Accelerating code on multi-cores with FastFlow[*]

Marco Aldinucci[†] Marco Danelutto[‡] Peter Kilpatrick[§] Massimiliano Meneghin[¶]
Massimo Torquati[‖]

May 31, 2011

Abstract

FastFlow is a programming framework specifically targeting cache-coherent shared-memory multi-cores. It is implemented as a stack of C++ template libraries built on top of lock-free (and memory fence free) synchronization mechanisms. Its philosophy is to combine programmability with performance. In this paper a new FastFlow programming methodology aimed at supporting parallelization of existing sequential code via offloading onto a dynamically created software accelerator is presented. The new methodology has been validated using a set of simple micro-benchmarks and some real applications.

keywords: offload, patterns, multi-core, lock-free synchronization, C++.

1 Introduction

Parallel programming is becoming more and more a *must* with the advent of multi-core architectures. While up to few years ago faster and faster execution of programs was mainly the result of increased clock speed and of improvements in single processor architecture, from now on improvements may only come from better and more scalable parallel programs.

Here we discuss a semi-automatic parallelization methodology for existing code which is based on streamization, i.e. on the introduction and exploitation in the user application of *stream* parallelism. The methodology is based on the identification of suitable stream parallel patterns within the user application. Once these patterns have been recognized the computation of a stream of tasks according to the patterns is delegated to a structured parallel library–FastFlow–targeting in a very efficient way common cache coherent multi-core architectures.

The proposed methodology is *semi-automatic* as i) the programmer is still in charge of identifying the appropriate stream parallel patterns, but ii) the stream parallel pattern implementation is completely and efficiently delegated to the FastFlow runtime system. This happens by way of offloading onto a software device behaving as an accelerator (FastFlow software accelerator) which realizes a parallel pattern (skeleton).

Stream parallelism is the well-known programming paradigm supporting the parallel execution of a stream of tasks by using a series of *sequential* or *parallel* stages [1]. A stream program can be naturally represented as a graph of independent *stages* (kernels or filters) that communicate explicitly over data channels. Parallelism is achieved by running each stage simultaneously on *subsequent* or *independent* data.

As with all kinds of parallel program, stream programs can be expressed as a graph of concurrent activities, and directly programmed using a low-level shared memory or message passing programming

[*]To appear as: *Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. Accelerating code on multi-cores with FastFlow, in: Proc. of Euro-Par 2011, LNCS, Bordeaux, France. Springer.*

[†]Computer Science Department, University of Torino, Italy

[‡]Computer Science Department, University of Pisa, Italy

[§]Computer Science Department, Queen's University Belfast, UK

[¶]IBM Dublin Research Lab, Ireland

[‖]Computer Science Department, University of Pisa, Italy framework. Although this is still a common approach, writing a correct, efficient and portable program in this way is a non-trivial activity. Attempts to reduce the programming effort by raising the level of abstraction through the provision of parallel programming frameworks date back at least three decades with a number of significant contributions.

Notable among these is the *skeletal* approach [2] (a.k.a. *pattern-based* parallel programming), which is becoming increasingly popular after being revamped by several successful parallel programming frameworks [3, 4, 5, 6]. Parallel patterns capture common parallel programming paradigms (e.g. MapReduce, ForAll, Divide&Conquer, etc.) and make them available to the programmer as high-level constructs equipped with well-defined functional and parallel semantics. Some of these attempts explicitly include stream parallelism as a major source of concurrency exploitation, such as *pipeline* (running each stage simultaneously on subsequent stream items), *farm* (running multiple independent stages in parallel, each operating on a different task), and *loop* (providing a way to generate cycles in a stream graph). The *loop* skeleton together with the *farm* skeleton can be effectively used to model recursive and Divide&Conquer computations.

The stream paradigm perfectly suits the need for reducing inter-core synchronization overheads in parallel programs for shared cache multi-cores. Therefore, it can be used to build an efficient run-time support for a high-level programming model aimed at the effective design of parallel applications.

The rest of paper discusses the idea of streamization (Sec. 2), outlines the main FastFlow features (Sec. 3), describes the stream acceleration methodology (Sec. 4) and gives experimental results (Sec. 5). Related work (Sec. 6) and Conclusions are then presented.

2 Code acceleration through streamization

The parallelization of a sequential code is typically tackled via data dependence analysis [7]. Having fixed a reference grain for the parallelization, the tasks to compute are released from strict sequential order in such a way that the program semantics is preserved. As a result, these objects are organized in a static or dynamically evolving graph of communicating tasks.

Instruction level parallelism is typically exploited at the hardware level within the single core, while coarser grain parallelism is expressed among cores at software level. In the latter case, the primary sources of parallelism exploitation are iterative and recursive tasks since they often model heavy kernels that can be unfolded into fully or partially independent tasks.

Also, if the parallel code is derived from existing sequential code, variable privatization and scalar/array expansion are often applied to further relax *false dependencies* [8, 7]. These techniques consist in various levels of duplication of some memory area. Variable privatization nicely couples with stream parallelism making possible *dynamic privatization.* Privatized variables can be copied into a dynamically created data structure (e.g. the stream task type `task_t` in Fig. 2, right, lines 44–46) and offloaded in a non-blocking fashion to an accelerator.

Computation offloading, which is typically used to feed hardware accelerators (e.g. via OpenCL or CUDA), naturally creates a stream of tasks for the accelerator, provided it is realized via non-blocking mechanisms. As with GPUs, streamization techniques may offer significant opportunities on shared-cache multi-cores.

We classify stream sources in two broad categories: *exo*- and *endo*-streams.

Exo-streams A stream parallel approach naturally matches the parallelization of applications that manage externally produced (exo) streams of input and output data. These applications are increasingly found in many domains, e.g. multimedia and networking. In many cases, the whole dataset is large and has to be processed online. Moreover, there may be few or no sources of data parallelism to allow use of classical data parallel techniques.

Endo-streams A stream parallel approach also matches those computations that internally (endo) generate streams. We recognize three distinct sources of endo-streams: recursive computations and iterative computations, with and without dependencies. Recursion (*Recursive kernels*) appears as a natural programming technique in many algorithms working with dynamic data structures such as graphs and trees. In many cases they are data intensive algorithms and require significant computational power. Recursion could be easily modeled as a streaming network using a cyclic graph, whereas it can not readily be modeled by way of a data parallel approach. In this case, stream items are generated *on-the-fly* and represent different invocations of the recursive kernel. *Iterative kernels with independent iterations* represent the simplest case of endo-stream sources and are typically parallelized using a data-parallel approach. Streamization can also be applied in this case (e.g. generating and then processing a stream of items representing the different iterations) and is particularly useful when dynamic loops (i.e. *while*) or *for* loops with conditional jumps in the body (i.e. *break* or *goto* statements) are used. In fact, in all cases when flag variables are used in the code to skip the next code section, classical data parallel techniques are difficult to apply and may lead to poor performance. In the presence of loop-carried dependencies (*Iterative kernels with dependencies*), streamization may lead to more efficient synchronization patterns because it reduces the synchronization overhead due to data sharing in shared memory systems and thus shortens the critical path of execution. In *doAcross* task scheduling, the dependencies across threads are typically cross-iteration dependencies, which means that the underling memory location cannot be privatized. The synchronization overhead must be paid at least once for each iteration of a loop. On the contrary, in a pipeline schedule, loop-carried dependencies can be mapped onto the same thread. The remaining dependencies will still have the same overhead, but privatization will better tolerate the latency.

3 The FastFlow parallel programming framework

FastFlow is a C++ parallel programming framework aimed at simplifying the development of efficient applications for multi-core platforms. The key vision of FastFlow is that ease-of-development and runtime efficiency can both be achieved by raising the abstraction level of the design phase, thus providing developers with a suitable set of parallel programming patterns that can be efficiently compiled onto the target platforms.

FastFlow is conceptually designed as a stack of layers that progressively abstract the shared memory parallelism at the level of cores up to the definition of useful programming constructs supporting structured parallel programming on cache-coherent shared memory multi- and many-core architectures [9].

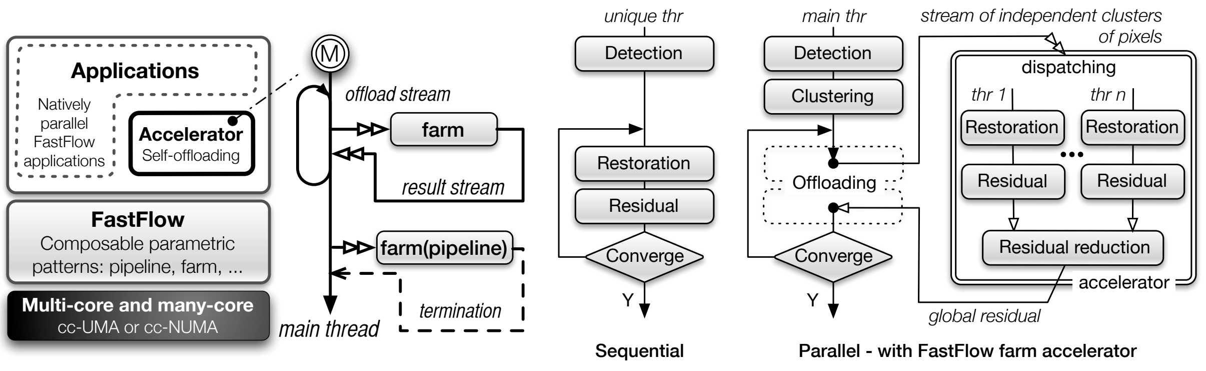
FastFlow's core is based on efficient Single-Producer-Single-Consumer (SPSC) and Multiple-Producer-Multiple-Consumer (MPMC) FIFO queues, which are implemented in a lock-free and wait-free fashion. On top of its core, FastFlow provides programmers with a set of patterns implemented as C++ templates: *farm*, *farm-with-feedback* (i.e. Divide&Conquer) and *pipeline* patterns, as well as their arbitrary nesting and composition. A FastFlow farm is logically built out of three entities: *emitter*, *workers*, *collector*. The emitter dispatches stream items to a set of workers which compute the output data. Results are then gathered by the collector back into a single stream.

Thanks to the lock-free implementation that significantly reduces cache invalidations in core-to-core synchronizations, FastFlow typically demonstrates increased speedup for fine-grained computations over other programming tools such as POSIX, Cilk, OpenMP, and Intel TBB [10]. For more information about the FastFlow implementation and features see [9].

4 Self-offloading on the FastFlow accelerator

A *FastFlow accelerator* is a software device that extends the FastFlow framework with a functional *self*-offloading feature, i.e. offloading from a thread running on the main CPU to other threads running on the main (multi-core) CPUs. The architecture of the accelerator is sketched in Fig. 1(a).

The main aim of self-offloading is to give the programmer an easy and semi-automatic way to introduce parallelism into a C/C++ sequential code by moving parts of the original code into the body of C++ methods, which will be executed in parallel according to the selected FastFlow skeleton (or skeleton composition). As shown in Fig. 2, this requires limited programming effort and may significantly speed up the original code by exploiting efficient stream parallelism.



(a) (b)

Figure 1: 1(a) FastFlow accelerator architecture with usage examples. 1(b). Flow charts of sequential and a FastFlow accelerated real case algorithm: two-step denoising

```
// Original code
#define N 1024
long A[N][N],B[N][N],C[N][N];          ①
int main() {
  // < init A,B,C>

  for(int i=0;i<N;++i) {
    for(int j=0;j<N;++j) {             ②

      int _C=0;
      for(int k=0;k<N;++k)             ❸
        _C += A[i][k]*B[k][j];
      C[i][j]=_C;

    }                                  ④
  }
}                                      ⑤
```

```
    // FastFlow accelerated code
    #define N 1024
①   long A[N][N],B[N][N],C[N][N];
    int main() {
      // < init A,B,C>

      ff::ff_farm<> farm(true /* accel */);
      std::vector<ff::ff_node *> w;
      for(int i=0;i<PAR_DEGREE;++i)
        w.push_back(new Worker);
      farm.add_workers(w);
      farm.run_then_freeze();

      for (int i=0;i<N;i++) {
②       for(int j=0;j<N;++j) {
          task_t * task = new task_t(i,j);     ➠
          farm.offload(task);
        }
④     }
      farm.offload((void *)ff::FF_EOS);
      farm.wait();  // Here join
⑤   }

    // Includes
    struct task_t {
      task_t(int i,int j):i(i),j(j) {}
      int i; int j;};

    class Worker: public ff::ff_node {
    public: // Offload target service
↻     void * svc(void *task) {
        task_t * t = (task_t *)task;
        int _C=0;
❸       for(int k=0;k<N;++k)
          _C += A[t->i][k]*B[k][t->j];
        C[t->i][t->j] = _C;
        delete t;
        return GO_ON;
      }
    };
```

Regions marked with white circled figures ①,②,④,⑤ are copy-pasted.
The region marked with the black circled figure (❸) has been selected to be accelerated with a farm. It is copied with renaming of variables that are concurrently changed, e.g. automatic variables in a loop. A stream of task_t variables is used to keep all different values of these variables.
Grey boxes create and run the accelerator; they are pre-determined according to the accelerator type.
The code marked with ➠ executes the offloading onto the accelerator; the target of the offloading is the `svc` method ↻ of the Worker class.

Figure 2: Derivation of FastFlow accelerated code from a simple sequential C++ application (matrix multiplication).

An accelerator is a collection of threads and has a global life-cycle with two stable states: *running* and *frozen*, plus several transient states. In a running state, all threads of an accelerator are logically able to run (either running or actively waiting on a non-blocking synchronization), whereas in a frozen state they are suspended (at the O.S. level). At any given time, due to non-blocking synchronizations, the total number of threads in running accelerators should typically be smaller (or equal) than core count for performance reasons. This kind of configuration typically benefits from O.S. affinity scheduling. In the case of a higher thread count, running threads will share available cores according to O.S. scheduling policies. The thread-to-core pinning is possible via FastFlow utility functions; automatic thread pinning/mapping are planned for future work.

The accelerator provides the programmer with one (untyped) streaming input channel and one (untyped) streaming output channel that can be dynamically *created* (and *destroyed*) from C++ code (either sequential or multi-threaded) as a C++ object (Fig. 2, right, lines 10–13). Thanks to the underlying shared memory architecture, messages flowing into these channels may carry both values and pointers to data structures.

When an accelerator is created (Fig. 2 lines 26–30), it can be switched on (Fig. 2 line 31): the accelerator threads are created and bound to system cores. A thread of a user can *wait* for an accelerator, i.e. suspend until the accelerator completes its input tasks, and then can put the accelerator into the frozen state. It is also possible to activate the accelerator asynchronously and pop output tasks from the accelerator's output channel using the *load_result* method.

A FastFlow accelerator is defined by a FastFlow skeletal composition augmented with an input stream and an output stream that can be, respectively, pushed and popped from outside the accelerator. Both the functional and extra-functional behaviour of the accelerator are fully determined by the chosen skeletal composition. For example, the *farm* skeleton provides the parallel execution of the same code (within a *worker* object) on independent items of the input stream. The *pipeline* skeleton provides the parallel execution of filters (or stages) exhibiting a direct data dependency. More complex behaviours can be defined by creating compositions of skeletons whose behaviour could be described using a (cyclic or acyclic) graph of tasks with well-defined functional and extra-functional semantics. Clear understanding of accelerator behaviour makes it possible to correctly parallelize segments of code.

The use of a farm accelerator is illustrated in Fig. 2. The code in Fig. 2 (left) shows a sequential program including three loops: simple matrix multiplication. Its accelerated version, shown in Fig. 2 (right), can be semi-automatically derived from the sequential by copy-pasting pieces of code into placeholders on a code template (parts in white background in the left column): for example, code marked with ①,②,④, and ⑤ are copied from left to right. The code that has been selected for offloading, in this case the body of a loop marked with ❸, is copied into the worker body after a suitable *renaming* of variables.

The accelerator shares the memory with its caller. As is well-known, transforming a sequential program into a parallel one requires regulation of possibly concurrent memory accesses. In low-level programming models this is usually done by using critical sections and monitors under the responsibility of the programmer. FastFlow does not prohibit these mechanisms, but promotes a methodology to avoid them. In very general terms, the sequential code statement can be correctly accelerated with FastFlow only mechanisms if the offloaded code and the offloading code (e.g. main thread) instances do not break any data dependency [7]. FastFlow helps the programmer in enforcing these conditions in two ways: *skeletons* and *streams*.

The *skeletal* structure of the accelerator induces a well-defined partial ordering among offloaded parts of code. For example, no-order for farm, a chain of dependencies for pipeline, a directed acyclic graph for farm-pipeline nesting/composition, and a graph for a farm-with-feedback. The synchronization among threads is enforced by *streams* along the paths of the particular skeleton composition, as in a data-flow graph. True dependencies (read-after-write) are admissible only along these paths. Streams can carry values or pointers, which act as synchronization tokens for indirect access to the shared memory.

Pragmatically, streams couple quite well with the needs of sequential code parallelization. In fact the creation of a stream to be offloaded on the accelerator can be effectively used to resolve anti-dependency (write-after-read) on variables since the stream can carry a copy of the values. For example, this happens when an iteration variable of an accelerated loop is updated after the (asynchronous) offload. This case naturally generalizes to all variables exhibiting a larger scope with respect to the accelerated code. The same argument can be used for output dependency (write-after-write). FastFlow accelerator templates accommodate all variables of this kind in one or more structs or C++ classes (e.g. `task_t`, lines 44–46) representing the input, and, if present, the output stream data type. All other data accesses can be resolved by just relying on the underlying shared memory (e.g. read-only, as with A in line 54, and single assignment as with C in line 55).

It is worth pointing out that the FastFlow acceleration methodology may not be fully automated. It has been conceived to ease the task of parallelization by providing the programmer with a methodology that helps in dealing with several common cases. However, many tasks require the programmer to make decisions, e.g. the selection of the code to be accelerated. In the example code in Fig. 2 there are several choices with different computation granularity: offload only the computations relative to index i, to both i and j, or to all three indices.

The programmer can control the communications among threads at higher level, and in particular can control when the thread reads from the input channel and writes to the output channel, the thread state (running and frozen) at different abstraction levels, the thread termination conditions, and the scheduling and the collection policies in the farm skeleton. In addition, as FastFlow equips the standard OS threads (e.g. POSIX threads) with additional synchronization mechanisms, the user retains the possibility to exploit thread native synchronization mechanisms (e.g. locks) and exploit thread library specific features (e.g. defining and using thread-specific storage). Also, almost any possible nesting of farm and pipeline skeletons is possible.

The low overhead added by the run-time support, together with the flexibility of the framework, widens the parallelization possibilities to a broader class of applications, and especially to programs performing frequent synchronizations.

5 Experimental Evaluation

The proposed self-offloading methodology has been validated on a set of three micro-benchmarks and two novel real-world applications. Several other complex applications —not presented here— have been parallelized using the FastFlow accelerator technique, such as a C.4.5 data classifier, a Gillespie simulator for biological systems and the Smith-Waterman string alignment. We refer back to [9] for an extensive listing.

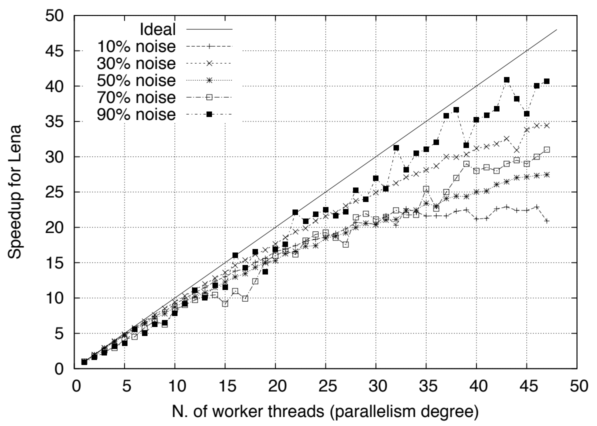
Two platforms are used in the evaluation: *8-core*) Intel workstation with 2 x quad-core Xeon E5520 Nehalem (16 HyperThreads) @2.26GHz; *48-core*) AMD Magny-Cours 4 x twelve-core Opteron 6174 @2.2GHz. Both run Linux x86_64.

5.1 Micro-benchmarks

We present the results obtained from the parallelization of three simple and well-known algorithms: dense square matrix multiplication, the Quicksort sorting algorithm, and the recursive computation of the n-th Fibonacci number.

The matrix multiplication consists in the parallelization of the “naïve” algorithm; the code is shown fully in Fig. 2; the parallel version achieves a speedup of 7.6 on the 8-core platform (1024x1024 integer matrices). In this case, a stream of submatrix multiplication tasks is created to feed the FastFlow accelerator.

The Quicksort benchmark has been parallelized using a Divide&Conquer parallel pattern (i.e. *farm* plus *loop* patterns). A worker receives a task with two indices describing a partition of a shared array from the emitter and executes a step of the Quicksort algorithm producing two tasks that return to the emitter, which in turn dispatches received tasks to workers. The workers switch to sequential processing at a given partition size threshold. The Fibonacci benchmark behaves similarly; the emitter also accumulates partial results. In both these micro-benchmarks the streams processed through the FastFlow accelerator are generated *on-the-fly* and composed of tasks corresponding to recursive calls to the main procedure. The Quicksort on a 50M integer array and Fibonacci(50) achieve speedups of 6.8 and 9.21, respectively, on the 8-core platform over-provisioned with 16 worker threads. The super-linear speedup achieved by Fibonacci is due to the HyperThreading technology, which, in contrast, brings no benefit to the Quicksort application.



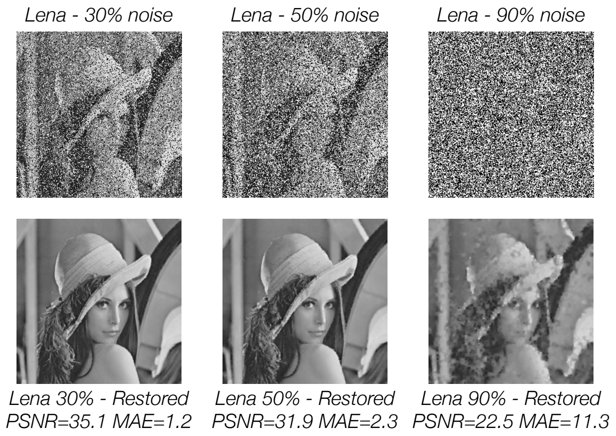

Lena - 30% noise
Lena - 50% noise
Lena - 90% noise
Lena 30% - Restored PSNR=35.1 MAE=1.2
Lena 50% - Restored PSNR=31.9 MAE=2.3
Lena 90% - Restored PSNR=22.5 MAE=11.3

Figure 3: Left) Speedup for the Lena image on the 48-core platform. Right) Restoration result with PSNR (Peak Signal-to-Noise Ratio) and MAE (Mean Absolute Error).

5.2 Applications

We discuss the results achieved when accelerating two applications with FastFlow. The applications are representative of two large and significant applications classes: the first performs classical data-parallel (with dependencies) computations and its acceleration is endo-stream parallel, while the second is a classical exo-stream parallel application. In both cases excellent results are achieved on state-of-the-art multi-core architectures.

Edge-Preserving Denoiser The edge-preserving denoiser is a two-step filter for removing salt-and-pepper noise (see Fig. 3 right). In the first step, an adaptive median filter is used to identify the set of noisy pixels; in the second step, these pixels are restored according to an iterative variational approach up to convergence. The detailed description of the sequential algorithm is beyond the scope of this paper; it ensures state-of-the-art restoration quality and execution time, and it is able to restore also very noisy images (e.g. 90% random noisy pixels) [11]. To ensure a high quality of restoration the algorithm features cross-dependencies among the noisy pixels, which induce a logical data-dependency pattern that cannot be solved with an *a priori* partitioning of the data set, as typically happens in data-parallelism. The idea behind the parallel porting of the proposed algorithm consists in clustering the noisy pixels into independent sets in such a way that cross-dependencies are respected. Following this, independent clusters, which can exhibit very different cardinalities, can be processed in parallel according to a farm paradigm. In particular, the clusters can be streamed (via offloading) to a FastFlow farm accelerator. The porting process, which is sketched in Fig. 1(b), required just a few hours development time. The clustering process, which is not present in the sequential version, has been designed from scratch as sequential code, and thus does not require concurrency skills.

Figure 3 reports the speedup achieved on the 48-core platform for the Lena 256x256 standard test image. The completion time of sequential processing grows linearly with noise ratio: from 9 to 180 seconds with 10% to 90% noise ratio. The parallel version speeds them up to a range of 0.4 to 4 seconds, respectively. Note that restoration quality metrics (PSNR and MAE) are comparable to or better than the best results in the area (e.g. Chan's method [11], while execution time for parallel execution is better than results in the literature [11, 9].

Stream File Compressor This application is a further development of an already parallel application: *pbzip2* [12], i.e. a parallel version of the widely used *bzip2* block-sorting file compressor. It uses pthreads and achieves very good speedup on SMP machines for large files. Small files (less then 1MB) are sequentially compressed. We extend it to manage streams of small files which can be compressed in parallel. In this case, in contrast with previous examples, the stream of data is not created within the application but exists independently of the application (e.g. comes from a POSIX stream, *find* shell command, etc.).

The original pbzip2 application is structured as a farm: the generic input file is read and split into independent parts (blocks) by a splitter thread; then each block is sent to a pool of worker threads which compress the blocks. The pool is hand-coded using pthread synchronizations and extensively hand-tuned. The FastFlow port of pbzip2 (*pbzip2_ff*) was developed by taking the original code of the workers and including it in a FastFlow farm pattern. Then, a second FastFlow farm whose workers execute the file compression sequentially was added. The two farms are run as two accelerators and fed by the main thread which selectively dispatches files to the two accelerators depending on the file size. The porting of the pbzip2 application to FastFlow has highlighted two aspects of the approach: (1) by using FastFlow it is possible to parallelize the algorithm using high-level parallel patterns rather than a hand-tuned mutex-based implementation without any performance penalty and with an actual performance improvement; (2) FastFlow non-blocking synchronizations exhibit good performance in comparison to traditional blocking synchronizations based on mutexes, even in worst-case scenarios such as coarse grained CPU-intensive elaborations where non-blocking behaviour might waste CPU cycles.

Table 1 compares the execution times of sequential bzip2, pbzip2 and pbzip2_ff on two different data sets: on the left, on large files shows that pbzip2_ff exhibits no significant slowdown against hand-tuned pbzip2; on the right, on files of various sizes shows the improved speedup of pbzip2_ff against pbzip2.

6 Related Work

The word accelerator is often used in the context of hardware accelerators. Usually accelerators feature a different architecture with respect to standard CPUs and thus, in order to ease exploitation of their computational power, specific libraries are developed. In the case of GPGPUs those (low-level) libraries include *Brook* [13], NVidia *CUDA*, and *OpenCL*. At a higher-level, *Offload* [14] enables offloading of parts of a C++ application, which are wrapped in offload blocks, onto hardware accelerators for asynchronous execution; *OMPSs* [15] enables the offloading of OpenCL and CUDA kernels as an OpenMP extension [16]. FastFlow, in contrast with these frameworks, does not target specific (hardware) accelerators but realizes a virtual accelerator running on the main CPUs and thus does not require the development of specific code.

Recent work [17] using the Charm++ programming model has demonstrated that accelerator extensions are able to achieve both code portability and speedup. However, in order to exploit the accelerator features, in contrast with FastFlow, the application has to be entirely rewritten using the Charm++ framework.

Streaming applications and patterns are also targeted by StreamIt [18], Intel Concurrent Collections (CnC) [19], and Intel TBB [5]. TBB, in particular, can be used to accelerate C/C++ programs in specific portions of code via parallel patterns (a.k.a. "algorithms") and thread-safe data containers. The only streaming pattern is the *pipeline* which, however, does not support non-linear streaming networks, which therefore have to be embedded in a pipeline with significant programming drawbacks. Farm and Divide&Conquer patterns are not natively provided, even if they can be simulated with lower-level features.

OpenMP [16] supports parallelization of sequential programs via pragmas that are, however, mainly designed to exploit loop-level data parallelism (e.g. *do_independent*) whereas the exploitation of other patterns

Table 1: On the left, compression and decompression time (S) on a single 1 GBytes file (528 MBytes compressed). On the right, execution time (S) and speedup over bzip2 in the case of a stream of 1078 files: 86% small (0–1 MBytes), 9% medium (1–10 MBytes), 4% large (10–50 MBytes), and 1% very large (50–100 MBytes). pbzip2 uses 16 threads. pbzip2_ff uses 16 threads for each accelerator.

	bzip2	pbzip2			pbzip2_ff		
# threads	1	4	8	16	4	8	16
compres	231	58.8	32.9	26.0	59.3	33.0	25.7
decompres	69	18.0	11.1	8.9	18.5	11.0	8.9

	bzip2	pbzip2		pbzip2_ff	
	Time (S)	Time (S)	Speedup	Time (S)	Speedup
comp.	538	97	5.5	72	7.5
decomp.	126	33	3.8	21	6.0

of parallelism (e.g. farm and Divide&Conquer) may require substantial re-factoring of the code.

A comparative performance study of FastFlow, OpenMP, and TBB on micro-benchmarks and the Smith-Waterman application is reported in [10].

7 Conclusions

In this paper the FastFlow accelerator, which represents an extension of the FastFlow framework specifically designed to support the easy porting of existing sequential C/C++ applications onto multi-cores using stream parallelism, is introduced. We identified exo- and endo-streams, showing that stream parallelism is applicable to a wide range of types of algorithm. The FastFlow accelerator exhibits well-defined functional and extra-functional behaviour represented by a skeleton composition; this helps in ensuring the correctness of the parallelization process. The main vehicle of parallelization is offloading of code kernels onto a number of additional threads running on the same CPU; we call this technique *self-offloading*. Code acceleration is supported by a methodology and by the unique ability of FastFlow to support very fine grain tasks on standard multi-cores.

The effectiveness of the proposed methodology has been demonstrated by a set of codes ranging from very simple kernels to real applications.

References

[1] Stephens, R.: A survey of stream processing. Acta Informatica **34**(7) (July 1997) 491–541

[2] Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computations. Research Monographs in Parallel and Distributed Computing. Pitman (1989)

[3] Vanneschi, M.: The programming model of ASSIST, an environment for parallel and distributed portable applications. Parallel Computing **28**(12) (December 2002) 1709–1732

[4] Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: Usenix OSDI '04. (December 2004) 137–150

[5] Intel Corp.: Threading Building Blocks. (2011) `http://www.threadingbuildingblocks.org/`.

[6] Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., Yelick, K.: A view of the parallel computing landscape. CACM **52**(10) (2009) 56–67

[7] Bernstein, A.J.: Program analysis for parallel processing. IEEE Trans. on Electronic Computers **EC-15**(5) (1966) 757–762

[8] Pop, A., Pop, S., Jagasia, H., Sjodin, J., Kelly, P.H.J.: Improving GCC infrastructure for streamization. In: Proc of the 2008 GCC Developers' Summit, Ottawa, Canada (June 2008)

[9] Aldinucci, M., Torquati, M.: FastFlow website. (2009) `http://mc-fastflow.sourceforge.net/`.

[10] Aldinucci, M., Meneghin, M., Torquati, M.: Efficient Smith-Waterman on multi-core with fastflow. In Danelutto, M., Gross, T., Bourgeois, J., eds.: Proc. of Intl. Euromicro PDP 2010: Parallel Distributed and network-based Processing, Pisa, Italy, IEEE (February 2010) 195–199

[11] Aldinucci, M., Drocco, M., Giordano, D., Spampinato, C., Torquati, M.: A parallel edge preserving algorithm for salt and pepper image denoising. Technical Report 138/2011, Università degli Studi di Torino, Dip. di Informatica, Italy (May 2011)

[12] Gilchrist, J.: Parallel data compression with bzip2. In: Proc. of IASTED Intl. Conference on Parallel and Distributed Computing and Systems. (2004) 559–564

[13] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for GPUs: stream computing on graphics hardware. In: ACM SIGGRAPH '04 Papers, New York, NY, USA (2004) 777–786

[14] Cooper, P., Dolinsky, U., Donaldson, A.F., Richards, A., Riley, C., Russell, G.: Offload - automating code migration to heterogeneous multicore systems. In: Proc of 5th Intl. Conference on High Performance Embedded Architectures and Compilers (HiPEAC). Volume 5952 of LNCS., Pisa, Italy, Springer (2010) 337–352

[15] Ferrer, R., Planas, J., Bellens, P., Duran, A., González, M., Martorell, X., Badia, R.M., Ayguadé, E., Labarta, J.: Optimizing the exploitation of multicore processors and GPUs with OpenMP and OpenCL. In: Proc of 23rd Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC). Volume 6548 of LNCS., Houston, TX, USA, Springer (2010) 215–229

[16] Park, I., Voss, M.J., Kim, S.W., Eigenmann, R.: Parallel programming environment for OpenMP. Scientific Programming **9** (2001) 143–161

[17] Kunzman, D.M., Kalé, L.V.: Towards a framework for abstracting accelerators in parallel applications: experience with cell. In: Proc. of the Conference on High Performance Computing (SC), Portland, Oregon, USA, ACM (2009) 1–12

[18] Thies, W., Karczmarek, M., Amarasinghe, S.P.: StreamIt: A language for streaming applications. In: Proc. of the 11th Intl. Conference on Compiler Construction (CC), London, UK (2002) 179–196

[19] Newton, R., Schlimbach, F., Hampton, M., Knobe, K.: Capturing and composing parallel patterns with Intel CnC. In: Proc. of 2nd USENIX Workshop on Hot Topics in Parallelism (HotPar '10), Berkley, CA, USA (June 2010)