other farm implementations on both synthetic benchmark and Smith-Waterman application.

As expected, the performance edge of FastFlow over the other frameworks is bold for fine-grained computations. This makes FastFlow suitable to implement a fast macro data-flow executor (actually wrapping around the order preserving farm), thus to achieve the automatic parallelisation of many classes of algorithms, including dynamic programming [6]. FastFlow will be released as open source library.

A preliminary version of this work has been presented at the ParCo conference [8].

6 Acknowledgments

We thank Marco Danelutto, Marco Vanneschi and Peter Kilpatrick for the many insightful discussions.

This work was partially funded by the project BioBITs ("Developing White and Green Biotechnologies by Converging Platforms from Biology and Information Technology towards Metagenomic") of Regione Piemonte, by the project FRINP of the "Fondazione della Cassa di Risparmio di Pisa", and by the WG Ercim CoreGrid topic "Advanced Programming Models".

References

[1] Marco Aldinucci. eskimo: experimenting with skeletons in the shared address model. *Parallel Processing Letters*, 13(3):449–460, September 2003.

[2] Marco Aldinucci, Sonia Campa, Pierpaolo Ciullo, Massimo Coppola, Silvia Magini, Paolo Pesciullesi, Laura Potiti, Roberto Ravazzolo, Massimo Torquati, Marco Vanneschi, and Corrado Zoccolo. The implementation of ASSIST, an environment for parallel and distributed programming. In *Proc. of 9th Intl Euro-Par 2003 Parallel Processing*, volume 2790 of *LNCS*, pages 712–721, Klagenfurt, Austria, August 2003. Springer.

[3] Marco Aldinucci, Massimo Coppola, Marco Danelutto, Marco Vanneschi, and Corrado Zoccolo. ASSIST as a research framework for high-performance grid programming environments. In *Grid Computing: Software environments and Tools*, chapter 10, pages 230–256. Springer, January 2006.

[4] Marco Aldinucci and Marco Danelutto. Stream parallel skeleton optimization. In *Proc. of PDCS: Intl. Conference on Parallel and Distributed Computing and Systems*, pages 955–962, Cambridge, Massachusetts, USA, November 1999. IASTED, ACTA press.

[5] Marco Aldinucci and Marco Danelutto. Skeleton based parallel programming: functional and parallel semantic in a single shot. *Computer Languages, Systems and Structures*, 33(3-4):179–192, October 2007.

[6] Marco Aldinucci, Marco Danelutto, Jan Dünnweber, and Sergei Gorlatch. Optimization techniques for skeletons on grid. In *Grid Computing and New Frontiers of High Performance Processing*, volume 14 of *Advances in Parallel Computing*, chapter 2, pages 255–273. Elsevier, October 2005.

[7] Marco Aldinucci, Marco Danelutto, and Peter Kilpatrick. Towards hierarchical management of autonomic components: a case study. In *Proc. of Intl. Euromicro PDP 2009: Parallel Distributed and network-based Processing*, pages 3–10, Weimar, Germany, February 2009. IEEE.

[8] Marco Aldinucci, Marco Danelutto, Massimiliano Meneghin, Peter Kilpatrick, and Massimo Torquati. Fastflow: Fast macro data flow execution on multi-core. In *Intl. Parallel Computing (PARCO)*, Lyon, France, September 2009.

[9] Marco Aldinucci, Marco Danelutto, and Paolo Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, July 2003.

[10] K. Arvind and Rishiyur S. Nikhil. Executing a program on the mit tagged-token dataflow architecture. *IEEE Trans. Comput.*, 39(3):300–318, 1990.

[11] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20(3):404–418, 2009.

[12] Holger Bischof, Sergei Gorlatch, and Roman Leshchinskiy. DatTel: A data-parallel C++ template library. *Parallel Processing Letters*, 13(3):461–472, 2003.

[13] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. Dag-consistent distributed shared memory. In *Proc. of the 10th Intl. Parallel Processing Symposium*, pages 132–141, Honolulu, Hawaii, USA, April 1996.

[14] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multi-threaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 1996.

[15] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. In *ACM SIGGRAPH '04 Papers*, pages 777–786, New York, NY, USA, 2004. ACM Press.

[16] Christopher Cole and Maurice Herlihy. Snapshots and software transactional memory. *Sci. Comput. Program.*, 58(3):310–324, 2005.

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

[18] Murray Cole. *Skeletal Parallelism home page*, 2009 (last accessed). `http://homepages.inf.ed.ac.uk/mic/Skeletons/`.

[19] Marco Danelutto, Roberto Di Meglio, Salvatore Orlando, Susanna Pelagatti, and Marco Vanneschi. A methodology for the development and the support of massively parallel programs. *Future Generation Compututer Systems*, 8(1-3):205–220, 1992.

[20] J. Darlington, A. J. Field, P.G. Harrison, P. H. J. Kelly, D. W. N. Sharp, R. L. While, and Q. Wu. Parallel programming using skeleton functions. In *Proc. of Parallel Architectures and Langauges Europe (PARLE'93)*, volume 694 of *LNCS*, pages 146–160, Munich, Germany, June 1993. Springer.

[21] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Usenix OSDI '04*, pages 137–150, December 2004.

[22] Michael Farrar. Smith-Waterman for the cell broadband engine.

[23] Michael Farrar. Striped Smith-Waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23(2):156–161, 2007.

[24] John Giacomoni, Tipp Moseley, and Manish 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)*, pages 43–52, New York, NY, USA, 2008. ACM.

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

[26] Intel Corp. *Intel Threading Building Blocks*, July 2009 (last accessed). `http://software.intel.com/en-us/intel-tbb/`.

[27] David Kirk. Nvidia cuda software and gpu parallel computing architecture. In *Proc. of the 6th Intl. symposium on Memory management (ISM)*, pages 103–104, New York, NY, USA, 2007. ACM.

[28] Edya Ladan-mozes and Nir Shavit. An optimistic approach to lock-free fifo queues. In *In Proc. of the 18th Intl. Symposium on Distributed Computing, LNCS 3274*, pages 117–131. Springer, 2004.

[29] Leslie Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, 1983.

[30] Yongchao Liu, Douglas Maskell, and Bertil Schmidt. CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Research Notes*, 2(1):73, 2009.

[31] H. Massalin and C. Pu. Threads and input/output in the synthesis kernal. *SIGOPS Oper. Syst. Rev.*, 23(5):191–201, 1989.

[32] Maged M. Michael and Michael L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

[33] Insung Park, Michael J. Voss, Seon Wook Kim, and Rudolf Eigenmann. Parallel programming environment for openmp. *Scientific Programming*, 9:143–161, 2001.

[34] M. Poldner and H. Kuchen. Scalable farms. In *Proc. of Intl. PARCO 2005: Parallel Computing*, Malaga, Spain, September 2005.

[35] Bill Pottenger and Rudolf Eigenmann. Idiom recognition in the Polaris parallelizing compiler. In *Proc. of the 9th Intl. Conference on Supercomputing (ICS '95)*, pages 444–448, New York, NY, USA, 1995. ACM Press.

[36] S. Prakash, Yann Hang Lee, and T. Johnson. A nonblocking algorithm for shared queues using compare-and-swap. *IEEE Trans. Comput.*, 43(5):548–559, 1994.

[37] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism.* O'Reilly, 2007.

[38] J. Serot. Tagged-token data-flow for skeletons. *Parallel Processing Letters*, 11(4):377–392, 2001.

[39] Adam Szalkowski, Christian Ledergerber, Philipp Kraehenbuehl, and Christophe Dessimoz. SWPS3 - fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2. *BMC Research Notes*, 1(1), 2008.

[40] Adam Szalkowski, Christian Ledergerber, Philipp Krähenbühl, and Christophe Dessimoz. *SWPS3 – fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2*, 2008.

[41] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. of the 11th Intl. Conference on Compiler Construction (CC)*, pages 179–196, London, UK, 2002. Springer-Verlag.

[42] Philippas Tsigas and Yi Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *SPAA '01: Proc. of the 13th ACM symposium on Parallel algorithms and architectures*, pages 134–143, New York, NY, USA, 2001. ACM.

[43] UniProt Consortium. *UniProt web site*, July 2009 (last accessed).

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