Università di Pisa
Dipartimento di Informatica

Technical Report: TR-10-03

Accelerating sequential programs using FastFlow and self-offloading

Marco Aldinucci Marco Danelutto Peter Kilpatrick
Massimiliano Meneghin Massimo Torquati

February 12th, 2010

address: Largo B. Pontecorvo 3, 56127 Pisa, Italy. tel: +39 050 2212700 fax: +39 050 2212726

Accelerating sequential programs using FastFlow and self-offloading

Marco Aldinucci* Marco Danelutto Peter Kilpatrick†
Massimiliano Meneghin‡ Massimo Torquati

February 12th, 2010

Abstract

FastFlow is a programming environment specifically targeting cache-coherent shared-memory multi-cores. FastFlow is implemented as a stack of C++ template libraries built on top of lock-free (fence-free) synchronization mechanisms. In this paper we present a further evolution of FastFlow enabling programmers to offload part of their workload on a dynamically created software accelerator running on unused CPUs. The offloaded function can be easily derived from pre-existing sequential code. We emphasize in particular the effective trade-off between human productivity and execution efficiency of the approach.

Keywords Multi-core, parallel programming, streaming, skeletons, accelerator, non-blocking, synchronization, lock-free, function offload.

1 Introduction

The entire hardware industry has been moving to multi-core, which nowadays equips the large majority of computing platforms. The rapid shift toward multi-core technology has many drivers that are likely to sustain this trend for several years to come. In turn, software technology is also responding to this pressure [6]. Certainly, in the long term, writing parallel programs that are efficient, portable, and correct must be no more onerous than writing such programs for sequential computers. To date, however, parallel programming has not embraced much more than low-level communication and synchronization libraries. In the hierarchy of abstractions, it is only slightly above toggling absolute binary in the front panel of the machine. We believe that, among many, one of the reasons for such failure is the fact that programming multi-core is still

*Computer Science Department, University of Torino, Italy. Email: adinuc@di.unito.it
†Computer Science Department, Queen's University Belfast, UK.
‡IBM Technology Campus, Dublin Software Lab, Ireland.

perceived as a branch of high-performance computing with the consequent excessive focus on absolute performance measures. By definition, the raison d'être for high-performance computing is high performance, but MIPS, FLOPS and speedup need not be the only measure. Human productivity, total cost and time to solution are equally, if not more, important [21]. While a substantial methodological change will be required to allow effective design of parallel applications from scratch, the shift to multi-core is required to be graceful in the short term: existing applications should be ported to multi-core systems with moderate effort (despite the fact that they could be redesigned with larger effort and larger performance gain).

In this paper we present the *FastFlow accelerator*, i.e. a software accelerator based on *FastFlow* lock-free programming technology, and a methodology enabling programmers to seamlessly (and semi-automatically) transform a broad class of existing C/C++ program to parallel programs. The FastFlow software accelerator, in contrast with classic hardware accelerators, allows execution of streams of tasks on unused cores of the CPU(s).

The FastFlow accelerator is build on top of the *FastFlow* programming environment, which is a stack of C++ template libraries that, conceptually, progressively abstract the shared memory parallelism at the level of cores up to the definition of useful programming constructs and patterns (skeletons) [5, 25, 8]. Skeletons subsume a well-defined parallel semantics, which is used to ensure the correctness of the program when offloading tasks from a possibly sequential framework to a parallel one. FastFlow is discussed in Sec. 2.

As we shall see in Sec. 3, the FastFlow accelerator ultimately consists in a specific usage of the FastFlow framework. However, while FastFlow, in the general case, *requires* redesign of the application, the FastFlow accelerator suggests an easy and rapid way to improve the performance of existing C++ applications. This is further reinforced by the relative popularity (especially among non-specialists) of accelerator APIs, such as *OpenCL*, *CUDA*, IBM's Dynamic Application Virtualization [10], and annotation languages such as *OpenMP*. As we shall see in Sec. 3.2 and Sec. 4, one of the advantages of the FastFlow accelerator with respect to these environments is the tiny overhead introduced by the non-blocking lock-free synchronization mechanism which enables the parallelization of very fine grain activities, and thus broadens the applicability of the technique to legacy codes. Finally, in Sec. 4 we report on experiments with the proposed technique using a couple of simple yet significant examples: the C++ Mandelbrot set application from Nokia TrollTech's QT examples [19], and a heavily hand-tuned C-based N-queens solver [23].

2 The FastFlow parallel programming environment

As Fig. 1 shows, FastFlow is conceptually designed as a stack of layers that progressively abstract the shared memory parallelism at the level of cores up to

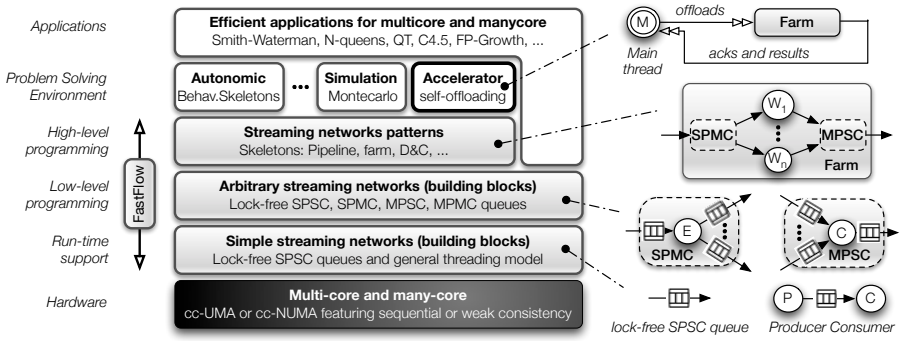


Figure 1: FastFlow layered architecture with abstraction examples.

the definition of useful programming constructs and patterns. The abstraction process has two main goals: 1) to promote high-level parallel programming, and in particular skeletal programming, i.e. pattern-based explicit parallel programming; 2) to promote efficient programming for multi-core.

2.1 Hardware

FastFlow is specifically designed for cache-coherent multiprocessors, and in particular commodity homogenous multi-core (e.g. Intel core, AMD K10, etc.). It supports multiprocessors exploiting any memory consistency, including very weak consistency models. FastFlow implementation is always lock-free, and for several memory consistency models is also memory fence-free (e.g., sequential consistency, total store ordering, and the x86 model). On other models (e.g., Itanium and Power4, 5, and 6), a store fence before an enqueue is needed [9].

At the current development status, FastFlow does not include any specific optimization targeting cc-NUMA platforms, although their support is planned and under design. Also, it currently does not automatically generate code running on GPUs (and other accelerators), even if it admits and supports the linking of code running on hardware accelerators. The full support of heterogenous platforms is currently under evaluation.

2.2 Run-time support

Taking inspiration from Fastforward queues [9] and Lamport's wait-free protocols [17], the second tier provides mechanisms to define simple streaming networks whose *run-time support* is implemented through correct and efficient lock-free Single-Producer-Single-Consumer (SPSC) queues.

The FastFlow run-time support layer realizes the two basic features: *parallelism exploitation*, i.e. the creation, destruction and life cycle control of different flows of control sharing the memory, and *asynchronous one-to-one communica-*

```
bool push(void * const data) {
  if (!data) return false;
  if (buf[pwrite]==NULL) {
   // WriteFence(); (e.g. for non x86 CPU)
    buf[pwrite] = data;
    pwrite+=(pwrite+1 >= size) ? (1−size): 1;
    return true;
  }
  return false;
}
```

```
bool pop(void ** data) {
  if (!data || (buf[pread]==NULL))
    return false;
  *data = buf[pread];
  buf[pread]=NULL;
  pread += (pread+1 >= size)?(1−size): 1;
  return true;
}
```

Figure 2: Actual FastFlow SPCS `pop` and `push` C++ implementation.

tion channels, supporting the synchronization of different flows of control. They are implemented by way of lock-free Single-Producer-Single-Consumer (SPSC) queue equipped with non-blocking `push` and `pop` operations.

While the former point can be addressed using quite standard technology (i.e. the wrapping of existing threading libraries, such as POSIX threads), the second exhibits a number of performance pitfalls on commodity shared-memory cache-coherent multiprocessors (as many commodity multi-core are). In particular, traditional lock-free implementations (such as Lamport's solution [17]) of SPSC queues are correct under sequential consistency only, where none of the current multi-cores implement sequential consistency. Also, some correct queue implementations induce a very high invalidation rate – and thus reduced performance – because they exhibit the sharing of locations that are subject to alternative invalidations from communication partners (e.g. head and tail of a circular buffers). The implementation does not suffer from the ABA problem [18], and it remains correct also in the case that only a reference instead of the full message is communicated using the queues. The FastFlow SPSC queue implementation (shown in Fig. 2) is largely inspired by Fastforward queues [9]. As with Fastforward queues, the `push` operation (issued by the producer) always reads and writes `pwrite` (i.e. tail pointer) only, and the `push` (issued by the consumer) always reads and writes `pread` (i.e. head pointer) only. This approach substantially differs from the traditional one (e.g. in Lamport's queues) where both the producer and the consumer access both the head and tail pointers causing the continuous invalidation of cache lines holding head and tail pointers.

2.3 Low-level programming

One small, but significant, abstraction step is evident in the *low-level programming* layer, which provides one-to-many, many-to-one, and many-to-many synchronizations and data flows. In the FastFlow approach these forms of communication are supported by SPMC (Single-Producer-Multiple-Consumer), MPSC (Multiple-Producer-Single-Consumer), MPMC (Multiple-Producer-Multiple-Consumer) queues, respectively. They can be directly used as general asymmetric asynchronous channels among threads. Clearly, messages flowing through these channels may carry memory pointers (that behave also as synchronization to-

kens), since we are exploiting the underlying hardware cache-coherent shared memory. Abstractly, these queues realize a general message passing API on top of a hardware shared memory layer.

SPMC, MPSC, and MPMC queues can be realized in several different ways, for example using locks, or in a lock-free fashion in order to avoid lock overhead (which is a non-negligible overhead in multi-core architectures). However, these queues could not be *directly* programmed in a lock-free fashion without using at least one atomic operation, which is typically used to enforce the correct serialization of updates from either many producers or many consumers at the same end of the queue. These operations, however, induce a memory fence, thus a cache invalidation/update[1], which can seriously impair the performance of parallel programs exhibiting frequent synchronizations (e.g. for fine-grain parallelism).

With FastFlow we advocate a different approach to the implementation of these queues, which require neither locks nor atomic operations. SPMC, MPSC, and MPMC queues are realized by using only SPSC queues and an arbiter thread, which enforce the correct serialization of producers and consumers. As shown in Fig. 1, this arbiter thread is called *Emitter* (E) when it is used to dispatch data from one channel to many channels, *Collector* (C) when it is used to gather data from many channels and push the messages into one channel, and *Collector-Emitter* (CE) when it behaves both as Collector and Emitter (a.k.a. Master-workers pattern).

Notice that, at this level, FastFlow does not make any decision about thread scheduling and their mapping onto the core; the programmer should be fully aware of all programming aspects and their potential performance drawback, such as load-balancing and memory alignment and hot-spots.

2.4 High-level programming

The next layer up, i.e. *high-level programming*, provides a programming framework based on parallelism exploitation patterns (*skeletons* [25]). They are usually categorized in three main classes: Task, Data, and Stream Parallelism. FastFlow specifically focuses on Stream Parallelism, and in particular provides: *farm*, *farm-with-feedback* (i.e. Divide&Conquer), *pipeline*, and their arbitrary nesting and composition. The set of skeletons provided by FastFlow could be further extended by building new C++ templates.

Stream Parallelism can be used when there exists a partial or total order in a computation. By processing data elements in order, local state may be maintained in each filter. The set of skeletons provided by FastFlow could be further extended by building new C++ templates on top of the Fastflow low-level programming layer.

Task Parallelism is explicit in the algorithm and consists of running the same or different code on different executors (cores, processors, machines, etc.).

[1] Notice that building a lock also requires an atomic operation unless working under sequential consistency for which a number of algorithms that do not require atomic operations exist, e.g. Lamport's Bakery algorithm [16].

Different flows-of-control (threads, processes, etc.) may communicate with one another as they work. Communication usually takes place to pass data from one thread to the next as part of the same data-flow graph.

Data Parallelism is a method for parallelizing 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 supports Loop-level Parallelism where successive iterations of a loop working on independent or read-only data are parallelized in different flows-of-control and concurrently executed.

While many of the programming frameworks mentioned in Sec. 5 offer Data and Task Parallel skeletons, only few of them offer Stream Parallel skeletons (such as TBB's *pipeline*). None of them offers the *farm* skeleton, which exploits functional replication of a set of *workers* and abstracts out the parallel filtering of successive *independent* items of the stream under the control of a scheduler, as a first-class concept.

We refer to [5] for implementation details and to [3, 4] for a performance comparison against POSIX locks, Cilk, OpenMP, and TBB. FastFlow is available at `http://sourceforge.net/projects/mc-fastflow/` under GPL.

3 Self-offloading on the FastFlow accelerator

A *FastFlow accelerator* is a software device wrapping a high-level FastFlow program, i.e. a skeleton or a composition of skeletons, and providing the application programmer with a functional *self-offloading* feature, since the offload happens on the same hardware device, i.e. CPU cores. The primary aim of self-offloading is to provide the programmer with an easy and semi-automatic path to introducing parallelism into a C/C++ sequential code by moving or copying parts of the original code into the body of C++ methods, which will be executed in parallel according to a FastFlow skeleton (or skeleton composition). This requires limited programming effort and it may speed up the original code by exploiting unused cores.

A FastFlow accelerator provides the programmer with one (untyped) streaming input channel and one (untyped) streaming output channel that can be dynamically *created* (and *destroyed*) from a C++ code (either sequential or multi-threaded) as a C++ object (Fig. 3 lines 26–30). Thanks to the underlying shared memory architecture, messages flowing into these channels may carry both values and pointers to data structures.

An accelerator, which is a collection of threads, has a global lifecycle with two stable states: *running* and *frozen*, plus several transient states. The running state happens when all threads are logically able to run (i.e. they are ready or running at the O.S. level). The frozen state happens when all threads are suspended (at the O.S. level). Transitions from these two states involve calls to the underlying threading library (and to the O.S.).

Once created, an accelerator can be run (line 31), making it capable of accepting tasks on the input channel. When running, the threads belonging to

```
1  // Original code
2  #define N 1024
3  long A[N][N],B[N][N],C[N][N];                    ①
4  int main() {
5    // < init A,B,C>
6
7    for(int i=0;i<N;++i) {                         ②
8      for(int j=0;j<N;++j) {
9
10       int _C=0;                                  ❸
11       for(int k=0;k<N;++k)
12         _C += A[i][k]*B[k][j];
13       C[i][j]=_C;
14
15     }                                            ④
16   }
17 }                                                ⑤
```

```
20 // FastFlow accelerated code
21 #define N 1024
22 long A[N][N],B[N][N],C[N][N];                    ①
23 int main() {
24   // < init A,B,C>
25
26   ff::ff_farm<> farm(true /* accel */);
27   std::vector<ff::ff_node *> w;
28   for(int i=0;i<PAR_DEGREE;++i)
29     w.push_back(new Worker);
30   farm.add_workers(w);
31   farm.run_then_freeze();
32
33   for (int i=0;i<N;i++) {                        ②
34     for(int j=0;j<N;++j) {
35       task_t * task = new task_t(i,j);           ➽
36       farm.offload(task);
37     }                                            ④
38   }
39   farm.offload((void *)ff::FF_EOS);
40   farm.wait();  // Here join                     ⑤
41 }
42
43 // Includes
44 struct task_t {
45   task_t(int i,int j):i(i),j(j) {}
46   int i; int j;};
47
48 class Worker: public ff::ff_node {
49 public: // Offload target service
50   void * svc(void *task) {                       ➼
51     task_t * t = (task_t *)task;
52     int _C=0;                                    ❸
53     for(int k=0;k<N;++k)
54       _C += A[t->i][k]*B[k][t->j];
55     C[t->i][t->j] = _C;
56     delete t;
57     return GO_ON;
58     }
59 };
```

Regions marked with white circled figures ①,②,③,⑤ are copy-pasted.
The region marked with 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 3: Derivation of FastFlow accelerated code from a simple sequential C++ application (matrix multiplication).

an accelerator might fall into an *active waiting* state. These state transitions exhibit a very low overhead and do not involve the O.S. Threads not belonging to the accelerator could *wait* for an accelerator, i.e. suspend until the accelerator completes its input tasks (receives the *End-of-Stream*, unique is propagated in transient states of the lifecycle to all threads) and then put it in the frozen state. At creation time, the accelerator is configured and its threads are bound into one or more cores. Since the FastFlow run-time is implemented via non-blocking threads, they will, if not frozen, fully load the cores in which they are placed, no matter whether they are actually processing something or not. Because of this, the accelerator is usually configured to use "spare" cores (although over-provisioning could be forced). If necessary, output tasks could be popped from the accelerator output channel.

3.1 Accelerating standard C++ codes: how to

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 is 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 [2, 1]; whose behaviour could be described using (acyclic or cyclic) data flow graphs. As we will see, clear knowledge of accelerator behaviour makes it possible to correctly parallelize segments of code.

The use of a farm accelerator is exemplified in Fig. 3. The code in the left column of the figure (lines 1–17) shows a sequential program including three loops: a simple matrix multiplication. Its accelerated version is shown on the right column (lines 20–59). The accelerated version 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 the offloading, in this case the body of a loop marked with ❸, is copied into the worker body after a suitable *renaming* of variables.

Because it is composed of threads, the accelerator shares the memory with its caller (and other threads of the same process). 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, according to Bernstein's conditions. 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 parallelisation. 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)

1. Choose a part of the code to be accelerated (e.g. a heavy kernel), understand the data dependencies, e.g. loop with independent iterations, data dependencies between functions or basic blocks, or more complex dependencies.
2. Choose a skeletal composition that models the required parallel execution schema.
3. Copy and paste the chosen code into the accelerator parts according to the skeleton template, e.g. in the farm worker, emitter (data scheduling), collector (data gathering and reduction).
4. Update accelerated code to access the memory via either stream values or pointers, if necessary.
5. Fill the skeleton template with accelerator creation and management code.
6. Substitute accelerated code with offloading calls.

Table 1: Self-offloading methodology.

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 A at line 54, and single assignment as C at line 55). The general methodology to accelerate existing C++ codes using the FastFlow accelerator is described in Table 1.

It is worth noticing that the FastFlow acceleration methodology may not be fully automated. It has been conceived to ease the task of parallelisation 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. 3 there are several choices with different computation granularity: offload only the index i or the indexes i and j, or all three indexes. Also, the correctness of the final code depends on the programmer: they should ensure that the accelerated code is thread safe, streams have a suitable type and their pointers are correctly cast, memory accesses are properly renamed, etc. FastFlow, like C/C++ itself, gives to the programmer much flexibility that should be used with great care.

3.2 Effectiveness and performance

The FastFlow accelerator aims to provide good speedup with moderate effort. Applications accelerated with FastFlow, in contrast with fully-fledged FastFlow applications, are not *fully* parallel. As with the other accelerators, Amdahl's law applies. Thus, the maximum speedup of an accelerated application depends primarily on which parts of the code have been offloaded, and on what fraction of the overall execution time is spent in that code. Equally important for per-

formance is the quality of the parallel code running on the accelerator in terms of computation vs communication size, load balancing, memory alignment, data locality and avoidance of false-sharing. For these problems FastFlow provides the programmer with specific tools to tune the performance: a parallel memory allocator, mechanisms to control task scheduling, and a mechanism to trace the execution of the workers' threads. A description of these tools goes beyond this paper: we refer to the FastFlow documentation for further details [5].

A significant advantage of the FastFlow accelerator with respect to other tools is the low latency of the run-time and the high flexibility of the framework. This, in turn, widens the parallelization possibilities to a broader class of applications, and especially those programs performing frequent synchronizations (e.g. fine-grain parallelism).

4 Experiments

In this section we show the performance of the FastFlow Accelerator using two well-known applications: a Mandelbrot set explorer and an N-queens solver. In both cases the code is third party and has been designed as sequential code; then it has been made parallel with the FastFlow farm accelerator. All experiments reported in the following sections have been executed on two platforms:

Andromeda Intel workstation with 2 quad-core Xeon E5520 Nehalem (16 HyperThreads) @2.26GHz with 8MB L3 cache and 24 GBytes of main memory.

Ottavinareale Intel workstation with 2 quad-core Xeon E5420 Harpertown @2.5GHz with 6MB L2 cache and 8 GBytes of main memory.

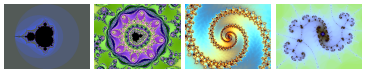
With the exception of very long runs, all presented experimental results are taken as an average of 5 runs exhibiting very low variance. All tested codes are available at the FastFlow website [5].

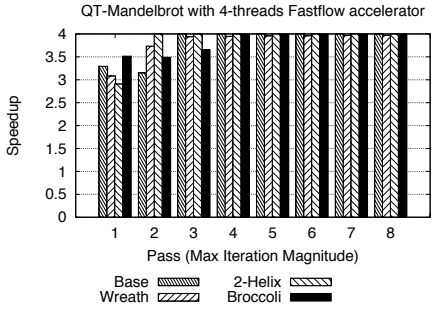
4.1 Interactive Mandelbrot set application

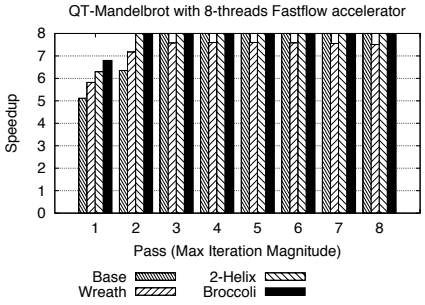
The "QT Mandelbrot" is an interactive application that computes the Mandelbrot set [19]. It is part of the Trolltech QT examples and it consists of two classes: `RenderThread.cpp`, i.e. a `QThread` subclass that renders the Mandelbrot set, and `MandelbrotWidget.cpp`, a `QWidget` subclass that shows the Mandelbrot set on screen and lets the user zoom and scroll. The application is multi-threaded (the two classes are run as QT threads) but threads are not used to speed the computation up since the whole computation is done within a single thread; rather they are used to decouple two different activities and to enhance responsiveness. During the time when the worker thread is recomputing the fractal to reflect the new zoom factor position, the main thread scales the previously rendered pixmap to provide immediate feedback. This use of threads is quite common in real life applications, where the user interface must remain responsive while some heavy operation is taking place.

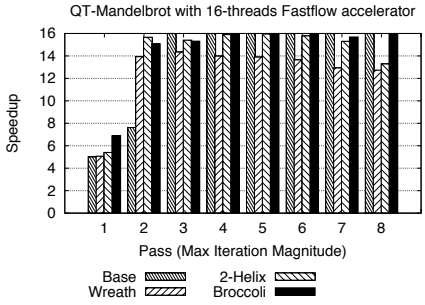
Tests on Andromeda: 8-core 16-hyperthreads Intel platform

Original QT-Mandelbrot execution time breakdown (mS)

Pass	Base	Wreath	2-Helix	Broccoli
				
1	79	140	154	128
2	98	382	435	120
3	696	1364	651	120
4	2682	1526	663	121
5	10430	1570	670	120
6	42203	1722	672	121
7	168532	2312	682	120
8	673203	4673	721	121

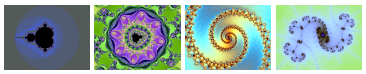


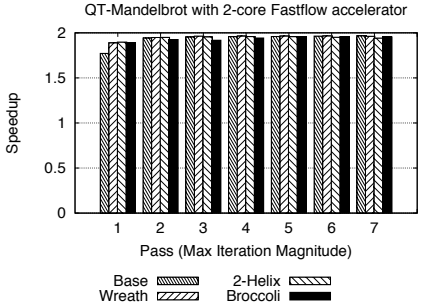


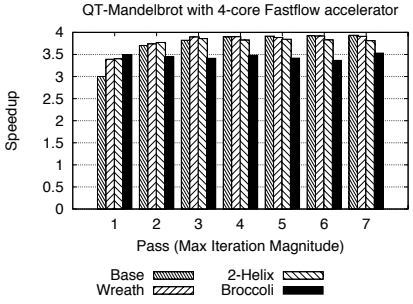


Tests on Ottavinareale: 8-core Intel platform

Original QT-Mandelbrot execution time breakdown (mS)

Pass	Base	Wreath	2-Helix	Broccoli
				
1	64	133	133	106
2	177	398	391	107
3	617	1422	586	107
4	2370	1594	594	107
5	9379	1638	595	107
6	37437	1795	597	107
7	150025	2413	606	107





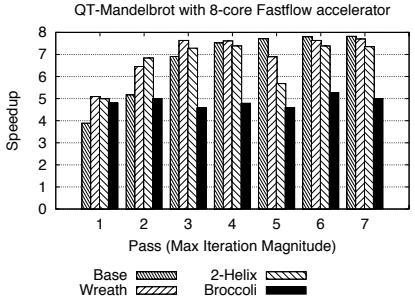


Figure 4: Original QT-Mandelbrot execution time with progressive precision (passes) in 4 different regions of the Mandelbrot set and the speedup obtained with FastFlow on two multi-core platforms (Andromeda and Ottavinareale).

Although it is a well-understood problem, the fully-fledged parallelization of the whole application is not trivial. The two threads synchronise with each other via QT events; to guarantee responsiveness the `MandelbrotWidget` thread may start, restart, and abort the activity of `RenderThread`. This kind of behavior, as well as the integration of QT threads with other threading libraries, makes porting to frameworks such as TBB and OpenMP non-trivial. The FastFlow accelerated version makes parallel the `RenderThread` by using a farm accelerator on the outer loop that traverses the pixmap of the Mandelbrot set. The farm accelerator is created once, then run and frozen each time a compute and interrupt signal is raised from `MandelbrotWidget`. The accelerated version can be easily derived by applying the methodology in Sec. 3.1. Figure 4 presents experimental results obtained by running the original code and the FastFlow accelerated version for 2, 4, 8 and 16 threads. As shown in the figure, the application has been tested for 8 refinement passes of the pixmap (according to the original algorithm) in 4 different regions of the plane exhibiting different execution times (and different regularity); in terms of Amdahl's law, the smaller this time, the smaller the fraction of the application that can be made parallel, the smaller the maximum speedup. As is clear from the figure, the FastFlow accelerator is able to boost the sequential application close to ideal speedup in almost all cases.

4.2 N-queens problem

The N-queens problem is a generalization of the well-known 8-queens problem. N-queens have to be placed on an NxN sized chessboard such that no queen can attack any of the others. The objective is to count all possible solutions. One of the fastest sequential implementations available for solving the problem is the heavily optimised C code written by Jeff Somers [23]. Somer's algorithm calculates one half of the solutions (considering one half of the columns), then flips the results over the "Y axis" of the board. Every solution can be reflected that way to generate another unique solution. That is because a solution cannot be symmetrical across the Y axis.

We attempted to accelerate the execution time of the sequential code using FastFlow. The FastFlow version uses the farm construct without the collector entity. A stream of independent tasks, each corresponding to an initial placement of a number of queens on the board, is produced and offloaded into the farm accelerator. The placement of the remaining queens in a task is handled by one of the accelerator's worker threads. In order to speed up the code, we simply applied the methodology described in Sec. 3.1. We copied the part of code that we wished to accelerate into the *svc* method of the Worker class; defined the stream type in such a way that it contained all the local variables that must be passed to the worker thread for the computation; and produced the stream of tasks from the initial placement of a given number of queens. No additional data structure or optimization has been added to the new code version.

Table 2 shows the execution times for the original sequential and the FastFlow accelerated versions for different board sizes. In all the tests we used 16

Andromeda: 8-core 16-hyperthreads platform					
Board size	# of solutions	Seq. Time	FastFlow Time	# of tasks	Speedup
18×18	666,090,624	5:53	34	1710	10.4
19×19	4,968,057,848	44:56	4:23	2072	10.2
20×20	39,029,188,884	6:07:21	35:41	2482	10.3
21×21	314,666,222,712	$\sim$ 2.2days	5:07:19	2943	10.3

Ottavinareale: 8-core platform					
Board size	# of solutions	Seq. Time	FastFlow Time	# of tasks	Speedup
18×18	666,090,624	6:52	1:06	1710	6.24
19×19	4,968,057,848	53:28	8:26	2072	6.34
20×20	39,029,188,884	7:16:27	1:8:56	2482	6.52
21×21	314,666,222,712	$\sim$ 2.7days	9:48:28	2943	6.69

Table 2: N-queens execution time breakdown on two different multi-core platforms (Andromeda and Ottavinareale).

worker threads and the stream has been produced from the initial placement of 4 queens (the resulting number of tasks is shown in the table). As can be seen, more than 10x speedup in the execution time has been obtained without any particular code optimization.

5 Related Work

In computing the word accelerator is used to refer to mechanisms that are used to speed up computation. The most widespread accelerators are hardware ones: the standard CPU is coupled with dedicated hardware optimized for a specific kind of computation. Examples include cryptographic accelerators, which have been developed to perform processor-intensive decryption/encryption; TCP/IP Offload Engines, which process the entire TCP/IP stack; and finally the well-known Graphics Processing Units (GPUs), which initially targeted graphics computations and are now increasingly used for a wider range of computationally intensive applications. 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 GPUs those libraries include *Brook*, NVidia *CUDA* and *OpenCL*.

Brook [7] provides extensions to the C language with single program multiple data (SPMD) operations on streams. It abstracts the stream hardware as a coprocessor to the host system. 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 GPUs via the OpenCL framework [13] and CUDA [14]. FastFlow accelerator differs from that of the previous libraries because it does not target specific accelerators; instead it make possible the usage of some of the cores as a virtual accelerator.

A recent work [15], using the Charm++ programming model [12], has demon-

strated that accelerator extensions are able to obtain good performance. Furthermore, code written with these extensions is portable without changing the application's source code. However, in order to exploit the accelerator features, the application has to be entirely rewritten using the Charm++ framework; this is not necessary in FastFlow.

Stream processing is extensively discussed in literature. Stream languages are often motivated by the application style used in image processing, networking, media processing, and a wide and growing number of problems in finance.

StreamIt [24] 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.

S-Net [22] is a coordination language to describe the communications of asynchronous sequential components (a.k.a. boxes) written in a sequential language (e.g. C, C++, Java) through typed streams. The overall design of S-Net is geared towards facilitating the composition of components developed in isolation.

Streaming applications are also targeted by TBB [11] through the *pipeline* construct. However, TBB does not support any kind of non-linear streaming network, which therefore has to be embedded in a pipeline with significant drawbacks in terms of expressivity and performance. As an example, a streaming network structured a workflow (a direct acyclic graph, actually) can be embedded in pipeline but this require pipeline stages to bypass data in which they have no interest. This clearly requires to change both the interfaces of the stages and their business logic and can be hardly made parametric. In addition, artificial data dependencies are (uselessly) introduced in the application with the consequent performance drawback.

OpenMP [20] is a very popular thread-based framework for multi-core architectures. It mostly targets Data Parallel programming and provides means to easily incorporate threads into sequential applications at a relatively high level. In an OpenMP program data needs to be labeled as shared or private, and compiler directives have to be used to annotate the code.

Both OpenMP and TBB can be used to accelerate serial C/C++ programs in specific portions of code, even if they do not natively include farm skeletons, which are instead realised by using lower-level features such as the *task* annotation in OpenMP and the *parallel_for* construct in TBB. OpenMP does not require restructuring of the sequential program, while with TBB, which provides thread-safe containers and some parallel algorithms, it is not always possible to accelerate the program without some refactoring of the sequential code.

In our vision, FastFlow falls between the easy programming of OpenMP and the powerful mechanisms provided by TBB. The FastFlow accelerator allows one to speed-up execution of a wide class of existing C/C++ serial programs with just minor modifications to the code. To the best of our knowledge none of the mentioned frameworks supports lock-free (and CAS-free) synchronizations.

6 Conclusions

In this paper we introduced the FastFlow accelerator which represents a further evolution of the FastFlow framework specifically designed to support the semi-automatic parallelization of existing sequential C/C++ applications on multi-cores. 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 on the same CPU; we call this technique *self-offloading*.

All in all, the work addresses an increasingly crucial problem for modern software engineering: how to make existing applications capable of effectively using modern multi-core systems with limited human effort. In this the FastFlow accelerator is supported by a semi-formal 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 simple but challenging applications. The FastFlow library and the code for all the applications in Sec. 4 are available under GPL at the FastFlow website [5].

References

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

[2] M. Aldinucci, M. Danelutto, and P. Kilpatrick. Autonomic management of non-functional concerns in distributed and parallel application programming. In *Proc. of Intl. Parallel & Distributed Processing Symposium (IPDPS)*, pages 1–12, Rome, Italy, May 2009. IEEE.

[3] 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 Parallel Computing (ParCo)*, Lyon, France, Sept. 2009.

[4] M. Aldinucci, M. Meneghin, and M. Torquati. Efficient Smith-Waterman on multi-core with FastFlow. In *Proc. of Intl. Euromicro PDP 2010: Parallel Distributed and network-based Processing*, Pisa, Italy, Feb. 2010. IEEE.

[5] M. Aldinucci and M. Torquati. *FastFlow website*, 2010. `http://mc-fastflow.sourceforge.net/`.

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

[7] 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*, pages 777–786, New York, NY, USA, 2004. ACM Press.

[8] M. Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.

[9] 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)*, pages 43–52, New York, NY, USA, 2008. ACM.

[10] IBM Corp. *IBM Dynamic Application Virtualization*, 2010. `http://www.alphaworks.ibm.com/tech/dav`.

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

[12] L. V. Kalé. Performance and productivity in parallel programming via processor virtualization. In *Proc. of the 1st Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)*, Madrid, Spain, Feb. 2004.

[13] Khronos Compute Working Group. *OpenCL*, Nov. 2009. `http://www.khronos.org/opencl/`.

[14] D. Kirk. NVIDIA CUDA software and GPU parallel computing architecture. In *Proc. of the 6th Intl. Symposium on Memory Management (ISMM)*, pages 103–104, New York, NY, USA, 2007. ACM.

[15] D. M. Kunzman and L. V. Kalé. Towards a framework for abstracting accelerators in parallel applications: experience with cell. In *SC '09: Proc. of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.

[16] L. Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

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

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

[19] Nokia Corp. *Qt Cross-platform application and UI framework*, 2010. `http://qt.nokia.com/`.

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

[21] D. Reed. *High-Performance Computing: Where'd The Abstractions Go?* BLOG@CACM, May 2009.

[22] A. Shafarenko, C. Grelck, and S.-B. Scholz. Semantics and type theory of S-Net. In *Proc. of the 18th Intl. Symposium on Implementation and Application of Functional Languages (IFL'06)*, TR 2006-S01, pages 146–166. Eötvös Loránd University, Faculty of Informatics, Budapest, Hungary, 2006.

[23] J. Somers. *The N Queens Problem: a study in optimization*, 2010. `http://jsomers.com/nqueen_demo/nqueens.html`.

[24] 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)*, pages 179–196, London, UK, 2002. Springer-Verlag.

[25] Wikipedia. *Algorithmic skeleton*, 2009. `http://en.wikipedia.org/wiki/Algorithmic_skeleton`.