# Efficient streaming applications on multi-core with FastFlow: the biosequence alignment test-bed[1]

Marco ALDINUCCI [a,2], Marco DANELUTTO [b], Massimiliano MENEGHIN [b], Massimo TORQUATI [b], and Peter KILPATRICK [c]

[a] *Computer Science Dept., University of Torino, Italy*
[b] *Computer Science Dept., University of Pisa, Italy*
[c] *Computer Science Dept., Queen's University Belfast, U.K.*

**Abstract.** Shared-memory multi-core architectures are becoming increasingly popular. While their parallelism and peak performance is ever increasing, their efficiency is often disappointing due to memory fence overheads. In this paper we present FastFlow, a programming methodology based on lock-free queues explicitly designed for programming streaming applications on multi-cores. The potential of FastFlow is evaluated on micro-benchmarks and on the Smith-Waterman sequence alignment application, which exhibits a substantial speedup against the state-of-the-art multi-threaded implementation (SWPS3 x86/SSE2).

**Keywords.** Lock-free queues, multi-threading, multi-core, stream parallel programming, software pipeline, SCM, Smith-Waterman, local sequence alignment, bioinformatics.

## Introduction

The success of future multi- and many-core chips depends mainly on advances in system software technologies (compilers, run-time support, programming environments) in order to utilise fully the on-chip parallelism. The tighter coupling of on-chip resources changes the communication to computation ratio that influences the design of parallel algorithms. Modern Single Chip Multiprocessor (SCM) architectures introduce the potential for low overhead inter-core communications, synchronisations and data sharing due to fast path access to the on-die caches. These caches, organised in a hierarchy, are also a potential limiting factor of these architectures since access patterns which are not carefully optimised may lead to relevant access contention for shared cache and invalidation pressure for replicated caches. In fact SCM and, especially, shared-cache multi-core

[1]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".

[2]Corresponding Author: Computer Science Department, University of Torino, Corso Svizzera 185, Torino, Italy; E-mail: aldinuc@di.unito.it

architectures are showing, even at the current scale, severe scalability limitations when programmed in the traditional way (e.g. using threads and monitors).

Performance on SCM is limited by the same factors as those that arise in shared-memory parallel architectures. Some of these limitations stem from the use of atomic memory transactions, which exhibit a rather high latency and tend to pollute the shared memory hierarchy. In reality, those operations are not strictly required when concurrent threads operate in a pipeline fashion, because data can be streamed from one stage to the next using fast lock-free queues. In this paper we show that this lock-free approach can be extended from simple pipelines to any *streaming network*. Such networks can be used directly to build efficient applications working on data streams; and, indirectly, to realise the implicit parallelisation of a wide class of algorithms. In fact a parallel Macro Data-Flow interpreter can be designed as a (cyclic) streaming network (e.g. according to the *farm* or *master-worker* paradigms/skeletons) [2,4,16,3].

FastFlow streaming networks are build upon two lower-level companion concepts: lock-free Multiple-Producer-Multiple-Consumer (MPMC) queues and a parallel lock-free memory allocator (MA). Both are realised as specific networks of threads connected via lock-free Single-Producer-Single-Consumer (SCSP) queues, which admit a very efficient implementation on cache-coherent SCM [9]. These concepts are implemented as a C++ template library.

Here we discuss how the FastFlow library can be used to build a widely used parallel programming paradigm (a.k.a. skeleton), i.e. the streaming stateful farm; and we compare its raw scalability against a hand-tuned Pthread-based counterpart on a dual quad-core Intel platform. The FastFlow farm skeleton can be rapidly and effectively used to boost the performance of many existing real-world applications, for example the Smith-Waterman local alignment algorithm [18]. In the following we show that a straightforward porting of the multi-threaded x86/SSE2-enabled SWPS3 implementation [19] onto FastFlow is twice as fast as the SWPS3 itself, which is a hand-tuned high-performance implementation.

## 1. Related Work

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. Many languages and libraries are available for programming stream applications. Some are general purpose programming languages that hide the detail of the underlying architectural. Stream languages enable the explicit specification of producer-consumer parallelism between coarse grain units of computation; examples include *StreamIt* [20], *S-Net* [17], *Brook* [6], and *CUDA* [11]. Some other languages, such as the *Intel Threading Building Block* (TBB), provide explicit mechanisms for both streaming and other parallel paradigms, while others, such as *OpenMP* and *Cilk*, although targeted particularly at data parallelism, can with greater programming effort be used to implement streaming applications.

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 task contains Java-like sequential code. The interconnection types provided 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.

S-Net [17] is a coordination language to describe the communications of asynchronous sequential components (a.k.a. boxes) written in sequential language (e.g. C, C++, Java) through typed streams. S-net boxes are entirely stateless, they are connected to other boxes by one single input and one single output typed streams and operate only in a input-process-output cycle. Complex stream networks are inductively defined using a set of four network combiners, which can express serial and parallel composition of two different networks as well as serial and parallel replication of a single network.

Brook [6] provides extensions to the 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 blocking behaviour: the execution of a kernel must complete before the next kernel can execute. This is the same execution model as is available on graphics processing units (GPUs). CUDA [11], which is an infrastructure from NVIDIA, presents features similar to those of Brook, but programmers are required to use low-level mechanisms to manage memory hierarchies.

Streaming applications are also targeted by TBB [10] through the *pipeline* construct. FastFlow 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). This approach is fully aligned with those traditionally followed within the skeleton research community [7,16,1,4,5,15,3]. However, TBB does not support any kind of non-linear streaming network, which therefore has to be embedded in a pipeline. This results in a non-trivial programming and performance drawback since pipeline stages must bypass data that they are not interested with.

Giacomini et al. [9] highlight the fact that traditional locking queues feature a high overhead on today's multi-core. Revisiting Lamport's work [12], which proves the correctness of lock-free mechanisms for concurrent Single-Producer-Single-Consumer (SPSC) queues on systems with memory sequential consistency commitment, they proposed a set of lock-free cache-optimised protocols for today's multi-core architectures. They also prove the benefit of those mechanisms on pipeline applications. Exploiting a lock-free SPSC, FastFlow substantially extends the work of Giacomini et al., from simple pipelines to *any streaming network*.

## 2. Fast Streaming Networks on Multi-core

FastFlow aims to provide a set of low-level mechanisms capable of supporting low-latency and high-bandwidth data flows in a network of threads running on a cache-coherent SCM. These flows, which are typical of streaming applications, are assumed to be mostly unidirectional and asynchronous. On these architectures the key issues concern 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 build on top of MPMC queues

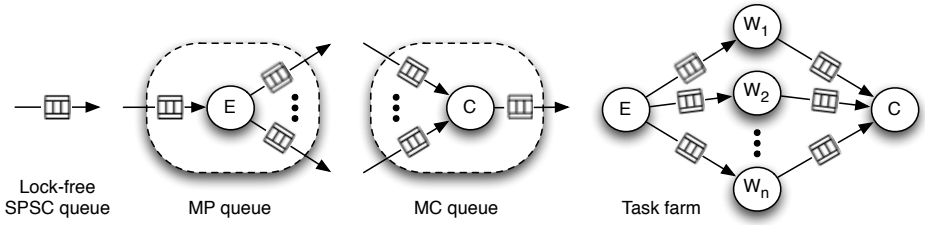


**Figure 1.** FastFlow concepts: Lock-free SPSC queue, MP queue, MC queue, Emitter (E) and Collector (C), and a streaming network implementing a task farm.

and can be substituted by either an OS standard allocator or a third-party allocator (e.g. TBB scalable allocator [10]). The FastFlow memory allocator substantially boosts FastFlow applications that use dynamic memory allocation but it does not have any impact on applications using static memory allocation, and thus is not discussed in this paper.

The key intuition behind 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. CAS: Compare-And-Swap) that behave as memory fences. The FastFlow design follows a different approach: 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 to 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 [9]). FastFlow networks do not suffer from the ABA problem [14] since MPMC queues are built by explicitly linearising correct SPSC queues using Emitters and Collectors.

The performance advantage of this solution derives from the relative speed of the copy with respect to the memory fence, and its impact on cache invalidation. This also depends on the size and the memory layout of copied data. The former point is addressed by using data pointers instead of data and enforcing that data is not concurrently written: in many cases this can be derived from the semantics of the skeleton that has been implemented using MPMC queues (for example, this is guaranteed in a stateless farm and many other cases). The latter point is addressed by using a suitable memory allocator (not presented in this paper).

## 3. Experimental Evaluation

We evaluate the performance of FastFlow with two families of applications: a synthetic micro-benchmark and the Smith-Waterman local sequence alignment algorithm. All experiments are executed on a shared memory Intel platform with two quad-core Xeon E5420 Harpertown 2.5GHz 6MB L2 cache and 8 GBytes of main memory.

```
void Emitter() {
  for(i=0;i<streamLen;++i){
    task=create_task();
    queue=
      SELECT_WORKER_QUEUE();
    queue->PUSH(task);
  }
}

void Worker() {
  while(!end_of_stream){
    myqueue->POP(&task);
    do_work(task);
  }
}

int main() {
  spawn_thread(Emitter);
  for(i=0;i<nworkers;++i){
    spawn_thread(Worker);
  }
  wait_end();
}
```

**Figure 2.** Micro-benchmark pseudo-code.

### *3.1. Micro-benchmarks*

Micro-benchmarks mock a parallel filtering application via a farm skeleton with a parametric synthetic load in the worker; by varying this load it is possible to evaluate the speedup for different computation grains, and therefore the overhead of the communication infrastructure. The pseudo-code of the micro-benchmark is sketched in Fig.2. A task is a pointer to a pre-allocated array of data, whereas the *do_work()* function just modifies the data pointed to by the task and then spends a fixed amount of time. Notice that the analysis of communication latency and bandwidth among a linear pipeline (through a lock-free SPSC queue) is beyond the scope of this paper (see related works [9]).

As is clear from the comparison of the two families of curves in Fig. 3, the standard implementation based on Pthread exhibits no speedup at all with fine grain tasks, while the proposed implementation exhibits a reasonable speedup, even for very fine computation grains (e.g. 5 $\mu$S and even finer grains such as 0.5 $\mu$S), and almost ideal speedup for medium to coarse grains. The communication overhead between successive stages is the most important limiting factor for fine-grain streaming networks. On the tested architecture we experienced that a put/get operation on a lock-based queue exhibits at least a threefold cost with respect to the same operation on a lock-free implementation (0.09 – 0.11 $\mu$s vs 0.03 – 0.04 $\mu$s).

### *3.2. 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 discover similar regions between two nucleotide or protein sequences, encoded as a string of characters in an alphabet (e.g. {A,C,G,T}). The sequence similarities can be determined by computing their optimal local alignments using the Smith-Waterman (SW) algorithm [18]. SW is a dynamic programming algorithm that is guaranteed to find the optimal local alignment with respect to the scoring system being used. Instead of looking at the total sequence, the SW algorithm compares segments of all possible lengths and optimises the similarity measure. The cost of this approach is fairly expensive in terms of memory space and computing time ($\mathcal{O}(mn)$, where $n$ and $m$ are the lengths of the two sequences), which is increasingly significant with the rapid growth of biological sequence databases.

For SW and similar algorithms, the emerging multi- and many-core architectures represent a concrete opportunity to cut computation time to acceptable figures, even for large datasets.

Moreover, it provides an ideal test-bed to evaluate the FastFlow performance on a real-world algorithm because:

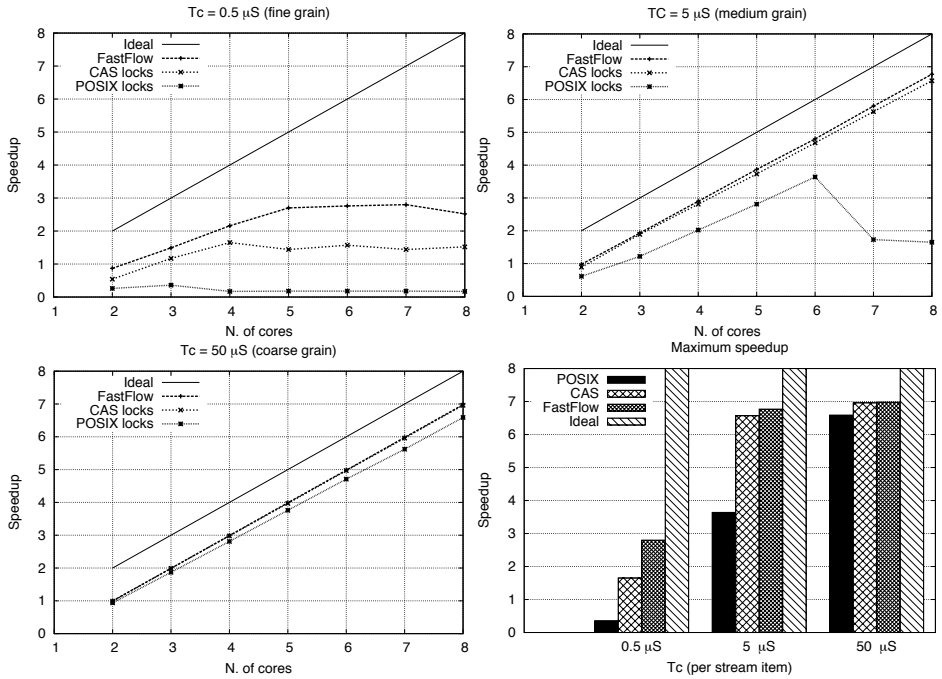


**Figure 3.** The speedup of different implementations of farm for different computation grains.

- SW features several efficient parallel hand-tuned implementations to be used as reference implementations;
- it works on a long stream of independent tasks;
- it makes possible variation of the computational grain from very fine to coarse by simply changing the query length.

Recent works in this area focus on the implementation of the SW algorithm on many-core architectures like GPUs [13] and Cell/BE [8] and on multi-core architectures exploiting the x86/SSE2 instruction set [8]. From them we selected the SWPS3, which has been extensively optimised to run on Cell/BE and on x86/64 CPUs with SSE2 instructions [19]. SWPS3 is an extension of Farrar's work for the Striped Smith-Waterman algorithm described in [8].

The original SWPS3 version is designed as a master-worker computation where the master process distributes the workload to a set of worker processes. The master process handles file I/O and communicates with the worker processes over bidirectional pipes to supply them with database sequences and to collect alignment scores. Every worker computes the alignment of the query sequence with a separate database sequence. We modified the original code to turn it into a FastFlow application by simply (almost syntactically) substituting processes with threads and pipes with FastFlow queues. The master thread (emitter) reads the sequence database and produces a stream of pairs: <*query sequence, subject sequence*>. The query sequence remains the same for all the subject sequences contained in the database. The worker threads compute the Smith-Waterman

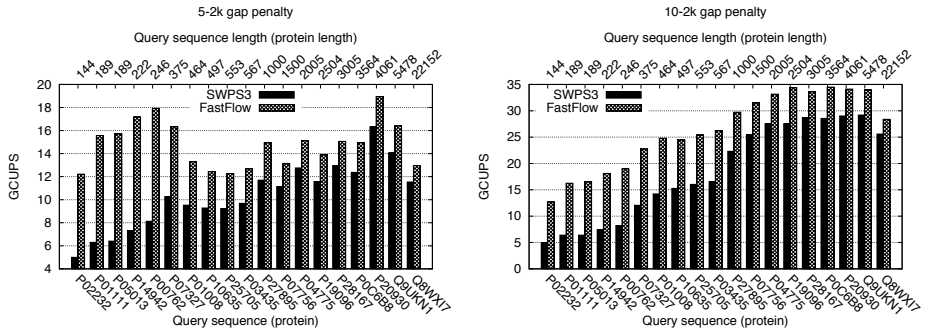


**Figure 4.** Smith-Waterman sequence alignment algorithm: FastFlow against SWPS3 implementation for $(5-2k)$ and $(10-2k)$ gap penalties evaluated on Release 57.5 of 07-Jul-09 of UniProtKB/Swiss-Prot (contains 471472 sequence entries, comprising 167326533 amino-acids abstracted from 181042 references). The two implementations share exactly the same sequential (x86/SSE2 vectorised) code.

algorithm on the input pairs using the SSE2 instructions set. The collector thread gets the resulting score and produces the output string (score and sequence name).

Figure 4 reports the performance comparison between SWPS3 and the FastFlow version of the SW algorithm for x86/SSE2 executed on the test platform described above. The scoring matrix BLOSUM50 is used for the tests with a gap penalty of 10-2 k and 5-2 k respectively. All the other parameters in the SWPS3 implementation are used with their default values. As can be seen from the figures, the FastFlow implementation outperforms the original SWPS3 x86/SSE2 version for all the sequences tested. SWPS3 achieves a peak performance of up to 16.31 and 29.19 GCUPS with a gap penalty of 5-2 k and 10-2 k, respectively, whereas the FastFlow version reaches a peak performance of up to 18.94 GCUPS with a gap penalty of 5-2 k and 34.38 GCUPS with a gap penalty of 10-2 k. The GCUPS (Giga-Cell-Updates-Per-Second) is a commonly used performance measure in bioinformatics and is calculated by multiplying the length of the query sequence by the length of the database divided by the total elapsed time.

The smaller the query sequences are, the bigger is the performance gain. This is mainly due to the lower overhead of the FastFlow communication channels with respect to the standard POSIX channels exploited by SWPS3. Other results, not shown here due to space limitations, demonstrate the further performance gain obtained by the FastFlow implementations when multiple query sequences are tested in a single FastFlow run. In fact, in this case, contrary to the SWPS3 implementation, FastFlow does not require the flushing of farm queues when the query changes.

## 4. Conclusions

FastFlow extends the use of lock-free SPSC queues from the implementation of the linear pipeline to any streaming network, including task farming. FastFlow networks make it possible to build very fast streaming applications on commodity multi-core architectures even for very fine-grained tasks. As an example, the FastFlow version of the Smith-Waterman algorithm, obtained from a third-party high-performance implementation by simply substituting communication primitives, is always faster when compared to the

original version and exhibits double its speedup on fine-grained datasets. The presented results should be considered as a "feasibility study" of the proposed approach. We envisage, in the medium term, FastFlow as part of the run-time support of a higher-level skeletal programming framework for a multi- and many-core C++ template-based library that will be released as open-source. Comparison with other multi-core-specific programming frameworks such as Intel TBB and OpenMP is among planned activities.

## References


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

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

[3] M. Aldinucci, M. Danelutto, and P. 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, Feb. 2009. IEEE.

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

[5] H. Bischof, S. Gorlatch, and R. Leshchinskiy. Dattel: A data-parallel c++ template library. *Parallel Processing Letters*, 13(3):461–472, 2003.

[6] 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.

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

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

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

[11] D. 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.

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

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

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

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

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

[17] 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.

[18] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J Mol Biol*, 147(1):195–197, March 1981.

[19] A. Szalkowski, C. Ledergerber, P. Krähenbühl, and C. Dessimoz. *SWPS3 – fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2*, 2008.

[20] 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.