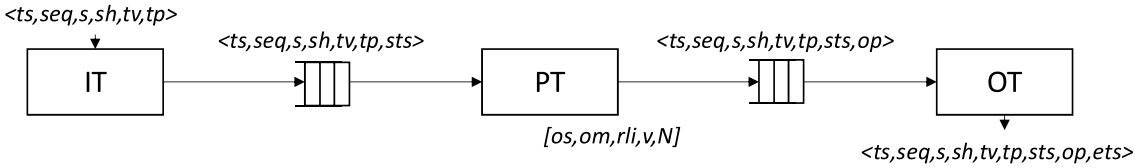
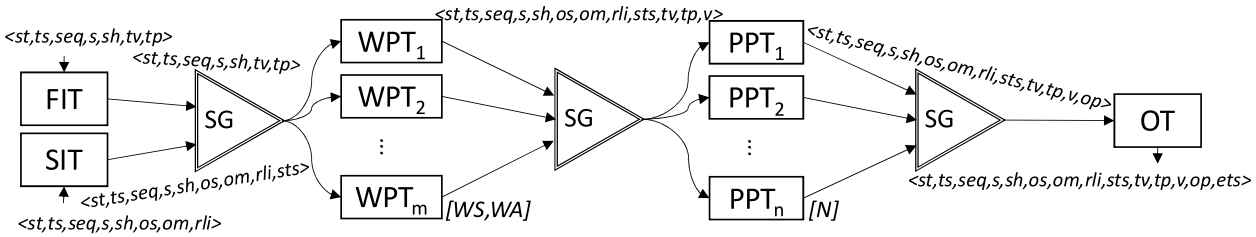
CHALMERS
UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG





Concurrent Data-Structures Applied to Financial Data-Stream Processing

Applying Concurrent Lock-Free Data-Structures to the design and development of a Financial Options Pricing Stream Processor

Master's thesis in Computer Systems and Networks (MPCSN)

ALFONSO ALHAMBRA MORON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY AND UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2016

Concurrent Data-Structures Applied to Financial Data-Stream Processing

Applying Concurrent Lock-Free Data-Structures to the design and development of a Financial Options Pricing Stream Processor

ALFONSO ALHAMBRA MORON

Department of Computer Science and Engineering
Division of Networks and Systems
Distributed Computing and Systems Research Group
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden 2016

Concurrent Data-Structures Applied to Financial Data-Stream Processing
Applying Concurrent Lock-Free Data-Structures to the design and development of a Financial Options Pricing Stream Processor
ALFONSO ALHAMBRA MORON

Supervisor: Philippas Tsigas, Department of Computer Science and Engineering
Supervisor: Ioannis Nikolakopoulos, Department of Computer Science and Engineering
Examiner: Marina Papatriantafilou, Department of Computer Science and Engineering

Department of Computer Science and Engineering
Division of Networks and Systems
Distributed Computing and Systems Research Group
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 (0)31 772-10 00

Cover: Data structures and operators diagram of the first and last iterations of the financial options pricing stream processor.

Typeset in LaTeX
Gothenburg, Sweden 2016

Concurrent Data-Structures Applied to Financial Data-Stream Processing
ALFONSO ALHAMBRA MORON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This Thesis focuses on the efficient utilization of lock-free concurrent data structures in the scope of financial data-stream processing to achieve low latency and high throughput parallel solutions responding to the continuously increasing high throughput and low latency demand to process financial streams of data [17, 14, 30].

The two main problems address in the scope of this Thesis are options pricing and risk assessment based on volatility aggregation. A proof-of-concept financial stream processing engine has been designed and developed consuming a stream of data representing the real-time behavior of the underlying stock exchange market, and a stream of data representing the specifications of the option contracts to be priced to produce an output stream of priced option contracts.

The throughput and latency results obtained when evaluating the different proposed solutions suggest that the *ScaleGate* data-structure, [7, 22], when efficiently used expediting its behavior with a heartbeat mechanism, satisfactorily responds to the aforementioned high throughput and low latency demand in addition to guaranteeing the correct ordering of the resulting output stream in non-decreasing timestamp order.

Keywords: Shared Memory Parallelism, Lock-Free Synchronization, *ScaleGate*, Stream Aggregate, Stream Join, Throughput, Latency, Finance, Options Pricing, Volatility.

Acknowledgements

I would like thank my supervisors, Yiannis and Philippas, for the chance of taking part in the great research being performed by the Division of Networks and Systems. The last months working closely with you have helped me growing not only as a student but also at a personal and professional level. Thank you very much for all your expert advice, for all your patience in some of the endless meetings and specially for being always ready to help with any issue I had.

I would also like to thanks my examiner, Marina, and all the members of the Division of Networks and Systems here at Chalmers for all the advice, encouragement and support. Having had the chance to learn first-hand about your research in the weekly seminars have strongly inspired me.

I cannot forget about all the docents I have had the opportunity to learn from here at Chalmers, back in the Autonomous University of Madrid and previously in my school and high-school, as well as all the professionals that I have had the chance to collaborate with and learn from in my different internships during the last years. All of you have contributed one way or another to the production of this Thesis. Thank you all.

I would like to thank "la Caixa" foundation for all the financial and personal support. I feel greatly honored to have been funded by the 2014 "la Caixa" Europe scholarship during the last two years.

I would also like to thank Parallel Scalable Solutions AB for splendidly facilitating me the NOBLE Professional Edition library which I have used in any experiment in which an efficient concurrent lock-free queue was needed.

I would like to specially thank my family, for the constant and unconditional support, and all my friends who have always been there when needed. Thank you all.

Finally, I would like to show my heartfelt gratitude to Alma. You have helped me growing at all levels, you have shared everything with me, you have helped me become the person I am right now. Words are not enough to express my gratitude. Thank you.

Alfonso Alhambra Moron, Gothenburg, June 2016

Contents

List of Figures

List of Tables

Listings

1
Introduction

The main focus of this Thesis is the research towards the application of the stream-processing technologies being developed by the Distributed Computing and Systems Research group to data-streaming challenges detected in the scope of finance.

The main objective of this research effort is, on the one hand, to test and improve, based on experimental results, the concurrent data-structures being developed to enhance parallelization in the context of data-streaming processing and, on the other hand, to profit from this technology in the domain of financial applications, in which more and more data need to be processed everyday on a streaming fashion and energy restrictions pose a constant challenge towards optimizing the usage of computing resources.

1.1 Context and Motivation

To better introduce the context of this Thesis it is key to understand the concept of data-streaming. Paraphrasing [7]: "Data streaming emerged as an alternative to store-then-process computing. In data-streaming, continuous queries (defined as directed acyclic graphs of interconnected operators) are executed by stream processing engines that process incoming data in a real-time fashion, producing results on an on-going basis". A brief analysis of the introduced definition helps understanding the main difference between store-then-process computing and data-streaming computing: the lack of backup storage in the former. In addition to this, it is common to assume that the data set being analyzed in data-streaming computing is an infinite dataset of which every time only a small subset is available for the stream processing engine and for a limited amount of time to produce results.

It is natural to understand that in this context, and as highlighted in [13], cited in [7], low-latency, due to the real time requirements of the results, and high throughput, depending on the amount of data to be processed every second, are key requirements for the stream processing of increasingly large data volumes making parallelism [25] a necessity. In line with this, the Distributed Computing and Systems Research Group at Chalmers is researching towards the definition of concurrent, linearizable and lock-free data-structures [7] in order to enhance the parallelization of the processes of aggregating the input streaming data and aggregating the output generated by parallelized operators.

A natural application scenario of the data stream processing paradigm is the domain of financial applications. Stock prices can naturally be seen as a stream of data and derived calculations such as option prices or market volatility are suitable operators in the data-streaming context applied to this scenario. A brief review of the state of the art literature in this field, e.g. [14], [17], is enough to become aware that financial workloads demand highly energy efficient stream processing solutions with low-latency and high-throughput requirements. In addition to this, the low-latency and high-throughput requirements make the energy challenges tougher by forcing the datacenters dealing with financial calculations to be located physically close to the financial data sources, which, as stated in [17], removes the freedom to site the datacenters in geographical regions with more preferential energy prices.

In this context, the main hypothesis of this Thesis is that the application of the aforementioned concurrent data-structures being developed by the Distributed Computing and Systems Research group to the financial analytics scenario can optimize the parallelized usage of computing resources, thus, reducing latency and increasing throughput without increasing the energy consumption.

1.2 Goals, Challenges and Limitations

Two main computational finance problems are addressed in the scope of this Thesis. The first of them consisting on pricing option contracts, [8, 29], the second of them focusing on risk assessment in the underlying stock exchange market based on the volatility, [41], of the returns of the different financial assets traded in the market.

Both problems are addressed both individually and also in conjunction given the strong connection between the latter and the former: most option pricing approaches, [8, 4, 29, 38, 10], use as an input parameter the volatility modeling the behavior of the underlying asset returns in order to expect wider or narrower changes in the future behavior of the underlying asset returns leading to higher or lower option prices.

The main goal driving the approach to the aforementioned computational finance problems are the achievement of high-throughput, low-latency stream processing solutions enabling the production of a stream of priced options according to one or more input streams modeling the behavior of the underlying stock market and optionally the requirements of the option contracts to be priced in terms of strike and expiration time. For this reason, an experimental stream processing engine has been developed in the context of this Thesis in order to test in terms of throughput latency and correctness the different single-threaded or multi-threaded solutions proposed in the scope of this Thesis to add the stream processing engine options pricing and or volatility aggregation functionality.

1.3 Structure

This Thesis is divided into three main parts, the first part, to which Chapters 1-5 belong, introduces all the necessary background to understand the research effort performed in the context of this thesis, the second part, to which Chapters 6-12 belong, introduces the research ideas produced in the context of the Thesis, and the third part, to which Chapters 13-16 belong, reports the achieved experimental results and discusses the main contributions.

2

Shared-Memory Parallelism and Lock-Free Synchronization

As anticipated in Section 1.3, this is the first one of the three chapters introducing all the necessary background to understand the research efforts performed in the context of this Thesis.

This chapter focuses on the scope of shared-memory parallelism and lock-free synchronization introducing the technical background needed to understand how to efficiently profit from the computational power offered by the multi-core hardware architectures available nowadays.

Section 2.1 below introduces the concept of shared-memory parallelism and elaborates on how the C programming language supports the production of shared-memory parallel solutions given the fact that it is the language of choice for the experiments performed in the context of this Thesis for the reasons discussed in that section. Section 2.2 briefly analyzes the C++ underlying memory model and its emulation in C in order to support the atomic operations needed to achieve lock-free synchronization. Section 2.3 elaborates on the memory management problem when multiple threads or processes concurrently access shared-memory. Finally, Section 2.4 briefly analyzes the impact of the cache architecture in performance outlining how to design solutions which make an efficient use of caches.

2.1 Shared-memory parallelism in C and pthreads

As explained in [25], shared-memory parallelism focuses on the efficient utilization of shared-memory multiprocessor architectures parallelizing tasks among multiple processors in order to benefit from an increase in computational power. Given the increasing computational power needs financial streaming applications have, as discussed in Section 1.1, shared-memory parallelism is a key background component to understand the different solutions proposed in the next chapters of this Thesis.

Paraphrasing [25], most real-world computational problems cannot be effectively parallelized without incurring the costs of inter-processor communication and coordination. And the computational problems approached in this Thesis are not an exception as it will be seen in detail in further chapters. For this reason, the choice of the tools used to implement the different solutions introduced in this Thesis has been

strongly determined by the way the aforementioned inter-processor communication and coordination costs can be minimized.

The last observation motivates the election of the C programming language in order to produce all the programs introduced in Chapter 5, which describes the experimental framework designed and developed in the context of this Thesis in order the face the challenges outlined in Section 1.2. Widely used higher level languages such as Java or C# would have simplified the development of the different solutions introduced in the next chapters by automatizing inter-processor communication and coordination tasks such as garbage collection at the costs of using a general purpose garbage collector not specifically optimized for the parallelization needs of the specific computational tasks parallelized in the context of this Thesis. In contrast, C and C++ allow for a finer grained memory management allowing the development of efficient memory management mechanisms tailored for the specific inter-processor communication and coordination needs the implemented solutions may have. In addition to this, C and C++ enable a fine grained control of the hardware resources by enabling for example the association of different threads to specific processing units or the possibility of executing atomic operations supported by the hardware architectures which enables the development of efficient lock-free solutions. In addition to this, C++ offers more development tools than C such as the support to develop object oriented code or the built-in memory model introduced in Section 2.2. However, given the existence of the GCC atomic built-ins which emulate this underlying memory model in C, and the fact that none of the solutions proposed in the current Thesis explicitly benefits from the object oriented paradigm, the C programming language was finally chosen for the development of all the solutions here reported. The following sections deepen in the scope of shared-memory parallelism and how it integrates with the C programming language.

2.1.1 Processes, threads, and Pthreads

Processes and threads are the main abstractions provided by modern operating systems to support the parallel or pseudo-parallel execution of different task in the underlying machine.

In [44], a process is described as an instance of an executing program, including the current values of the program counter, registers, and variables. And a thread is defined as a light-weight process which shares with other threads the same address space associated to the parent process as opposed to processes, each of which have its own address space which they do not share with other processes. This difference between processes and threads make threads the right choice to implement shared-memory parallel solutions.

As introduced in [28], Pthreads are defined as a set of C language programming types and procedure calls implementing the standardized, hardware-independent, thread programming interface specified by the IEEE POSIX 1003.1c standard (1995).

Henceforth, the word thread will refer to a Pthread when it is used in an implementation related context and to the abstract definition of thread introduced in the previous paragraph when discussing solutions independently of the implementations details.

2.1.2 OpenMP

As described in [37], OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran, on most platforms, processor architectures and operating systems, including Solaris, AIX, HP-UX, Linux, OS X, and Windows. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior which uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the supercomputer.

In early iterations of the experiments performed in the scope of this Thesis, this framework has been used to price options contracts in parallel as it is done in [29] which inspires the solutions presented in Chapter 7. However, given the finer grained control achievable when using Pthreads in C instead of the OpenMP framework, most of the proposed solutions in This thesis are based on the latter.

2.1.3 Thread safety

As explained in [25] the concurrent correctness problem in terms of safety properties, is much harder than the sequential version due to the vast number of ways that the steps of concurrent threads can be interleaved.

When many threads concurrently access a shared memory region, it is essential for them to synchronize in order to guarantee the correct behavior of the concurrent algorithm no matter how they interleave in each possible execution.

2.1.3.1 Lock-based synchronization

Lock-based synchronization guarantees thread safety properties through mutual exclusion achieved by the usage of semaphores, mutual exclusion locks or condition variables supported by most of the modern operating systems if not all of them [44].

Lock-based synchronization solutions need to be carefully addressed in order to avoid deadlocks, which occur when each thread belonging to a set of threads is waiting for an event that only another thread in the set can cause [44], as well as livelocks and starvation, which can eventually prevent the set of concurrent threads to fulfill their purpose.

In addition to the aforementioned threats, lock-based synchronization is expensive in terms of operating system overhead due to the changes of state required for all the participating threads when interacting with the different lock-based synchronization data-structures.

2.1.3.2 Lock-free synchronization

As described in [22], lock-free synchronization is a relaxed form of wait-free synchronization. Wait-free synchronization ensures that any operation on a shared memory object can complete in bounded number of steps, independently of other contending threads, whereas lock-free synchronization ensures that at least one of the contending operations makes progress in a finite number of its own steps.

Lock-free synchronization is usually achieved through the usage of atomic synchronization primitives reducing the operating system overhead discussed in the previous section when discussing lock-based synchronization.

2.1.4 Thread safe pseudo-random numbers generation in C

A very good example of a non-thread-safe mechanism is the default pseudo-random number generator in C. If two different threads execute the default `rand()` function defined in the `stdlib.h` header, both of them will access the same space of memory, the global random generator state, without synchronizing to do so leading to an unexpected behavior of the concurrent threads making it necessary for the threads to synchronize to avoid concurrently executing the `rand()` function.

As an alternative to this pseudo-random number generator, the `drand48_r` family of re-entrant pseudo-random number generators, also defined in the `stdlib.h` header, can be used in a thread-safe manner because instead of relying on a global random generator state, the `drand48_r` pseudo-random generators update a buffer provided and managed by the calling threads, which avoids the aforementioned risk of having two threads concurrently updating the global random generator state without synchronizing to do so.

2.2 The C++11 Memory Model and its emulation in C

As explained in [45], the C++11 memory model supports atomic operations able to create inter-thread ordering constraints according to the different memory orders of varying strength: relaxed, consume, acquire, release, acquire-release, and sequential consistency.

This underlying memory model helps producing fine grained lock-free solutions, but it is not supported by default as part of neither the C99 standard nor the C11 standard. The way of emulating this memory model in C involved the careful

usage of memory barriers or fences in conjunction with the blind to memory orders atomic primitives. However, the latest versions of the GCC compiler support a set of built in functions for memory model aware atomic operations which emulate the aforementioned memory model making it possible to develop in C fine grained lock-free solutions in a very similar manner as it is done in C++.

2.3 Memory Management

One of the most critical task related to shared-memory parallelism is memory management. In single-threaded programming, as soon as the thread making use of a given portion of memory considers it does not need to access it anymore it is safe to free that portion of memory in order to, for example, be able to allocate and use it again for a different purpose. However, if multiple threads concurrently access a given portion of memory, and one of them considers it does not need to access it anymore, it is not safe to free that portion of memory as done in the single-threaded case because other concurrent threads may still need to access that portion of memory. For this reason, it is necessary to keep track of whether if any given address can be safely freed or not in order to avoid situations in which a thread may access an already freed memory address leading to the failure of the program.

2.3.1 Reference Counting

Reference counting is probably the most widely spread, intuitive, and general purpose memory management technique. As described in [15], it consists on keeping track, for each allocated memory address, of the number of variables having a pointer pointing to that address so that when the counter reaches zero the address can safely be freed.

Even though it may seem a straight forward and safe solution, several threats need to be taken into account in order to make it properly work. For example, in case a cyclic linked list is not referenced by any other variable apart from the nodes in the list itself, the list is safe to be freed, but all the addresses associated to the nodes have a reference counter greater than zero due to the previous node in the list holding a reference to it, preventing the reference counting mechanism from freeing the memory occupied by the list. For this reason, reference counting us usually extended by additional complementary memory management techniques to identify and prevent threats such as the aforementioned one.

2.3.2 Hazard Pointers

Hazard pointers based memory management, [34], is an efficient memory management technique specifically designed to manage the memory occupied by the different nodes or blocks underlying to concurrent data-structures.

This technique consists on keeping track of a small set of memory addresses flagged as hazardous by each of the concurrent threads utilizing the concurrent data-structure, and adding the memory addresses which the different threads consider safe to free to temporary lists which trigger, when they reach a certain number of elements, a scan routine in which all the nodes in the list which are not flagged as hazardous by other threads are safely freed.

This technique tends to be less expensive in terms of execution time and memory than reference-counting based techniques, however, it has to be very carefully used in order to guarantee according to the semantics of the concurrent data-structures making use of this memory management technique that addressed not flagged as hazardous are actually not hazardous when executing the aforementioned scan routine.

2.4 The impact of caches in performance

As explained in [11], given the performance gap between processing units and random access memory, cache hierarchies play a fundamental role in the performance of any computer architecture.

An efficient cache hierarchy efficiently managed by the operating system can considerably reduce the overhead of retrieving bytes from the random access memory making the memory system virtually behave almost as fast as the processing units making use of it. For this reason, it is important to avoid whenever it is possible cache misses in order to minimize the overhead of accessing the actual random access memory hardware.

2.4.1 Cache misses and contention

Resource contention is a conflict over access to a shared resource such as random access memory, disk storage, cache memory, internal busses or external network devices aroused when more than one thread compete for the usage of the shared resource.

Concurrent solutions making excessive use of synchronization variables which are constantly checked and updated by multiple different threads may lead to a higher cache contention, leading to the invalidation of entries in the different caches accessed by the different threads increasing the number of cache misses and consequently reducing the performance of the solution. For this reason, it is important to make a conscious effort towards the minimization of the aforementioned contention in order to produce efficient concurrent solutions which make a good use of the cache memory hierarchy.

2.4.2 Cache misses and memory alignment

The different levels of cache memory hierarchies are usually divided in blocks or cache lines of different size containing the cached parts of the underlying random access memory. In case a variable is mapped to two or more blocks, every time at least one of the blocks is replaced, all the cached memory corresponding to that variable is invalidated leading to more cache misses when trying to access that variable.

One way to minimize the aforementioned problem is to align the different data-structures to the smallest cache line sizes, which are the ones belonging to the L1 cache, by adding padding bytes. This would avoid situations in which a variable smaller than one block is mapped to two due to memory miss-alignment, resulting in a more efficient use of the cache hierarchy.

3
Data-Streaming

As anticipated in Section 1.3, this is the second one of the three chapters introducing all the necessary background to understand the research efforts performed in the context of this Thesis.

This chapter focuses on the data-streaming model introducing the technical background needed to understand how to efficiently produce relevant and useful results on-the-go when a stream of data such as the succession of trades registered in a stock exchange market needs to be processed in real time.

Henceforth the data-streaming-model will be understood as described in [22]: in the data-streaming model, input tuples coming from one or multiple input streams are consumed by *Continuous Queries* (or simply queries in the following), which subsequently produce one or more output streams. A query, defined as a directed acyclic graph (DAG) with additional input and output edges, produces results "continuously" while consuming input tuples. Vertexes represent operators that consume tuples (from at least one input stream) and produce output tuples (for at least one output stream). Edges define how input and output tuples flow among the operators of a query.

In order to let the tuples flow from one operator to another, concurrent data-structures need to be used allowing for one or multiple concurrent readers and writers to add or retrieve tuples. Sections 3.1 and 3.2 below introduce the two main concurrent data-structures which are used for this purpose in the scope of this thesis, namely concurrent queues, and *ScaleGate*.

Another common need in the scope of stream-processing is to process tuples on a sliding-window basis taking into account for the production of aggregated results subsets of the entire dataset represented by the stream of tuples. Section 3.3 elaborates on this introducing two alternative sliding-window models present in the literature.

3.1 Concurrent queues

In [25], a concurrent queue is described as a concurrent container that provides *first-in-first-out* (FIFO) fairness. In other words, a concurrent data-structure providing the `enq` and `deq` methods to respectively add elements to the end of the

queue and remove and return elements from the other end of the queue, commonly referred to as *head*:

- `enq(x)`: the enqueue method, henceforth referred to as `enq` in all the listings it is used, let the calling writer thread add an element, `x`, to the container which will be removed from the queue and returned to one and only one of the reader threads only after all the elements already present in the container have already been removed and returned to other readers.
- `deq()`: the dequeue method, henceforth referred to as `deq` in all the listings it is used, removes in case there are one more elements available in the container, the element which have stayed in the container the longest and returns it to the calling reader thread.

Given the aforementioned semantics, concurrent queues are widely used in the scope of data streaming as they allow a stream of tuples processed by a single writer to traverse the queue in the same order as they are served.

In order for concurrent queues to achieve strong safety and liveliness requirements, including linearizability [26], and lock-freedom [5] in case a lock-free implementation is used, the concurrent queue writers and readers need to carefully synchronize either on a lock-based manner relying on blocking solutions such as spin-locks or on a lock-free manner carefully using atomic construct to guarantee the proper behavior of the queues. This has motivated the research for the creation of efficient lock-free queues [35, 16].

In order to better understand the impact of the different queue implementations available in the state of the art literature on the performance of the different stream processing solutions proposed in the context of this thesis, three of the queue implementations from the NOBLE[1] library [42, 43] have been used for all the experiments reported in Chapter 13:

- **Standard spin-lock based queue (LB)**: the standard spin-lock based queue is referred in [42] as LB (lock-based).
- **Lock-free block structure bounded memory queue (LF_BB)**: the lock-free block structure bounded memory queue introduced in [16] referred in [42] as LF_BB (lock-free block-bounded).
- **Lock-free dynamic structure bounded memory queue (LF_DB)**: the lock-free dynamic structure bounded memory queue introduced in [35] referred in [42] as LF_DB (lock-free dynamic-bounded).

3.2 *ScaleGate*

As described in [22], *ScaleGate* is a recently proposed abstract data type tailored for the parallelization needs that arise in many stream processing problems.

[1]NOBLE Professional Edition is the lock-free concurrent data-structures library in C and C++ commercialized by Parallel Scalable Solutions AB, `http://www.non-blocking.com/`

In order to properly define the semantics of the *ScaleGate* data-structure it is necessary to understand the concept of *ready* tuples introduced in [6] as follows:

***Ready tuple**: Let t_i^j be the i-th tuple in a timestamp-sorted physical stream j. Tuple t_i^j is ready to be processed if $t_i^j.ts \leq merge_{ts}$; where $merge_{ts}$ is the minimum among the latest timestamps from each timestamp-sorted physical stream j, i.e. $merge_{ts} = \min_j \left\{ \max_i \left(t_i^j.ts \right) \right\}$.*

As it is proven in [6], if a set of concurrent writers add tuples in non-descending timestamp order to a concurrent data-structure in which tuples are sorted in non-descending timestamp order, consuming these tuples only once they are ready according to the definition above guarantees that the resulting output stream of tuples is properly sorted in non-descending timestamp order.

The last observation is key to understand the semantics of the *ScaleGate* data-structure, which similarly to the concurrent queue data type described in the previous section, it is a concurrent data-structure in which multiple concurrent writer and reader threads can add or retrieve tuples respectively using the `addTuple` and `getNextReadyTuple` methods whose behavior is different from the formerly introduced `enq` and `deq` methods as described in [22]:

- **`addTuple(timestamp,tuple,sourceID)`**: the `addTuple` method allows a tuple from the source thread, uniquely identified by the `sourceID` identifier, to be merged by *ScaleGate* in the resulting timestamp-sorted stream of ready tuples.
- **`getNextReadyTuple(readerID)`**: the `getNextReadyTuple` method provides to the calling reader thread, uniquely identified by the `readerID` identifier, the next earliest ready tuple that has not been yet consumed by the former.

Altogether, the differences between the *ScaleGate* data-structure and the concurrent queue data-structure introduced in the previous chapters can be understood when analyzing what happens if a set of multiple writer threads concurrently add tuples in non-descending timestamp order to each of the data-structures a set of multiple reader threads concurrently retrieve tuples from the data-structures.

In the case of the formerly introduced concurrent queue data type, each of the tuples added by the set of concurrent writer threads would be retrieved by one and only one of the concurrent reader threads. In case two or more tuples added by the same writer thread are retrieved by a single reader thread, these tuples would be retrieved in the same order in which they were added to the queue and consequently, properly sorted in non-descending timestamp order. However, if two tuples retrieved by a single reader thread were added to the queue by two different writer threads, there is no guarantee whether or not they may be ordered in non-descending timestamp order.

In the case of the newly introduced *ScaleGate* data-structure, each of the tuples added by the set of concurrent writer threads would be retrieved by all of the concurrent reader threads resulting in all of the reader threads consuming exactly the same stream of tuples instead of a subset of it each as it happened with the queue. All the tuples belonging to the stream of tuples consumed by all the reader threads are guaranteed to be ordered in non-descending timestamp order.

In order to benefit from the aforementioned semantics of the *ScaleGate* data-structure in the experiments performed in the context of this Thesis, the *ScaleGate* data-type has been implemented in C together with an adaptation of the Hazard Pointers based memory management mechanism introduced in Section 2.3.2.

3.3 Sliding-Windows

Given the unbounded and dynamic nature of the data processed in the scope of data streaming, one common challenge is to process subsets of the streams of tuples on a sliding-window basis for example to produce aggregate results [6].

3.3.1 The Window Size Only Sliding-Windows Model

In the window size only sliding-windows model, operators are computed over sliding-windows which are defined by its window size parameter, henceforth referred to as WS. Sliding windows can be time-based, (e.g., to group tuples received no earlier than 5 minutes before the reception of the newest tuple in the window), or tuple-based, (e.g., to group the last 10 received tuples).

As it can be understood, this sliding-windows model involves keeping track always of the most recently received tuples discarding older tuples from the sliding windows when newer tuples are received.

3.3.2 The Window Size and Window Advance Sliding-Windows Model

In the window size and window advance sliding-windows model, as described in [6], operators are computed over sliding-windows which are defined by the parameters size, henceforth referred to as WS, and advance, henceforth referred to as WA. Sliding windows can be time-based, (e.g., to group tuples received during periods of 5 minutes every 2 minutes) or tuple-based (e.g., to group the last 10 received tuples every 3 incoming tuples).

As it can be understood, this sliding-windows model involves keeping track at a time of more than one sliding window as each tuple can contribute to one or more overlapping sliding windows. Section 10.4 further elaborates on the definition of this sliding-window model which is used by the operator introduced in that section.

4

Finance

As anticipated in Section 1.3, this is the third and last one of the three chapters introducing all the necessary background to understand the research efforts performed in the context of this Thesis.

This chapter focuses on the context of finance describing the computational finance problems detected in the literature which are approached in this Thesis in the scope of data streaming optimizing the utilization of the available hardware resources through shared-memory parallelism.

4.1 Relevant Financial Problems for this Thesis

After a careful study of the state of the art literature in the scope of computational finance with a special focus on [14, 17, 29, 1, 18, 39, 10, 9, 3, 38, 30, 12, 2], further reviewed in Chapter 14, two main financial problems were identified as relevant for the goals of this Thesis, namely the options pricing problem and the risk assessment problem which is approached in the context of this Thesis as the assessment of the volatility of the underlying traded assets prices over time.

4.1.1 Options Pricing

As described in [8], an option is a security that gives its owner the right to trade in a fixed number of shares of a specified common stock at a fixed price at any time on or before a given date. The act of making this transaction is referred to as exercising the option. The fixed price is termed the strike price, and the given date, the expiration or maturity date. A call option gives the right to buy the shares, and a put option gives the right to sell the shares.

The most commonly traded options are American and European options, sometimes referred to as plain vanilla options:

- **American Options**: American options can be exercised at any time between the date of purchase and the expiration or maturity date.
- **European Options**: European options are different from American options in that they can only be exercised at the end of their lives.

In addition to the aforementioned option types, non-standard options, commonly referred to as exotic options, are also traded in different markets. These last kind of options are either variations on the payoff profiles of the plain vanilla options or are wholly different products based on options.

Given the stochastic nature of the underlying assets the problem of pricing different types of options becomes a complex and expensive estimation problem based on different input parameters according to the underlying market information available, and the specified strike price and maturity. In addition to this, the increasing volume of the financial data streams modeling the behavior of the underlying stock markets and the expensive data center energy consumption and ownership cost make it paramount in the scope of computational finance to research towards efficient options pricing solutions which can process financial tuples achieving a high throughput and a low latency in the most energy efficient way as possible.

According to [17, 10], two main models are used nowadays to approach the options pricing problems, Monte Carlo models, briefly introduced in Section 4.1.1.1, and binomial or grid-based models, briefly introduced in Section 4.1.1.2. In addition to the aforementioned two models the Black-Scholes formula [4], briefly introduced in Section 4.1.1.3, gives a theoretical estimate of the price of European options.

4.1.1.1 Monte Carlo Models

As described in [3], Monte Carlo models are computational algorithms which rely on repeated random sampling to obtain numerical results. In order to do so, the following steps are usually followed:

- **Definition of a domain of possible inputs**: in the scope of options pricing this domain would be most of the times the set of possible values that the underlying asset can have at any point of time before the expiration date, or only at the expiration date.
- **Random generation of inputs from a probability distribution over the aforementioned domain**: in the scope of options pricing this probability distribution would deem more probable prices closer to the observed price of the asset at the moment of pricing.
- **Performance of a deterministic computation on the inputs**: in the scope of options pricing this deterministic computation would consist most of the times on discounting the payoffs of the option at the given input points to the instant of time in which the option is priced based on an assumed risk-less interest rate.
- **Aggregation of the results**: in the scope of options pricing according to the previous steps this deterministic aggregation would consist most of the times on averaging the discounted payoffs weighted by the aforementioned probability distribution.

As it can be easily seen, and as highlighted also in [3], Monte Carlo models are especially suitable for GPU based solutions given the embarrassingly parallel nature of the second and third aforementioned steps.

4.1.1.2 Binomial Models

As described in [8, 29], binomial model based options pricing algorithms rely on underlying binomial tree models representing all the different paths that the underlying asset price could follow if time were discretized from the option pricing instant to the expiration date, and in each discrete step the underlying stock price could go either up or down. In order to price options based on this, the following steps are usually followed:

- **Iterating to model the underlying binomial tree**: in this first step, the underlying binomial tree model is traversed from the options pricing instant to the option expiration date calculating the expected price of the underlying asset and the corresponding payoff of the option contracts in the scenario represented by each node in case of need.
- **Backtracking to obtain the desired option price**: in this second step, the underlying binomial tree model built in the previous step is iterated backwards implicitly aggregating the expected payoff of the underlying asset at the option pricing instant.

It is worth noticing that in order to bound the number of nodes in the different levels of the underlying binomial tree, most binomial three based options pricing algorithms force the underlying trees to be recombinant so that the number of nodes per level is proportional to the level instead of two raised to the level.

4.1.1.3 Black-Scholes

As anticipated earlier in this chapter, the Black-Scholes formula [4] gives a theoretical estimate of the price of European call (C) and put (P) options as expressed in Equation 4.1:

$$\begin{aligned} C(S,t) =& N(d_1)S - N(d_2)Ke^{-r(T-t)} \\ P(S,t) =& Ke^{-r(T-t)} - S + C(S,t) \\ =& N(-d_2)Ke^{-r(T-t)} - N(-d_1)S \\ d_1 =& \frac{1}{\sigma\sqrt{T-t}}\left[\ln\left(\frac{S}{k}\right) + \left(r + \frac{\sigma^2}{2}\right)(T-t)\right] \\ d_2 =& d_1 - \sigma\sqrt{T-t} \end{aligned} \tag{4.1}$$

- $N(.)$: cumulative distribution function if the standard normal distribution.
- $T-t$: time to maturity.
- S: spot price of the underlying asset.
- K: strike price.
- r: assumed continuous risk-less interest rate.
- σ: the volatility of returns of the underlying asset.

4.1.2 Volatility

As anticipated earlier in this chapter, the second key computational finance problem identified as relevant for the goals of this Thesis is the risk assessment problem which is approached in the context of this Thesis as the assessment of the volatility of the underlying traded assets prices over time because it is the way risk is assessed in order to price options based on the models introduced in the previous section.

As described in [41], the standard deviation, henceforth referred to as σ, is a measure which is used to quantify the amount of variation or dispersion of a set of data values or a random variable, X.

Given the discrete and finite nature of the sets of stock prices which whose volatility is expected to be priced in the scope of this Thesis, the formal standard deviation definition for a bounded discrete random variable $X = \{X_1, X_2, ..., X_N\}$ is the one formally introduced in this section. However, before directly introducing the standard deviation formula for a bounded discrete random variable, it is worth understanding the definition of the mean or first moment, $\mathrm{E}[X]$ or μ, as described in Equation 4.2.

$$\mathrm{E}\left[X\right] = \frac{1}{N}\sum_{i=1}^{N} X_i = \mu \tag{4.2}$$

As it can be seen, $\mathrm{E}[X] = \mu$ is the well-known mean or average of the discrete random variable. Elaborating from this definition towards the definition of the standard deviation, σ, a first measure of how fast a discrete random variable spreads out is its variance, $\mathrm{Var}(X)$, which is the mean of a new discrete random variable representing, for each value in the original random variable, the difference from the original value to the mean of the original random variable raised to the square, so that the sign of the difference does not allow positive and negative deviations to compensate each other. This is formally expressed in Equation 4.3.

$$\mathrm{Var}\left(X\right) = \mathrm{E}\left[\left(X-\mu\right)^2\right] = \frac{1}{N}\sum_{i=1}^{N}\left(X_i-\mu\right)^2 \tag{4.3}$$

The small problem with variance for its usage as a volatility measure is that given the square operation in Equation 4.3, it is not expressed in the same units as the original random variable. In other words, if the random variable units are dollars, \$, the variance is expressed in dollars raised to the square, $\$^2$. This motivates the introduction of the standard deviation, σ, as a volatility measure, which is simply the square root of the variance [41], as expressed in Equation 4.4, which takes it back to the same units as the original random variable.

$$\sigma = \sqrt[2]{\mathrm{Var}\left(X\right)} = \sqrt[2]{\mathrm{E}\left[\left(X-\mu\right)^2\right]} = \sqrt[2]{\frac{1}{N}\sum_{i=1}^{N}\left(X_i-\mu\right)^2} \tag{4.4}$$

5

Framework

In this chapter the experimental framework[1] designed and developed in order to quantitatively assess the quality of the different operators defined for the different financial stream processing engine iterations described in detail in Chapters 6-12 is introduced.

Section 5.1 below introduces the data sources that have been used to generate the streams of tuples used in the different experiments reported and analyzed in Chapter 13. Section 5.2 briefly describes the financial stream processor developed in the context of this Thesis whose internal structure and the behavior of its operators is described in detail in Chapters 6-12. Section 5.3 introduces all the auxiliary programs which complete the framework enabling the production of the input data files to be used by the aforementioned program to model streams of tuples, the analysis of the output generated by the stream processing engine in order to obtain the throughput and latency metrics which are reported and analyzed in Chapter 13, and the generation of the different scripts to launch and keep track of all the experiments performed in the different test machines. Finally, Section 5.4 elaborates on the physical test environment introducing the programming language used to implement the aforementioned programs and the machines in which the experiments reported and analyzed in Chapter 13 have been performed.

5.1 Data Sources

In order to simulate a financial stream of data representing the realistic behavior of a reasonably scaled financial market, the data contained in the sample *Daily Trades File*[2] reporting all the trades registered in the NYSE and NASDAQ markets the 5th of August of 2015 with μs timestamps precision provided by the NYSE Market Data[3] reporting authority, whose structure is described in detail in Chapter 5 in [36], have been used to produce the two different kinds of input data files consumed by the financial stream processing engine described in Section 5.2 to price option contracts

[1] All the programs and tools introduced in this chapter are maintained in the private Git repository `https://bitbucket.org/ioaniko/alfonso_thesis_work`. Please contact its administrator, Ioannis Nikolakopoulos <ioaniko@chalmers.se> in order to be granted access in case of need.

[2] Sample *Daily Trades File* available for free in the official NYSE ftp repository `ftp://ftp.nyxdata.com/HistoricalDataSamples/DailyTAQ/` under the name EQY_US_ALL_TRADE_20150805.zip.

[3] The NYSE Market Data reporting authority (`http://www.nyxdata.com/`) keeps track of all the financial information registered in the NYSE and NASDAQ stock exchanges.

based on the behavior of the underlying financial market and the requirements for each specific option contract.

Section 5.1.1 below describes the financial stream of data resulting from the first of the two kinds of input data files consumed by the financial stream processing engine, and Section 5.1.2 below describes the options settings stream of data resulting from the second of the two kinds of input data files consumed by the financial stream processing engine since the last iteration introduced in Chapter 12.

5.1.1 Financial Stream

The financial stream models the information contained in the aforementioned *Daily Trades File* as a stream of financial tuples representing the behavior of the underlying stock markets.

This is done by modeling each of the selected entries in the *Daily Trades File* as a financial tuple representing a trade transaction in which a given number of shares of a given stock are traded at a given price per share at a given instant of time. Sections 6.2 and 12.2 describe in detail the structure and internal representation of the financial tuples belonging to the financial stream specifying the mapping between the different fields of the entries in the *Daily Trades File* and the different fields in each financial tuple consumed by the stream processing engine.

5.1.2 Options Settings Stream

The options settings stream models the specifications of the options contracts to be priced by the options pricing stream processing engine in accordance to the most up to date financial data from the aforementioned stream processed by the stream processing engine.

Each tuple belonging to this second stream represents an option contract to be priced for a given underlying stock, at a given time, with a given strike and maturity assuming a given risk-less interest rate.

This is achieved by creating for each of the tuples in the financial stream an options settings tuple with the same physical and logical timestamps as the corresponding financial tuple in order to produce two input datasets which can be fed synchronously by two different input threads simulating two simultaneous streams of tuples in real time, a randomly chosen underlying symbol among the traded symbols represented in the financial stream, a randomly chosen options strike in the order of magnitude of the trade prices seen for the chosen symbol in the financial stream, and a randomly chosen maturity and assumed risk-less interest rate. Section 12.2 describes in detail the structure and internal representation of the options settings tuples belonging to the options settings stream.

5.2 Main Program: The Options Pricing Financial Stream Processing Engine

The main program developed in the context of this Thesis is the options pricing financial stream processing engine which addresses the two main financial problems identified in Section 4.1, namely options pricing and volatility aggregation, in the scope of data streaming, processing the two aforementioned streams of tuples producing as a result, an output stream of tuples representing the priced option contracts according to the observed behavior of the underlying stock exchange market.

The structure of this stream processing engine is adapted to the directed-acyclic-graph model introduced in Chapter 3, the nodes and edges in the graph corresponding respectively to threads executing different operators and the different concurrent data-structures used to transfer tuples from one thread to another.

Two special types of threads are present in all the iterations of the stream processing engine in order to model the input data streams and to output the processed tuples:

- **Input threads**: the input threads retrieve tuples from the binary files representing the two data streams introduced in Section 5.1 generated by the input generator program introduced in Section 5.3.2 below. Consequently, these threads do not retrieve tuples from a concurrent data-structure as the rest of the threads in the stream processing engine do. In order to properly model the behavior of an input stream of data adding tuples to the concurrent data-structures from which the first process threads retrieve tuples, the input threads can control the input rate in two different manners:
  - **Full speed rate**: serving tuples at full speed rate consists on letting the input threads serve the financial tuples read from the binary input files one by one as fast as they can iterate through the input binary file. As it will be further discussed in Section 13.1, this mode of operation leads to a saturation situation in the stream processing engine which allows for the assessment of the maximum achievable throughput but inflates the reported latency metrics due to the time spent by the tuples in the data-structures until they can be retrieved and processed.
  - **Constant rate**: serving tuples at a constant rate consists on controlling the speed at which the input threads serve the tuples read from the input binary files in order to avoid the aforementioned saturation situation.

  In addition to this, the input threads add each of the served tuples a process start timestamp, as described in detail in Section 6.2, in order to measure the time elapsed for each individual tuple from the instant of time it is added to the corresponding concurrent data-structure by the corresponding input thread to the instant of time it is finally retrieved by the output thread described below without letting these measurements affect the performance of the operators which process the tuple in the meantime.
- **Output thread**: the output thread corresponds always to the final node

of the underlying directed acyclic graph. Instead of adding tuples to the next concurrent data-structure as the rest of the threads do, it outputs the tuples it retrieves to the corresponding output binary file which keeps track of how the financial stream processing stream process each individual tuple in order to assess afterwards the correctness of the assigned volatility values and options prices and the performance of the stream processing engine in terms of throughput and latency. Before doing so, it adds each of the retrieved tuples a process end timestamp, as described in detail in Section 6.2, for the same reason why the input threads described above added a process start timestamp.

The design, development, and evaluation of the options pricing financial stream processing engine has been approached in an iterative manner as documented in detail in Chapters 6-12, each of which covers an iteration of the stream processing engine in which new functionality is added or the existing functionality is optimized towards a higher throughput or a lower latency.

Chapter 6 introduces the first iteration of the stream processing engine which consumes tuples only from the aforementioned financial stream and uses only one process thread to price option contracts always with the same fixed option strike, option maturity, assumed risk-less interest rate and assumed volatility, and the symbol and current trade price retrieved from the processed financial tuples.

Chapters 7-9 evolve the options pricing engine described above by letting it price in parallel several option contracts at a time with the same settings as in the first iteration employing to do so $n \geq 1$ option pricing threads in parallel, yet preserving the linearizability requirements for the ordering of the tuples output by the output thread. In other words, the output thread in the iterations introduced in Chapters 7-9 output the same stream of processed tuples in terms of ordering and assigned option prices as the one output in the first iteration of the stream processing engine.

Chapter 10 extends the functionality of the stream processing engine as defined in Chapter 9 to aggregate on a sliding-window basis the volatility of the underlying stock before pricing each option, and Chapter 11 parallelizes the sliding-window based volatility aggregator introduced in the previous iteration.

Finally, Chapter 12 extends the functionality of the stream processing engine as defined in Chapter 11 adding a second input thread to provide the options settings tuples from the options settings stream and extending the parallel sliding-window based volatility aggregation operator to let it also match the two streams of data so that options contracts can be priced according to the option strike, maturity and assumed risk-less interest rate specified in the options settings stream and the trade price and aggregated volatility assigned to the most recent tuple from the financial stream matching the options setting tuple given the specified underlying stock symbol. Chapter 13 reports the throughput and latency metrics obtained when totally or partially executing all the aforementioned iterations of the financial stream processing engine in the context of the framework introduced in this chapter.

5.3 Auxiliary Programs

As anticipated in the previous sections, a set of auxiliary programs, which are described in the following sections, complete the framework enabling the production of the input data files to be used by the aforementioned financial stream processing engine input threads to model the two streams of tuples introduced in Section 5.1, the analysis of the output generated by the stream processing engine in order to obtain the throughput and latency metrics which are reported and analyzed in Chapter 13, and the generation of the different scripts to launch and keep track of all the experiments performed in the different test machines.

5.3.1 Financial Dataset Analyzer

The financial dataset analyzer program represents the first building block in the framework described in this chapter. It analyzes the aforementioned *Daily Trades File* in order to identify the most traded symbols in the reported session discarding all the trades flagged as invalid.

The output of this program is used to determine the selection of target symbols provided to the input generator program introduced in the next section to generate the different binary input files modeling the streams of tuples introduced in Section 5.1.

Before concluding this section, it is worth highlighting some facts according to the output generated by this program when analyzing the NASDAQ session recorded the 5th of August 2015:

- Out of the 2972 symbols traded in the NASDAQ market that day. Only the ten most traded ones already accounted for more than 15% of the trades registered in the *Daily Trades File* and more than 50% of all the trades registered were associated only to the top 140 most traded symbols.
- The rate at which tuples were recorded in the *Daily Trades File* according to their physical timestamp followed a bathtub trend with peak rates close to respectively 60000 and 100000 tuples per minute at the opening and closing exchange times and an average rate between 10000 and 20000 tuples per minute during the rest of the trading hours.

5.3.2 Input Generator

The input generator program, as its name anticipates, is the program which produces the binary files modeling each of the tuples belonging to the two streams introduced in Section 5.1 which the stream processing engine consumes.

To do so, the input generator program needs to be provided with a *Daily Trades File* with the structure specified in Chapter 5 in [36], and a text file listing a selection of target traded symbols, with the same 16 ASCII characters format as specified in [36], to be taken into account for the generation of the streams. With this, only

the entries in the *Daily Trades File* whose symbol fields match one of the specified target symbols, which are not flagged as invalid, and which match the user specified filters, are used for the generation of the financial tuples in the stream described in Section 5.1.1.

The structure of the tuples belonging to both the binary files produced by this program contains all the fields successively introduced in Chapters 6-12, plus a set of padding bytes in order to align the tuples to the cache line length of the specific machine in which the program is executed in order to minimize the number of cache misses due to memory miss-alignment as discussed in Section 2.4.2.

All the tuples belonging to the financial stream according to the aforementioned filters applied to the *Daily Trades File* are assigned by the input generator, in addition to the physical timestamp retrieved from the *Daily Trades File*, a sequence number according to its physical timestamp order in the final financial stream of tuples, which represents a logical timestamp in the resulting output stream being the first tuple assigned the sequence number 0, the second one 1, and so on. For each of the financial tuples representing an entry in the *Daily Trades File*, an options settings tuple is generated as described in Section 5.1.2, with the same sequence number and physical timestamp. The main reason for this is to guarantee that when testing the stream processing engine at full speed rate the two streams of tuples are implicitly synchronized.

In addition to the production of the binary files modeling the two stream of tuples, the input generator program is able to produce if requested to do so, an ASCII version of the same files to facilitate further research on the streams of tuples with external analysis tools, and trade price and trade volume plots for each traded symbol belonging to the financial stream of tuples in order to provide a visual description of the produced financial stream of tuples and its behavior.

Before concluding this section, it is worth briefly summarizing the different datasets or pairs of streams produced making use of the input generator program based on the *Daily Trades File* recording the 5th of August 2015 session which have been used for different purposes in the context of this thesis:

- **The *micro* dataset**: the *micro* dataset was built using the last 10000 NASDAQ valid trade records registered the 5th of August 2015 between 11:40 and 13:50 by the 3 most traded symbols in the session by number of valid trade records, AAPL, FOXA, and CMCS A, according to the financial dataset analyzer program introduced in the previous section.
  The resulting input files have been used for debugging purposes as well as the fast production of preliminary results to expedite the implementation and assessment of new ideas during the development of the different iterations of the stream processing engine.
- **The *tiny* dataset**: the *tiny* dataset was built using the 190442 valid NASDAQ trade records registered the 5th of August 2015 between 11:40 and 13:50 by

the aforementioned 3 most traded symbols in the session by number of valid trade records.
The resulting input files have been used to proof check the preliminary results obtained with the *micro* dataset in case of need and to produce the full speed rate results to assess the maximum achievable throughput under a saturation situation achievable by the different iterations of the stream processing engine in order to control the input rate to avoid saturation in the production of the experimental results reported in Chapter 13.

- **The *main* dataset**: the *main* dataset was built using the 1705386 valid NASDAQ trade records registered the 5th of August 2015 between 09:25 and 16:05 by the 10 most traded symbols in the session by number of valid trade records, according to the financial dataset analyzer program introduced in the previous section.
  The resulting input files have been used to produce the final throughput and latency results reported and analyzed in Chapter 13.

5.3.3 Output Analyzer

The output analyzer program, as its name anticipates, is the program which taking as an input the binary output file generated by the options pricing stream processing engine each time it is executed, checks the correctness of the corresponding stream processing engine execution and calculates a series of performance metrics to quantitatively assess the quality of the operators and data-structures implemented and tested.

As anticipated in Section 5.1.1, each of the tuples retrieved by the output thread is stored in the aforementioned binary output file in the same order as it is retrieved by the aforementioned tread containing the process start and process end timestamps respectively assigned by the input and output threads as well as the physical and logical timestamps assigned by the input generator program and the values assigned by the different operators executed by the process threads.

With all the aforementioned information, the output analyzer program verifies for each execution if all the tuples from the input datasets have been correctly processed and if they have been output by the stream processing engine in the same order as the corresponding input thread served them. It also calculates the latency for each individual tuple as the difference between its process end timestamp and its process start timestamp and the overall achieved throughput as the difference between the last processed tuple process end timestamp and the first processed tuple process start timestamp divided by the total number of tuples in the binary file. In order to summarize the aforementioned latency metrics, it calculates the minimum, maximum, mean, median, first percentile, first quartile, third quartile and ninety ninth percentile of the latencies for all tuples and generates, using Gnuplot, latency plots representing the individual latency for each tuple and the aforementioned latency statistics.

Even though the output thread in the stream processing engine program introduced in Section 5.1.1 already have access to all the information needed to produce the aforementioned statistics, checks and plots, all this functionality has been implemented separately in the output analyzer program in order to minimize the impact of measuring the performance of the stream processing engine in performance of the stream processing engine itself.

5.3.4 Excel Master Index

In order to produce the experimental results which are reported and analyzed in Chapter 13, the stream processing engine as well as the output analyzer program described in the previous sections have been executed hundreds of times in the different test machines introduced in Section 5.4.2 having the carefully keep track of the specific settings used to execute the financial stream processing engine and the results produced by the output analyzer program.

In order to manage all of this, Excel has been used to produce a master index workbook to keep track of all the experiments performed in the context of this thesis. Apart from keeping track of all the settings configuring the different executions of the stream processing engine and the statistics and checks generated for each execution by the output analyzer program, this workbook automatized the generation of shell scripts to schedule and run both the stream processing engine and the output analyzer program and to import the output generated by the latter to the master index table. In addition to this, the production of preliminary throughput and latency plots for the different experiments reported in Chapter 13 has also been automatized making use of pivot tables and pivot charts, which made it easier to rapidly identify outliers and executions whose detailed latency plots needed to be revised to understand in detail the specific behavior of the different operators and data-structures used in the stream processing engine.

5.4 Test Environment

In order to produce all the experimental results reported in Chapter 13, all the iterations of the stream processing engine described in Chapters 6-12 have been carefully implemented in C and compiled with the GCC compiler as specified in Section 5.4.1 below. The resulting executables have been run in the two different Linux servers with shared-memory multi-processor architectures introduced in Section 5.4.2.

5.4.1 Language and Compiler: C and GCC

As discussed in Section 2.1, the C programming language is the one used in the context of this Thesis to implement the different iterations of the stream processing engine introduced in Chapters 6-12.

In particular, the C99 standard (ISO/IEC 9899:1999) has been followed with additions from the C11 standard (ISO/IEC 9899:2011) when needed.

The code in both machines has been compiled using the GCC compiler, version 4.9.2, which supports the atomic built-ins emulating the C++ memory model introduced in Section 2.2.

5.4.2 Machines

All the experiments reported in Chapter 13 have been produced executing the different iterations of the stream processing engine in the Linux servers belonging the department of Computer Science and Engineering at Chalmers cse-31228.cse.chalmers.se, henceforth referred to as 31228, and cse-hasgreen.cse.chalmers.se, henceforth referred to as Hasgreen.

Both machines are Linux servers with single-socket multiple-processors architectures, both of them with twice the number of virtual cores than physical CPUs through the use of hyper-treading. Sections 5.4.2.1, and 5.4.2.2 respectively report the main features of 31228 and Hasgreen in the context of the experiments performed in this Thesis.

5.4.2.1 31228: Intel Xeon

The 31228 machine is a Linux server running the CentOS Linux release 7.1.1503 distribution over the 3.10.0-229.20.1.el7.x86_64 Linux kernel.

It is powered by an Intel Xeon E5-2695 v3 chip with 14 physical-28 virtual processors at 2.30 GHz, with 35840KB of cache memory, 64B L1 cache line size, and μs clock precision.

5.4.2.2 Hasgreen: Intel Core i7

The Hasgreen machine is a Linux server running the Ubuntu Linux release 14.04.2 LTS over the 3.13.0-48-generic Linux kernel.

It is powered by an Intel Core i7-4770 chip with 4 physical-8 virtual processors at 3.40 GHz, with 8192KB of cache memory, 64B L1 cache line size, and μs clock precision.

6

Single-Threaded Binomial Options Pricing

As anticipated in Section 5.2, this is the first one of the seven chapters describing the financial stream processing engine that have been developed in the context of this Thesis.

In each of the Chapters 6 - 12, building on top of the previous chapter in Chapters 7 - 12, one iteration of the stream processing engine is covered introducing either more functionality to the stream processing engine or making the existing functionality more efficient. To do so, each chapter has four main sections: a first section to introduce the involved threads, a second one focusing on the structure of the tuples, another one introducing the data-structures used, and a final one focusing on the behavior of the operator. Every time new functionality is added, a detailed pseudocode of the new operator or operators can be found in the section dedicated to analyzing the behavior of the operators. Every time existing functionality is optimized, the section focusing on the behavior of the operators is extended analyzing the theoretically expected performance improvements in terms of throughput and latency.

In this chapter, the core functionality of the stream processing engine is introduced: the options pricing operator in its single-threaded version. The introduction of the behavior of the operator will be strongly based on the binomial options pricing code introduced by Shuo Li [29], based on the classical paper by Cox et al. [8]. Nevertheless, any operator implementing any of the models outlined in Section 4.1.1 can be used for the options pricing purpose without having to modify neither the structure of the involved threads, nor the structure of the tuples, or the data-structures used for synchronization purposes.

6.1 Involved Threads

As anticipated above, only one thread is dedicated to calculating option prices in this first iteration of the financial stream processing engine. In addition to this thread, as anticipated in Section 5.2, two threads are dedicated to retrieve tuples from the input dataset and serve them to the process thread, and to get the tuples output by the process thread and output them, in the case of the experiments, to an output binary file.

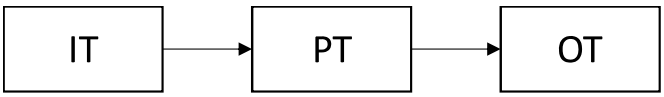


Figure 6.1: Involved threads

Figure 6.1 outlines how threads are arranged in this first version of the financial stream processing engine following the Directed Acyclic Graph (DAG) model introduced in Chapter 3. Each of the boxes in the diagram represents one thread and the arrows indicate the precedence of the threads in the stream processing engine:

- **Input Thread** (*IT*): the input thread, henceforth referred to as *IT*, is the thread in charge of retrieving tuples from the input stream and serving them to the process thread.
- **Process Thread** (*PT*): the process thread, henceforth referred to as *PT*, is the thread which applies the binomial options pricing operator to the generate option prices according to the financial information contained in the input tuples.
- **Output Thread** (*OT*): the output thread, henceforth referred to as *OT*, is the thread which reads the tuples output by the process thread and prints them to an output file to enable further analysis of the performance of the stream processing engine.

6.2 Structure of the Tuples

Input tuples are retrieved from the input financial stream introduced in Section 5.1.1 by *IT* and they are extended by each of the threads introduced in the previous section introducing all the needed additional fields.

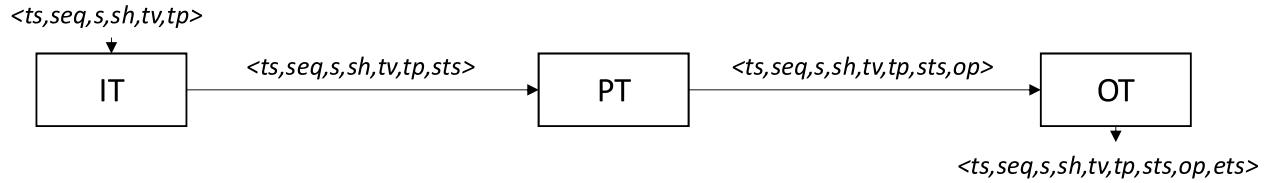


Figure 6.2: Structure of the tuples

Figure 6.2 extends the involved threads diagram introduced in Figure 6.1 specifying the structure of the tuples in each transition from one thread to another as well as the structure of the tuples read by *IT*, and the tuples output by *OT*.

Each tuple read by *IT* from the input financial stream, as described in Section 5.1.1, represents a trade transaction in which a given number of shares of a given stock are traded at a given price per share, at a given instant of time. To model this, the input financial tuples contain six fields, <*ts*, *seq*, *s*, *sh*, *tv*, *tp*>:

- **Timestamp** (*ts*): the timestamp field, henceforth referred to as *ts*, is the value representing the physical timestamp when the transaction took place in the monitored stock market. This fields is internally modeled as a `long` value indicating the number of elapsed microseconds from the trade market opening

hour minus a given threshold to the instant of time when the transaction took place. E.g. for NASDAQ trades, the number of elapsed microseconds from 09:25 a.m.

In the tuples used for the experiments documented in Chapter 13, this field is retrieved from the bytes $\{0, .., 11\}$ in the entries belonging to the *Daily Trades File* introduced in Section 5.1.1, whose structure is documented in Chapter 5 in the financial dataset specification [36].

- **Sequence number** (*seq*): the sequence number field, henceforth referred to as *seq*, is the value representing the unique sequence number assigned to each tuple in the input financial stream and behaves as a logical timestamp indicating the precedence order of the tuples in the stream. This field is internally modeled as a `long` value, 0 being the sequence number assigned to the first tuple in the stream, 1 being the sequence number associated to the second tuple in the stream, and so on.

  In the tuples used for the experiments documented in Chapter 13, this field is assigned to each tuple by the input generator program introduced in Section 5.3.2.
- **Symbol** (*s*): the symbol field, henceforth referred to as *s*, is the value representing the traded security symbol. E.g. AAPL for Apple Inc. or FOXA for Twenty-First Century Fox Inc. shares traded in the NASDAQ stock market. This field is internally modeled as an array with 16 ASCII `char` values, the first 6 characters representing the symbol root and the last 10 characters representing the symbol suffix.

  In the tuples used for the experiments documented in Chapter 13, this field is retrieved from the bytes $\{13, .., 28\}$ in the entries belonging to the aforementioned *Daily Trades File*.
- **Symbol hash** (*sh*): the symbol hash field, henceforth referred to as *sh*, is the value representing the unique hash value assigned to the traded security symbol. This field is internally modeled as an `int` value.

  In the tuples used for the experiments documented in Chapter 13, this hash value is assigned to each tuple by the aforementioned input generator program which makes sure to assign each distinct 16 characters symbol a unique hash value in the range $\{0, ..., \text{TOT\_SYM} - 1\}$, TOT_SYM being the total number of symbols in the input financial stream.
- **Trade volume** (*tv*): the trade volume field, henceforth referred to as *tv*, is the value representing the total volume of shares traded in the trade that is represented by the financial tuple. This field is internally modeled as a `long` value accounting for the number of shares of the traded security symbol that are traded in the transaction represented by the financial tuple.

  In the tuples used for the experiments documented in Chapter 13, this field is retrieved from the bytes $\{33, .., 41\}$ in the entries belonging to the aforementioned *Daily Trades File*.
- **Trade price** (*tp*): the trade price field, henceforth referred to as *tp*, is the value representing the price per share of the traded security symbol in the trade that is represented by the financial tuple. This field is internally modeled as a `double` value accounting for the price paid for each share.

In the tuples used for the experiments documented in Chapter 13, this field is retrieved from the bytes $\{42, .., 52\}$ in the entries belonging to the aforementioned *Daily Trades File*.

Before forwarding each tuple to *PT*, *IT* adds a process start physical timestamp representing the instant of time in which the tuple starts being processed by the stream processing engine. This action extends the tuples received by *PT* from six to seven fields, <*ts*, *seq*, *s*, *sh*, *tv*, *tp*, *sts*>:

- **Process start timestamp** (*sts*): the process start timestamp field, henceforth referred to as *sts*, is the value representing the instant of time in which each individual tuple started being processed by the stream processing engine. This field is internally modeled as a `long` value accounting for the number of elapsed microseconds from the epoch date of January 1, 1970.

  This value will be used after the execution of the stream processing engine to assess the latency in the processing of each individual tuple, hence minimizing the impact of monitoring performance on the execution of the experiments.

The process thread, *PT*, also adds a field to the tuples it processes representing the calculated option price for the given input tuple. This action extends the tuples received by *OT* from seven to eight fields, <*ts*, *seq*, *s*, *sh*, *tv*, *tp*, *sts*, *op*>:

- **Option price** (*op*): the option price field, henceforth referred to as *op*, is the value representing the option price assigned to each tuple by the options pricing operator. This field is internally modeled as a `double` value accounting for the price that should be paid for an option over one share of the underlying stock.

  Section 6.4 further elaborates on how this option price is calculated including which input values are taken by the operator, where are they taken from and how are they used to produce an option price.

Finally, before storing the output tuples in an output file, *OT* adds a process end physical timestamp representing the instant of time in which the tuple finishes being processed by the stream processing engine. This action extends the tuples stored by *OT* from eight to nine fields, <*ts*, *seq*, *s*, *sh*, *tv*, *tp*, *sts*, *op*, *ets*>:

- **Process end timestamp** (*ets*): the process end timestamp field, henceforth referred to as *ets*, is the value representing the instant of time in which each individual tuple finished being processed by the stream processing engine. This field is internally modeled as a `long` value accounting for the number of elapsed microseconds from the epoch date of January 1, 1970.

  This value, together with the aforementioned *sts* value, will be used after the execution of the stream processing engine to assess the latency in the processing of each individual tuple, hence minimizing the impact of monitoring performance on the execution of the experiments.

6.3 Used Data-Structures

The DAG introduced in the previous sections can be seen as a pipeline in which each thread represents one pipeline stage and the tuples processed by the financial stream processor have to follow the path indicated by the DAG through the stream processing engine. In order to achieve this, the different threads representing different stages of this pipeline need to synchronize and communicate through concurrent data-structures.

According to the structure of the DAG and the synchronization needs of the different threads involved in all the calculations, different concurrent data-structures are needed to guarantee the correct behavior of the stream processing engine.

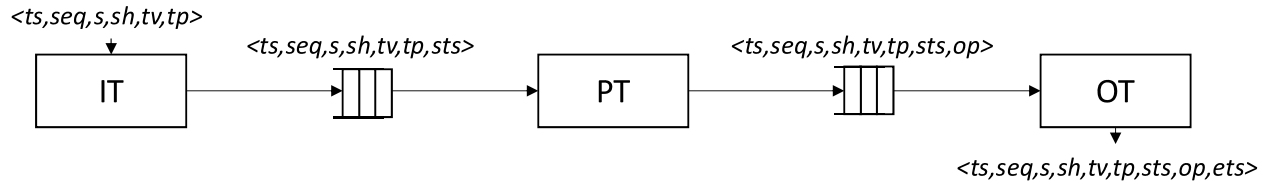


Figure 6.3: Used data-structures

Figure 6.3 extends the structure of the tuples diagram introduced in Figure 6.2 specifying the data-structures used in each transition from one thread to another. As it can be seen, each arrow in the DAG introduced in Figure 6.1 models one of the data-structures in the data-structures diagram in Figure 6.3:

- *IT* **to** *PT* **queue**: *IT* and *PT* share one instance of a concurrent lock-free queue from the NOBLE library [42, 43] introduced in Section 3.1. *IT* acts as the only writer thread in this queue whereas *PT* acts as the only reader thread in this queue. This way, tuples are processed by *PT* in the same order they are served by *IT* and always after they have been assigned the *sts* timestamp by *IT*.
- *PT* **to** *OT* **queue**: *OT* and *PT* also share one instance of a concurrent lock-free queue. This way, tuples are assigned the *ets* timestamp and stored in the output file in the same order as they are processed by *PT* (and transitively in the same order as they are served by *IT* to the previous queue) and always after they have been assigned the *op* value by *PT*.

6.4 Behavior of the Operators

As anticipated in the previous sections, what *PT* does is assigning each tuple an option price value, *op*, based on the financial information contained in the tuple. To do so, an options pricing operator is used. In the experiments performed in the scope of the current thesis, the binomial options pricing operator introduced by Shuo Li in [29], based on the classical paper by Cox et al. [8] is used. Section 6.4.1 below elaborates on the formal description of this operator and Section 6.4.2 integrates this operator in the data-structures diagram from Figure 6.3.

6.4.1 The Single-Threaded Binomial Options Pricing Operator

The binomial options pricing operator by Shuo Li [29] estimates the price of a European call option over one share of the underlying asset. As anticipated in Section 4.1.1, an European call option is a financial contract which gives the buyer (the owner or holder) the right, but not the obligation, to buy an underlying asset or instrument at a specified strike price on a specified date.

As it can be foreseen given its name, this options pricing operator belongs to the family of binomial models based options pricing operators introduced in Section 4.1.1.2. It means that in order to estimate a fair price for an option contract, it models the behavior of the underlying asset price in the interval of time between the option contract creation and the expiration date building a binomial tree and it uses this model to price the option contract.

The next lines elaborate on how this binomial tree model is built and how it is used to price the option contract assuming a continuous risk-less interest rate return on available secure assets such as bonds and the absence of arbitrage opportunities in the market.

The main idea in the binomial tree model is to discretize time in steps and assume two feasible events between one step and another: the underlying asset price can either go up, multiplied by a constant $u \geq 1$, or down, multiplied by a constant $d \leq 1$:

- Let S_{t_0} be the price of the underlying asset at time t_0.
- Let t be the time interval covered by one discrete step.
- Let f be the payoff of the option.
- Let u and d, $d \leq u$, be the aforementioned constants determining the two possible price variations in the transition from time t_0 to time $t_0 + t$.
- Let f_u and f_d be the payoff of the option in the events of the asset price going respectively up and down between t_0 and $t_0 + t$.
- Let p be the probability of the underlying asset price going up between t_0 and $t_0 + t$.
- Let K be the strike price of the option contract, in other words, the price at which the European call contract gives the buyer the right to buy shares of the underlying asset at the expiration date.

With this notation, the two possible scenarios which can be reached from a scenario at time t_0 after t time units can be easily visualized as a Bernoulli distribution tree model.

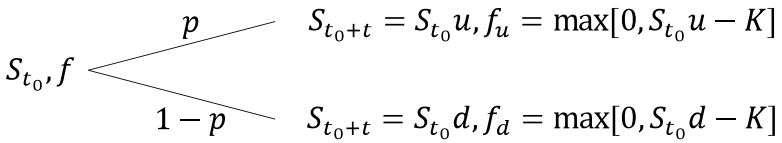


Figure 6.4: Reachable scenarios in one step in the underlying binomial tree model

Figure 6.4 summarizes the aforementioned two reachable scenarios with respect to the underlying asset price per share, S, and the payoff of the option contract, f:

- In the event of the underlying asset price going up, the stock price at time t_0+t would naturally be $S_{t_0}u$ and the payoff of the option would be $\max[0, S_{t_0}u-K]$. In other words, the difference between the market price and the arranged strike price lower bounded by 0 given the fact that the call contract gives the right, but not the obligation, to buy shares of the underlying asset at the expiration date.
- In the event of the underlying asset price going down, the stock price at time $t_0 + t$ would naturally be $S_{t_0}d$ and following the same reasoning as in the previous event, the payoff of the option would be $\max[0, S_{t_0}d - K]$.

Assuming a continuous risk-less interest rate return on available secure assets such as bonds and the absence of arbitrage opportunities in the market, every risk-less portfolio, understanding as a risk-less portfolio, a portfolio built at time t_0 whose payoffs at time $t_0 + t$ are the same under both the scenarios described in Figure 6.4, should behave exactly the same way as a bond with the same price at time t_0. Otherwise, a trivial arbitrage opportunity would consist on a long (buy) position at time t_0 in the bond or risk-less portfolio with the highest payoff at time $t_0 + t$ and a short (sell) position at time t_0 with the same value as the long position in the bond or risk-less portfolio with the lowest payoff at time $t_0 + t$ leading to the investment of 0 monetary units at time t_0 and the risk-less gain of the difference between the fixed payoffs of the bond and the risk-less portfolio at time $t_0 + t$, in other words, an arbitrage opportunity.

The previous observation is relevant when the possibility of building a risk-less portfolio based only on the option contract and the underlying asset is considered. Let this risk-less portfolio be composed of:

- A long (buy) position in Δ unit shares of the underlying asset.
- A short (sell) position in 1 unit of the call option.

A value for Δ so that this portfolio behaves as a risk-less portfolio can be obtained in the context of the one-step Bernoulli model introduced in Figure 6.4 by simply analyzing the value or payoff of the portfolio at time t_0 and at time $t_0 + t$ under the two possible scenarios:

- The value of this portfolio at time t_0 is $S_{t_0}\Delta - f$.
- The value or payoff of this portfolio at time $t_0 + t$ under the scenario in which the underlying asset price goes up with probability p is $S_{t_0}u\Delta - f_u$.
- The value or payoff of this portfolio at time $t_0 + t$ under the scenario in which the underlying asset price goes down with probability $1 - p$ is $S_{t_0}d\Delta - f_d$.

In order for this portfolio to satisfy the risk-less portfolio definition, it has to have the same payoff at time $t_0 + t$ under both scenarios:

$$S_{t_0}u\Delta - f_u = S_{t_0}d\Delta - f_d \tag{6.1}$$

Equation 6.1 can be trivially transformed to express Δ as a function of u, d, f_u, f_d, and S_0:

$$\begin{aligned} S_{t_0}u\Delta - f_u &= S_{t_0}d\Delta - f_d \\ S_{t_0}u\Delta - S_{t_0}d\Delta &= f_u - f_d \\ \Delta(S_{t_0}u - S_{t_0}d) &= f_u - f_d \\ \Delta &= \frac{f_u - f_d}{S_{t_0}u - S_{t_0}d} \end{aligned} \tag{6.2}$$

In order for the market to satisfy the absence of arbitrage assumption given the observation above about the implications of this assumption on the behavior of risk-less portfolios, the payoff at time $t_0 + t$ of this portfolio must be the same as that of a risk-less bond whose value at time t_0 were the same as the value of the portfolio at time t_0. This is, letting the assumed continuous risk-less interest rate be r:

$$S_{t_0}\Delta - f = (S_{t_0}u\Delta - f_u)e^{-rt} \tag{6.3}$$

Substituting in Equation 6.3 the value of Δ obtained in Equation 6.2 it is possible to express the price of the option at time t_0, f, as a function of u, d, f_u, f_d, r, and t:

$$\begin{aligned} S_{t_0}\Delta - f &= (S_{t_0}u\Delta - f_u)e^{-rt} \\ f &= S_{t_0}\Delta - (S_{t_0}u\Delta - f_u)e^{-rt} \\ f = S_{t_0}\frac{f_u - f_d}{S_{t_0}u - S_{t_0}d} &- \left(S_{t_0}u\frac{f_u - f_d}{S_{t_0}u - S_{t_0}d} - f_u\right)e^{-rt} \\ f &= \frac{f_u - f_d}{u - d} - \left(u\frac{f_u - f_d}{u - d} - f_u\right)e^{-rt} \end{aligned} \tag{6.4}$$

Another way to express the price of the option at time t_0, f, based on the same absence of arbitrage argument is expressing it as the expected payoff of the option at time $t_0 + t$, i.e. $pf_u + (1 - p)f_d$, discounted by the continuous risk-less interest rate r. This leads to an expression of f as a function of p, f_u, f_d, r, and t:

$$f = (pf_u + (1 - p)f_d)e^{-rt} \tag{6.5}$$

Refactoring Equation 6.4 towards Equation 6.5 enables the representation of the probability of the stock price going up in one step, p, as a function of u, d, r and t:

$$
\begin{aligned}
f &= \frac{f_u - f_d}{u - d} - \left(u \frac{f_u - f_d}{u - d} - f_u \right) e^{-rt} \\
f &= \frac{f_u - f_d - u f_u e^{-rt} + u f_d e^{-rt} + u f_u e^{-rt} - d f_u e^{-rt}}{u - d} \\
f &= \frac{f_u - f_d + u f_d e^{-rt} - d f_u e^{-rt}}{u - d} \\
f &= \left(\frac{f_u e^{rt} - f_d e^{rt} + u f_d - d f_u}{u - d} \right) e^{-rt} \\
f &= \left(\frac{e^{rt} - d}{u - d} f_u + \frac{u - e^{rt}}{u - d} f_d \right) e^{-rt} \\
p &= \frac{e^{rt} - d}{u - d} \qquad 1 - p = 1 - \frac{e^{rt} - d}{u - d} = \frac{u - e^{rt}}{u - d}
\end{aligned} \tag{6.6}
$$

It is relevant to notice that this expression of p only makes sense if $d \leq e^{rt} \leq u$. Otherwise, probability values out of the interval $[0, 1]$ would be possible. However, the assumption $d \leq e^{rt} \leq u$ comes together with the absence of arbitrage assumption:

- If $e^{rt} < d$, a trivial arbitrage opportunity would consist on a long (buy) position at time t_0 in the option and a short (sell) position at time t_0 with the same value as the long position in the bond leading to the investment of 0 monetary units at time t_0 and the gain, at time $t_0 + t$, of the difference between the payoff of the option and the payoff of the bond, the former being always bigger than the latter.
- If $e^{rt} > u$, a trivial arbitrage opportunity would consist on a long (buy) position at time t_0 in the bond and a short (sell) position at time t_0 with the same value as the long position in the option leading to the investment of 0 monetary units at time t_0 and the gain, at time $t_0 + t$, of the difference between the payoff of the bond and the payoff of the option, the former being always bigger than the latter.

Putting everything together, in the Bernoulli model introduced in Figure 6.4, when a continuous risk-less interest rate return on available secure assets such as bonds is assumed as well as the absence of arbitrage opportunities in the market, the probability of the price of the underlying asset going up, p, can be expressed as a function of the assumed risk-less interest rate, r, the time interval covered by one discrete step in the model, t, and the constants controlling how much the price of the underlying asset can go up, u, or down, d, from time t_0 to time $t_0 + t$ as shown in Equation 6.6.

The variable r can be well understood as an environmentally given constant that needs to be provided to the model from an analysis of the market in which the options are priced. The variable t is directly derived from the desired parameters to price an option, i.e. the time from the option pricing instant to the expiration

date and the number of steps to model in the underlying binomial tree model. But u and d still remain undefined by the analysis of one single step in the underlying binomial tree. It is necessary to move forward from the one step Bernoulli model introduced in Figure 6.4 to the full binomial tree model in order to determine which values of u and d would be correct in order for the binomial options pricing operator to produce reasonable option prices.

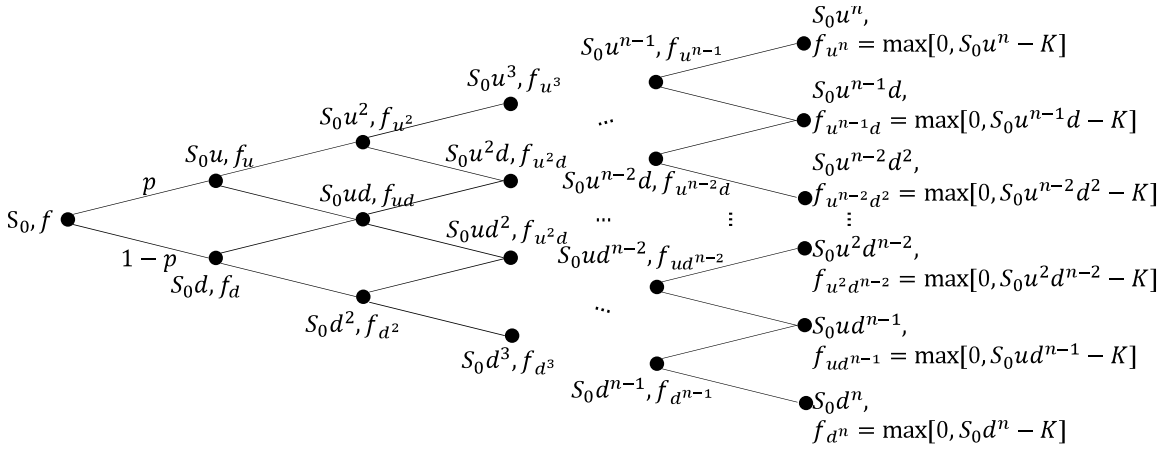


Figure 6.5: Extending the 1-step Bernoulli tree model to an n-steps recombinant binomial tree model

Figure 6.5 extends the 1-step Bernoulli tree model introduced in Figure 6.4 to a recombinant binomial tree model. In this extended model, each of the nodes together with its two child nodes represent one instance of the Bernoulli tree model introduced in Figure 6.4 and analyzed above. However, all of these groups of three nodes are integrated in the binomial tree model in Figure 6.5 in a recombinant manner. It means that given any node in the tree, the child node following the stock price going down path from the child node following the stock price going up path from the given node is the same node as the child node following the stock price going up path from the child node following the stock price going down path from the given node. This property establishes an upper bound on the number of nodes per level which grows linearly with the number of levels instead of exponentially, making it computationally feasible to model an orders of magnitude bigger number of steps, and it introduces a new constraint on the values u and d. In order for the recombinant behavior of the extended tree to be achieved, u and d must satisfy Equation 6.7.

$$ud = 1 \tag{6.7}$$

Equation 6.7 allows for the simplification of the equations in Figure 6.5 by substituting d by u^{-1}.

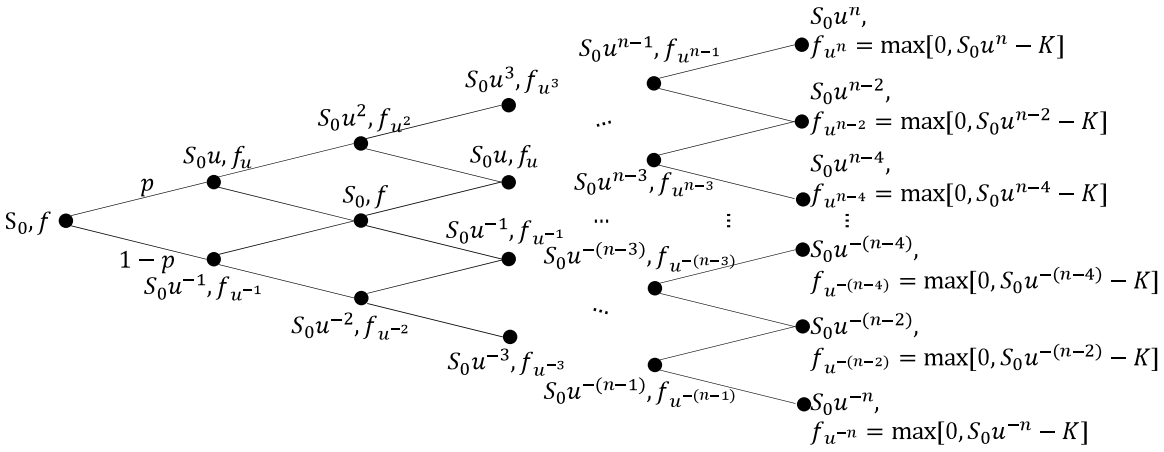


Figure 6.6: n-steps recombinant binomial tree model with simplified equations

Figure 6.6 applies Equation 6.7 to simplify the equations shown in Figure 6.5 according to the requirement of the binomial tree to be recombinant.

In addition to the requirement on the values of u and d described in Equation 6.7, Cox et al. [8] proof that in order to converge to the classical Black-Scholes options pricing formula [4] when the number of steps in the binomial tree tends to ∞, the values of u and d given the standard deviation, σ, of the underlying asset price distribution as a measure of its volatility, must be the ones shown in Equation 6.8.

$$u = e^{\sigma\sqrt{t}} \qquad d = e^{-\sigma\sqrt{t}} \tag{6.8}$$

With this, the variables u and d, are now defined as a function of the variable t, which is directly derived from the desired parameters to price an option, i.e. the time from the option pricing instant to the expiration date and the number of steps to model in the underlying binomial tree model, and the variable σ, which, as it happened before with r, can be well understood as an environmentally given constant that needs to be provided to the model from an analysis of the market in which the options are priced.

Putting everything together, the binomial options pricing operator introduced by Shuo Li [29] builds the binomial tree model introduced in Figure 6.5 and then it backtracks from the right most level in Figure 6.5 to the left most one discounting the expected option payoff as described in Equation 6.5 obtaining this way, when reaching the parent node in the tree, the fair option price. To do so, and as discussed above, five financial input values are taken in addition to one algorithm-internal parameter to define the granularity of the analysis:

- **Underlying stock price** (tp): the underlying stock price per share at the time of pricing the option contract, henceforth referred to as tp, in accordance to the notation introduced in Section 6.2. This value is provided to the operator as a `double` value accounting for the price per share of the underlying stock.

- **Option strike** (os): the strike or price at which the European call option contract gives the buyer the right to buy shares of the underlying asset at the expiration date, henceforth referred to as os (option strike). This value is provided to the operator as a `double` value accounting for the price the buyer has the right, but not the obligation, to pay per share of the underlying asset at the expiration date.
- **Option time to maturity** (om): the time from the option pricing instant to the expiration date of the option contract, henceforth referred to as om (option maturity). This value is provided to the operator as a `double` value accounting for the number of years from the option pricing instant to the expiration date of the priced option contract.
- **Risk-less interest rate** (rli): the guaranteed by the market risk-less interest rate, henceforth referred to as rli (risk-less interest). This value is provided to the operator as a `double` value accounting for the guaranteed continuous risk-less interest rate.
- **Volatility** (v): the volatility of the underlying stock, henceforth referred to as v (volatility). This value is provided to the operator as a `double` value accounting for the observed volatility of the distribution of prices of the underlying asset.
- **Number of steps** (N): the number of steps to be modeled, henceforth referred to as N (number of steps). This parameter determines how many levels the underlying binomial tree model explained above will have. On the one hand, the bigger N is, the finer grained and consequently accurate the assessment of the fair option price will be. On the other hand, the smaller N is, the cheaper in terms of memory and time the computations performed by the operator will be. Shuo Li [29] recommends modeling 2048 levels for a reasonable tradeoff between granularity and computational cost.

The notation introduced above together with the analysis of the underlying binomial tree model allows for the formal description of a single-threaded version of the binomial options pricing operator introduced by Shuo Li [29]. However, and in order to properly understand the pseudocode describing the operator, it is worth making two observations. The first one involves how many nodes of the tree need to be maintained in memory at a time and the second one suggests how the stage of building the binomial tree can be dramatically optimized by directly building the last level of the tree.

Regarding how many nodes of the tree needs to be maintained at a time, it is worth noticing that in order to backtrack in the tree applying Equation 6.5 to calculate the payoff of the option at a parent node given the payoffs of the options at both children nodes until the three parent node is reached, only up to two levels of the tree need to be maintained at a time. What is more, in this operation both levels can be simultaneously maintained in the same array if the child node following the stock price going down from the parent node is assigned to the same index as the parent node and the child node following the stock price going up from the parent node is assigned to the next index.

Regarding how the stage of building the binomial tree can be dramatically optimized by directly building the last level of the tree, it is worth reviewing the equations in the last level of Figure 6.6. These equations display a closed formula for the expected option payoff at each node in the right most level of the tree in the context of the binomial model which enables the operator to directly model this final level of the tree without having to iterate through the tree from left to right according to Figure 6.6. This last observation concludes also the previous one as it bounds to one the number of levels to be simultaneously maintained in memory during the tree model construction stage letting the two levels upper bound discussed before be the upper bound of levels to be maintained in memory at a time.

Listing 6.1: Binomial options pricing operator pseudocode

```
1  N = 2048 // The number of steps to be modeled
2  priceOption(tp, os, om, rli, v)
3    // Static array modeling the algorithm view of
4    // at most two levels of the tree at a time
5    call[N + 1]
6    // Pre-calculate constants
7    t = om / N
8    v_sqrt_t = v * sqrt(t)
9    rli_t = rli * t
10   incr_f = exp(rli_t)
11   disc_f = exp(-rli_t)
12   u = exp(v_sqrt_t)
13   d = exp(-v_sqrt_t)
14   pu = (incr_f - d) / (u - d)
15   pd = 1 - pu
16   pu_disc_f = pu * disc_f
17   pd_disc_f = pd * disc_f
18   // Directly build the last level of the binomial tree
19   for (i = 0 to N)
20     payoff = tp * exp(v_sqrt_t * (2 * i - N)) - os
21     call[i] = payoff > 0 ? payoff : 0
22   // Backtrack from the last to the first level
23   for (i = N - 1 to 0)
24     for (j = 0 to i)
25       call[j] = pu_disc_f * call[j + 1]
26                 + pd_disc_f * call[j]
27   return call[0]
```

Listing 6.1 formally introduces the pseudocode of the single-threaded version of the binomial options pricing algorithm introduced by Shuo Li [29]. In line 5 the size of the array to maintain up to two levels of the binomial tree at a time is set to be 1 plus the number of steps to be modeled, N, which is the number of nodes the last level of the binomial tree has. Lines 7-17 pre-calculate the constants analyzed above which are needed to build the last level of the tree and backtrack from that level.

The loop in lines 19-21 corresponds to the first stage of the operator in which the last level of the binomial tree is modeled directly storing in each position of the array the payoff off the option at that node in the last level according to the equations in Figure 6.6. Finally, the loops in lines 23-26 correspond to the second and final stage of the operator backtracking from the last level to the first level applying Equation 6.5 to traverse from each pair of child nodes to their parent node obtaining this way for the parent node in the first level a fair estimation of the option price. This value is the one returned in line 27.

Before concluding this section, it is worth analyzing the complexity of this operator in terms of time and memory. As it can easily be seen after a brief analysis of the pseudocode introduced in Listing 6.1, the only parameter affecting the cost of the operator in time and memory is N. In terms of time:

- Lines 7-17 calculate constants at a cost independent of the value of N. The cost of these operations is $O(1)$.
- Lines 19-21 iterate a single loop from 0 to N. The cost of these operations is $O(N)$.
- Lines 23-24 iterate two nested loops: the outer one from $N-1$ down to 0, and the inner one from 0 to the number of nodes in the right most level present in the tree at that iteration of the outer loop. The cost of these operations is $O(N^2)$.

Overall, the cost of the operator in terms of time is the one expressed in Equation 6.9.

$$O(N^2) \tag{6.9}$$

The cost in terms of memory is even easier to assess:

- The input variables (`tp`, `os`, `om`, `rli`, and `v`) plus the constant N itself occupy a fixed amount of memory independent of the value of N. The space occupied by these variables in memory is $O(1)$.
- The `call` array introduced in line 5 has the size $N+1$. The space occupied by this variable in memory is naturally $O(N)$.
- The temporary variables used by the algorithm which are initialized in lines 7-17 and line 20 occupy a fixed amount of memory independent of the value of N. The space occupied by these variables in memory is $O(1)$.

Overall, the cost of the operator in terms of memory is the one expressed in Equation 6.10.

$$O(N) \tag{6.10}$$

6.4.2 Integrating the Single-Threaded Binomial Options Pricing Operator

Getting back to the stream processing engine diagram from Section 6.3, it is worth noticing that out of the six values that the binomial options pricing operator uses as an input, namely the stock price (tp), option strike (os), option time to maturity (om), risk-less interest rate (rli), volatility (v) and number of steps (N), only one, tp, can be retrieved by PT from the tuples served by IT to the queue these two threads share as described in Section 6.3. In this first stage of the financial stream processing engine, the rest of the values are simply taken as constants:

- **Stock price** (tp): this value is taken from the tuples served by IT.
- **Option strike** (os): this value is taken in this stage of the financial options pricing stream processing engine as a constant, internally modeled as a `double` constant whose default value is 75.
- **Option time to maturity** (om): this value is taken in this stage of the financial options pricing stream processing engine as a constant, internally modeled as a `double` constant whose default value is 3.
- **Risk-less interest rate** (rli): this value is taken in this stage of the financial options pricing stream processing engine as a constant, internally modeled as a `double` constant whose default value is 0.06.
- **Volatility** (v): this value is taken in this stage of the financial options pricing stream processing engine as a constant, internally modeled as a `double` constant whose default value is 0.1.
- **Number of steps** (N): this value is taken in this stage of the financial options pricing stream processing engine as a constant, internally modeled as an `int` constant whose default value is 2048 according to [29].

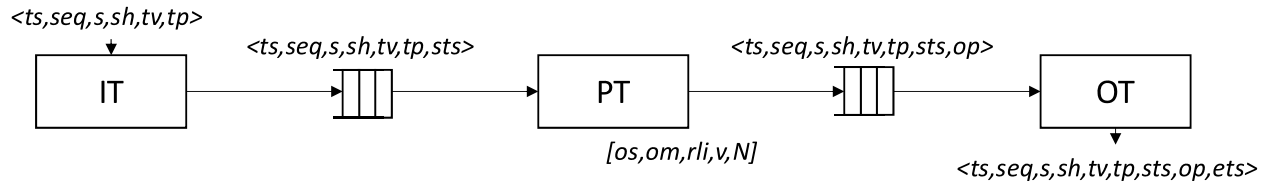


Figure 6.7: Operators and used constants

Figure 6.7 extends the data-structures diagram introduced in Figure 6.3 to represent the aforementioned constant values between square brackets close to the box representing the thread, PT, which executes the single-threaded binomial options pricing operator.

As it can be well understood, there is plenty of room for improvement starting from this first iteration of the financial stream processing engine. On the one hand, the binomial options pricing stage of the stream processing engine seen as a pipeline can be optimized. On the other hand, the constants os, om, rli and v need to be provided in a more useful way in order to let the stream processing engine price any option with any settings instead of just options with always the same fixed settings. The next iterations of the stream processing engine introduced in the next chapters

will improve the current binomial options pricing stream processing engine in both directions.

7

Batching Based Multi-Threaded Binomial Options Pricing

In the previous chapter, the core functionality of the stream processing engine, namely the options pricing operator in its single-threaded version, was introduced. As it could be seen in Equation 6.9 introduced in Section 6.4, the cost of the options pricing operation in terms of execution time grows quadratically with respect to the number of steps modeled in the underlying binomial tree. Given that a higher number of steps leads to a finer grained analysis resulting in a more accurate option price [8], this operator would become a scalability bottleneck in an options pricing stream processing engine if it is used in its single-threaded version as introduced in Chapter 6. For this reason, it is reasonable to approach the problem of parallelizing this operator. This chapter and the next two ones, Chapter 8, and Chapter 9, focus on this problem evolving the design introduced in the previous chapter.

The approach followed in this chapter to parallelize the single-threaded binomial options pricing operator is strongly inspired by the paper by Shuo Li [29]. In this paper, two approaches are followed to parallelize the operator making use of the OpenMP framework [37] introduced in Section 2.1.2. On the one hand, a pool of OpenMP threads process in parallel a set of different option contracts profiting from the fact that different option contracts are independent one to another. On the other hand, the vector processing capabilities of the Xeon Phi Coprocessor [40] are exploited by letting OpenMP unroll the loop in lines 24-26 in Listing 6.1 and processing both the loop in lines 19-21 and the aforementioned loop in lines 24-26 following the Single Instruction Multiple Data (SIMD) paradigm. Even though the second approach achieves both an increase in throughput and a decrease in latency by reducing the amount of time needed to process each specific option contract, given its high dependability on the specific hardware platform, only the first one, which would improve the stream processing engine throughput but not its latency, is followed in this chapter.

The following sections elaborate on how the *PT* thread introduced in Section 6.1 can be replaced by a batching helper thread and a set of OpenMP threads in order to price sets of option contracts in parallel ensuring that processed tuples are delivered to *OT* in the same order as the corresponding input tuples are served by *IT*.

7.1 Involved Threads

As anticipated above, the thread dedicated in the previous chapter to calculating option prices is replaced in this second iteration of the financial stream processing engine by a batching helper thread and a set of OpenMP threads. The two threads dedicated to retrieve tuples from the input dataset and serve them to the process threads, and to get the tuples output, in this case, by the batching helper thread, and output them, remain in this second iteration exactly as they were introduced in the first one.

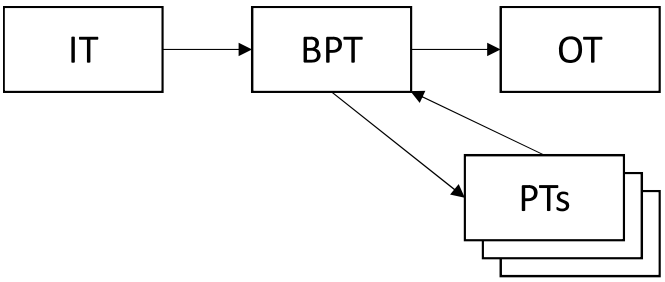


Figure 7.1: Involved threads

Figure 7.1 outlines how threads are arranged in this second version of the financial stream processing engine. As it can be seen, IT and OT remain as they were introduced in Section 6.1 while PT has been replaced by BPT and the PT OpenMP threads:

- **Batching Process Thread** (BPT): the batching process thread, henceforth referred to as BPT, is the thread which substitutes the former PT thread in the pipeline. This new thread will arrange in batches the tuples received from IT and it will serve the tuples to OT in the same order as they were served by IT every time a batch is processed.
- **Parallel (OpenMP) Process Threads** (PT): the parallel OpenMP process threads, henceforth referred to as PT, are the OpenMP threads which apply the binomial options pricing operator to the generate option prices according to the financial information contained in the input tuples the same way as the former PT thread did.

7.2 Structure of the Tuples

As introduced in Section 6.2, input tuples are retrieved from the input financial stream introduced in Section 5.1.1 by IT and they are extended by each of the threads introduced in the previous section introducing all the needed additional fields.

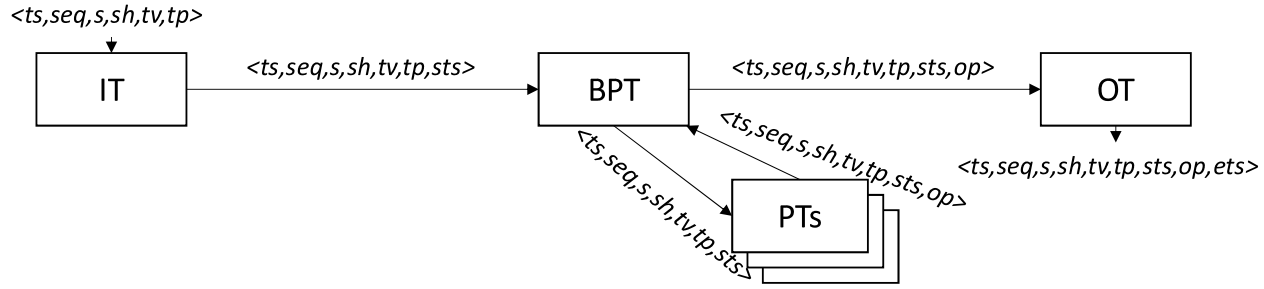


Figure 7.2: Structure of the tuples

Figure 7.2 extends the involved threads diagram introduced in Figure 7.1 specifying the structure of the tuples in each transition from one thread to another as well as the structure of the tuples read by IT, and the tuples output by OT.

As it can be seen, the structure of the tuples retrieved by IT, the tuples transferred from IT to BPT, the tuples served by BPT to OT, and the tuples output by OT is exactly the same as the structure of the tuples retrieved by IT, the tuples transferred from IT to PT, the tuples served by PT to OT, and the tuples output by OT in Figure 6.2. The tuples facilitated to the PT threads by BPT have the same structure, $<ts$, seq, s, sh, tv, tp, $sts>$, as the tuples retrieved by BPT from IT and it is each PT thread who adds the op field introduced in Section 6.2 to the tuples received from BPT before serving them back to BPT with the extended structure $<ts$, seq, s, sh, tv, tp, sts, $op>$.

7.3 Used Data-Structures

As introduced in Section 6.3, the DAG introduced in the previous sections, composed by the threads IT, BPT, and OT, can be seen as a pipeline in which each thread represents one pipeline stage and the tuples processed by the financial stream processor have to follow the path indicated by the DAG through the stream processing engine. In addition to this, the tuples processed by BPT need to be provided to the PT threads in batches and collected back by BPT once an option price is assigned to each tuple before serving them to OT. In order to achieve all of this, the IT, BPT, and OT threads synchronize and communicate through concurrent data-structures as in Section 6.3 and the BPT and the PT threads communicate through shared memory and synchronize using OpenMP.

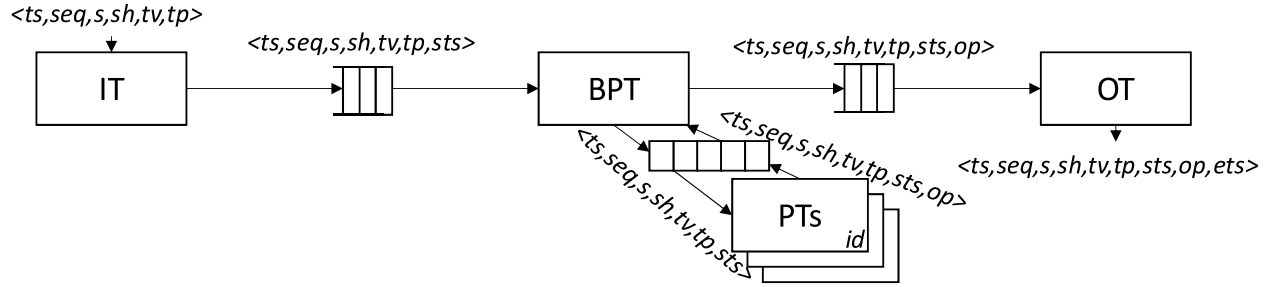


Figure 7.3: Used data-structures

Figure 7.3 extends the structure of the tuples diagram introduced in Figure 7.2 specifying the data-structures used in each transition from one thread to another. As anticipated above, *IT* and *BPT*, and *BPT* and *OT* communicate through concurrent lock-free queues as *IT* and *PT*, and *PT* and *OT* did in Section 6.3. The new data-structure introduced in this section is the one used for the communications and synchronization between *BPT* and the *PT* OpenMP threads:

- ***BPT* to *PT* threads and *PT* threads to *BPT* shared array**: *BPT* and the *PT* OpenMP threads communicate through a shared array of tuples. *BPT* iteratively adds tuples to the shared array until it is full. When the array is full, *BPT* temporarily stops reading tuples from the queue it shares with *IT* and lets OpenMP launch the *PT* OpenMP threads, each of which process one or more tuples in the array. Once each tuple in the shared array has been processed and assigned an option price by one and only one *PT* thread, *BPT* serves the processed tuples from the array to the queue it shares with *OT*. After this, the shared array is empty again and *BPT* can start reading tuples again from the queue shared with *IT* and adding them to the shared array.

7.4 Behavior of the Operators

As anticipated in the previous sections, the *PT* OpenMP threads assign each tuple an option price value, *op*, based on the financial information contained in the tuple, the same way as the former *PT* thread did in the previous chapter. It is the *BPT* thread the one responsible for arranging the tuples in batches so that they can be processed in parallel by the *PT* threads, and serving them the *OT* thread once they have been processed by the *PT* threads in the same order as they were provided by *IT*. Section 7.4.1 below elaborates on the formal description of this batching mechanism and Section 7.4.2 integrates it in the data-structures diagram from Figure 7.3.

7.4.1 The Batching Based Multi-Threaded Binomial Options Pricing Operator

The *BPT* thread can be understood as a centralized scheduler of *PT* threads. While the former *PT* thread simply kept retrieving tuples from the input queue, individually processing each of them using the operator whose pseudocode was introduced in Listing 6.1 in Section 6.4.1, and adding them to the output queue, the *BPT* thread batches the tuples retrieved from the input queue, let the *PT* OpenMP threads process them in parallel, and serves them to the output queue afterwards.

Given that the operator executed by the *PT* OpenMP threads is exactly the same as the one introduced in Section 6.4.1, and that the behavior of the *BPT* thread in this section is the key addition to the stream processing engine with respect to the previous iteration, this section focuses on how *BPT* interacts with the input queue, which is the one it shares with *IT*, the array it uses to communicate with the *PT* OpenMP threads, and the output queue, which is the one it shares with *OT*.

Listing 7.1: Batching-based multi-threaded binomial options pricing operator pseudocode

```
1  BS              // Batch size
2  BATCH[BS]       // Shared array of tuples with size BS
3
4  processTuples(inQueue, outQueue)
5    currPos = 0
6    while (TRUE)
7      BATCH[currPos] = inQueue.deq()
8      currPos = currPos + 1
9      if (currPos == BS)  // Batch full and ready
10       processBatch()
11       for (i = 0 to BS - 1)
12         outQueue.enq(BATCH[i])
13       currPos = 0
14
15 processBatch()
16 #pragma omp parallel for
17   for (i = 0 to BS - 1)
18     BATCH[i].op = priceOption(BATCH[i].tp, os, om, rli, v)
```

Listing 7.1 formally introduces the pseudocode describing the behavior of the *BPT* and *PT* OpenMP threads. Two main procedures are introduced in this listing. The `processTuples` procedure in lines 4-13 is the task executed by the *BPT* thread and the `processBatch` procedure in lines 15-18 lets the *BPT* thread offload the calculation of the option prices to the *PT* OpenMP threads. In addition to this, the shared array described in Section 7.3 is initialized in lines 1-2 and visible inside both the aforementioned procedures.

The infinite loop in lines 6-13 determines how the *BPT* thread processes each tuple it retrieves from the input queue, `inQueue`. In line 7, each retrieved tuple is stored in the current position of the shared `BATCH` array. The current position being 0 for the first iteration, as initialized in line 5, and the position after the previous one for the next iterations in which the retrieved tuples belong to the same batch, as updated in line 8. Line 9 checks if the batch of tuples is full after the addition of the tuple retrieved in line 7. In case the batch is full, the `processBatch` procedure is called in line 10 to let the *PT* OpenMP threads process all the tuples in the batch assigning them an option price, and the loop in lines 11-12 is executed afterwards to enqueue the tuples in the output queue, `outQueue`, in the same order as they were retrieved from the input queue, `inQueue`. Done so, the position in the batch is updated again to 0 in line 13, as it was done in line 5 for the very first iteration of the loop in lines 6-13, letting the *BPT* thread start filling the shared array from the beginning in the next execution of the loop, or in other words, start gathering together the next batch of tuples.

The for loop in lines 17-18 is parallelized using OpenMP with the directive shown in line 16. This automatically lets the OpenMP framework partition the set of BS iterations of the loop and assign each of the parallel PT OpenMP threads one subset of the iterations so that the PT OpenMP threads process the loop in parallel, each iteration being executed by one and only one PT OpenMP thread. The body of this loop, which contains the code executed for each iteration by the corresponding PT OpenMP thread, consists on line 18, in which the `priceOption` procedure introduced in Listing 6.1 in Section 6.4.1 is executed to add the corresponding tuple the *op* field, representing the option price assigned to the corresponding tuple based on its financial information. To do so, in this iteration of the stream processing engine, only the trade price, tp, is retrieved from the tuple, whereas the option strike, os, the option maturity, om, the risk-less interest rate, rli, and the volatility, v, are provided as fixed constants with the same values for all the tuples, as it was done in the previous iteration.

Before concluding this section, it is worth analyzing the complexity of this operator in terms of time and memory. As it can easily be seen after a brief analysis of the pseudocode introduced in Listing 7.1, the main parameters affecting the cost of the operator in time and memory, apart from N as in the previous chapter, are the batch size, BS, and the number of PT OpenMP threads, henceforth n. In terms of time:

- Lines 8-9 represent an $O(1)$ time overhead for each individual tuple in terms of scheduling.
- Line 7 and line 12 are executed once for each tuple as they were already executed by PT in the previous chapter.
- Finally line 18 is executed once for each tuple as it was done also by PT.

In this sense the execution time overhead in terms of scheduling is the one expressed in Equation 7.1.

$$O(1) \tag{7.1}$$

Nevertheless, given the semantics of the algorithm, it is worth noticing that:

- Each tuple in each batch needs to wait before being delivered to OT until the full batch of tuples is ready, which represents an average $O(BS)$ overhead per tuple.
- Each tuple in each batch also needs to wait until all the tuples in the batch have been processed. A fast analysis of loop 17-18 taking line 16 into account shows that the average overhead of this per tuple is $O(N^2) \cdot O(BS/n)$. The $O(N^2)$ contribution coming from the time complexity analysis performed in Section 6.4.1, the $O(BS/n)$ contribution coming from the BS iterations of the loop in lines 17-18, parallelized among n threads.

With these two observations, the time that each tuple needs to wait from the moment it is retrieved by BPT from the input queue until the moment it is enqueued by BPT to the output queue is expressed in Equation 7.2.

$$L(n) = O(BS) + O(N^2) \cdot O(BS/n) \tag{7.2}$$

Taking into account that given the discussion above, each option is priced by one and only one PT OpenMP thread, the time each tuple needs to wait from the moment it is retrieved by BPT from the input queue until the moment it is enqueued by BPT to the output queue is minimized when Equation 7.3 holds.

$$BS = n \tag{7.3}$$

If this condition holds, Equation 7.2, which represents the expected latency for each individual tuple in the system can be simplified to Equation 7.4.

$$L(n) = O(n) + O(N^2) \cdot O(1) = O(n) + O(N^2) \tag{7.4}$$

Given that the latency for each individual tuple in the first iteration of the stream processing engine is $O(N^2)$ as expressed in Equation 6.9. The overall overhead in terms of latency for each tuple in this iteration of the stream processing engine is the one expressed in Equation 7.5.

$$O(n) \tag{7.5}$$

In compensation for this overhead, the impact on throughput of this approach is straight forward. Given that the PT OpenMP threads share the load of the loop in lines 17-18 distributing iterations among the n threads, and that the synchronization overhead per tuple, as introduced in Equation 7.1, is negligible compared to the single-threaded operator overhead ($O(1) << O(N^2)$), the throughput of the stream processing engine presented in this iteration is expected to grow linearly with the number of threads as expressed in Equation 7.6

$$T(n) = n \cdot T(1) \tag{7.6}$$

With this, the expected latency and throughput of this iteration of the stream processing engine can be plotted as a function of the number of PT OpenMP threads used.

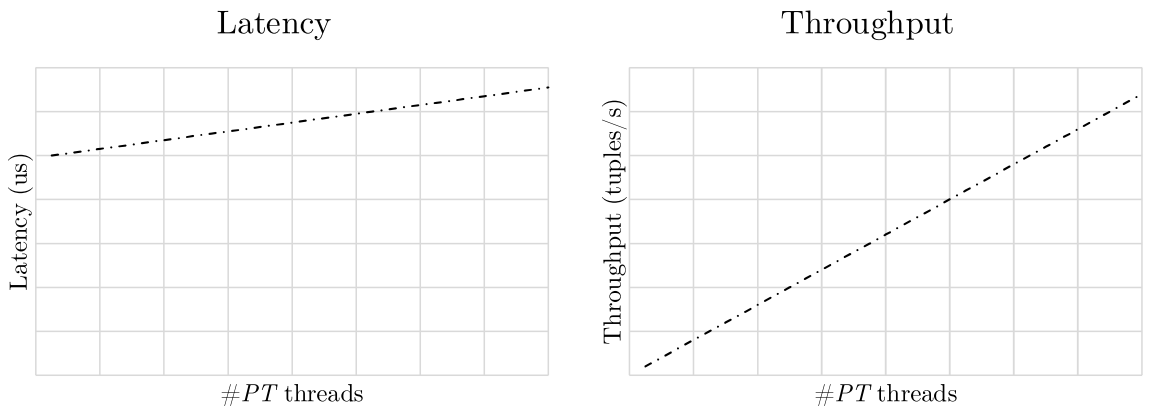


Figure 7.4: Latency and throughput as a function of the number of PT OpenMP threads

Figure 7.4 plots the expected latency and throughput of this iteration of the stream processing engine. It is worth noticing that the positive impact of parallelization in terms of throughput is stronger than its negative impact in latency. This is due to the high overhead of calculating each individual option price compared to the small overhead of scheduling which has been analyzed above.

The overhead in terms of memory is even easier to assess:

- The shared `BATCH` array initialized in line 2 with size BS tuples, or directly n if Equation 7.3 holds, plus the `BS` constant itself occupy in memory $O(BS)$ space, or directly $O(n)$ if Equation 7.3 holds.
- The temporary variables used by the algorithm which are initialized in lines 5 (`currPos`), 11 (`i` in `processTuples`), and 17 (`i` in `processBatch`) occupy a fixed amount of memory independent of the values of N, BS or n. The space occupied by these variables in memory is $O(1)$ per PT OpenMP thread, $O(n)$ at the global stream processing engine level.

Overall, the overhead of the batching-based multi-threaded version of the operator in terms of memory when Equation 7.3 holds is the one expressed in Equation 7.7.

$$O(n) \tag{7.7}$$

Adding the overhead above to the cost in terms of memory of the single-threaded volatility operator introduced in Equation 6.10 in Section 6.4.1, the total cost in terms of memory of the batching-based multi-threaded version of the options pricing operator in terms of memory when Equation 7.3 holds is the one expressed in Equation 7.8.

$$O(n) \cdot O(N) + O(n) \tag{7.8}$$

7.4.2 Integrating the Batching Based Multi-Threaded Binomial Options Pricing Operator

Getting back to the stream processing engine diagram from Section 7.3, it is worth noticing that, as anticipated in Section 7.4.1 and the same way as in the previous iteration of the stream processing engine, out of the six values that the binomial options pricing operator uses as an input, namely the stock price (tp), option strike (os), option time to maturity (om), risk-less interest rate (rli), volatility (v) and number of steps (N), only one, tp, can be retrieved by BPT from the tuples served by IT to the queue these two threads share as described in Section 7.3. In this second iteration of the financial stream processing engine, the rest of the values are simply taken as constants with the default values introduced in Section 6.4.2.

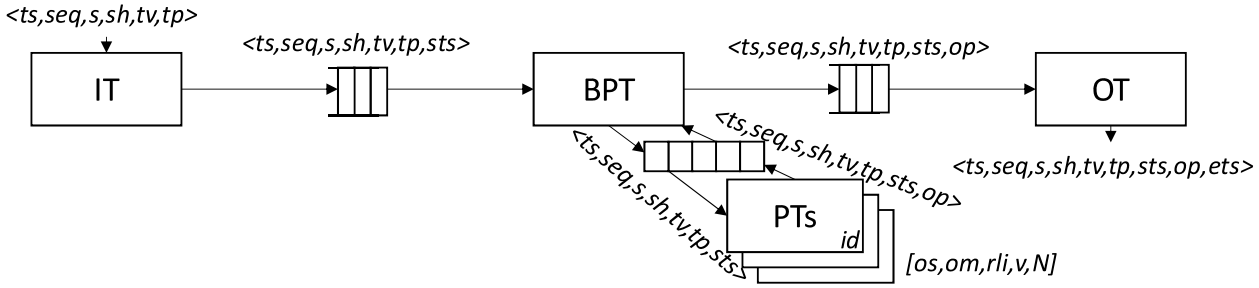


Figure 7.5: Operators and used constants

Figure 7.5 extends the data-structures diagram introduced in Figure 7.3 to represent the aforementioned constant values between square brackets close to the boxes representing the PT OpenMP threads, each of which executes the single-threaded binomial options pricing operator introduced in the previous chapter.

Overall, this second iteration of the stream processing engine responds to the first line of improvement outlined in the last paragraph in Section 6.4.2 concluding Chapter 6.

8

Queue-*ScaleGate*-Based Multi-Threaded Binomial Options Pricing

In the previous chapter, a batching based approach to the options pricing operator parallelization problem was introduced and integrated in the stream processing engine. This approach involved synchronizing a centralized batching helper thread, BPT, and a pool of OpenMP option pricing threads, the PT OpenMP threads. As a result, the expected throughput linearly increased with the number of PT OpenMP threads at the cost of a small overhead in the expected latency due to the fact that tuples needed to wait for all the tuples belonging to their same batch to be retrieved by BPT before starting being processed.

In this chapter, an alternative approach allowing tuples to start being processed as soon as they are retrieved from the input queue populated by the input thread is introduced. It involves letting multiple option pricing threads competitively retrieve tuples from the input queue and using the *ScaleGate* [21] data-structure introduced in Section 3.2 to let these threads collaboratively re-order the tuples in the same order as they were added to the input queue by the input thread. This new approach also distributes the synchronization effort to which the BPT thread in the previous approach was entirely dedicated, avoiding the risk of letting the centralized scheduling thread eventually become a scalability bottleneck.

The following sections elaborate on how the PT thread introduced in Section 6.1 and the queue it enqueued tuples to can be replaced by a set of parallel PT threads and a concurrent *ScaleGate* data-structure in order to price sets of option contracts in parallel ensuring that processed tuples are delivered to OT in the same order as the corresponding input tuples are served by IT.

8.1 Involved Threads

As anticipated above, the thread dedicated in the previous chapter to batching the tuples and synchronizing the pool of parallel OpenMP threads is replaced in this third iteration by a set of parallel options pricing threads which do not need a synchronization thread as in the previous approach because all the option pricing threads will implicitly synchronize by using the concurrent queue and *ScaleGate*.

The two threads dedicated to retrieve tuples from the input dataset and serve them to the process threads, and to get the tuples output, in this case, by the parallel option pricing threads, and output them, remain in this third iteration exactly as they were introduced in the first one.

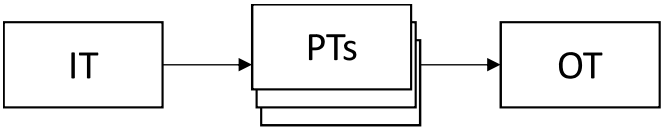


Figure 8.1: Involved threads

Figure 8.1 outlines how threads are arranged in this third version of the financial stream processing engine. As it can be seen, *IT* and *OT* remain as they were introduced in Section 6.1 while the *BPT* thread and the pool of *PT* OpenMP threads introduced in Section 7.1 have been replaced by a set of parallel option pricing threads. Given that the internal behavior of these options pricing threads is basically the same as in the first iteration, they have been assigned the same name in this chapter as the former *PT* thread introduced in Section 6.1 in Chapter 6.

8.2 Structure of the Tuples

As introduced in Section 6.2, input tuples are retrieved from the input financial stream introduced in Section 5.1.1 by *IT* and they are extended by each of the threads introduced in the previous section introducing all the needed additional fields.

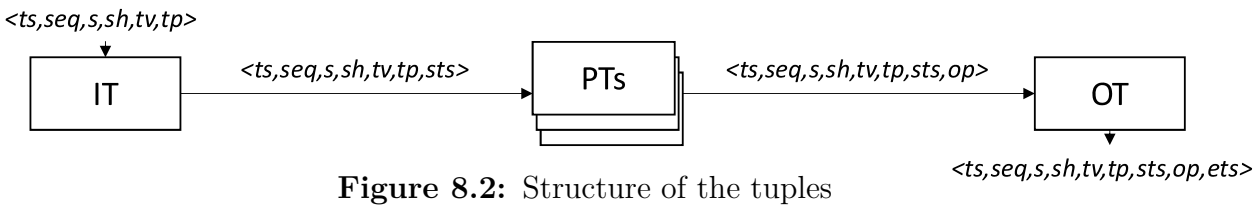


Figure 8.2: Structure of the tuples

Figure 8.2 extends the involved threads diagram introduced in Figure 8.1 specifying the structure of the tuples in each transition from one thread to another as well as the structure of the tuples read by *IT*, and the tuples output by *OT*.
As it can be seen, the structure of the tuples retrieved by *IT*, the tuples transferred from *IT* to the parallel *PT* threads, the tuples served by the parallel *PT* threads to *OT*, and the tuples output by *OT* is exactly the same as the structure of the tuples retrieved by *IT*, the tuples transferred from *IT* to the former *PT* thread, the tuples served by the former *PT* thread to *OT*, and the tuples output by *OT* in Figure 6.2.

8.3 Used Data-Structures

The key difference between this third iteration of the financial stream processing engine and the two first iterations discussed in Chapters 6 and 7 is that the options

pricing stage of the DAG is parallelized temporarily subdividing the single physical stream of financial tuples which flowed from IT to OT into a set of physical streams of financial tuples, one for each PT thread, all of them belonging to the same logical stream and naturally having all the tuples in each physical stream traversing the PT threads in ascending timestamp order.

On the one hand, subdividing the single physical stream of tuples produced by IT is straight forward given the semantics of the concurrent queue. Having IT enqueuing tuples in ascending timestamp order and letting the set of PT threads competitively retrieve tuples from the queue results in a partition of the physical stream produced by IT into the set of physical streams traversing each PT thread, each tuple output by IT belonging to one and only one of these streams and having all the tuples in each stream traversing the corresponding PT thread in ascending timestamp order.

On the other hand, recombining the set of physical streams of tuples traversing each PT thread is not as straight forward as the aforementioned task. If a queue was used for this purpose, the resulting output stream would not be guaranteed to have all the tuples in the same order as they were served by IT, preventing the stream processing engine from meeting the linearizability requirement. Fortunately, the *ScaleGate* concurrent data-structure, as anticipated in Section 3.2, solves this problem by letting the concurrent writers insert tuples in the proper order and allowing readers to consume tuples only once they are ready to be consumed ensuring that the resulting output stream is ordered in ascending timestamp order the same way as the stream produced by IT was.

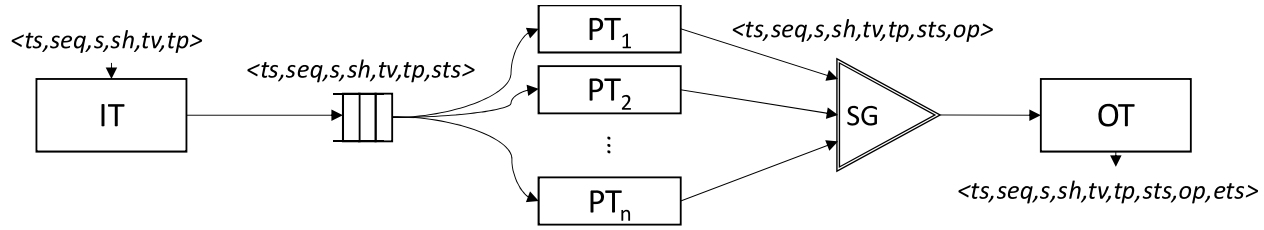


Figure 8.3: Used data-structures

Figure 8.3 extends the structure of the tuples diagram introduced in Figure 8.2 specifying the data-structures used in each transition from one thread to another:

- IT **to the** PT **threads queue**: As reasoned above, IT and the parallel PT threads keep sharing a concurrent lock-free queue from the NOBLE library [42, 43] introduced in Section 3.1. This time, IT acts as the only writer thread in this queue as it did in the two previous iterations, and the PT threads act as a set of multiple readers in this queue. This way, as anticipated above, tuples are processed by one and only one PT thread in the same order they are served by IT and always after they have been assigned the *sts* timestamp by IT.
- PT **threads to** OT ***ScaleGate***: As anticipated above, the PT threads and OT share an instance of *ScaleGate* in which the PT threads act as a set of

multiple writers and *OT* acts as a single reader. This way, the set of physical streams of tuples traversing the *PT* threads is recombined again into a single physical stream letting *OT* assign the tuples the *ets* timestamp and serve them in the same order as *IT* enqueued them in the queue above only after they have been added the *op* field by the corresponding *PT* thread.

8.4 Behavior of the Operators

As discussed in the previous sections, the binomial options pricing operator is parallelized in this third iteration of the financial stream processing engine by letting a set of parallel *PT* threads price options in parallel synchronizing by concurrently dequeuing tuples from the queue to which *IT* enqueues input tuples and adding them once they are processed to the *ScaleGate* instance they share with *OT*. Section 8.4.1 below elaborates on the formal description of the parallel *PT* threads and Section 8.4.2 concludes the presentation of this third iteration of the financial stream processing engine by extending the data-structures diagram introduced in Figure 8.3 to take into account the constant values that are provided to the binomial options pricing operator by the *PT* threads.

8.4.1 The Queue-*ScaleGate*-Based Multi-Threaded Binomial Options Pricing Operator

Similarly as the former *PT* thread from the first iteration of the stream processing engine simply kept retrieving tuples from the input queue, individually processing each of them using the operator whose pseudocode was introduced in Listing 6.1 in Section 6.4.1, and adding them to the output queue, each of the new *PT* threads keeps retrieving tuples from the input queue, processing them using the aforementioned operator, and adding them, this time, to the output *ScaleGate* instance.

Given that the operator executed by the *PT* threads is exactly the same as the one introduced in Section 6.4.1, and that the usage of the queue and *ScaleGate* data-structures is the key addition to the stream processing engine with respect to the first iteration, this section focuses on how the *PT* threads interact with the input queue which they share with *IT*, and the output *ScaleGate* instance which they share with *OT*.

Listing 8.1: Queue-*ScaleGate*-based multi-threaded binomial options pricing operator pseudocode

```
1 ID  // Thread local PT thread id
2 processTuples(inQueue, outScaleGate)
3   while (TRUE)
4     tuple = inQueue.deq()
5     tuple.op = priceOption(tuple.tp, os, om, rli, v)
6     outScaleGate.addTuple(tuple.seq, tuple, ID)
```

Listing 8.1 formally introduces the pseudocode describing the behavior of the parallel *PT* threads, each of which directly executes the `processTuples` procedure in lines 2-6. In addition to this, a unique identifier, `ID`, is assigned to each individual *PT* thread, as expressed in line 1, to allow them properly synchronize when concurrently adding tuples to the *ScaleGate* data-structure, `outScaleGate`.

The infinite loop in lines 3-6 determines how the *PT* threads process each tuple they retrieve from the input queue, `inQueue`. In line 4, a tuple is retrieved from the input queue, `inQueue`. It is important to take into account that given the semantics of the queue, as discussed before, only the thread which dequeued that tuple is able to process it. Other threads will dequeue other tuples but not the one dequeued by this thread. In line 5, the `priceOption` procedure introduced in Listing 6.1 in Section 6.3 is executed to add the corresponding tuple the *op* field, representing the option price assigned to the corresponding tuple based on its financial information. To do so, in this iteration of the stream processing engine, as in the two previous ones, only the trade price, *tp*, is retrieved from the tuple. The *os*, *om*, *rli*, and *v* values are provided as fixed constants with the same values for all the tuples. Finally in line 6, the tuple which was added the *op* field in the previous line is added to the output `ScaleGate` instance so that *OT* can retrieve it once it becomes a ready tuple.

Before concluding this section, it is worth analyzing the complexity of this operator in terms of time and memory. As it can be understood after a brief analysis of the pseudocode introduced in Listing 8.1, the main parameter affecting the cost of the operator in time and memory, apart from N as in Chapter 6, is the number of parallel *PT* threads, henceforth n.

In terms of execution time there is not an explicit batching overhead as there was in the previous iteration, as carefully analyzed in Section 7.4.1. However, adding a tuple to the *ScaleGate* data-structure in line 6 instead of enqueuing it in a concurrent queue as it was done in the previous two iterations, adds a non-negligible overhead in terms of execution time as the operation of inserting a node in the proper position of the skip list maintained by the *ScaleGate* instance involves:

- A random number generation in order to determine to how many levels of the underlying skip list the inserted *ScaleGate* node carrying the tuple will belong to, which represents an $O(1)$ overhead in terms of execution time.
- A search operation for each of the levels the *ScaleGate* node carrying the tuple will belong to. Under a non-saturation situation, assuming that the amount of nodes persisting in the *ScaleGate* instance at a time is proportional to the number of *PT* threads, $O(n)$, a reasonable expected overhead in terms of execution time, as far as n does not exceed by far 2 raised to the number of *ScaleGate* levels, is $O(\log n)$.
- An insertion operation for each of the levels the *ScaleGate* node carrying the tuple will belong to, which represents an $O(1)$ overhead under a non-saturation situation but redounds in a more intensive use of the atomic synchronization primitives due to having to insert the tuple in multiple levels of the skip list as

opposed to simply synchronizing with other threads to insert the tuple in the tail of a concurrent queue, which makes the algorithm more prone to cache misses which can represent a non-negligible $O(1)$ time overhead.

- The utilization of the Hazard Pointers based memory reclamation mechanism [34] introduced in Section 2.3.2. This mechanism has been simplified given the semantics of the *ScaleGate* data-structure replacing the first stage of the `scan` routine by a search of the oldest tuple having a hazard pointer referencing it, and modifying the second stage of the `scan` routine replacing the search operation by a comparison against the sequence number of the tuple which was selected in the modified first stage. In addition to this, only one hazard pointer per thread has been used in the C implementation of the *ScaleGate* instance. As a result, the average overhead of managing memory per tuple is upper bound by $O(\log n)$.

Altogether, the expected execution time overhead derived from the analysis of the costs of using an *ScaleGate* instance instead of a queue assuming a non-saturation situation is the one expressed in Equation 8.1.

$$O(\log n) \tag{8.1}$$

If this overhead is compared against the overhead for the batching based mechanism shown in Equation 7.1, it can be seen that the usage of the *ScaleGate* data-structure instead of the batching mechanism introduced in the previous chapter seems to be slightly more expensive in terms of execution time, and may have a negative impact on latency and throughput. However, this difference in practice, as it will be appreciated in the experimental results in Section 13.2.2, is negligible given the number of usable threads bounded by the physical resources of the used machines and the proportion between the overhead of pricing options and the overhead of using either the batching mechanism or *ScaleGate* ($O(N^2) >> O(\log n) \sim O(1)$).

However, it is worth noticing that, given the semantics of the *ScaleGate* data-structure, the tuples added to the *ScaleGate* instance cannot be retrieved by *OT* until a new tuple is added by the same thread which added it, and all the tuples in the *ScaleGate* instance with an earlier timestamp also satisfy this condition. This observation implies that even in the best possible case, independently of how the *PT* threads interact with each other, any tuple being processed by the stream processing engine needs to spend, after being served by *IT* and before reaching *OT*, at least the time required to price two options: the time it takes to price the option corresponding to that tuple executing in the corresponding thread the `priceOption` procedure introduced in Listing 6.1 in Section 6.3, and the time it takes to price the option corresponding to the next tuple processed by the same *PT* thread. As a result, even when using only one *PT* thread, the latency of processing a tuple would be duplicated with respect to the latency that would be achieved in the first and second iterations of the stream processing engine. This motivates the addition of a `NULL` tuple with the same sequence number as the properly added tuple in the *ScaleGate* instance every time a tuple is added. Given the small overhead of

using *ScaleGate* compared to the cost of the options pricing operator as discussed above, this modification to the algorithm presented in Listing 8.1 would result in an improvement in latency with a negligible impact on throughput, as it would unlock the tuples served and ready to be retrieved yet preserving the inter-thread synchronization semantics by letting the `NULL` tuple serve as a control tuple to prevent tuples added by other threads from becoming prematurely ready.

Listing 8.2: Queue-*ScaleGate*-based multi-threaded binomial options pricing operator pseudocode adding `NULL` control tuples

```
1 ID  // Thread local PT thread id
2 processTuples(inQueue, outScaleGate)
3   while (TRUE)
4     tuple = inQueue.deq()
5     tuple.op = priceOption(tuple.tp, os, om, rli, v)
6     outScaleGate.addTuple(tuple.seq, tuple, ID)
7     outScaleGate.addTuple(tuple.seq, NULL, ID)
```

Listing 8.2 extends the `processTuples` routine introduced in Listing 8.2 adding the aforementioned `NULL` tuple with the same sequence number as the properly added tuple in the *ScaleGate* instance every time a tuple is added. In line 7 it can be seen how the `NULL` tuple is added with the same sequence number as the tuple added in line 6 in both Listing 8.1 and Listing 8.2.

With this edition, the aforementioned duplicated latency problem in the case of one PT thread is mitigated getting back to similar latencies as in the previous two iterations for the single-threaded execution. However, it is still possible with this setup, as soon as two or more PT threads are used, to reach a situation in which one tuple needs to spend, after being served by IT and before reaching OT, at least the time required to price two options: if two PT threads, PT_1, and PT_2 concurrently add tuples to the *ScaleGate* instance, it is feasible to reach the following scenario:

- PT_1 and PT_2 just processed and inserted in the *ScaleGate* instance the tuples with sequence numbers $x - 1$ and $x - 2$.
- PT_1 retrieves the tuple with sequence number x from the queue.
- Almost simultaneously, PT_2 retrieves the tuple with the next sequence number, $x + 1$, from the queue.
- Both threads process the tuples simultaneously.
- PT_1 inserts the tuple with sequence number x in the *ScaleGate* instance and immediately after that, it inserts the `NULL` tuple with the same sequence number, x.
- Almost simultaneously, PT_2 inserts the tuple with sequence number $x + 1$ in the *ScaleGate* instance and immediately after that, it inserts the `NULL` tuple with the same sequence number, $x + 1$.

In the scenario above:

- The tuples with sequence number $x-1$ and $x-2$, both the proper tuples and the `NULL` tuples, must become ready, if they were not already ready, right after the insertion of the non `NULL` tuples with sequence numbers x and $x+1$.
- The tuple with sequence number x becomes ready right after the insertion of the `NULL` tuple with sequence number x, thus, being able to be directly retrieved by OT after having spent, since retrieved from the queue, the time needed to execute the `priceOption` procedure only once plus the small overhead to add this tuple and the `NULL` tuple to the *ScaleGate* instance.
- The tuple with sequence number $x+1$, as opposed to the one with sequence number x, needs to wait until PT_1 adds another tuple with a sequence number greater than x to let the `NULL` tuple with sequence number x, which allowed the non `NULL` tuple with sequence number x to become ready right after being added, become ready. As a result, the non `NULL` tuple with sequence number $x+1$ needs to spend, since retrieved from the queue, at least twice the time needed to execute the `priceOption` procedure before being successfully retrieved by OT.

The analysis presented above can be trivially extended to three or more PT threads arriving at the conclusion that the expected time spent by each tuple in the stream processing engine, under a non-saturation scenario, between the instant of time it is added to the queue by IT and the instant of time it is finally retrieved by OT is lower bounded by the time consumed by the `processTuple` procedure to process one tuple ($O(N^2)$) plus the overhead of adding two tuples to the *ScaleGate* instance ($O(\log n)$), and upper bounded by the time consumed by the `processTuple` procedure to process up to two tuples ($O(N^2)$), plus the overhead of adding three tuples to the *ScaleGate* instance ($O(\log n)$). With the exception of the single-threaded setup in which the upper bound is equal to the aforementioned lower bound following the reasoning which motivated above the addition of the `NULL` tuple in line 7 in Listing 8.2. Altogether, the expected latency for each tuple is the one expressed in Equation 8.2.

$$L(n) = O(N^2) + O(\log n) \tag{8.2}$$

If this expected latency is compared against the expected latency for the batching based approach introduced in Equation 7.4 in the previous chapter, it is tempting to conclude that the parallelization approach introduced in this chapter will produce better latency results. However, it is important to keep in mind that the strongest factor in both equations is the $O(N^2)$ contribution derived from executing the `priceOption` procedure to price the option contracts, and as discussed above, the semantics of the *ScaleGate* data-structure, even when using the aforementioned `NULL` tuple mechanism, make the constant controlling this contribution to the latency greater than in the previous iteration. In other words, if the contribution to the latency derived from the usage of the queue plus *ScaleGate* mechanism is separated from the contribution to the latency derived from the actual pricing of the option contract associated to a given input tuple, the $O(N^2)$ contribution will not disappear as it did in the transition from Equation 7.4 to Equation 7.5 in the

previous chapter. Instead, the expression of the overhead in terms of latency for each tuple in this iteration of the stream processing engine is the one expressed in Equation 8.3.

$$O(N^2) + O(\log n) \tag{8.3}$$

As it can be seen, Equations 8.2 and 8.3 have the same big O notation expression. The difference, as discussed above, can be found in the constant controlling the $O(N^2)$ contribution. This contribution not being zero in Equation 8.3 as it was in Equation 7.5 is the main reason why, even though as discussed above, Equation 8.2 seemed to deem the current iteration more efficient in terms of latency than the previous one, in the experimental results presented in Section 13.2.2, the previous iteration of the stream processing engine leads to lower latency records.

In terms of throughput, given that the parallel PT threads subdivide the input physical stream into n physical streams, and that the synchronization overhead derived from using *ScaleGate* is orders of magnitude smaller than the single-threaded operator overhead ($O(\log n) << O(N^2)$), the throughput is expected to grow linearly with the number of threads as expressed in Equation 8.4.

$$T(n) = n \cdot T(1) \tag{8.4}$$

Altogether, the expected latency and throughput of this iteration of the stream processing engine can be plotted as a function of the number of parallel PT threads used.

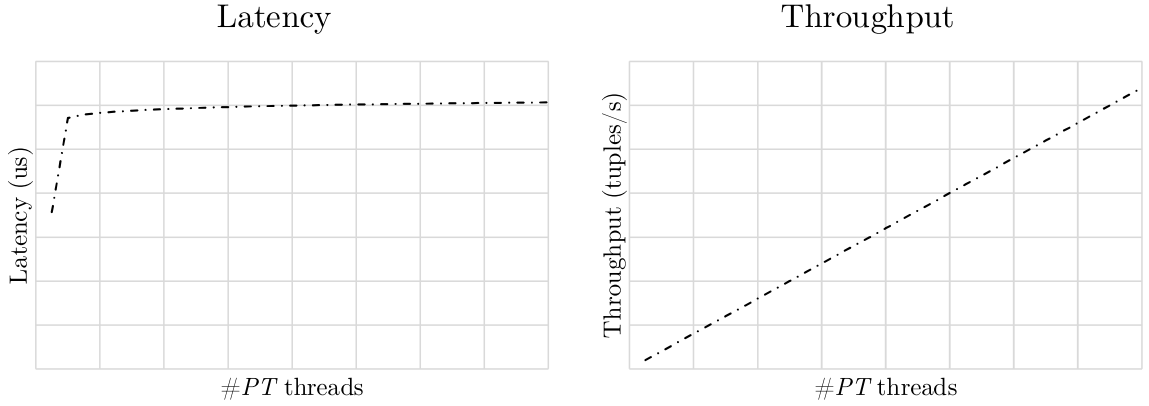


Figure 8.4: Latency and throughput as a function of the number of parallel PT threads

Figure 8.4 plots the expected latency and throughput of this iteration of the stream processing engine. It is worth observing how in the transition from one PT thread to two or more PT threads the latency overhead dramatically increases in comparison the further latency increases due to the addition of the $O(N^2)$ contribution in Equation 8.3. In terms of expected throughput, the tendency is similar to the previous iteration. However, as it will be shown in Section 13.2.2, the current

iteration will scale better in terms of throughput under scenarios in which not all the processing units have the same conditions given the load balancing flexibility achieved by the combined use of the queue and *ScaleGate*.

The overhead in terms of memory is easier to assess:

- The thread local `ID` assigned to each of the n PT threads occupy in memory $O(1)$ space per PT thread, $O(n)$ space globally.
- The temporary variable `tuple` initialized in line 4 represents an $O(1)$ space overhead per PT thread, consequently an $O(n)$ space overhead at the global stream processing engine level.
- The underlying skip list maintained by the *ScaleGate* data-structure has a similar size as the underlying concurrent data-structures maintained by the queues.

Overall, the overhead of the queue-*ScaleGate*-based multi-threaded version of the operator in terms of memory is the one expressed in Equation 8.5.

$$O(n) \tag{8.5}$$

Adding the overhead above to the cost in terms of memory of the single-threaded volatility operator introduced in Equation 6.10 in Section 6.4.1, the total cost in terms of memory of the queue-*ScaleGate*-based multi-threaded version of the options pricing operator is the one expressed in Equation 8.6.

$$O(n) \cdot O(N) + O(n) \tag{8.6}$$

8.4.2 Integrating the Queue-*ScaleGate*-Based Multi-Threaded Binomial Options Pricing Operator

Getting back to the stream processing engine diagram from Section 8.3, it is worth noticing that, as anticipated in Section 8.4.1, and the same way as in the previous two iterations of the stream processing engine, out of the six values that the binomial options pricing operator uses as an input, tp, os, om, rli, v, and N, only one, tp, can be retrieved by the PT threads from the tuples served by IT to the queue it shares with them as described in Section 8.3. In this third iteration of the financial stream processing engine, as in the previous two ones, the rest of the values are simply taken as constants with the default values introduced in Section 6.4.2.

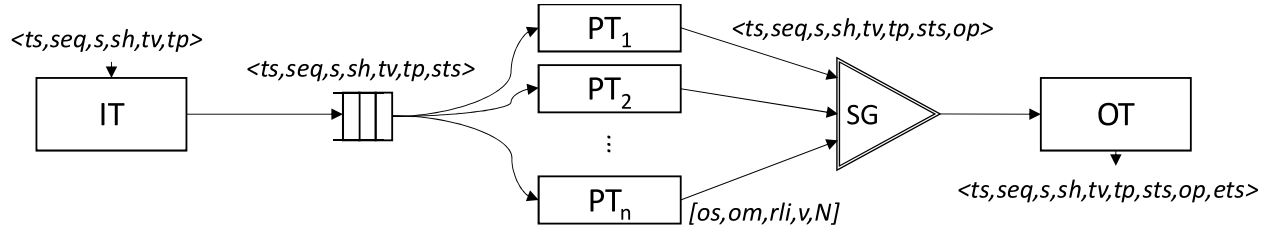


Figure 8.5: Operators and used constants

Figure 8.5 extends the data-structures diagram introduced in Figure 8.3 to represent the aforementioned constant values between square brackets close to the boxes representing the PT threads, each of which executes the single-threaded binomial options pricing operator introduced in Chapter 6.

Overall, this third iteration of the stream processing engine, as the second one also did, and the fourth one will do, responds to the first line of improvement outlined in the last paragraph in Section 6.4.2 concluding Chapter 6.

9

ScaleGate-*ScaleGate*-Based Multi-Threaded Binomial Options Pricing

In the previous chapter, an alternative approach to the options pricing operator parallelization problem was introduced and integrated in the stream processing engine. This approach involved temporarily partitioning the physical input stream of tuples into a set of physical disjoint streams by letting a set of parallel *PT* threads compete to retrieve the tuples served by the input thread to the input queue and using a *ScaleGate* instance to let the parallel *PT* threads collaboratively re-combine the physical streams of tuples into one output physical stream having the tuples in exactly the same order as they were originally served by the input thread. As a result, the synchronization among threads became distributed instead of centralized as in the batching based approach discussed in Chapter 7, and the load balancing properties of the stream processing engine became more flexible because the competitive way of retrieving tuples from the input queue would allow faster processing units to retrieve and process more tuples than slower processing units. However, the expected impact on latency of this solution, as extensively discussed in Section 8.4.1, could be potentially even higher than that of the batching based parallelization approach introduced in Chapter 7 due to the semantics of the output *ScaleGate* instance.

In this chapter, a new alternative approach is introduced by replacing the input queue in the previous approach by another *ScaleGate* instance and letting the *PT* threads determine whether to process or not a tuple based on their thread identifier and the tuple sequence number. Letting the *PT* threads retrieve the tuples from the *ScaleGate* instance instead of the former queue makes it possible for all the tuples to be visible by all the threads, which enables the implementation of a heartbeat mechanism to expedite the output *ScaleGate* behavior preserving the output stream ordering requirements. With this, the synchronization effort which became distributed in the previous iteration of the stream processing engine remains distributed in this new iteration avoiding performance bottlenecks, and the latency expectations discussed in Section 8.4.1 can be improved thanks to the heartbeat mechanism.

The following sections elaborate on how the queue instance in the previous iteration of the stream processing engine can be replaced by a *ScaleGate* instance,

and how it would affect the way the PT threads interact with the input and output data-structures.

9.1 Involved Threads

As anticipated above, this fourth iteration of the stream processing engine is a modification of the previous one in which the data-structure used to transmit tuples from the input thread to the option pricing threads is the main structural change. For this reason, the way to arrange the different threads in this iteration is exactly the same as in the previous one.

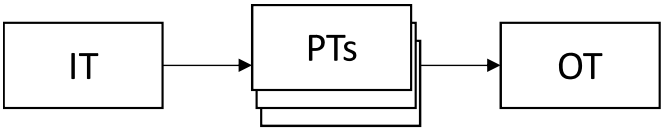


Figure 9.1: Involved threads

Figure 9.1 outlines how threads are arranged in this fourth version of the financial stream processing engine. As it can be seen, IT, the PT threads and OT remain as they were introduced in Figure 8.1 in Section 8.1.

9.2 Structure of the Tuples

Similarly as it happened with the involved threads in the previous section, the structure of the tuples in this fourth iteration of the financial stream processing engine remains the same as in the previous iteration.

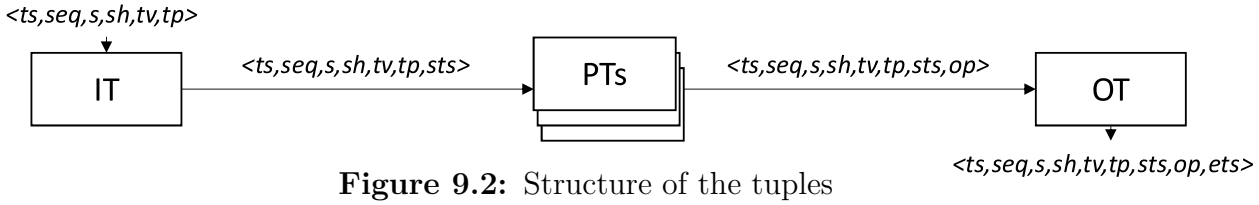


Figure 9.2: Structure of the tuples

Figure 9.2 extends the involved threads diagram introduced in Figure 9.1 specifying the structure of the tuples in each transition from one thread to another as well as the structure of the tuples read by IT, and the tuples output by OT.

As it can be seen, the structure of the tuples retrieved by IT, the tuples transferred from IT to the parallel PT threads, the tuples served by the parallel PT threads to OT, and the tuples output by OT is exactly the same as the structure of the tuples retrieved by IT, the tuples transferred from IT to the parallel PT threads, the tuples served by the parallel PT threads to OT, and the tuples output by OT in Figure 8.2.

9.3 Used Data-Structures

The key difference between this fourth iteration of the financial stream processing engine and the previous iteration discussed in Chapter 8 is the usage of a *ScaleGate* instance instead of a queue to let the *IT* and *PT* threads communicate.

This change with respect to the previous iteration implies that the input physical stream served by *IT* will no longer be implicitly partitioned by the *PT* threads when retrieving tuples from the former input queue. Instead, all the *PT* threads will have access to the full physical stream of tuples served by *IT* thanks to the input *ScaleGate* instance semantics. This allows them to explicitly partition the input physical stream into a set of physical streams output by each *PT* thread and generate additional control streams based on the tuples the *PT* threads receive but do not assign a price to in order to expedite the behavior of the output *ScaleGate* instance which represented in the previous iteration, as analyzed in Section 8.4.1, a potential latency bottleneck.

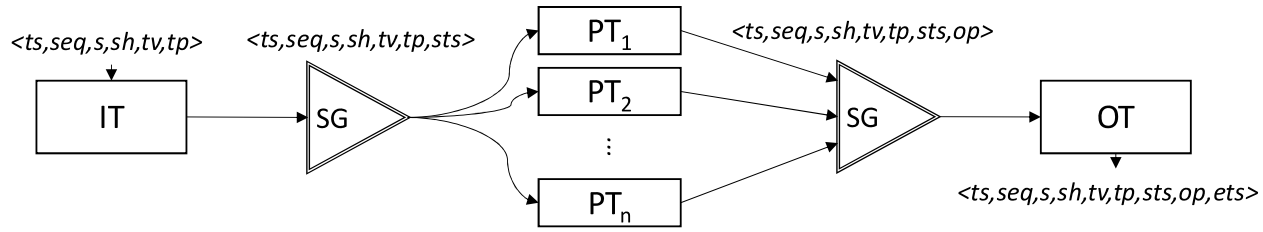


Figure 9.3: Used data-structures

Figure 9.3 extends the structure of the tuples diagram introduced in Figure 9.2 specifying the data-structures used in each transition from one thread to another:

- *IT* **to the** *PT* **threads** ***ScaleGate***: As reasoned above, *IT* and the parallel *PT* threads share in this fourth iteration of the stream processing engine a *ScaleGate* instance instead of a queue having *IT* as the only writer thread and the *PT* threads as a set of multiple readers. This way, as anticipated above, all the tuples are now received by all the *PT* threads in the same order they are served by *IT* making it possible for them to expedite the behavior of the output *ScaleGate* instance using the heartbeat mechanism which will be introduced in Section 9.4.1.
- *PT* **threads to** *OT* ***ScaleGate***: The same way it was done in the previous iteration as discussed in Section 8.3, the *PT* threads and *OT* keep sharing an instance of *ScaleGate* in which the *PT* threads act as a set of multiple writers and *OT* acts as a single reader. This way, the set of physical streams of tuples output by the *PT* threads is recombined again into a single physical stream letting *OT* assign the tuples the *ets* timestamp and serve them in the same order as *IT* added them to the input *ScaleGate* instance only after they have been added the *op* field by the corresponding *PT* thread.

9.4 Behavior of the Operators

As discussed in the previous sections, the binomial options pricing operator is parallelized in this fourth iteration of the financial stream processing engine in a manner very similar to how it was parallelized in the previous iteration. A set of parallel *PT* threads price options in parallel synchronizing by retrieving from the input *ScaleGate* instance all the tuples added by *IT*, making sure one and only one *PT* thread assign an option price to each tuple by using the tuple sequence number and the threads unique identifier to determine whether to assign a tuple a price or not, and adding the priced tuples once they are processed to the output *ScaleGate* instance they share with *OT*. Section 9.4.1 below elaborates on the formal description of the parallel *PT* threads and Section 9.4.2 concludes the presentation of this fourth iteration of the financial stream processing engine by extending the data-structures diagram introduced in Figure 9.3 to take into account the constant values that are provided to the binomial options pricing operator by the *PT* threads.

9.4.1 The *ScaleGate-ScaleGate*-Based Multi-Threaded Binomial Options Pricing Operator

In contrast with the previous iteration of the financial stream processing engine in which the *PT* threads simply retrieved tuples from the input queue, added them an option price, and added them to the output *ScaleGate* instance, the *PT* threads in this fourth iteration of the stream processing engine have two main additional task to perform given the fact that all of them have access to all the tuples served by *IT*:

- The *PT* threads have to determine whether they should assign a tuple an option price or not based on their unique thread identifier and the tuple sequence number in order to guarantee that for each input tuple one and only one *PT* thread assigns it an option price and adds it to the output *ScaleGate* instance.
- The *PT* threads can also make use of the information contained in the tuples they do not assign a price to, in particular their sequence number, in order to expedite the behavior of the output *ScaleGate*. They can do so by adding `NULL` tuples with the same sequence number as the tuples they do not assign an option price to letting the non-`NULL` tuples added to the output *ScaleGate* instance faster become ready without having to wait for all the *PT* threads to price up to two options as discussed in Section 8.4.1. This way of expediting the output *ScaleGate* behavior will be henceforth referred to as the heartbeat mechanism, being the aforementioned `NULL` tuples heartbeat tuples.

Listing 9.1: *ScaleGate-ScaleGate*-based multi-threaded binomial options pricing operator pseudocode

```
1 ID  // Thread local PT thread id in the range {0,...,n-1}
2 processTuples(inScaleGate, outScaleGate, H)
3   cnt = 0
4   while (TRUE)
5     tuple = inScaleGate.getNextReadyTuple(ID)
```

```
6       if (tuple.seq mod n == ID)
7         tuple.op = priceOption(tuple.tp, os, om, rli, v)
8         outScaleGate.addTuple(tuple.seq, tuple, ID)
9         outScaleGate.addTuple(tuple.seq, NULL, ID)
10      else if (H > 0)
11        cnt = cnt + 1
12        if (cnt == H)
13          outScaleGate.addTuple(tuple.seq, NULL, ID)
14          cnt = 0
```

Listing 9.1 formally introduces the pseudocode describing the behavior of the parallel *PT* threads, each of which executes the `processTuples` procedure in lines 2-14. In addition to this, as in the previous iteration, a unique identifier, `ID`, is assigned to each individual *PT* thread, as expressed in line 1, to allow them properly synchronize when retrieving tuples from the input *ScaleGate* instance, `inScaleGate`, determining whether they should add the tuples a price or not, and concurrently adding the tuples they add a price to the output *ScaleGate* instance, `outScaleGate`.

The infinite loop in lines 4-14 determines how the *PT* threads process each tuple they retrieve from the input *ScaleGate* instance, `inScaleGate`. In line 5, the next ready tuple input by *IT* is retrieved from the input *ScaleGate* instance, `inScaleGate`. It is worth taking into account that even though the pseudocode above represents this operation straight forward in one line, the retrieval of a tuple from a *ScaleGate* instance in practice, given its non-blocking nature, conveys a retry loop in which the next ready tuple is requested iteratively until it is ready, having the *ScaleGate* instance returning a `NULL` tuple, which needs to be ignored by the caller, every time the `getNextReadyTuple` procedure is called and there is not a non-`NULL` tuple ready to be provided to the caller. This observation applies also to the `deq` operation in queues in case a non-blocking implementation is used. It is also important to take into account that given the semantics of *ScaleGate*, as discussed before, each of the tuples added by *IT* to the input *ScaleGate* instance, `inScaleGate`, will eventually be retrieved by all the *PT* threads in line 5.

The last observation above justifies the check performed in line 6, which corresponds to the aforementioned task of determining whether a tuple should be added an option price or not by the current thread according to the tuple sequence number and the current thread identifier. As it can be seen, this check consists on basically checking whether the identifier modulo the number of *PT* threads, henceforth n, matches the identifier of the thread performing the check belonging to the range $\{0, ..., n-1\}$. This way, the tuples which are added by *IT* to `inScaleGate` with consecutive sequence numbers are paired to *PT* threads for the option pricing task in a round robin fashion achieving a fair load balance assuming similar processing capacities in all the nodes running the *PT* threads, and allowing each tuple to start being processed by the corresponding *PT* thread as soon as they are retrieved from `inScaleGate` and the aforementioned $O(1)$ time complexity check is performed. The

PT thread for which this check is successful for a given tuple moves forward to line 7 in which the `priceOption` procedure introduced in Listing 6.1 in Section 6.3 is executed to add the corresponding tuple the *op* field, representing the option price assigned to the corresponding tuple based on its financial information. To do so, as in the previous three iterations of the stream processing engine, only the trade price, *tp*, is retrieved from the tuple. The *os*, *om*, *rli*, and *v* values are provided as fixed constants with the same values for all the tuples. Finally, in line 8, the tuple which was added the *op* field in the previous line is added to the output `ScaleGate` instance so that *OT* can retrieve it once it becomes a ready tuple, and in line 9, a `NULL` tuple is added with the same sequence number as the tuple added in line 8 to mitigate the duplicated latency problem in the case of having a single *PT* thread which was discussed in Section 8.4.1. This last action in also performed for the same reason by *IT* when adding tuples to `inScaleGate` in this iteration of the stream processing engine and all further iterations in which it adds tuples to a *ScaleGate* instance.

In case the check in line 6 fails for the current thread, it means another thread will take care of pricing the corresponding option but the current thread can still use the sequence number of the retrieved tuple to expedite the output *ScaleGate* instance, `outScaleGate`, behavior helping the non-`NULL` tuple added by the thread which succeeded in line 6 earlier become ready. This is the motivation behind the heartbeat mechanism executed by the non successful in line 6 threads which have the chance to execute lines 10-14 in Listing 9.1. This heartbeat mechanism is controlled by the *H* input parameter which determines how many non-priced tuples should elapse between two consecutive heartbeat tuples. A value of 0 or lower implying the heartbeat mechanism is simply not used as the check in line 10 would fail triggering the next iteration of the loop in lines 4-14, a value of 1 letting every non-priced tuple trigger a heartbeat tuple, and a value *H* greater than 1 letting the *PT* threads output a heartbeat tuple every *H* non-processed tuples. To allow this, the counter of non-priced tuples, `cnt`, initially initialized to zero in line 3, is updated in line 11 every time a tuple is deemed not to be priced by the current thread according to the check in line 6. This update is followed by the check in line 12 to determine whether or not the specified amount of non-priced tuples seen by the current thread has reached the *H* threshold. In case the check in line 12 succeeds, the thread is allowed to add a heartbeat tuple to the output *ScaleGate* instance executing line 13 which adds a `NULL` tuple to `outScaleGate` with the same sequence number as the non-priced tuple. As a result, if a non-`NULL` tuple was waiting to be ready in `OutScaleGate` with a sequence number higher than the last `NULL` or non-`NULL` tuple added by the current thread and a sequence number lower than or equal to the sequence number of the heartbeat tuple, the thread which adds the heartbeat tuple would immediately stop preventing that tuple from becoming ready yet preserving the ordering requirements in the output stream retrieved by *OT*. Finally, in line 14 the `cnt` counter is set to zero again as in line 3 to let another set of *H* non-priced tuples be retrieved by the current thread before adding another heat beat tuple to `OutScaleGate`.

Before concluding this section, it is worth analyzing the complexity of this operator in terms of execution time and memory. As it can be understood after a brief analysis of the pseudocode introduced in Listing 9.1, the main parameters affecting the cost of the operator in terms of execution time and memory, apart from N as in the previous chapters, are the number of parallel PT threads, n, and the H parameter.

In terms of execution time it is worth starting the analysis referencing back to the $O(\log n)$ expected execution time overhead derived from the analysis of the costs of using the `outScaleGate` *ScaleGate* instance instead of a queue assuming a non-saturation situation which was expressed in Equation 8.1. In the current iteration of the stream processing engine, the H parameter with values tending to 1 from above would result in a more intensive usage of the output *ScaleGate* instance potentially multiplying by up to n the amount of nodes inserted in `outScaleGate`. However, as the purpose of these nodes is to expedite the behavior of that *ScaleGate* instance helping tuples faster become ready as discussed above, and assuming once more a non-saturation situation, the amount of nodes maintained at a time in the underlying skip list in `outScaleGate` is expected to remain in a similar range as in the previous iteration leaving the `outScaleGate` usage overhead with the same big O notation expression as in the previous iteration controlled perhaps by a slightly bigger constant as expressed in Equation 9.1.

$$O(\log n) \tag{9.1}$$

The experimental results in Section 13.2.2 actually confirm the expectations above as it will be seen when comparing the throughput achieved using this iteration of the stream processing engine with $H = 0$ and $H = 1$.

In this iteration of the stream processing engine, the input queue is also replaced by a *ScaleGate* instance. However, having IT acting as a single writer, the insertions performed by IT will not have the $O(\log n)$ search overhead they have in the output *ScaleGate* instance to which multiple threads concurrently add tuples. In this sense, only the memory management overhead discussed in Section 8.4 lets the input *ScaleGate* overhead remain as expressed in Equation 9.2.

$$O(\log n) \tag{9.2}$$

It is also worth mentioning that given the semantics of *ScaleGate*, the multiple readers, in this case the PT threads, do not have to compete with each other attempting to retrieve the head tuple and retrying if another concurrent thread succeeded doing so at the same time. This should reduce the contention overhead with respect to the usage of a queue as an input data-structure.

To conclude the execution time overhead analysis, it is worth noticing that the newly added logic with respect to the previous iteration in lines 6, and 10-14 in Listing 9.1 consists on a series of $O(1)$ operations performed by each PT thread for each retrieved tuple. Adding this overhead to the overheads of using the input

and output *ScaleGate* instances, the expected execution time overhead assuming a non-saturation situation is the one expressed in Equation 9.3.

$$O(\log n) \tag{9.3}$$

The expression above of the expected execution time overhead is exactly the same as in the previous iteration. However, its impact in latency as far as the heartbeat mechanism is enabled is expected to be strongly different from that of the previous iteration analyzed in Section 8.4.1. Given that the latency for each individual tuple in the first iteration of the stream processing engine is $O(N^2)$ as expressed in Equation 6.9. And taking into account that the heartbeat mechanism reduces unnecessary waiting time for the tuples in the output *ScaleGate* instance, the overall overhead in terms of latency for each tuple in this iteration of the stream processing engine according to 9.3 should be $O(\log n)$ instead of $O(N^2) + O(\log n)$ as expressed in Equation 8.3 for the previous iteration of the stream processing engine. This means that the expected latency for each tuple is expected to be the one expressed in Equation 9.4 but having the $O(N^2)$ contribution a lower control constant than in Equation 8.2.

$$L(n) = O(N^2) + O(\log n) \tag{9.4}$$

In terms of throughput, given that the options pricing overhead is equally distributed among the n parallel PT threads, and that the synchronization overhead derived from using the input and output *ScaleGate* instances and executing the newly introduced control logic with respect to the previous iteration is orders of magnitude smaller than the single-threaded operator overhead ($O(\log n) << O(N^2)$), the throughput is expected to grow linearly with the number of threads as expressed in Equation 9.5.

$$T(n) = n \cdot T(1) \tag{9.5}$$

Altogether, the expected latency and throughput of this iteration of the stream processing engine having H greater than 0 and close to 1 can be plotted as a function of the number of parallel PT threads used.

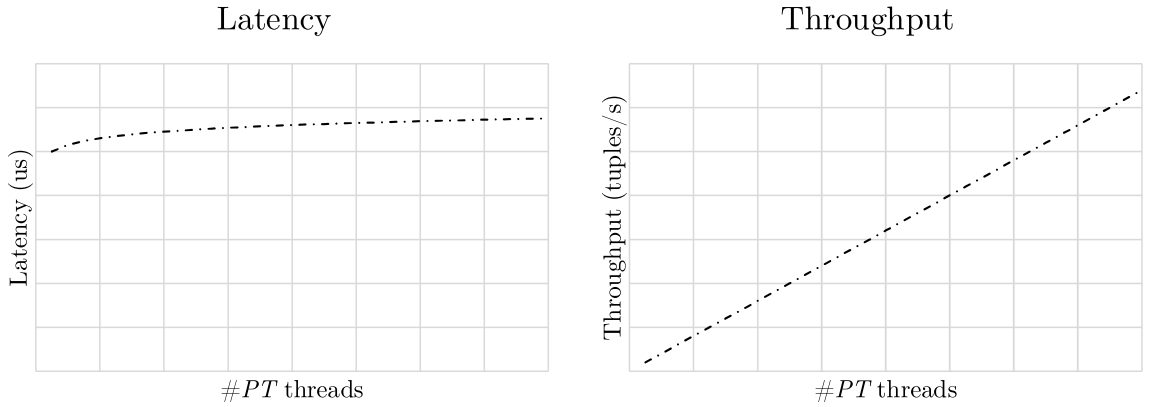


Figure 9.4: Latency and throughput as a function of the number of parallel *PT* threads

Figure 9.4 plots the expected latency and throughput of this iteration of the stream processing engine. It is worth observing how in the transition from one *PT* thread to two or more *PT* threads the latency overhead no longer dramatically increases in comparison the further latency increases as it did in the previous iteration of the stream processing engine. This is achieved thanks to the usage of the aforementioned heartbeat mechanism made possible by the fact that all the *PT* threads have access to all the tuples served by *IT*. In terms of expected throughput, the tendency is similar to the previous iterations.

The overhead in terms of memory is even easier to assess:

- The thread local `ID` assigned to each of the n *PT* threads occupy in memory $O(1)$ space per *PT* thread, $O(n)$ space globally.
- The temporary variable `tuple` initialized in line 5 as well as the temporary value cnt initialized in line 3 and the input parameter, H, represent an $O(1)$ space overhead per *PT* thread, consequently an $O(n)$ space overhead at the global stream processing engine level.
- The underlying skip lists maintained by the *ScaleGate* data-structures have a similar size as the underlying concurrent data-structures maintained by the former queues.

Overall, the overhead of the *ScaleGate-ScaleGate*-based multi-threaded version of the operator in terms of memory is the one expressed in Equation 9.6.

$$O(n) \tag{9.6}$$

Adding the overhead above to the cost in terms of memory of the single-threaded volatility operator introduced in Equation 6.10 in Section 6.4.1, the total cost in terms of memory of the *ScaleGate-ScaleGate*-based multi-threaded version of the options pricing operator is the one expressed in Equation 9.7.

$$O(n) \cdot O(N) + O(n) \tag{9.7}$$

9.4.2 Integrating the *ScaleGate-ScaleGate*-Based Multi-Threaded Binomial Options Pricing Operator

Getting back to the stream processing engine diagram from Section 9.3, it is worth noticing that, as anticipated in Section 9.4.1, and the same way as in the previous three iterations of the stream processing engine, out of the six values that the binomial options pricing operator uses as an input, tp, os, om, rli, v, and N, only one, tp, can be retrieved by the PT threads from the tuples served by IT to the *ScaleGate* instance it shares with them as described in Section 9.3. The rest of the values are simply taken as constants with the default values introduced in Section 6.4.2. In addition to this, in this fourth iteration of the stream processing engine the H parameter needs to be specified and provided to the parallel PT threads to determine whether to use or not the heartbeat mechanism and how exhaustively to use it.

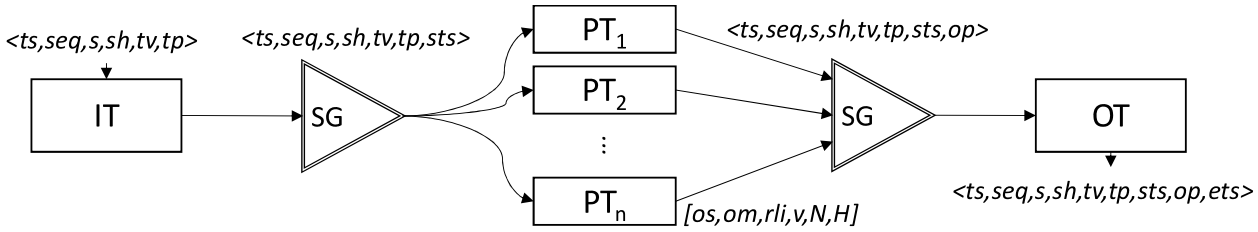


Figure 9.5: Operators and used constants

Figure 9.5 extends the data-structures diagram introduced in Figure 9.3 to represent the aforementioned constant values between square brackets close to the boxes representing the PT threads, each of which executes the single-threaded binomial options pricing operator introduced in Chapter 6. The H parameter has also been added to the square brackets as a value determining the behavior of the operator more widely understood as the actions performed by the PT threads for each input tuple, which involves executing the heartbeat mechanism as explained in the previous section.

Overall, this fourth iteration of the stream processing engine, as the second and third ones also did, responds to the first line of improvement outlined in the last paragraph in Section 6.4.2 concluding Chapter 6. The next three chapters address the second line of improvement, namely providing the v, os, om, and rli values in a more useful way in order to let the stream processing engine price any option with any settings instead of just options with always the same fixed settings.

10

Single-Threaded Volatility Aggregation

In the previous three chapters, three different approaches to the options pricing operator parallelization problem was introduced and integrated in the stream processing engine providing three alternative solutions for the first line of improvement of the stream processing engine outlined in Section 6.4.2. Even though these three approaches achieve an expected throughput increase proportional to the number of process threads at the cost of a small latency overhead, only the trade price, tp, is taken from the tuples representing the actual market behavior whereas many other constants such as the volatility, option strike, option maturity, and risk-less interest rate, are simply taken as constants, which is not a realistic solution in a dynamic market setup.

Among the aforementioned constants, volatility, is especially relevant for the proper functioning of an options pricing mechanism as it is the value which summarizes the speed at which the underlying stock prices change in the market, which determines the behavior of the underlying binary tree model as discussed in [8, 4]. It has been extensively argued in the literature, as anticipated in Section 4.1.2, that the market volatility is not static at all and changes with time [1, 18, 39]. For this reason, it is important to be able to keep track of its evolution in time, which is the problem this chapter focuses on. In particular, a sliding-window-based volatility aggregation mechanism will be introduced, understanding market volatility as the standard deviation [41] of the most recently observed trade prices for each symbol.

The following sections elaborate on how the stream processing engine as it was defined in the previous chapter can be extended by adding a volatility aggregation thread in the pipeline right after IT and before the PT threads, so that the tuples can be extended before calculating an option price with a volatility field obtained from the aggregated calculation of the volatility of all the tuples contributing to the most recent sliding-window the corresponding tuple does not contribute to.

10.1 Involved Threads

As anticipated above, the threads introduced in the previous chapter to retrieve tuples from the input dataset and serve them to the process threads, to calculate option prices, and to get the tuples output by the option pricing threads and output

them, remain in this fifth iteration as they were introduced in the previous one with the addition of a new volatility aggregation thread which will process the tuples after they have been assigned the process start timestamp by the input thread and before the option pricing threads start pricing the corresponding options.

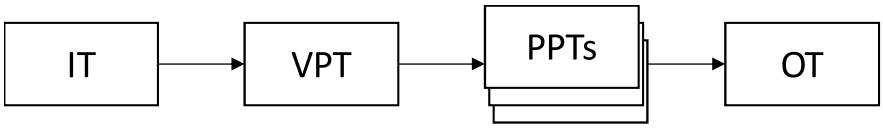


Figure 10.1: Involved threads

Figure 10.1 outlines how threads are arranged in this fifth version of the financial stream processing engine. As anticipated above, the new addition is the volatility aggregation thread placed between the input thread and the parallel option pricing threads whose name has been updated in this fifth iteration given the fact that now there are two different kinds of process threads, the volatility aggregation thread and the options pricing threads:

- **Volatility Aggregation Process Thread** (VPT): the volatility aggregation process thread, henceforth referred to as VPT, is the aforementioned addition to the stream processing engine pipeline. This new thread will maintain a set of sliding-windows for each traded symbol in order to aggregate the volatility of the most recent trades and extend the tuples served by IT with an aggregated volatility result so that the option pricing threads, which will process the tuples afterwards, do so with a realistic and updated measure of the underlying stock volatility.
- **Options Pricing Parallel Process Threads** (PPT): the parallel option pricing process threads, henceforth referred to as PPT, have basically the same function as the former PT threads introduced in Section 9.1, namely pricing options. The only difference apart from the newly assigned name is that these threads will no longer provide a predefined constant volatility value to the options pricing algorithm. They will instead provide the updated volatility value assigned by the volatility aggregation thread to the tuples.

10.2 Structure of the Tuples

As anticipated above, the new VPT thread extends the structure of the tuples it processes adding a new field summarizing the market volatility which needs to be taken into account to perform the options pricing calculation for each individual tuple instead of using a default, constant and outdated value.

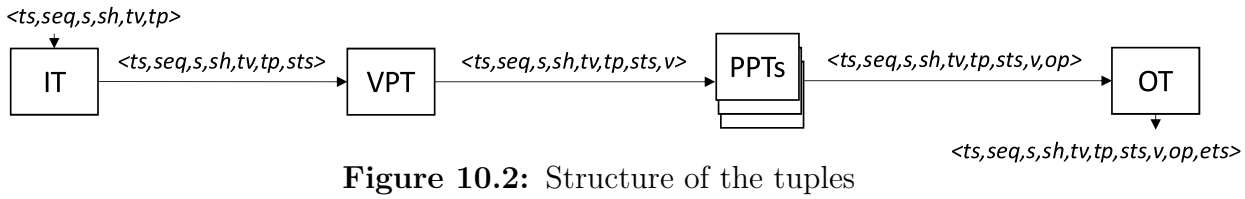


Figure 10.2: Structure of the tuples

Figure 10.2 extends the involved threads diagram introduced in Figure 10.1 specifying the structure of the tuples in each transition from one thread to another as well as the structure of the tuples read by IT, and the tuples output by OT.

As it can be seen, with the addition of VPT, the structure of the tuples processed by the PPT threads and output by OT is extended. The tuples retrieved by IT and the tuples received by VPT have exactly the same structure as the tuples IT retrieved and the former PT threads received in the previous iteration of the financial stream processing engine. The volatility aggregation thread, VPT, adds a field to the tuples it process representing the aggregated volatility for the given input tuple, which extends the tuples received by the PPT threads from seven to eight fields, <ts, seq, s, sh, tv, tp, sts, v>:

- **Volatility** (v): the volatility field, henceforth referred to as v, is the value representing the volatility aggregated from the most recent tuples with the same symbol as the tuple processed by the VPT thread. This field is internally modeled as a `double` value accounting for the standard deviation of the distribution of the finite discrete distribution of the most recent trade prices weighted by the trade volumes.

  Section 10.4 further elaborates on how this volatility is calculated including which input values are taken by the operator, where are they taken from and how are they used to produce a volatility value.

As in the previous iterations, the option pricing threads, PPT, add the option price, op, field to the tuples the VPT thread provides them with, extending its structure from eight to nine fields, <ts, seq, s, sh, tv, tp, sts, v, op>. And the OT thread adds the process end timestamp, ets, field completing the structure of the tuples output by the stream processing engine with ten fields, <ts, seq, s, sh, tv, tp, sts, v, op, ets>.

10.3 Used Data-Structures

Given the single-threaded nature of the volatility aggregation mechanism introduced in this fifth iteration of the stream processing engine, the VPT thread has the same synchronization needs the single-threaded volatility operator had in the first iteration introduced in Section 6.3. It will synchronize with IT using a single writer, single reader concurrent queue and with the PPT threads the same way as the former IT thread synchronized with the former PT threads in the previous iteration of the stream processing engine because what VPT outputs is the same stream output by IT with one more field in each tuple informing about the market volatility at the time of pricing the options.

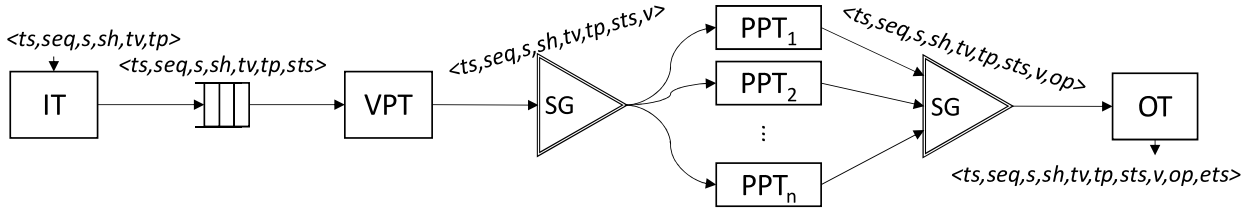


Figure 10.3: Used data-structures

Figure 10.3 extends the structure of the tuples diagram introduced in Figure 10.2 specifying the data-structures used in each transition from one thread to another. As anticipated above, IT and VPT share a concurrent single writer, single reader queue exactly the same way as the former IT and PT did in the first iteration of the financial stream processing engine as discussed Section 6.3. VPT and the PPT threads share a *ScaleGate* instance having VPT as a single writer and the PPT threads as a set of multiple readers exactly the same way as the former IT and PT threads did in the fourth iteration of the financial stream processing engine as discussed in Section 9.3. Finally, the PPT threads and OT share a *ScaleGate* instance also the same way as the former PT threads and OT did in the fourth iteration of the financial stream processing engine as discussed in Section 9.3.

10.4 Behavior of the Operators

As anticipated in the previous sections, what VPT does is assigning each tuple a volatility value, v, based on the financial information contained in the tuple and the most recent tuples with the same symbol. To do so, a sliding-window-based volatility aggregation operator is used. In the experiments performed in the scope of the current thesis, as suggested in [8], the tradditional definition of standard deviation [41] is used as a measure of the volatility of the distribution of trade prices of the most recent tuples with the same symbol as the tuple to which VPT assigns a volatility value. This is done on a sliding-window basis using the window size and window advance sliding-window model introduced in Section 3.3.2, assigning the tuples the volatility value derived from the consumption of the most recent window the tuple does not contribute to. With this, the PPT threads can provide the options pricing operator the volatility aggregated by the VPT tread instead of a default constant and outdated value as it was done in the previous iteration. In this sense, the only difference in the behavior of the PPT threads with respect to the PT threads in the previous operation is the last parameter provided to the `priceOption` procedure in line 7 in Listing 9.1, which is now retrieved from the tuple as the first one, *tp*, was. Section 10.4.1 below elaborates on the formal description of the sliding-window-based volatility aggregation operator and Section 10.4.2 integrates this operator in the data-structures diagram from Figure 10.3.

10.4.1 The Single-Threaded Volatility Aggregation Operator

As described in [41], the standard deviation, henceforth referred to as σ, is a measure which is used to quantify the amount of variation or dispersion of a set of data values or a random variable, X, in this case, the set of most recent stock prices for a given symbol. Given the discrete and finite nature of the set of most recent stock prices, the formal standard deviation definition which is relevant in the scope of the volatility aggregation operator is the standard deviation for a bounded discrete random variable $X = \{X_1, X_2, ..., X_N\}$ introduced in Section 4.1.2. This definition was based on the definition of the mean or first moment, $\mathrm{E}[X]$ or μ, as described in Equation 10.1.

$$\mathrm{E}[X] = \frac{1}{N}\sum_{i=1}^{N} X_i = \mu \tag{10.1}$$

As it can be seen, $\mathrm{E}[X] = \mu$ is the well-known mean or average of the discrete random variable. Elaborating from this definition towards the definition of the standard deviation, σ, a first measure of how fast a discrete random variable spreads out is its variance, $\mathrm{Var}(X)$, which is the mean of a new discrete random variable representing, for each value in the original random variable, the difference from the original value to the mean of the original random variable raised to the square, so that the sign of the difference does not allow positive and negative deviations to compensate each other. This is formally expressed in Equation 10.2.

$$\mathrm{Var}(X) = \mathrm{E}\left[(X-\mu)^2\right] = \frac{1}{N}\sum_{i=1}^{N}(X_i-\mu)^2 \tag{10.2}$$

The small problem with variance for its usage as a volatility measure is that given the square operation in Equation 10.2, it is not expressed in the same units as the original random variable. In other words, if the random variable units are dollars, \$, the variance is expressed in dollars raised to the square, $\$^2$. This motivates the introduction of the standard deviation, σ, as a volatility measure, which is simply the square root of the variance [41], as expressed in Equation 10.3, which takes it back to the same units as the original random variable.

$$\sigma = \sqrt[2]{\mathrm{Var}(X)} = \sqrt[2]{\mathrm{E}\left[(X-\mu)^2\right]} = \sqrt[2]{\frac{1}{N}\sum_{i=1}^{N}(X_i-\mu)^2} \tag{10.3}$$

Equation 10.3 provides a clear definition of the standard deviation, σ, which can be easily translated to code so that given a set of stock prices, a standard deviation value can be easily obtained. Nevertheless, it is worth noticing that if this formula is used as it is, in order to be able to start calculating the standard deviation of a given set of values, it is necessary to first have all the values together so that the mean, μ, can be calculated and then this value can be used to calculate the standard deviation σ.

This last observation leads to a clear imbalance in the effort spent by one thread processing the standard deviation after the arrival of one value:

- **Values which are not the last value in a set of values**: until the set of values is ready to be processed, arriving values can only be added in an attempt to start pre-calculating the mean, μ, and inserted to an array or linked list so that they can be used afterwards to calculate the standard deviation, σ.
  It is easy to see that with an efficient data-structure to temporarily store the values, the cost of this operation in terms of execution time is $O(1)$.
- **Values which are the last value in a set of values**: once the set of values is ready, it is necessary to go through all the values and calculate the sum in Equation 10.3.
  It is also easy to see that again with an efficient data-structure to temporarily store the values, the cost of this operation in terms of execution time is $O(N)$.

In addition to this, given the fact that all the values in the set need to be maintained in memory, Equation 10.3 leads also to an $O(N)$ memory overhead per set of values of size N.

Altogether, Equation 10.3 as it has been introduced allows for the implementation of a stateful sliding-window-based volatility aggregator with the aforementioned imbalance in terms of execution time and overhead in terms of memory. This would lead to the aggregator maintaining $O(N)$ sized windows with window update and window consume operations with execution time costs respectively $O(1)$ and $O(N)$.

Fortunately, a simple succession of arithmetic transformations on Equation 10.3 leads to an expression of the standard deviation, σ, which does not suffer from the aforementioned drawbacks.

Applying the definition of μ as introduced in Equation 10.1:

$$\sigma = \sqrt[2]{\frac{1}{N}\sum_{i=1}^{N}(X_i - \mu)^2} = \sqrt[2]{\frac{1}{N}\sum_{i=1}^{N}\left(X_i - \frac{1}{N}\sum_{j=1}^{N}X_j\right)^2}$$

Raising the expression in the parenthesis to the square:

$$= \sqrt[2]{\frac{1}{N}\sum_{i=1}^{N}\left(X_i^2 + \left(\frac{1}{N}\sum_{j=1}^{N}X_j\right)^2 - \frac{2X_i}{N}\sum_{j=1}^{N}X_j\right)}$$

Separating the sum iterating i from 1 to N into three sums:

$$= \sqrt[2]{\frac{1}{N}\sum_{i=1}^{N}X_i^2 + \frac{1}{N}\sum_{i=1}^{N}\left(\frac{1}{N}\sum_{j=1}^{N}X_j\right)^2 - \frac{1}{N}\sum_{i=1}^{N}\frac{2X_i}{N}\sum_{j=1}^{N}X_j}$$

Extracting 2 from the third sum as a common factor and refactoring that sum:

$$= \sqrt[2]{\frac{1}{N}\sum_{i=1}^{N} X_i^2 + \frac{1}{N}\sum_{i=1}^{N}\left(\frac{1}{N}\sum_{j=1}^{N} X_j\right)^2 - \frac{2}{N}\sum_{i=1}^{N} X_i\left(\frac{1}{N}\sum_{j=1}^{N} X_j\right)}$$

Applying the definition of $\mathrm{E}[X]$ as introduced in Equation 10.1 in the second and third sums:

$$= \sqrt[2]{\frac{1}{N}\sum_{i=1}^{N} X_i^2 + \frac{1}{N}\sum_{i=1}^{N}\mathrm{E}[X]^2 - \frac{2}{N}\sum_{i=1}^{N} X_i\mathrm{E}[X]}$$

Transforming the second sum into a multiplication given the fact that all the factors have the same value, $\mathrm{E}[X]^2$, and extracting $\mathrm{E}[X]$ in the third sum as a common factor:

$$= \sqrt[2]{\frac{1}{N}\sum_{i=1}^{N} X_i^2 + \frac{1}{N}\left(N\mathrm{E}[X]^2\right) - 2\mathrm{E}[X]\frac{1}{N}\sum_{i=1}^{N} X_i}$$

Applying the definition of $\mathrm{E}[X]$ as introduced in Equation 10.1 in the first and third sums and applying simple arithmetic:

$$= \sqrt[2]{\mathrm{E}[X^2] + \mathrm{E}[X]^2 - 2\mathrm{E}[X]\,\mathrm{E}[X]} = \sqrt[2]{\mathrm{E}[X^2] + \mathrm{E}[X]^2 - 2\mathrm{E}[X]^2} = \sqrt[2]{\mathrm{E}[X^2] - \mathrm{E}[X]^2}$$

With the last expression obtained the chain of arithmetic transformations above, the expression of the standard deviation, σ, which is extended in Equation 10.4 below applying once more Equation 10.1, is obtained.

$$\sigma = \sqrt[2]{\mathrm{E}[X^2] - \mathrm{E}[X]^2} = \sqrt[2]{\sum_{i=1}^{N} X_i^2 - \left(\sum_{i=1}^{N} X_i\right)^2} \tag{10.4}$$

This new way of expressing the standard deviation, σ, disentangles the aforementioned imbalance in the effort spent by one thread processing the standard deviation after the arrival of one value:

- **Values which are not the last value in a set of values**: Until the set of values is ready to be processed, arriving values and their squares can be added to start pre-processing respectively $\mathrm{E}[X]$ and $\mathrm{E}[X^2]$. Instead of having to add these values to a temporary array or linked list, a simple counter can be updated to keep track of the number of values in the set.
  It is easy to see that the cost of this operation in terms of execution time is $O(1)$.
- **Values which are the last value in a set of values**: On arrival of the last value completing the set, this value can be processed as the values which are not the last value and afterwards compute the standard deviation, σ, of the whole set of values in five simple steps:

1. Dividing the sum of values by the counter of values obtaining $\mathrm{E}[X]$.
2. Raising the aforementioned value to the square obtaining $\mathrm{E}[X]^2$.
3. Dividing the sum of squares of the values by the counter of values obtaining $\mathrm{E}[X^2]$.
4. Subtracting $\mathrm{E}[X]^2$ from $\mathrm{E}[X^2]$ obtaining $\mathrm{E}[X^2] - \mathrm{E}[X]^2$.
5. Calculating the square root of the value obtained in the previous step, finally obtaining $\sigma = \sqrt[2]{\mathrm{E}[X^2] - \mathrm{E}[X]^2}$.

The cost of this procedure in terms of execution time is also $O(1)$.

With this, no matter whether a value is the last value belonging to a set or not, the time spent to process it in the scope of one set of values whose standard deviation is being calculated is always $O(1)$.

In addition to this, with the aforementioned procedure, each value in the set no longer needs to occupy an individual position in memory as in the case of Equation 10.3. Instead, only an accumulator of values, an accumulator of their squares and a counter of values need to be maintained in memory, which reduces the memory overhead per set of values of size N from $O(N)$ to $O(1)$.

Altogether, Equation 10.4 as it has been introduced allows for the implementation of a stateless sliding-window-based volatility aggregator with $O(1)$ sized windows and $O(1)$ execution time window update and window consume operations.

In order to integrate Equation 10.4 in the paradigm of sliding-window stream processing so that the sliding-window-based volatility operator executed by VPT can be formally defined, it is worth reviewing the window size and window advance sliding-window model introduced in Section 3.3.2, which is the window model which VPT implements due to its algebraic properties, which allow the parallelization introduced in the next chapter of the single-threaded volatility aggregator that is introduced in this chapter.

As described in Section 3.3.2, according to the sliding-window model introduced in [6], operators are computed over a sliding-window, defined by the parameters size, henceforth referred to as WS, and advance, henceforth referred to as WA. Sliding-windows can be time-based, (e.g., to group tuples received during periods of 5 minutes every 2 minutes) or tuple-based (e.g., to group the last 10 received tuples every 3 incoming tuples). Given the nature of the data to be processed by the financial stream processing engine, the former type, time-based, is the one implemented by VPT letting the ts field in the tuples determine which windows they contribute to.

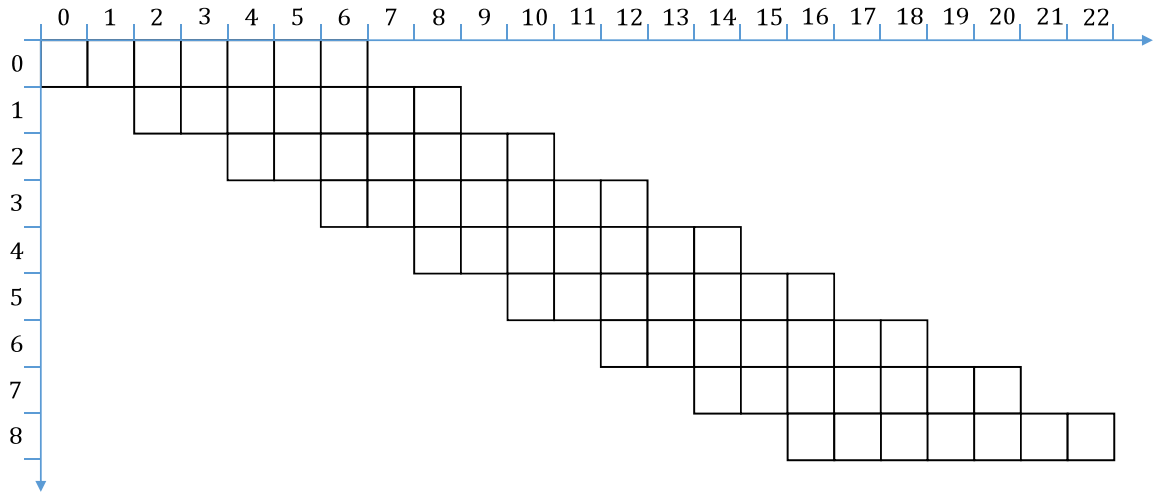


Figure 10.4: Sliding-window model visualization. $WS = 7$, $WA = 2$

Figure 10.4 helps visualizing the window size and window advance time-based sliding-window model. The x axis represents the time discretized in intervals of one time unit and the y axis assigns each window an index, 0 being the index for the very first sliding-window in chronological order of expiration. The white horizontal successions of squares in the diagram represent the different sliding-windows in the two-dimensional discrete space defined by the aforementioned x and y axes. For visualization purposes, small values for WS (7) and WA (2) have been chosen for the illustrative diagrams in this section. However, as it will be discussed in further sections, different values will be chosen for the stream processing engine with a big impact on both the quality of the volatility values calculated and the performance of the operator. It is worth observing the superposition of sliding-windows in time. Tuples with timestamps bigger than WA will contribute always to two windows of more, in the sense that their financial information will be used to calculate the volatility aggregated for these windows. This raises the need to maintain more than one window at a time for each traded symbol in VPT.

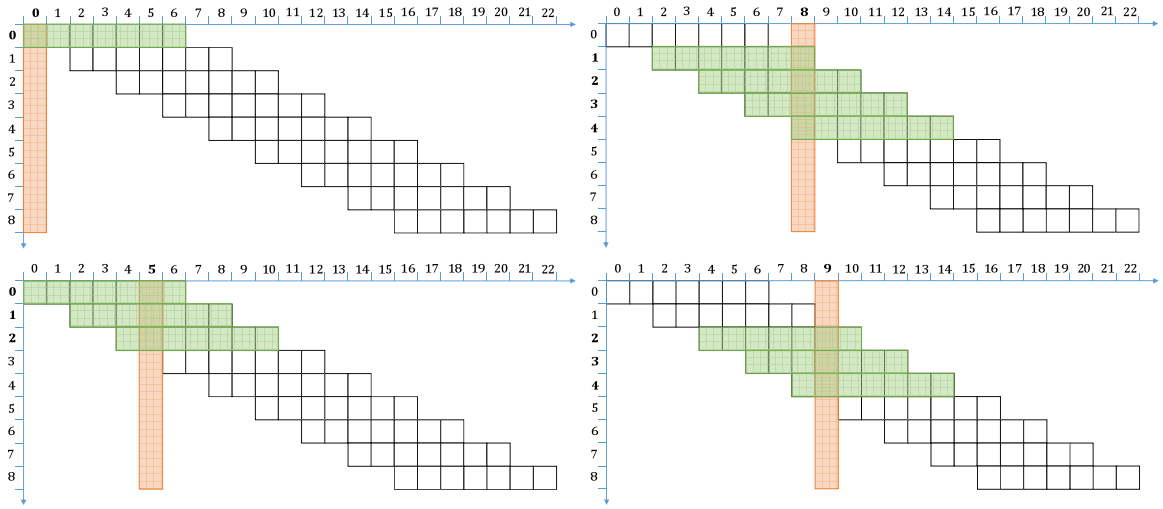


Figure 10.5: Windows different tuples contribute to. $WS = 7$, $WA = 2$

Figure 10.5 illustrates which windows different tuples contribute to, based on their timestamp having $WS = 7$, and $WA = 2$. According to the upper left diagram, a tuple with timestamp 0 would only contribute to window 0 whereas a tuple with timestamp 5, according to the diagram below the previous one, would contribute to windows 0, 1, and 2. The upper right diagram shows how a tuple with timestamp 8 would contribute to windows 1, 2, 3, and 4, while according to the lower right diagram, a tuple with timestamp 9 would contribute to windows 2, 3, and 4.

The visual procedure to determine which windows a tuple contributes to given its timestamp is pretty simple. As illustrated in Figure 10.5, it consists on projecting a vertical line from the tuple's timestamp, ts, the orange rectangles in Figure 10.5, and determining which windows it intersects with, the green rectangles in Figure 10.5. However, in order to provide a formal description of the sliding-window-based volatility operator, it is necessary to formalize the calculations to determine the bounds of the set of windows a tuple contributes to given its timestamp. This can be done by defining a family of transformations from the domain of non-negative discrete timestamps, the x axis in Figure 10.4, to the range of non-negative window identifiers, the y axis in Figure 10.4, as expressed in Equation 10.5.

$$f : X \to Y \tag{10.5}$$

- X: the timestamps space, $\{0, ...\}$, x axis in Figure 10.4.
- Y: the window identifiers space, $\{0, ...\}$, y axis in Figure 10.4.

The first transformation worth defining is PREV_WIN, which determines the last window a tuple does not contribute to given its timestamp, ts, as expressed in Equation 10.6.

$$\begin{aligned} \text{PREV\_WIN} : \; & X \longrightarrow Y \\ & ts \longrightarrow \text{PREV\_WIN}(ts) = \lfloor (ts - WS) / WA \rfloor \end{aligned} \tag{10.6}$$

The second transformation worth defining is POST_WIN, which determines the first window a tuple does not contribute to given its timestamp, ts, as expressed in Equation 10.7.

$$\begin{aligned} \text{POST\_WIN} : \; & X \longrightarrow Y \\ & ts \longrightarrow \text{POST\_WIN}(ts) = \lfloor (ts + WA) / WA \rfloor \end{aligned} \tag{10.7}$$

Based on the PREV_WIN transformation, FIRST_WIN can be defined to determine the first window a tuple contributes to given its timestamp, ts, which is the window right after the one obtained in Equation 10.6, as expressed in Equation 10.8.

$$\begin{aligned} \text{FIRST\_WIN} : \; & X \longrightarrow Y \\ & ts \longrightarrow \text{FIRST\_WIN}(ts) = \text{PREV\_WIN}(ts) + 1 \end{aligned} \tag{10.8}$$

Based on the POST_WIN transformation, LAST_WIN can be defined to determine the last window a tuple contributes to given its timestamp, ts, which is the window right before the one obtained in Equation 10.7, as expressed in Equation 10.9.

$$\begin{aligned} \text{LAST\_WIN}: \quad X &\longrightarrow Y \\ ts &\longrightarrow \text{LAST\_WIN}(ts) = \text{POST\_WIN}(ts) - 1 \end{aligned} \tag{10.9}$$

Altogether, the set of windows a tuple contributes to given its timestamp, ts, is $\{\text{FIRST\_WIN}(ts), ..., \text{LAST\_WIN}(ts)\}$.

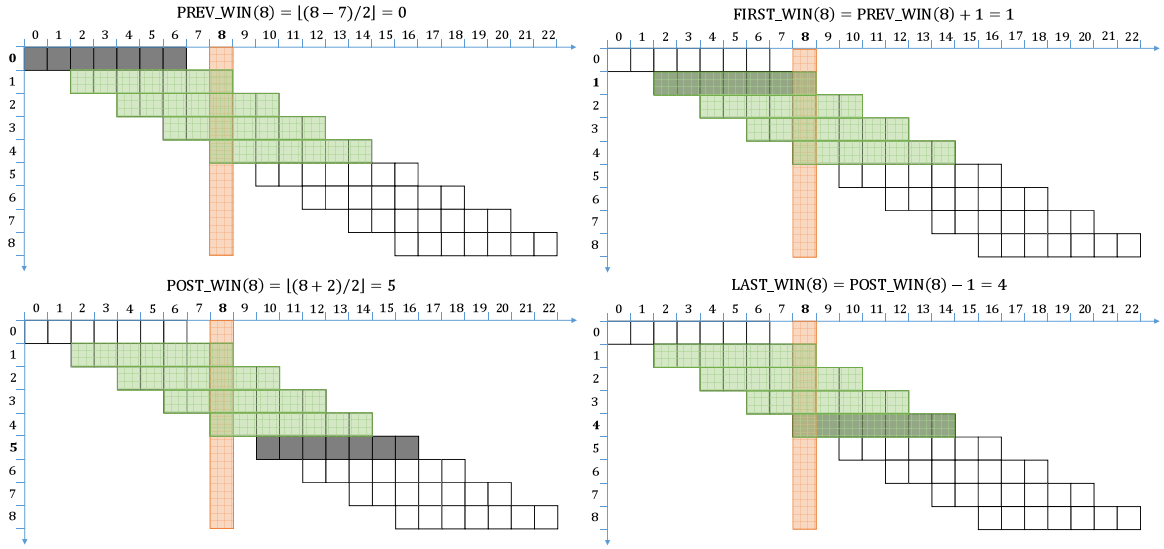


Figure 10.6: Illustration of the PREV_WIN, POST_WIN, FIRST_WIN, and LAST_WIN transformations. $WS = 7$, $WA = 2$, $ts = 8$

Figure 10.6 illustrates the four transformations introduced above using the sliding-window diagram introduced in Figure 10.4 with $WS = 7$ and $WA = 2$, highlighting in grey the windows matching the four different transformations given a tuple with timestamp 8. According to the upper left diagram, the last window the tuple does not contribute to is $\text{PREV\_WIN}(8) = \lfloor (8-7)/2 \rfloor = 0$. The first window the tuple does not contribute to, according to the diagram below the previous one is $\text{POST\_WIN}(8) = \lfloor (8+2)/2 \rfloor = 5$. The upper right diagram shows that the first window the tuple contributes to is $\text{FRIST\_WIN}(8) = \text{PREV\_WIN}(8) + 1 = 1$, while according to the lower right diagram, the last window the tuple contributes to is $\text{LAST\_WIN}(8) = \text{POST\_WIN}(8) - 1 = 4$.

Having formalized the transformations which determine the set of windows a tuple contributes to given its timestamp, ts, it is worth determining how many windows an aggregator needs to keep track of at a time for each traded symbol. In other words, if there is a boundary on the size of the set of windows a tuple contributes to. The answer to this question is obviously yes, Equation 10.10, derived from Equations 10.6 and 10.7 expresses this upper boundary henceforth referred to as MAXW.

$$\text{MAXW} = \lceil WS/WA \rceil \tag{10.10}$$

As it can be seen in Figure 10.4, having $WS = 7$ and $WA = 2$, MAXW $= \lceil 7/2 \rceil = 4$, which is the maximum number of windows which can be seen overlapping in the diagram in the time instants $8 + 2 * n$ for $n \geq 0$.

As discussed above, the MAXW boundary determines the maximum number of windows which need to be maintained by a volatility aggregator at a time. The reason for this is that given the nature of the data to be processed by the financial stream processing engine, only the last window a tuple does not contribute to given its timestamp, ts, or according to the transformations above, the window whose identifier is PREV_WIN(ts), is the window whose aggregated volatility is assigned to the tuple being processed by the volatility aggregator. Given the fact that tuples arrive in non-decreasing timestamp order, once a window has been consumed, it will be no longer updated and only its aggregated value may be used for future tuples while they have the same PREV_WIN as the tuple which triggered its consumption, in other words, the first tuple with that PREV_WIN. With all of this, whenever a tuple arrives to the aggregator, the window with identifier PREV_WIN(ts) can be consumed if it has not been consumed and dropped before and all the windows older than that one can be simply dropped as well as that one, once consumed, if they have not been dropped by an earlier tuple with the same PREV_WIN. Afterwards, only up to MAXW windows need to be updated, thus, having to maintain in memory at most MAXW windows at a time per stock symbol.

The last observation allows for the usage of a fixed size circular buffer of windows to keep track of the MAXW windows which need to be maintained for each traded symbol. In order to formally define the volatility aggregation operator using the aforementioned circular buffer, it is necessary to formalize the calculations to determine which position in the circular array of windows does a window occupy given its window identifier. This can be done by defining a transformation from the formerly introduced domain of non-negative window identifiers, the y axis in Figure 10.4, to the range of the corresponding circular buffer indexes, as expressed in Equation 10.11.

$$f : Y \to Z \tag{10.11}$$

- Y: the window identifiers space, $\{0, ...\}$, y axis in Figure 10.4.
- Z: the circular buffer indexes space, $\{0, ..., \text{MAXW} - 1\}$.

The transformation WIN_BIDX determines the index in the circular buffer of MAXW windows corresponding to the position a window occupies in it given its window identifier as expressed in Equation 10.12.

$$\begin{array}{lrcl} \text{WIN\_BIDX}: & Y & \longrightarrow & Z \\ & id & \longrightarrow & \text{WIN\_BIDX}(id) = id \mod \text{MAXW} \end{array} \tag{10.12}$$

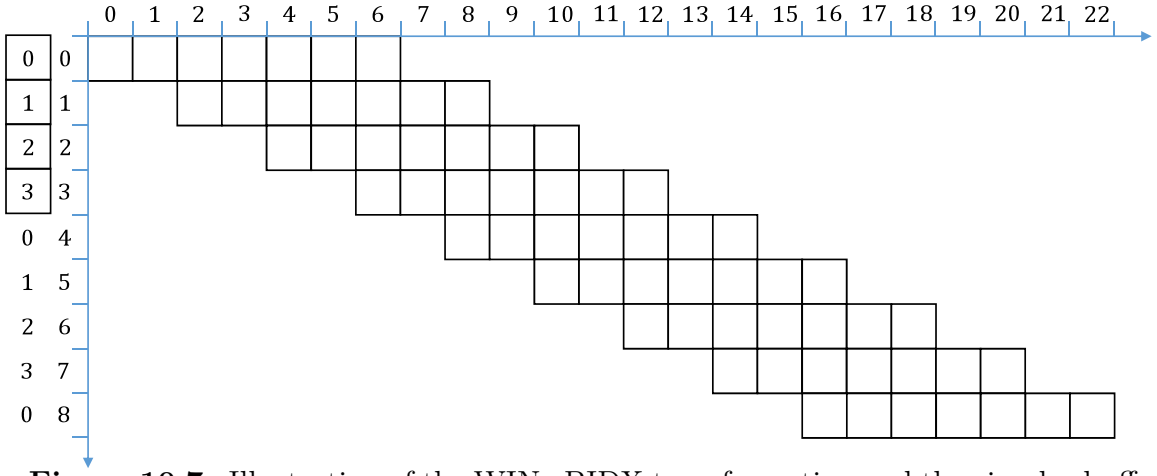


Figure 10.7: Illustration of the WIN_BIDX transformation and the circular buffer of windows. $WS = 7$, $WA = 2$, MAXW $= \lceil 7/2 \rceil = 4$

Figure 10.7 illustrates the WIN_BIDX transformation introduced above using the sliding-window diagram introduced in Figure 10.4 with $WS = 7$ and $WA = 2$, which lead to $MAXW = 4$ as discussed above. The Z range for this transformation has been represented in parallel with the Y range in the left most side of the figure depicting for the first MAXW window an illustration of the circular buffer in its initial state. With all of this, having $WS = 7$ and $WA = 2$, the windows with identifiers $\{0 + i \cdot MAXW, 1 + i \cdot MAXW, 2 + i \cdot MAXW, 3 + i \cdot MAXW\}$, for $i \in \{0, ...\}$ are respectively assigned to the buffer indexes $\{0, 1, 2, 3\}$.

Having introduced the formal definition of volatility in both the standard manner, Equation 10.3, and optimized for computational purposes, Equation 10.4, having reviewed the window size and window advance sliding-windows model and formally defined the PREV_WIN, POST_WIN, FIRST_WIN, LAST_WIN, and WIN_BIDX transformations as well as the MAXW boundary, the single-threaded sliding-window-based volatility aggregation operator executed by the VPT thread can be formally defined.

First of all, each volatility aggregation window updated and consumed by the volatility aggregator can be modeled as a data-structure with three variables according to the volatility calculation procedure introduced when analyzing Equation 10.4:

- **Trade price sum** (*tps*): the trade price sum variable, henceforth referred to as *tps*, is the variable summing all the prices per share of the traded security symbol that are traded in the transactions represented by all the financial tuples contributing to the volatility aggregation window. This variable is internally modeled as a `double` value accounting for the sum of the prices paid for each individual share. Its value on initialization of the volatility aggregation window is naturally 0.
- **Trade price to the square sum** (*tpsqs*): the trade price to the square sum variable, henceforth referred to as *tpsqs*, is the variable summing the squares of all the prices per share of the traded security symbol that are

traded in the transactions represented by all the financial tuples contributing to the volatility aggregation window. This variable is internally modeled as a `double` value accounting for the sum of the squares of the prices paid for each individual share. Its value on initialization of the volatility aggregation window is naturally 0.

- **Trade volume sum (*tvs*)**: the trade volume sum variable, henceforth referred to as *tvs*, is the variable summing all the trade volumes for all the financial tuples contributing to the volatility aggregation window, in other words, the count of values contributing to the discrete random variable of trade prices whose volatility is output when consuming the window. This variable is internally modeled as a `long` value accounting for the sum of the trade volumes for each tuple contributing to the window. Its value on initialization of the volatility aggregation window is naturally 0.

In accordance to the sliding-window paradigm, there are two main ways to interact with a volatility aggregation window:

- **Update the volatility aggregation window**: the update procedure consists on letting the window account for the information contained in one tuple contributing to it. In the case of the volatility aggregation window, this implies updating the aforementioned variables:
  - Adding to *tps* the *tp* field contained in the contributing tuple as many times as the *tv* field in the same tuple specifies.
  - Adding to *tpsqs* the *tp* field contained in the contributing tuple, raised to the square, as many times as the *tv* field in the same tuple specifies.
  - Adding to *tvs* the *tv* field contained in the contributing tuple.
- **Consume the volatility aggregation window**: the consume procedure consist on iterating through the data accumulated by the window and obtaining an aggregated result summarizing the information aggregated by the window from the contributing tuples which performed the aforementioned update procedure on the window. In the case of the volatility aggregation window this implies using the aforementioned three variables to compute the volatility as described when analyzing Equation 10.4.

In addition to the two main procedures above, an instance of volatility aggregation window can be reset in order to be reused modelling a different window once the window represented by that instance has expired and is no longer of use. This additional procedure, consisting on resetting to zero the values of the three variables, will be useful for the definition of the circular array of volatility aggregation windows maintained by the volatility aggregators as discussed earlier in this section.

Listing 10.1: Volatility aggregation window pseudocode

```
1 volatilityWindow
2   tps = 0, tpsqs = 0, tvs = 0
3
4 update(tp, tv)
5   tps = tps + tp * tv
```

```
6     tpsqs = tpsqs + tp * tp * tv
7     tvs = tvs + tv
8
9   consume()
10    if (tvs > 0)
11      etp = tps / tvs
12      etpsq = tpsqs / tvs
13      v = sqrt(etpsq - etp * etp)
14    else
15      v = 0
16    return v
17
18  reset()
19    tps = 0
20    tpsqs = 0
21    tvs = 0
```

Listing 10.1 formally defines the volatility aggregation window introduced above. Lines 1-2 represent the structure of the volatility aggregation window with the three aforementioned variables, *tps*, *tpsqs*, and *tvs* initialized to 0 by default. The `update` procedure is described in lines 4-7, having in lines 5, 6, and 7 respectively the updates to the *tps*, *tpsqs*, and *tvs* variables described when introducing the update procedure above. The `consume` procedure is defined in lines 9-16. The safety check in line 10 prevents the window from dividing by 0 assuming that the volatility for a window in which not a tuple has contributed is simply 0 as expressed in line 15. Lines 11-13 perform the volatility aggregation according to Equation 10.4 using the aforementioned three variables *tps*, *tpsqs*, and *tvs* to calculate $\mathrm{E}[X]$ in line 11, $\mathrm{E}[X^2]$ in line 12, and finally $\sqrt{\mathrm{E}[X^2] - \mathrm{E}[X]^2}$ in line 13. This value, or the value 0 in case not a tuple contributed to the window, is returned in line 16. Finally, in lines 18-21 the `reset` procedure is defined by setting back to 0 the values of the variables *tps*, *tpsqs*, and *tvs* respectively in lines 19, 20, and 21.

Having formally defined the volatility aggregation window above, the sliding-window-based volatility aggregator for a given traded symbol maintaining a circular buffer of MAXW volatility aggregation windows can be formally defined as a data-structure also with three variables:

- **Volatility aggregation windows buffer (*wbuff*)**: the volatility aggregation windows buffer variable, henceforth referred to as *wbuff*, is the variable modeling the aforementioned circular buffer of volatility aggregation windows which let the volatility aggregator maintain in memory the MAXW most recent volatility aggregation windows. This variable is internally modeled as an array of MAXW `volatilityWindow` instances, each of which is initialized by default as described when formally defining the volatility aggregation windows.
- **Last timestamp (*lastts*)**: the last timestamp variable, henceforth referred to as *lastts*, stores the last processed tuple timestamp, *ts*, field value in order to

keep track of which specific windows are maintained every time in the $wbuff$ circular buffer, and to determine whether if a new window has to be consumed or not, and when a set of windows in the buffer need to be discarded and replaced by newer windows. This variable is internally modeled as a *long* value exactly the same way as the ts field in the financial tuples is modeled as introduced in Section 6.2. Its value on initialization of the volatility aggregator is 0 by default.

- **Last volatility** (*lastv*): the last volatility variable, henceforth referred to as *lastv*, stores the volatility calculated for the last processed tuple whose timestamp is stored in the previous variable, *lastts*. This value needs to be maintained by the volatility aggregator because as soon as a window is consumed, it has to be discarded and replaced by a new one in order to let the tuple which triggered the window consumption contribute to the up to MAXW windows it has to contribute to. With this, in case a new tuple arrives with a greater or equal timestamp as the tuple which triggered the window consumption but with the same PREV_WIN, the window to be consumed to assign a volatility value to that tuple is no longer present in $wbuff$, as reasoned above, but the volatility resulting for its consumption is the one stored in *lastv*. This variable is internally modeled as a `double` value accounting for the volatility assigned to the tuple with timestamp *lastts* resulting from the consumption of the window whose identifier in the Y space discussed when introducing the ts to window identifiers transformations is PREV_WIN(*lastts*). Its value on initialization of the volatility aggregator is 0 by default.

For each of the tuples with the same symbol as the underlying stock associated to the volatility aggregator there is one single way to interact with the volatility aggregator consisting on letting the aggregator process the tuple. This procedure involves providing the tuple ts, tp, and tv fields to the aggregator in order for it to assign the tuple a volatility value resulting from the consumption of the window whose identifier is PREV_WIN(ts), and to update all the windows in the range {FIRST_WIN(ts), ..., LAST_WIN(ts)} providing the tp and ts values as discussed when introducing the `update` procedure for the volatility aggregation window.

Listing 10.2: Single-threaded sliding-window-based volatility aggregator for a single traded symbol pseudocode

```
1  volatilityAggregator
2    wbuff[MAXW], lastts = 0, lastv = 0
3
4  processTuple(ts, tp, tv)
5    if (PREV_WIN(lastts) < PREV_WIN(ts) and
6        PREV_WIN(ts) - PREV_WIN(lastts) <= MAXW)
7      lastv = wbuff[WIN_BIDX(PREV_WIN(ts))].consume()
8      for (i = FIRST_WIN(lastts) to PREV_WIN(ts))
9        wbuff[WIN_BIDX(i)].reset()
10   else if (PREV_WIN(lastts) < PREV_WIN(ts) and
11            PREV_WIN(ts) - PREV_WIN(lastts) > MAXW)
12     lastv = 0
```

```
13      for (i = 0 to MAXW - 1)
14        wbuff[i].reset()
15    for (i = FIRST_WIN(ts) to LAST_WIN(ts))
16      wbuff[WIN_BIDX(i)].update(tp, tv)
17    lastts = ts
18    return lastv
```

Listing 10.2 formally defines the sliding-window-based volatility aggregator for a given traded symbol introduced above. Lines 1-2 represent the structure of the aggregator with the aforementioned variable, $wbuff$, having space for MAXW `volatilityWindow` instances and the $lastts$, and $lastv$ variables initialized to 0 by default.

The `processTuple` procedure is formally defined in lines 4-18, having the window consumption and replacement logic in lines 5-14, and the window update logic in lines 15-16. The assertion in lines 5-6 checks whether or not the processed tuple given its timestamp, ts, should trigger the consumption of a window present in $wbuff$. In case this assertion holds, the window is consumed and the resulting volatility value is stored in the $lastv$ variable in line 7 and the loop in lines 8-9 is executed to reset all the windows which need no longer be maintained in $wbuff$, including the window consumed in line 7, in order to start keeping track of the last windows the processed tuple may contribute to which were not yet present in $wbuff$. The assertion in lines 10-11 checks if the unfeasible but not impossible case in which the tuple should trigger the consumption of a window which is not present in $wbuff$ is given. This would happen if two consecutive tuples processed by the aggregator are separated in time enough for the full set of windows the first one contributed to not to intersect with the set of windows the second one contributes to plus the last window it did not contribute to. In this unfeasible situation, the volatility for the window consumed by the second tuple, which had never a tuple contributing to it in the aggregator, is assumed to be 0 as expressed in line 12, and the loop in lines 13-14 is executed resetting all the now outdated windows in $wbuff$ in order to allocate space for the up to MAXW windows the second tuple contributes to.

Having updated in case of need the $lastv$ variable and reset the outdated tuples in $wbuff$ in lines 5-14, the tp and tv values of the tuple being processed can be used to update all the windows it contributes to according to its timestamp. This is done in the loop in lines 15-16 which iterates from the first window the processed tuple contributes to according to its timestamp, FIRST_WIN(ts), to the last window the processed tuple contributes to according to its timestamp, LAST_WIN(ts), executing in line 16 the `update` procedure introduced in Listing 10.2, for the corresponding windows in $wbuff$ which can be located thanks to the WIN_BIDX transformation. Once all the windows have been updated, the $lastts$ variable is updated in line 17 to keep track of the tuple which was just processed and the previously updated if needed $lastv$ value is returned in line 18 representing the volatility value to be assigned to the tuple by the VPT thread when adding the v field to the tuple to

let the PPT threads price the corresponding option contract taking into account the aggregated volatility instead of a constant outdated value as in the previous iterations of the financial stream processing engine.

Having formally defined the sliding-window-based volatility aggregator for a given traded symbol, the single-threaded sliding-window-based volatility aggregator for a set of traded symbols with symbol hash, sh, in the range $\{0, ..., \text{MAXSH} - 1\}$ can be formally defined as an array with MAXSH `volatilityAggregator` instances indexed by symbol hash.

Listing 10.3: Single-threaded sliding-window-based volatility aggregator for multiple traded symbols pseudocode

```
1 volatilityAggregators
2   aggrs[MAXSH]
3
4 processTuple(sh, ts, tp, tv)
5   return aggrs[sh].processTuple(ts, tp, tv)
```

Listing 10.3 formally defines the sliding-window-based volatility aggregator for a set of traded symbols with symbol hash, sh, in the range $\{0, ..., \text{MAXSH} - 1\}$ introduced above. Lines 1-2 represent the structure of the aggregator with the aforementioned array of aggregators, $aggrs$, having space for MAXSH `volatilityAggregator` instances. The `processTuple` procedure is described in lines 4-5, and consists only in using the sh field in the processed tuple to identify the corresponding single-threaded volatility aggregator for the symbol associated to the tuple and letting that aggregator process the tuple based on its ts, tp, and tv fields executing in line 5 the `processTuple` procedure introduced in Listing 10.2 which allows the VPT thread to assign the processed tuple the resulting aggregated volatility returned in line 5.

The last described procedure, `processTuple`, is the one executed by the VPT thread every time a tuple is retrieved from the queue it shares with IT and before adding it to the *ScaleGate* instance it shares with the PPT threads. It is worth mentioning that the same way as IT did in the previous iteration of the financial stream processing engine, VPT will also insert a `NULL` tuple with the same sequence number as the non-`NULL` tuple added to the *ScaleGate* instance to avoid the double latency problem for single writers in a *ScaleGate* instance discussed in Section 8.4.1.

Before concluding this section, it is worth analyzing the complexity of the single-threaded sliding-window based volatility aggregation operator in terms of execution time and memory as it was done in Section 6.4 with the single-threaded binomial option pricing operator. As it can easily be seen after a brief analysis of the pseudocode introduced in Listings 10.1, 10.2, and 10.3, the three parameters affecting the cost of the operator in time and memory are MAXSH, WS, and WA, the latter two ones determining together the MAXW boundary as described in Equation 10.10. In terms of execution time:

- Line 5 in Listing 10.3 triggers the execution of the `processTuple` procedure introduced in lines 4-18 in Listing 10.2 for the `volatilityAggregator` instance associated to the processed tuple symbol. The cost of locating the `volatilityAggregator` instance is $O(1)$ as it consists simply on a direct access to the position indexed by the tuple symbol hash, sh, in the hash indexed array of aggregators.
- The assertion in lines 5-6 in Listing 10.2 have also an $O(1)$ execution time cost.

  In case it succeeds:
  - The `consume` procedure introduced in lines 9-16 in Listing 10.1 is executed in line 7 in Listing 10.2. As discussed when analyzing Equation 10.4, and as it can be verified analyzing lines 9-16 in Listing 10.1, the cost in terms of execution time of this procedure is $O(1)$.
  - The `reset` procedure introduced in lines 18-21 in Listing 10.1 is executed up to MAXW times in the worst possible case in the loop in lines 8-9 in Listing 10.2. It is easy to see that the cost in terms of execution time of the `reset` procedure consisting on simply 3 variable value updates is $O(1)$ concluding that the worst possible execution time cost of the loop in lines 8-9 in Listing 10.2 is O(MAXW).

  In case it fails:
  - The assertion in lines 10-11 in Listing 10.2 is checked incurring in an $O(1)$ execution time cost.

    In case it succeeds:
    - The variable update performed in line 12 in Listing 10.2 has an $O(1)$ execution time cost.
    - The execution of the loop in lines 13-14 in Listing 10.2 has an $O(\text{MAXW})$ cost in terms of execution time following the same reasoning as when analyzing the cost of the loop in lines 8-9 in Listing 10.2.
- The loop in lines 15-16 in Listing 10.2 is executed for all the tuples. It implies executing in line 16 the `update` procedure introduced in lines 4-7 in Listing 10.1 $\text{LAST\_WIN}(ts) - \text{FIRST\_WIN}(ts) + 1$ times, ts being the provided tuple timestamp. Given the definition of MAXW, $\text{LAST\_WIN}(ts) - \text{FIRST\_WIN}(ts) + 1 \leq \text{MAXW}\ \forall sh \in \{0, ...\}$. As discussed when analyzing Equation 10.4, and as it can be verified analyzing lines 4-7 in Listing 10.1, the cost in terms of execution time of the `update` procedure is $O(1)$. With this, the cost in terms of execution time of the loop in lines 15-16 in Listing 10.2 is $O(\text{MAXW})$.
- Finally, the variable update and return operations in lines 17 and 18 respectively in Listing 10.1 are executed for all the tuples with an $O(1)$ execution time cost.

Overall, the cost of the operator in terms of time is the one expressed in Equation 10.13.

$$O(\text{MAXW}) = O(WS/WA) \tag{10.13}$$

The cost in terms of memory is also easy to assess:

- The input variables in the different analyzed procedures plus the constants WS, and WA, and the different temporary variables used inside the different procedures analyzed occupy a fixed amount of memory independent of the values of WS, WA, and MAXSH. The space occupied by these variables in memory is $O(1)$.
- The `wbuff` array introduced in line 2 in Listing 10.2 has space for MAXW `volatilityWindow` instances, each of which occupy in memory, has anticipated when analyzing Equation 10.4, and as it can be verified analyzing line 2 in Listing 10.1, $O(1)$ space. With this, the space occupied by one instance of `volatilityAggregator` in memory is $O(\text{MAXW})$.
- The `aggrs` array introduced in line 2 in Listing 10.3 has space for MAXSH instances of `volatilityAggregator`, each of them occupying $O(\text{MAXW})$ space in memory. With this, the space occupied in memory by the instance of `volatilityAggregators` used by the VPT thread is $O(\text{MAXSH} \cdot \text{MAXW})$.

Overall, the cost of the operator in terms of memory is the one expressed in Equation 10.14.

$$O(\text{MAXSH} \cdot \text{MAXW}) = O(\text{MAXSH} \cdot WS/WA) \tag{10.14}$$

10.4.2 Integrating the Single-Threaded Volatility Aggregation Operator

Getting back to the stream processing engine diagram from Section 10.3, the two main values that the VPT thread needs to be provided with in order to initialize each of the `volatilityAggregator` instances maintained by the `volatilityAggregators` instance it uses to support the sliding-windows model, discussed in detail in the previous section, are WS and WA. MAXSH is also necessary in order to determine how many `volatilityAggregator` instances to maintain in the `volatilityAggregators` instance but it does not have a direct impact on the volatility values output by the operator nor the execution time cost of the operator as discussed in the previous section.

- **Window size** (WS): this value is taken as a constant, internally modeled as a `long` constant expressing the size of the windows in microseconds. In a production setup, pricing option contracts with maturities in the range of months or years, reasonable WS values consequently stay in the range of months of years. However, in the context of this thesis, having a fine grained financial dataset spanning only one day in time as described in Section 5.1, the WS and WA constants have been homotetically scaled to fit the size of the experimental dataset yet preserving the proportion they would preserve in a reasonable production setup in order not to alter the MAXW boundary which determines the execution time and memory cost of the operator, together with the MAXSH constant in terms of memory as described in Equations 10.13 and 10.14. With this, the default value chosen for the WS constant in the context of this thesis is $3600000000\mu s = 1h$.

- **Window advance** (WA): this value is also taken as a constant, internally modeled as a `long` constant expressing the advance of the windows in microseconds. In a production setup, the smaller the window advance is, the faster the volatility aggregator would react to sudden changes in the behavior of the underlying asset leading to more up to date volatility values being provided to the PPT threads. WA values in the range of minutes would be acceptable for WS values in the range of years or months. With this, the default value chosen for the WA constant in the context of this thesis is $50000\mu s = 50ms$.

It is worth noticing that in contrast with the previous four iterations of the stream processing engine, out of the six values that the binomial options pricing operator uses as an input, tp, os, om, rli, v, and N, two of them, tp, and since this fifth iteration also v, can be retrieved by the PPT threads from the tuples served by VPT to the *ScaleGate* instance it shares with them as described in Section 10.4.1. The rest of the values are still taken as constants with the default values introduced in Section 6.4.2. In addition to this, since the fourth iteration of the stream processing engine the H parameter needs to be specified and provided to the parallel PPT threads to determine whether to use or not the heartbeat mechanism introduced in the previous chapter and how exhaustively to use it.

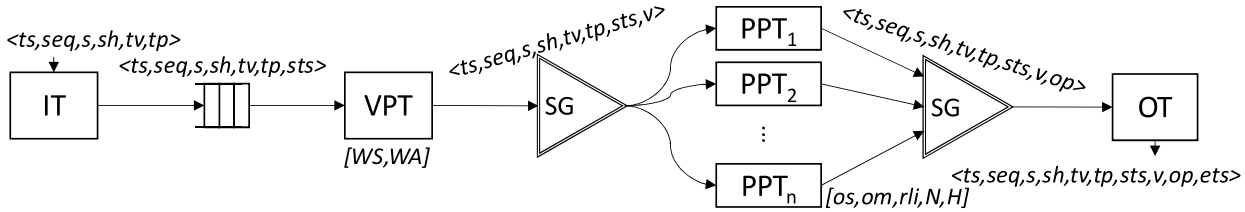


Figure 10.8: Operators and used constants

Figure 10.8 extends the data-structures diagram introduced in Figure 10.3 to represent the aforementioned constant values between square brackets close to the boxes representing the VPT and PPT threads, which execute respectively the single-threaded sliding-window-based volatility aggregation operator introduced in the previous section and the multi-threaded binomial options pricing operator introduced in the previous chapter retrieving the volatility, v, this time from the tuples instead of taking it as a fixed outdated constant.

Overall, this fifth iteration of the stream processing engine, as anticipated in Section 9.4.2, responds to the second line of improvement outlined in the last paragraph in Section 6.4.2 concluding Chapter 6.

11

Multi-Threaded Volatility Aggregation

In the previous chapter, the single-threaded version of the sliding-window-based volatility aggregation operator was introduced and integrated in the financial stream processing engine enabling the options pricing threads to use the reliable and up to date volatility values aggregated by this operator based on the financial information contained in the tuples being processed by the financial stream processing engine. As it could be seen in Equation 10.13 introduced in Section 10.4, the cost of the aforementioned volatility aggregation operator in terms of execution time grows linearly with the proportion between the window size, WS, and window advance, WA, of the underlying sliding-window model. Given that, as discussed in Section 10.4.2, window sizes in the range of months or years are needed in production setups and the smaller the window advance, the more valuable the volatility aggregation values are in terms of responsiveness to sudden changes in the market and timeliness, this proportion can become considerably big making it possible for the operator to become a scalability bottleneck in an options pricing stream processing engine if it is used in its single-threaded version. This problem can be actually appreciated in Section 13.2.3 where the throughput and latency median results obtained when executing the fifth iteration of the stream processing engine introduced in the previous chapter with one volatility aggregation thread and n parallel options pricing threads are reported and discussed. For this reason, it is reasonable to approach in this chapter the problem of parallelizing the volatility aggregation operator.

It is worth taking into account that, in contrast with the nature of the formerly introduced options pricing operator, in which the option prices assigned to the different tuples were totally independent from the behavior of other tuples in the same stream, all the tuples processed by the volatility aggregation operator update the status of the underlying sliding-window model, these changes affecting the volatility values assigned to further tuples in the stream. For this reason, the approach followed in this chapter to parallelize the single-threaded volatility aggregation operator differs from the approaches followed in Chapters 7-9 to parallelize the binomial options pricing operator in which the workload derived from pricing one option contract given a specific tuple was not distributed among the concurrent threads. In this occasion the workload the single-threaded version of the volatility aggregation operator performed for each individual tuple needs to be distributed among the parallel volatility aggregation threads in order to let all of them be aware of changes in the underlying sliding-window model every time a new tuple is processed by all of

them.

The following sections elaborate on how the VPT thread introduced in Section 10.1 can be replaced by a set of parallel volatility aggregation threads in order to concurrently update the shared underlying sliding-windows model ensuring that processed tuples are delivered to the PPT threads in the same order as the corresponding input tuples are served by IT and assigning them the same volatility values the single-threaded version of the volatility aggregation operator would assign them.

11.1 Involved Threads

As anticipated above, the thread dedicated in the previous chapter to calculating the aggregated volatility of the most recent tuples for a given symbol on a sliding-window basis is replaced in this sixth iteration of the financial stream processing engine by a set of parallel volatility aggregation threads. The threads dedicated to retrieve tuples from the input dataset and serve them to the process threads, to get the tuples output, in this case by the volatility aggregation threads and assign them an option price, and to get the tuples output by the options pricing threads and output them, remain in this sixth iteration exactly as they were introduced in the previous one.

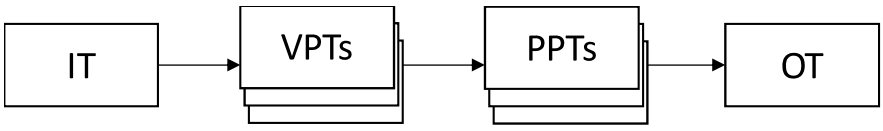


Figure 11.1: Involved threads

Figure 11.1 outlines how threads are arranged in this sixth version of the financial stream processing engine. As it can be seen, IT and OT remain as they were introduced in Section 6.1 and the PPT threads remain as the PT threads were introduced in Section 8.1 renamed to PPT threads in Section 10.1, while the VPT thread introduced in Section 10.1 has been replaced by a set of parallel volatility aggregation threads. Given that the new parallel volatility aggregation threads distribute the workload performed in the previous iteration by the single-threaded volatility aggregation thread, they have been assigned the same name in this chapter as the former VPT thread introduced in Section 10.1 in Chapter 10.

11.2 Structure of the Tuples

Similarly as it happened when parallelizing the options pricing operator in Chapters 7-9, the structure of the tuples in this sixth iteration of the financial stream processing engine remains the same as in the previous iteration.

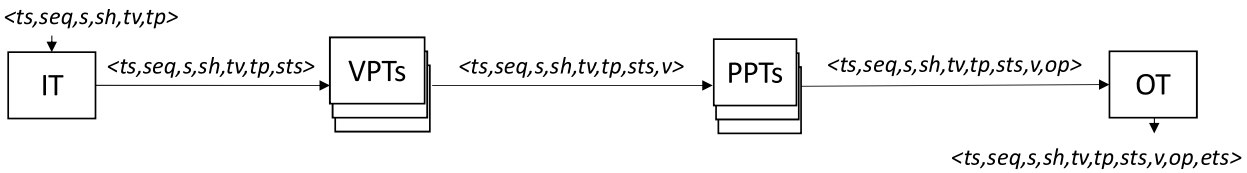


Figure 11.2: Structure of the tuples

Figure 11.2 extends the involved threads diagram introduced in Figure 11.1 specifying the structure of the tuples in each transition from one thread to another as well as the structure of the tuples read by IT, and the tuples output by OT.

As it can be seen, the structure of the tuples retrieved by IT, the tuples transferred from IT to the now parallel VPT threads, the tuples extended by the now parallel VPT threads and facilitated to the PPT threads, the tuples served by the parallel PPT threads to OT, and the tuples output by OT is exactly the same as the structure of the tuples retrieved by IT, the tuples transferred from IT to the former VPT thread, the tuples extended by the former VPT thread and facilitated to the PPT threads, the tuples served by the parallel PPT threads to OT, and the tuples output by OT in Figure 10.2.

11.3 Used Data-Structures

As anticipated in the introduction of this chapter, all the VPT threads need to have access to all the tuples served by IT in the same order so that they can concurrently update the underlying sliding-windows to aggregate volatility producing exactly the same output as the single-threaded volatility aggregator. As discussed in Section 9.3, letting IT add all the tuples it assigns a timestamp to a *ScaleGate* instance shared with a set of readers enables all the readers to have access to all the tuples served by IT in the same order. For this reason, similarly as it was done in Section 9.3 to let all the tuples added by IT reach all the parallel PT threads, the IT thread and the VPT threads in this sixth iteration of the financial stream processing engine will share a *ScaleGate* instance.

Moving forward in the DAG, given the fact that since the fourth iteration of the financial stream processing engine all the PPT threads need to have access to all the tuples being processed by the stream processing engine in order to have the chance to use the heartbeat mechanism introduced in Section 9.4.1 to expedite the behavior of the *ScaleGate* instance they share with OT, the *ScaleGate* instance that VPT and the PPT threads shared in the previous iteration of the financial stream processing engine will also be shared between the VPT threads and the PPT threads in this sixth iteration of the financial stream processing engine.

With all of this, the stream of tuples served by IT, is accessed by all the VPT threads, which will coordinate to, as explained in detail in Section 11.4.1, add to the *ScaleGate* instance they share with the PPT threads one and only one non-`NULL` extended tuple for each tuple received from IT letting each of the PPT threads

retrieve from the *ScaleGate* instance they share with the VPT threads exactly the same stream of extended tuples they retrieved in the previous iteration of the stream processing engine consequently letting the OT thread retrieve and output the same stream of tuples with an option price assigned based on the volatility aggregated by the VPT threads the same way as it did in the previous iteration with one VPT thread.

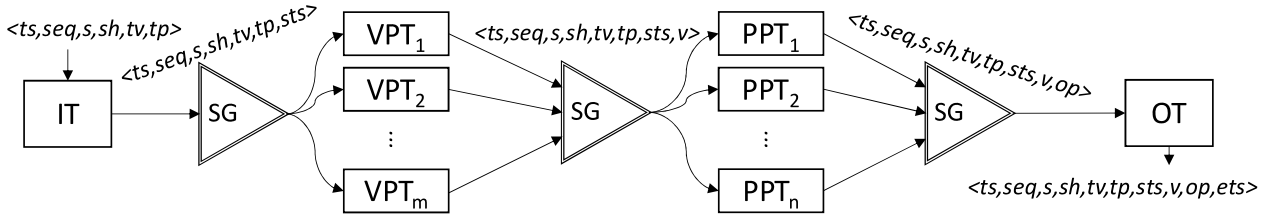


Figure 11.3: Used data-structures

Figure 11.3 extends the structure of the tuples diagram introduced in Figure 11.2 specifying the data-structures used in each transition from one thread to another:

- IT **to the** VPT **threads** ***ScaleGate***: As reasoned above, IT and the parallel VPT threads share in this sixth iteration of the stream processing engine a *ScaleGate* instance having IT as the only writer thread and the VPT threads as a set of multiple readers. This way, as anticipated above, all the tuples are received by all the VPT threads in the same order they are served by IT making it possible for them to update in parallel the underlying sliding-windows model and synchronize to output, for each tuple served by IT, one and only one tuple with the same value in the volatility, v, field the former VPT thread in the previous section would have assigned to that tuple. This also gives the VPT threads the chance to expedite the behavior of the *ScaleGate* instance they share with the PPT threads in a similar manner as the PPT threads in the two previous iterations of the stream processing engine expedited the behavior of the output *ScaleGate* instance.
- VPT **threads to the** PPT **threads** ***ScaleGate***: As reasoned above, the parallel VPT threads and the parallel PPT threads share in this sixth iteration of the stream processing engine also a *ScaleGate* instance having the VPT threads as a set of multiple writers and the PPT threads as a set of multiple readers. This way, as anticipated above, all the tuples concurrently assigned a volatility value by the VPT threads are received by all the PPT threads in the same order they were initially served by IT making it possible for them to expedite the behavior of the output *ScaleGate* instance using the heartbeat mechanism introduced in Section 9.4.1.
- PPT **threads to** OT ***ScaleGate***: The same way it was done in the previous iterations as initially discussed in Section 8.3, the PPT threads and OT keep sharing an instance of *ScaleGate* in which the PPT threads act as a set of multiple writers and OT acts as a single reader. This way, the set of physical streams of tuples output by the PPT threads is recombined again into a single physical stream letting OT assign the tuples the *ets* timestamp and serve them in the same order as IT added them to the input *ScaleGate* instance only after

they have been added the v and op fields by the corresponding VPT and PPT threads.

11.4 Behavior of the Operators

As anticipated in the previous sections, what the VPT threads do in this sixth iteration of the stream processing engine is distributing the volatility aggregation workload assigned in the previous iteration of the financial stream processing engine to the VPT thread. With this, the PPT threads in this sixth iteration of the financial stream processing engine keep having exactly the same behavior as in the previous one given the fact that the stream of tuples they retrieve from the *ScaleGate* instance they share with the parallel VPT threads has exactly the same information as the one they retrieved in the previous version of the stream processing engine from the *ScaleGate* instance they shared with the VPT thread had.

Section 11.4.1 below elaborates on the formal description of the parallel version of the sliding-window-based volatility aggregation operator and Section 11.4.2 integrates this operator in the data-structures diagram from Figure 11.3.

11.4.1 The Multi-Threaded Volatility Aggregation Operator

As introduced in Section 10.4.1 when formally defining the single-threaded version of the sliding-window based volatility aggregation operator executed by the VPT thread in the previous iteration of the stream processing engine, what the single-threaded volatility aggregation operator maintained for each traded symbol was a circular buffer of volatility aggregation windows. The size of the aforementioned buffer was determined by the MAXW boundary, derived from the window size, WS, and window advance, WA, parameters of the underlying sliding-window model as introduced in Equation 10.10, which determined the maximum number of windows to be maintained in memory at a time. This circular buffer made it possible for VPT to assign the tuples processed by the stream processing engine, which arrived in non-decreasing timestamp order, the volatility resulting from consuming the last window they did not contribute to, and to let each of them contribute to the up to MAXW windows they should contribute to according to their timestamp, ts.

As it could be seen in Section 10.4.1 when analyzing the cost of the single-threaded volatility aggregation operation in terms of execution time, the O(MAXW) contributions to the execution time overhead of the single-threaded operator came from the iteration over all the up to MAXW volatility aggregation windows every time a window was consumed and that window together with all the older windows still present in $wbuff$ had to be reset in the loops in lines 8-9 and 13-14 in Listing 10.2, and the iteration over all the up to MAXW windows a tuple contributed to, updating the affected windows in the loop in lines 15-16 in Listing 10.2. These O(MAXW) contributions are actually the reason, as anticipated in the introduction

of this chapter, why the problem of parallelizing the sliding-window-based volatility operator is approached in this chapter and it is worth analyzing them in order to determine how to properly parallelize the operator.

The first observation which is worth doing is about the inter-dependability among different windows in the underlying sliding-window model. As it could be seen in Section 10.4.1, one tuple can contribute to many windows according to its timestamp, which motivated the aforementioned long loops, but once a tuple has contributed to a window, this contribution does not affect at all how it contributes to other windows as it can be seen when analyzing the `update` procedure introduced in lines 4-7 in Listing 10.1. The same observation can be done for the `reset` and `consume` procedures introduced in the same listing. For this reason, is feasible to simply partition the $wbuff$ circular buffer formerly maintained by the VPT thread for each traded symbol letting each of the m VPT threads in the current iteration of the stream processing engine maintain a portion of $wbuff$ containing MAXWT volatility aggregation windows, as expressed in Equation 11.1 below, instead of all the MAXW volatility aggregation windows maintained by the former VPT thread in the previous iteration.

$$\text{MAXWT} = \lceil \text{MAXW}/m \rceil \tag{11.1}$$

Given the order in which the different windows in the former $wbuff$ circular buffer were iterated in all the loops in Listing 10.2, always in ascending window identifier order iterating through all the consecutive windows from the first window in the loop to the last one, partitioning the $wbuff$ array in a round robin fashion distributing each set of m consecutive volatility aggregation windows among the m available VPT threads, would ensure that the workload represented by the former loops is equally distributed among the m VPT threads. In particular, this would ensure that each of the m VPT threads would iterate through up to MAXWT windows, being the difference between the number of iterations of each loop performed by one VPT thread or another lower than or equal to one.

The last observations motivate the definition of two new transformations, in addition to the ones introduced in Section 10.4.1, in order to formally define the multi-threaded version of the sliding-windows based volatility aggregator.

The first of these transformations can be defined from the domain of circular buffer indexes introduced in Section 10.4.1, to the range of VPT thread identifiers, as expressed in Equation 11.2.

$$f : Z \rightarrow T \tag{11.2}$$

- Z: the circular buffer indexes space, $\{0, ..., \text{MAXW} - 1\}$.
- T: the VPT thread identifiers space, $\{0, ..., m - 1\}$.

The transformation BIDX_TID determines the identifier of the VPT thread to which the given index of the former $wbuff$ circular buffer of MAXW volatility aggregation windows will be assigned, as expressed in Equation 11.3.

$$\begin{aligned} \text{BIDX\_TID}: \quad Z &\longrightarrow T \\ bidx &\longrightarrow \text{BIDX\_TID}(bidx) = bidx \mod m \end{aligned} \tag{11.3}$$

The second of these transformations can be defined also from the domain of circular buffer indexes introduced in Section 10.4.1, to the range of circular buffer subset indexes maintained by each VPT thread, as expressed in Equation 11.4.

$$f : Z \rightarrow W \tag{11.4}$$

- Z: the circular buffer indexes space, $\{0, ..., \text{MAXW} - 1\}$.
- W: the circular buffer subset indexes space, $\{0, ..., \text{MAXWT} - 1\}$.

The transformation BIDX_TBIDX determines the index in the $wbuff$ subset maintained by the corresponding VPT thread to which the given index of the former $wbuff$ circular buffer will correspond, as expressed in Equation 11.5.

$$\begin{aligned} \text{BIDX\_TBIDX}: \quad Z &\longrightarrow W \\ bidx &\longrightarrow \text{BIDX\_TBIDX}(bidx) = \lfloor bidx/m \rfloor \end{aligned} \tag{11.5}$$

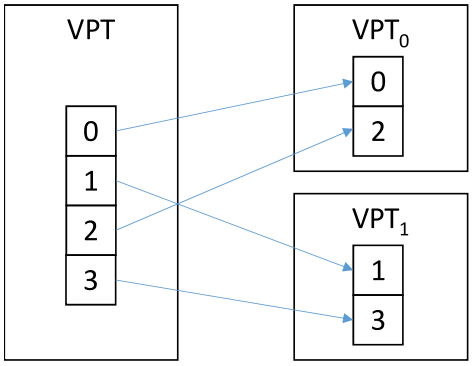


Figure 11.4: Illustration of the BIDX_TID, and BIDX_TBIDX transformations. $WS = 7$, $WA = 2$, $m = 2$

Figure 11.4 illustrates the BIDX_TID, and BIDX_TBIDX transformations introduced above building on top of the sliding-window diagram introduced in Figure 10.4 in Section 10.4.1 with $WS = 7$, $WA = 2$, and in this section $m = 2$, which led to MAXW $= 4$ as discussed in Section 10.4, and MAXWT $= 2$ according to Equation 11.1. As it can be seen, the windows which were assigned to index 0 in the former $wbuff$ maintained by VPT in the previous iteration of the financial stream processing engine are assigned to the VPT thread with identifier 0 in this sixth iteration of the financial stream processing engine by the BIDX_TID transformation, occupying in the subset of $wbuff$ maintained by this thread the position with index 0 according to the BIDX_TBIDX transformation. Similarly, the windows which were assigned to index 1 are assigned to the VPT thread with identifier 1 occupying

the position with index 0, the windows which were assigned to index 2 are assigned to the VPT thread with identifier 0 occupying this time the position with index 1, and finally the windows which were assigned to index 3 are assigned to the VPT thread with identifier 1 occupying the position with index 1.

Having formally defined the BIDX_TID, and BIDX_TBIDX transformations as well as the MAXWT boundary, the multi-threaded sliding-window based volatility aggregation operator executed by the VPT threads can be formally defined.

First of all, the sliding-window-based volatility aggregator for a given traded symbol maintaining a circular buffer of MAXW volatility aggregation windows formally defined as a data-structure with three variables in Section 10.4.1 can be redefined with the same three variables, but reducing the size of $wbuff$ from MAXW to MAXWT, and adding a fourth variable to store the corresponding VPT thread identifier:

- VPT **thread identifier** (tid): the VPT thread identifier variable, henceforth referred to as tid, stores the thread identifier of the VPT thread which owns the current instance of parallel volatility aggregator in order to keep track of which subset of the original $wbuff$ circular buffer of MAXW volatility aggregation windows is represented by the new $wbuff$ variable which maintains MAXWT of the original MAXW volatility aggregation windows.

For each of the tuples with the same symbol as the underlying stock associated to the parallel volatility aggregator there is, as it happened in Section 10.4.1 with the single-threaded volatility aggregator, one single way to interact with the it consisting on letting the parallel aggregator process the tuple. This procedure involves providing the tuple ts, tp, and tv fields to the aggregator in order for it to assign the tuple a volatility value resulting from the consumption of the window whose identifier is PREV_WIN(ts) if and only if that window is assigned by the BIDX_TID transformation to its subset of the original $wbuff$, and to update all the windows in the range {FIRST_WIN(ts), ..., LAST_WIN(ts)} maintained in its subset of the original $wbuff$ providing the tp and ts values as discussed in Section 10.4.1 when introducing the update procedure for the volatility aggregation window.

Listing 11.1: Multi-threaded sliding-window-based volatility aggregator for a single traded symbol pseudocode

```
1  pVolatilityAggregator
2    wbuff[MAXWT], lastts = 0, lastv = 0, tid
3
4  FIRST_WINL(ts, tid) =
5    if (BIDX_TID(WIN_BIDX(FIRST_WIN(ts))) <= tid)
6      FIRST_WIN(ts)+tid-BIDX_TID(WIN_BIDX(FIRST_WIN(ts)))
7    else
8      FIRST_WIN(ts)+tid-BIDX_TID(WIN_BIDX(FIRST_WIN(ts)))+m
9
10 PREV_WINR(ts, tid) =
```

```
11    if (BIDX_TID(WIN_BIDX(PREV_WIN(ts))) >= tid)
12      PREV_WIN(ts)+tid-BIDX_TID(WIN_BIDX(PREV_WIN(ts)))
13    else
14      PREV_WIN(ts)+tid-BIDX_TID(WIN_BIDX(PREV_WIN(ts)))-m
15
16  LAST_WINR(ts, tid) =
17    if (BIDX_TID(WIN_BIDX(LAST_WIN(ts))) >= tid)
18      LAST_WIN(ts)+tid-BIDX_TID(WIN_BIDX(LAST_WIN(ts)))
19    else
20      LAST_WIN(ts)+tid-BIDX_TID(WIN_BIDX(LAST_WIN(ts)))-m
21
22  processTuple(ts, tp, tv)
23    if (PREV_WIN(lastts) < PREV_WIN(ts) and
24        PREV_WIN(ts) - PREV_WIN(lastts) <= MAXW)
25      if (BIDX_TID(WIN_BIDX(PREV_WIN(ts))) == tid)
26        lastv = wbuff[BIDX_TBIDX(WIN_BIDX(PREV_WIN(ts)))]
27                .consume()
28      for (i = FIRST_WINL(lastts, tid) to
29            PREV_WINR(ts, tid)
30            in steps of size m)
31        wbuff[BIDX_TBIDX(WIN_BIDX(i))].reset()
32    else if (PREV_WIN(lastts) < PREV_WIN(ts) and
33              PREV_WIN(ts) - PREV_WIN(lastts) > MAXW)
34      if (BIDX_TID(WIN_BIDX(PREV_WIN(ts))) == tid)
35        lastv = 0
36      for (i = 0 to MAXWT - 1)
37        wbuff[i].reset()
38    for (i = FIRST_WINL(ts, tid) to
39          LAST_WINR(ts, tid)
40          in steps of size m)
41      wbuff[BIDX_TBIDX(WIN_BIDX(i))].update(tp, tv)
42    lastts = ts
43    if (BIDX_TID(WIN_BIDX(PREV_WIN(ts))) == tid)
44      return lastv
```

Listing 11.1 formally defines the parallel sliding-window-based volatility aggregator for a given traded symbol introduced above. Lines 1-2 represent the structure of the parallel aggregator, `pVolatilityAggregator`, with the aforementioned variable, $wbuff$, having space in this sixth iteration of the stream processing engine for MAXWT `volatilityWindow` instances, the $lastts$, and $lastv$ variables initialized to 0 by default, and the aforementioned tid variable storing the thread identifier of the corresponding VPT thread.

The `FIRST_WINL` transformation represented in lines 4-8 adapts the FIRST_WIN transformation introduced in Section 10.4.1 to the scope of the loops iterated by

the parallel volatility aggregator. Instead of determining the identifier of the first window a tuple contributes to; it determines the index of the first window assigned to the thread with identifier *tid* whose window identifier is higher than the identifier of the window determined by the former `FIRST_WIN` transformation.

Similarly, the `PREV_WINR`, and `LAST_WINR` transformations respectively represented in lines 10-14, and 16-20, respectively adapt the PREV_WIN, and LAST_WIN transformations introduced in Section 10.4.1 to the scope of the loops iterated by the parallel volatility aggregator. Instead of respectively determining the identifier of the last window a tuple does not contribute to, and the last window a tuple contributes to, they respectively determine the indexes of the last windows assigned to the thread with identifier *tid* whose window identifiers are lower than the identifier of the windows respectively determined by the former `PREV_WIN`, and `LAST_WIN` transformations.

The `processTuple` procedure is formally defined in lines 22-44, having the window consumption and replacement logic in lines 23-37, and the window update logic in lines 38-41. The assertion in lines 23-24 corresponds to the assertion in lines 5-6 in Listing 10.2 which checked whether or not the processed tuple given its timestamp, ts, should trigger the consumption of a window present in the original $wbuff$. In case this assertion holds, the window cannot be directly consumed and the resulting volatility stored in the *lastv* variable as it was done in line 7 in Listing 10.2 because in this iteration of the stream processing engine, a window present in the original $wbuff$ is only present in one of the $wbuff$ subsets maintained by one of the VPT threads. For this reason, lines 26-27, which perform the aforementioned window consumption and *lastv* variable update actions performed in line 7 in Listing 10.2, are protected by the assertion in line 25 which verifies whether or not the volatility aggregation window to be consumed is present in the current VPT thread view of the original $wbuff$. The loop in lines 28-31 corresponds to the loop in lines 8-9 in Listing 10.2 and makes use of the aforementioned `FIRST_WINL` and `PREV_WINR` transformations to reset in line 31 all the windows present in the current VPT thread view of $wbuff$ which need no longer be maintained in order to start keeping track of the subset of the last windows the processed tuple may contribute to which were not yet present in the current VPT thread view of $wbuff$. Exactly the same way as lines 5-31 in Listing 11.1 correspond to lines 5-9 in Listing 10.2 with the addition of the assertion in line 25 in Listing 11.1, lines 32-37 in Listing 11.1 correspond to lines 10-14 in Listing 10.2 with the addition of the assertion in line 34 in Listing 11.1 which performs exactly the same check as the one in line 25.

Having updated in case of need the *lastv* variable and reset the outdated tuples in the current VPT thread view of $wbuff$ in lines 23-37, the tp and tv values of the tuple being processed can be used to update all the windows present in the current VPT thread view of $wbuff$ the processed tuple contributes to according to its timestamp, ts. This is done using the aforementioned `FIRST_WINL` and `LAST_WINR` transformations in the loop in lines 38-41 which equitatively subdivides the workload of the loop in lines 15-16 in Listing 10.2 among the m parallel VPT threads. Once

all the windows have been updated, the *lastts* variable is updated in line 42 as it was done in line 17 in Listing 10.2 to keep track of the tuple which was just processed. Finally, the previously updated if needed *lastv* value is returned in line 44 representing the volatility value to be assigned to the tuple by the VPT thread if and only if, as checked in line 43, the window consumed to update the *lastv* value was ever present in the current VPT thread view of $wbuff$. This way, only the VPT thread in charge of that window adds the v field to the tuple to let the PPT threads price the corresponding option contract taking into account the aggregated volatility as in the previous iteration of the financial stream processing engine.

The same way as it was done in the previous iteration of the financial stream processing engine, having formally defined the parallel sliding-window-based volatility aggregator for a given traded symbol, the parallel sliding-window-based volatility aggregator for a set of traded symbols with symbol hash, sh, in the range $\{0, ..., \text{MAXSH} - 1\}$ can be formally defined as an array with MAXSH `pVolatility-Aggregator` instances indexed by symbol hash.

Listing 11.2: Multi-threaded sliding-window-based volatility aggregator for multiple traded symbols pseudocode

```
1 pVolatilityAggregators
2   aggrs[MAXSH]
3
4 processTuple(sh, ts, tp, tv)
5   return aggrs[sh].processTuple(ts, tp, tv)
```

Listing 11.2 formally defines the multi-threaded sliding-window-based volatility aggregator for a set of traded symbols with symbol hash, $sh \in \{0, ..., \text{MAXSH} - 1\}$. As it can be easily seen, the only difference between the pseudocode in this listing and the pseudocode in Listing 10.3 is the name given to the `pVolatilityAggregators` data-structure in line 1 and the data-structure of the items contained in the *aggrs* array in line 2, which are `pVolatilityAggregator` instances in this sixth iteration of the financial stream processing engine instead of `volatilityAggregator` instances.

The last described procedure, `processTuple`, is the one executed by the parallel VPT threads every time a tuple is retrieved from the *ScaleGate* instance they share with IT. In case a volatility value is returned, which will happen, as discussed above, for one and only one VPT thread for each individual tuple, the tuple will be added the v field by that VPT thread and added to the *ScaleGate* instance shared with the PPT threads followed by a `NULL` tuple with the same sequence number as the non-`NULL` tuple added to the *ScaleGate* instance to avoid the double latency problem for single writers in a *ScaleGate* instance discussed in Section 8.4.1. The rest of the VPT threads for which a volatility value is not returned, execute the heartbeat logic introduced in Lines 10-14 in Listing 9.1 in Section 9.3 to expedite the behavior of the *ScaleGate* instance shared with the PPT threads exactly the same way these threads expedite the behavior of the *ScaleGate* instance they share with OT.

It is worth observing that the heartbeat mechanism plays a more crucial role in the behavior of the *ScaleGate* instance the VPT and PPT threads share than the role it played in the behavior of the *ScaleGate* instance shared by the PPT threads and OT. The reason for this is that while the PPT threads add tuples to the *ScaleGate* instance they share with OT in a round robin fashion having each of the n PPT threads adding one tuple to the *ScaleGate* instance for every n processed tuples, the VPT threads, given the algorithms described above, will have the first thread adding a set of tuples to the *ScaleGate* instance until the next window is consumed triggering the addition of tuples by the second thread instead of the first one, and so on. With this, the cycle until any VPT thread starts adding tuples again to the *ScaleGate* instance after it stops adding tuples is considerably longer than in the case of the PPT threads making the heartbeat mechanism crucial for the control of the latency as it will be seen in Sections 13.2.4 and 13.2.5 where the latency and throughput achieved when executing the parallel volatility aggregation stage of the stream processing engine respectively alone and followed by the options pricing stage are reported.

Before concluding this section, it is worth analyzing the complexity of the multi-threaded sliding-window based volatility aggregation operator in terms of execution time and memory. As it can easily be seen after a brief analysis of the pseudocode introduced in Listings 11.1, and 11.2, in addition to the three parameters affecting the cost of the operator in terms of execution time and memory identified in Section 11.4.1, MAXSH, WS, and WA, the latter two ones determining together the MAXW boundary as described in Equation 10.10, the number of parallel VPT threads, m, also determines the cost of the operator. As anticipated when distributing the $MAXW$ windows in the former $wbuff$ array among the m available VPT threads and as it can be verified analyzing the loops in lines 28-31, 36-37, and 38-41 in Listing 11.1, the cost of the parallel volatility aggregation operator in terms of execution time is the one expressed in Equation 11.6:

$$O(\text{MAXWT}) = O(\text{MAXWT}/m) = O(WS/WA/m) \tag{11.6}$$

Following a similar reasoning as when analyzing the expected latency for each tuple in the fourth iteration of the stream processing engine in Section 9.4.1, in which the PT threads retrieve tuples from a *ScaleGate* instance as multiple readers and serve tuples to a *ScaleGate* instance as multiple writers, expediting its behavior with the heartbeat mechanism, exactly the same way as the VPT threads do in this iteration, the expected latency for each tuple making use of the parallel volatility aggregation operator is the one expressed in Equation 11.7:

$$L(m) = O(\text{MAXWT}/m) + O(\log m) \tag{11.7}$$

In terms of throughput of the multi-threaded volatility aggregator, given that the volatility aggregation overhead is equally distributed among the m parallel VPT threads, and that the synchronization overhead derived from using the input and

output *ScaleGate* instances and executing the newly introduced control logic with respect to the previous iteration is orders of magnitude smaller than the multi-threaded operator overhead ($O(\log m) << O(WS/WA/m)$), the throughput is expected to grow linearly with the number of threads as expressed in Equation 11.8.

$$T(m) = m \cdot T(1) \tag{11.8}$$

Altogether, the expected latency and throughput of the multi-threaded volatility aggregator having H greater than 0 and close to 1 can be plotted as a function of the number of parallel VPT threads used.

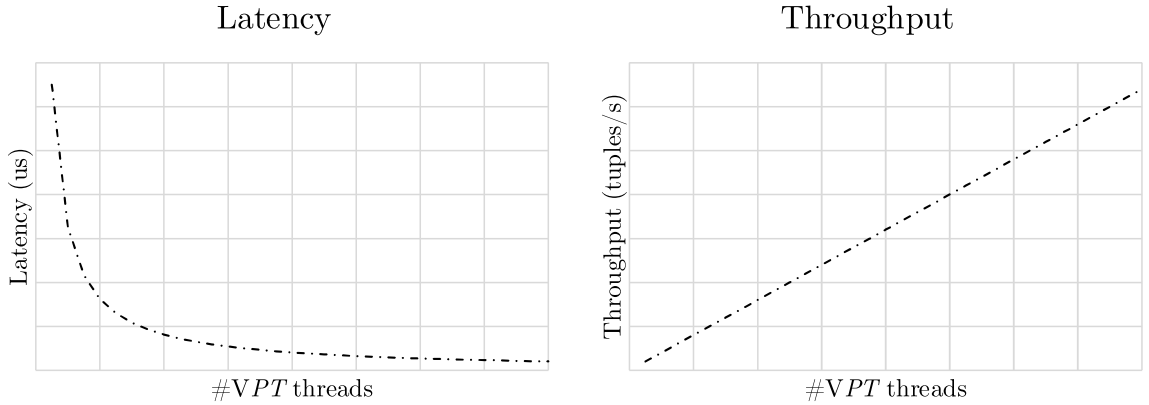


Figure 11.5: Latency and throughput as a function of the number of parallel VPT threads

Figure 11.5 plots the expected latency and throughput of the parallel volatility aggregation operator introduced in this section. It is worth observing how in terms of latency, as opposed to the expectations shown when executing in parallel the options pricing operator, the tendency is to decrease inversely proportionally to the number of volatility aggregation threads, the reason for this is that in this case, the workload for each individual tuple is distributed among all the volatility aggregation threads allowing the latency to decrease. In terms of expected throughput, the tendency is similar to the one observed when executing the options pricing operator in parallel in previous iterations.

In terms of memory, given the fact that the former $wbuff$ array with MAXW volatility aggregation windows is partitioned among the m VPT threads maintaining each or them up to MAXWT = $\lceil$MAXW/$m\rceil$ volatility aggregation windows, the cost of the operator in terms of memory per thread is the one expressed in Equation 11.9, and the cost of the operator at a global level is the one expressed in Equation 11.10 which matches exactly the cost of the single-threaded operator introduced in the previous chapter:

$$O(\text{MAXSH} \cdot \text{MAXWT}) = O(\text{MAXSH} \cdot \text{MAXW}/m) = O(\text{MAXSH} \cdot WS/WA/m) \tag{11.9}$$

$$O(\text{MAXSH} \cdot \text{MAXW}) = O(\text{MAXSH} \cdot WS/WA) \tag{11.10}$$

11.4.2 Integrating the Multi-Threaded Volatility Aggregation Operator

Getting back to the stream processing engine diagram from Section 11.3, it is worth noticing that, as anticipated in Section 11.4.1, and the same way as in the previous iteration of the stream processing engine, the two main values that the VPT threads need to be provided with in order to initialize each of the `pVolatilityAggregator` instances maintained by the `pVolatilityAggregators` instance it uses to support the sliding-windows model, discussed in detail in the previous section, are WS and WA. Out of the six values that the binomial options pricing operator uses as an input, tp, os, om, rli, v, and N, two of them, tp, and v, can be retrieved by the PPT threads from the tuples served by the VPT threads to the *ScaleGate* instance they share with them as described in Section 11.3. The rest of the values are simply taken as constants with the default values introduced in Section 6.4.2. In addition to this, in this sixth iteration of the stream processing engine the H parameter needs to be specified and provided to both the parallel VPT threads and the PPT threads to determine whether to use or not the heartbeat mechanism and how exhaustively to use it.

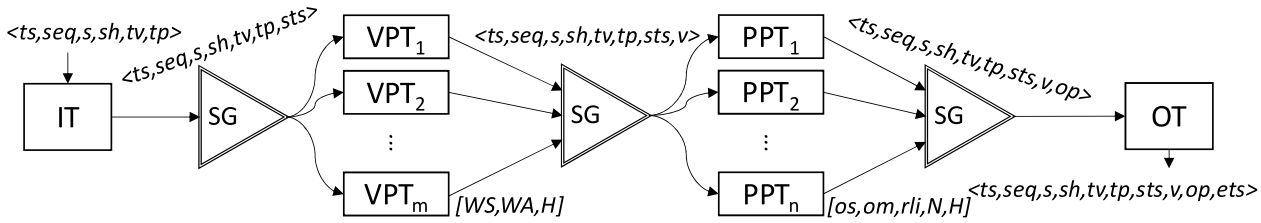


Figure 11.6: Operators and used constants

Figure 11.6 extends the data-structures diagram introduced in Figure 11.3 to represent the aforementioned constant values between square brackets close to the boxes representing the VPT and PPT threads, which execute respectively the multi-threaded sliding-window-based volatility aggregation operator introduced in the previous section and the multi-threaded binomial options pricing operator introduced in Chapter 9 retrieving the volatility, v, as in the previous iteration, from the tuples instead of taking it as a fixed outdated constant.

Overall, this sixth iteration of the stream processing engine, as anticipated in Section 9.4.2, responds to the second line of improvement outlined in the last paragraph in Section 6.4.2 concluding Chapter 6.

12

Multi-Threaded Volatility Aggregation and Stream Matching

In the previous chapter, the sliding-window-based volatility aggregator introduced in Chapter 10 was parallelized linearly increasing the throughput expectations at the same time as the latency expectations were reduced. However, out of the six values that the options pricing threads provided to the binomial options pricing operator as an input, tp, os, om, rli, v, and N, yet only two of them, tp, and v, could be retrieved from the tuples served by the volatility aggregation threads in order to price option contracts.

In a realistic options pricing setup, in order to profit from the operators so far introduced and optimized, it is necessary to allow for the specification of the option strike, os, option maturity, om, and assumed risk-less interest rate, rli, for each of the options contracts to be priced. As it can be understood, whereas the financial information contained in the tuples processed in the previous iterations of the stream processing engine represented the behavior of the underlying market independently of the actions of the clients willing to acquire option contracts, the aforementioned information, os, om, and rli, is solely determined by the specific desires the clients of the financial stream processing engine may have, which needs to be modeled, given its different nature, as a separate logical stream of data different from the financial stream of tuples that have been used in the previous iterations.

For this reason, in this chapter the functionality of the stream processing engine introduced in the previous iteration will be extended in order to support two different logical streams, the first of them with the same tuples as the financial input stream that was already processed by the previous versions of the stream processing engine, the second of them allowing for the specification of the different option contracts settings.

The following sections elaborate on how the *ScaleGate* instance the IT and VPT threads shared in the previous iteration of the stream processing engine can be used to receive not one but two logical streams of tuples, and how the functionality of the VPT threads can easily be extended to match the two streams of tuples producing an extended stream of tuples to be provided to the PPT threads summarizing both the most relevant and up to date information retrieved from the financial stream of tuples and the options settings information retrieved from the new stream of option settings tuples.

12.1 Involved Threads

As anticipated above, this seventh iteration of the financial stream processing engine profits from the semantics of the *ScaleGate* instance which the input thread, IT, and the volatility aggregation threads, VPT, shared in the previous iteration in order to input not only one logical stream of financial tuples as it was done in all the previous iterations of the stream processing engine, but also a second logical stream of option settings in order to specify for each option contract to be priced the option strike, *os*, option maturity, *om*, and assumed risk-less interest rate, *rli*.

With this, the former IT thread is renamed to FIT to specify that it takes care of serving the financial input stream of tuples, and the new SIT thread is added to serve the settings input stream of tuples. In addition to this, the functionality of the former VPT threads is extended in order to match the two streams of tuples as it will be explained in detail in Section 12.4.1, which motivates its change of name in this iteration of the financial stream processing engine from VPT to WPT.

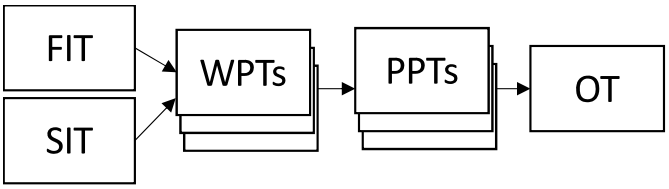


Figure 12.1: Involved threads

Figure 11.1 outlines how threads are arranged in this seventh iteration of the financial stream processing engine. As anticipated above, the former IT thread has been replaced by the FIT and SIT threads, the former serving the same stream of tuples IT served and the latter serving the new stream of tuples to determine for each specific option contract to be priced the option strike, *os*, option maturity, *om*, and the assumed risk-less interest rate, *rli*. The former VPT threads have been renamed to WPT given their extended functionality, and the PPT threads and OT remain exactly the same as in the previous iteration with the only difference that the PPT threads will now retrieve from the tuples served by the WPT threads, not only the *tp* and *v* fields, but also the *os*, *om*, and *rli* fields.

12.2 Structure of the Tuples

In this seventh iteration of the financial stream processing engine, input financial tuples are retrieved from the input financial stream introduced in Section 5.1.1 by FIT, and input options settings tuples are retrieved from the input options settings stream introduced in Section 5.1.2 by SIT. Both streams of tuples are matched by the WPT threads as it will be explained in detail in Section 12.4.1 resulting in an extended streams of tuples which will be further extended by the PPT threads and OT allowing the latter thread to output a stream of tuples with the same options settings, and in the same order as the tuples served by the SIT thread, with the most

up to date financial information added by the WPT threads from the tuples served by the FIT thread and with an option price assigned based on all this information by the PPT threads.

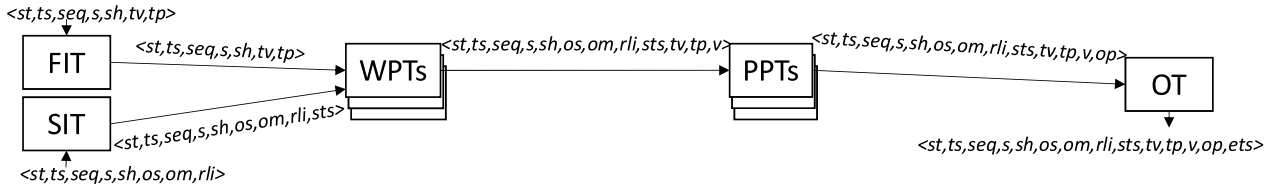


Figure 12.2: Structure of the tuples

Figure 12.2 extends the involved threads diagram introduced in Figure 12.1 specifying the structure of the tuples in each transition from one thread to another as well as the structure of the tuples read by FIT and SIT, and the tuples output by OT.

Each tuple read by FIT from the input financial stream, as described in Section 5.1.1, represents, as in the previous iterations of the stream processing engine the tuples read by IT represented, a trade transaction in which a given number of shares of a given stock are traded at a given price per share, at a given instant of time. To model this, the input financial tuples contained six fields in the previous iterations of the stream processing engine, $<ts$, seq, s, sh, tv, $tp>$. However, given the addition of a second logical stream of tuples, a new additional field needs to be added to these tuples to specify the logical stream type once they reach the WPT threads, extending the structure of the tuples retrieved by FIT from six to seven fields $<st$, ts, seq, s, sh, tv, $tp>$:

- **Stream Type** (st): the stream type field, henceforth referred to as st, is the value identifying the logical stream of tuples to which the tuple belongs. This field is internally modeled as a `char` value uniquely identifying the logical stream type. Its value is `'F'` for all the tuples belonging to the stream served by FIT.

Each tuple read by SIT from the input options settings stream, as described in Section 5.1.2, represents an option contract to be priced for a given underlying stock, at a given time, with a given strike and maturity assuming a given risk-less interest rate. To model this, the input financial tuples contain eight fields, $<st$, ts, seq, s, sh, os, om, $rli>$:

- **Stream Type** (st): the stream type field, henceforth referred to as st, has the same function and internal representation as the st field in the financial tuples read by FIT described above. Its value is `'S'` for all the tuples belonging to the stream served by SIT.
- **Timestamp** (ts): the timestamp field, henceforth referred to as ts, is the value representing the physical timestamp when the option contract starts being effective. This field is internally modeled the same way as the so called field is modeled in the financial input tuples read by FIT.

- **Sequence number** (*seq*): the sequence number field, henceforth referred to as *seq*, is the value representing the unique sequence number assigned to each tuple in the input options settings stream and behaves as a logical timestamp indicating the precedence order of the tuples in the stream the same way the so called field behaved in the financial stream of tuples read by FIT. This field is internally modeled the same way as the so called field is modeled in the financial input tuples read by FIT.
- **Symbol** (*s*): the symbol field, henceforth referred to as *s*, is the value representing the underlying security symbol as the so called field represented in the financial input tuples read by FIT. This field is internally modeled the same way as the so called field is modeled in the financial input tuples read by FIT.
- **Symbol hash** (*sh*): the symbol hash field, henceforth referred to as *sh*, is the value representing the unique hash value assigned to the underlying security symbol as the so called field represented in the financial input tuples read by FIT. This field is internally modeled the same way as the so called field is modeled in the financial input tuples read by FIT, and it is used by the WPT threads to match the two streams of tuples together with the *ts* field as it will be explained in detail in Section 12.4.1.
- **Option strike** (*os*): the option strike field, henceforth referred to as *os*, represents the price at which the European call option contract gives the buyer the right to buy shares of the underlying asset at the expiration date. This field is internally modeled as a `double` value accounting for the price the buyer has the right, but not the obligation, to pay per share of the underlying asset at the expiration date, in accordance to how it is provided to the options pricing operator as specified in Section 6.4.1.
- **Option time to maturity** (*om*): the option time to maturity field, henceforth referred to as *om*, represents the time from the option pricing instant to the expiration date of the option contract. This field is internally modeled as a `double` value accounting for the number of years from the option pricing instant to the expiration date of the priced option contract, in accordance to how it is provided to the options pricing operator as specified in Section 6.4.1.
- **Risk-less interest rate** (*rli*): the risk-less interest rate field, henceforth referred to as *rli*, represents the assumed to be guaranteed by the market risk-less interest rate. This value is internally modeled as a `double` value accounting for the assumed guaranteed continuous risk-less interest rate, in accordance to how it is provided to the options pricing operator as specified in Section 6.4.1.

The same way as the IT thread in the previous iterations of the financial stream processing engine added all the tuples the *sts* timestamp to keep track of when the tuples started being processed, it is the new SIT thread in this iteration of the stream processing engine the one in charge of adding the tuples the *sts* timestamp given the fact that the options settings tuples are the ones which really trigger the event of pricing an option contract.

12.3 Used Data-Structures

As anticipated earlier in this chapter, this iteration of the stream processing engine profits from the semantics of the *ScaleGate* instance the former IT and VPT threads shared in the previous iteration of the stream processing engine to let the newly introduced FIT and SIT threads provide tuples belonging to the financial and options settings logical streams ordered by logical and physical timestamp to the WPT threads.

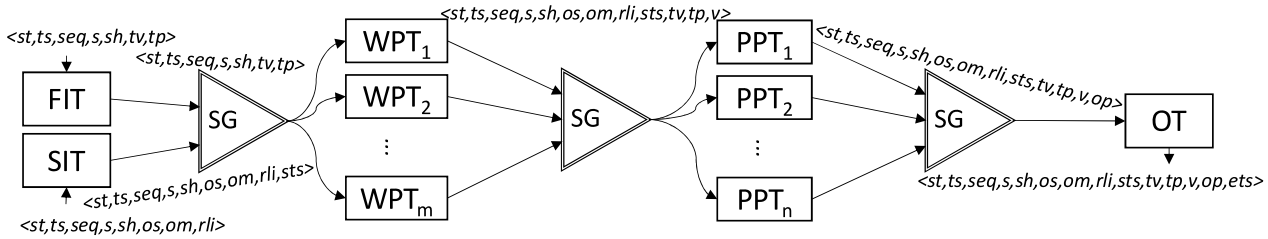


Figure 12.3: Used data-structures

Figure 12.3 extends the structure of the tuples diagram introduced in Figure 12.2 specifying the data-structures used in each transition from one thread to another. As it can be seen, the only difference with respect to the data-structures diagram introduced in Figure 11.3 in Section 11.3, is the fact that now the FIT and SIT threads act as multiple writers in the first *ScaleGate* instance in contrast with how the IT thread acted in the previous iteration of the stream processing engine as a single writer.

The WPT threads act as multiple readers in the first *ScaleGate* instance and multiple writers in the second *ScaleGate* instance the same way as the former VPT threads did in the previous iteration of the stream processing engine. The PPT threads act as multiple readers in the second *ScaleGate* instance and multiple writers in the third *ScaleGate* instance the same way they did in the previous iteration of the stream processing engine. And OT acts as a single reader in the third *ScaleGate* instance as it did in the previous iteration of the stream processing engine.

12.4 Behavior of the Operators

As anticipated in Section 12.2, in addition to updating the underlying sliding-window model as the VPT threads did in the previous iteration of the financial stream processing engine, the WPT threads in this iteration of the stream processing engine will match the tuples served by SIT with the most recent tuples served by FIT with the same underlying stock symbol in order to extend the tuples served by SIT with the most up to date financial information from the financial stream served by FIT. This allows the PPT threads to provide the options pricing procedure with the *os*, *om*, and *rli* fields specified in the options settings stream served by SIT, the *tp* field from the most recent financial tuple served by FIT which matched the corresponding options settings tuple served by SIT, and the v field the volatility

aggregation operator assigned to the aforementioned financial tuple provided by FIT.

Section 12.4.1 below elaborates on the formal description of the aforementioned parallel sliding-window-based volatility aggregation and stream matching operator and Section 12.4.2 integrates this operator in the data-structures diagram from Figure 12.3.

12.4.1 The Multi-Threaded Volatility Aggregation and Stream Matching Operator

In order to formally define the multi-threaded volatility aggregation and stream matching operator, it is worth starting by extending the definition of the sliding-window-based multi-threaded volatility aggregator for a given traded symbol introduced in Section 11.4.1, `pVolatilityAggregator`, which was defined as a data-structure with four variables. The first of these variables, $wbuff$, maintained up to MAXWT of all the MAXW windows in its view of the global $wbuff$ circular buffer introduced in Section 10.4.1, the second and third of them, $lastts$, and $lastv$, kept track respectively of the last processed financial tuple physical timestamp and the volatility value assigned to it, and the fourth of them, tid, stored the corresponding VPT thread identifier, WPT thread identifier in this iteration of the financial stream processing engine. In addition to these four variables, in order to support the stream matching functionality, two additional variables are needed to keep track of the tp and tv fields of the last processed financial tuple:

- **Last trade price** ($lasttp$): the last trade price variable, henceforth referred to as $lasttp$, stores the tp field of the last financial tuple processed by the aggregator and stream matcher in order to make it possible for the WPT threads to add the option settings tuples the most recently seen trade price in the financial stream of tuples.
- **Last trade volume** ($lasttv$): the last trade volume variable, henceforth referred to as $lasttv$, stores the tv field of the last financial tuple processed by the aggregator and stream matcher in order to make it possible for the WPT threads to add the option settings tuples the most recently seen trade volume in the financial stream of tuples.

For each of the financial or options settings tuples with the same symbol as the underlying stock associated to the multi-threaded volatility aggregator and stream matcher there is, as it happened in Section 11.4.1 with the multi-threaded volatility aggregator, one single way to interact with it consisting on letting the parallel aggregator and stream matcher process the tuple. This procedure involves, in case the tuple belongs to the financial stream of tuples served by FIT, performing the same actions the aggregator introduced in the previous iteration of the stream processing engine performed but keeping track of the assigned volatility and the provided tp and tv fields instead of outputting the assigned volatility, and in case the tuple belongs to the options settings stream of tuples served by SIT, providing the volatility, trade price and trade volume values kept track of to let the corresponding WPT

thread extend the tuple with the most up to date financial information from the financial stream of tuples.

Listing 12.1: Multi-threaded sliding-window-based volatility aggregator and stream matcher for a single traded symbol pseudocode

```
1  pVolatilityAggregatorM
2    wbuff[MAXWT]
3    lastts = 0, lastv = 0, tid, lasttv = 0, lasttp = 0
4
5  processTuple(st, ts, [tp, tv])
6    if (st == 'F')
7      if (PREV_WIN(lastts) < PREV_WIN(ts) and
8          PREV_WIN(ts) - PREV_WIN(lastts) <= MAXW)
9        if (BIDX_TID(WIN_BIDX(PREV_WIN(ts))) == tid)
10         lastv = wbuff[BIDX_TBIDX(WIN_BIDX(PREV_WIN(ts)))]
11                 .consume()
12       for (i = FIRST_WINL(lastts, tid) to
13            PREV_WINR(ts, tid)
14            in steps of size m)
15         wbuff[BIDX_TBIDX(WIN_BIDX(i))].reset()
16     else if (PREV_WIN(lastts) < PREV_WIN(ts) and
17              PREV_WIN(ts) - PREV_WIN(lastts) > MAXW)
18       if (BIDX_TID(WIN_BIDX(PREV_WIN(ts))) == tid)
19         lastv = 0
20       for (i = 0 to MAXWT - 1)
21         wbuff[i].reset()
22     for (i = FIRST_WINL(ts, tid) to
23          LAST_WINR(ts, tid)
24          in steps of size m)
25       wbuff[BIDX_TBIDX(WIN_BIDX(i))].update(tp, tv)
26     lastts = ts
27     if (BIDX_TID(WIN_BIDX(PREV_WIN(ts))) == tid)
28       lasttp = tp
29       lasttv = tv
30   else if (st == 'S')
31     if (BIDX_TID(WIN_BIDX(PREV_WIN(ts))) == tid)
32       return <lasttp, lasttv, lastv>
```

Listing 12.1 formally defines the parallel sliding-window-based volatility aggregator and stream matcher for a given traded symbol introduced above. Lines 1-3 represent the structure of the parallel aggregator and stream matcher, `pVolatilityAggregatorM`, with the aforementioned variable, $wbuff$, having space as in the previous iteration of the stream processing engine for MAXWT `volatilityWindow` instances, the *lastts*, and *lastv* variables initialized to 0 by default, the aforementioned *tid* variable storing the thread identifier of the corresponding WPT thread, and the new *lastv*, and *lasttp* variables also initialized to 0 by default.

The `processTuple` procedure is formally defined in lines 5-32. The two last input parameters, corresponding to the *tp*, and *tv* fields of the financial tuples are represented in brackets as optional parameters because only when processing tuples belonging to the financial stream these parameters can be provided and used inside the procedure. The assertion in line 6 checks whether the tuple belongs to the financial stream of tuples served by FIT or the options settings stream of tuples served by SIT. In case it belongs to the former, lines 7-29 are executed which perform the same actions as lines 23-44 in Listing 11.1 with the exception of lines 28 and 29 which, instead of returning the volatility value to extend the financial tuples as line 44 in Listing 11.1 did in the previous iteration, keep track of the processed tuple trade price, *tp*, and trade volume, *tv*, updating respectively the *lasttp*, and *lasttv* variables and not returning anything. This way, the WPT threads can execute the heartbeat mechanism for every tuple they process belonging to the financial input stream, and any time a tuple belonging to the options settings stream arrives, the values stored in the *lasttp* and *lasttv* correspond to the most recently processed tuple belonging to the financial stream of tuples served by FIT. In case the assertion in line 6 fails because the tuple belongs to the options settings stream, the assertion in line 30 confirms if the tuple belongs to the options settings stream and in case it does, the assertion in line 31 checks if the window consumed to update the *lastv* value was ever present in the current WPT thread view of $wbuff$, and line 32 returns, only for the WPT thread in which the assertion in line 31 holds, the *lasttp*, *lasttv*, and *lastv* values to let the WPT thread extend the tuple with the financial information from the last processed financial tuple.

The same way as it was done in the previous iteration of the financial stream processing engine, having formally defined the parallel sliding-window-based volatility aggregator and stream marcher for a given traded symbol, the parallel sliding-window-based volatility aggregator and stream matcher for a set of traded symbols with symbol hash, *sh*, in the range $\{0, ..., \text{MAXSH} - 1\}$ can be formally defined as an array with MAXSH `pVolatilityAggregatorM` instances indexed by symbol hash.

Listing 12.2: Multi-threaded sliding-window-based volatility aggregator and stream matcher for multiple traded symbols pseudocode

```
1 pVolatilityAggregatorMs
2   aggrs[MAXSH]
3
4 processTuple(sh, ts, [tp, tv])
5   if (st == 'F')
6     aggrs[sh].processTuple(ts, tp, tv)
7   else if (st == 'S')
8     return aggrs[sh].processTuple(ts)
```

Listing 12.2 formally defines the multi-threaded sliding-window-based volatility aggregator and stream matcher for a set of traded symbols with symbol hash, $sh \in$

$\{0, ..., \text{MAXSH} - 1\}$. Similarly as it happened in the previous chapter, the pseudocode in this listing differs from the pseudocode in Listing 11.2 in the name given to the `pVolatilityAggregatorMs` data-structure in line 1, and the data-structure of the items contained in the $aggrs$ array in line 2, which are `pVolatilityAggregatorM` instances in this seventh iteration of the financial stream processing engine instead of `pVolatilityAggregator` instances. In addition to this, the two last input parameters in the `processTuple` procedure in line 4, corresponding to the tp, and tv fields of the financial tuples are represented in brackets as optional parameters as it happened in line 5 in Listing 12.1, and the assertions in lines 5, and 7 are added to determine whether to provide or not the tp, and tv input parameters to the corresponding volatility aggregator and stream matcher instance according to whether the tuple belongs to the financial stream of tuples served by FIT or the options settings stream of tuples served by SIT.

The last described procedure, `processTuple`, is the one executed by the parallel WPT threads every time a tuple is retrieved from the *ScaleGate* instance they share with FIT and SIT. In case the $lasttp$, $lasttv$, and $lastv$ values are returned, which will happen, as discussed above, for one and only one WPT thread for each individual options settings tuple, the tuple will be added the tp, tv, and v fields by that WPT thread and added to the *ScaleGate* instance shared with the PPT threads followed by a `NULL` tuple with the same sequence number as the non-`NULL` tuple added to the *ScaleGate* instance to avoid the double latency problem for single writers in a *ScaleGate* instance discussed in Section 8.4.1. In case nothing is returned, the heartbeat logic introduced in Lines 10-14 in Listing 9.1 in Section 9.3 is executed to expedite the behavior of the *ScaleGate* instance shared with the PPT threads exactly the same way it was done in the previous iteration.

Before concluding this section, it is worth analyzing the complexity of the multi-threaded sliding-window based volatility aggregation and stream matching operator in terms of execution time and memory. As it can easily be seen after a brief analysis of the pseudocode introduced in Listings 12.1, and 12.2, the same four parameters identified in Section 11.4.1, MAXSH, WS, and WA, the latter two ones determining together the MAXW boundary as described in Equation 10.10, and the number of parallel WPT threads, m, determine the cost of the operator in terms of execution time and memory. Given the fact that the assertions in lines 6 and 30 in Listing 12.1, the assertion in line 31 in Listing 12.1, the variable updates in lines 28 and 29 in Listing 12.1, the return action in line 32 in Listing 12.1, and the assertions in lines 5 and 7 in Listing 12.2 have all $O(1)$ execution time cost and the rest of the code is exactly the same as the one analyzed in the previous iteration of the stream processing engine, the cost of the parallel volatility aggregation and stream matching operator in terms of execution time has the same big O notation as the cost of the parallel volatility aggregation operator as expressed in Equation 12.1:

$$O(\text{MAXWT}) = O(\text{MAXWT}/m) = O(WS/WA/m) \tag{12.1}$$

Following a similar reasoning as in the previous iteration, the expected latency for each tuple making use of the parallel volatility aggregation as stream matching operator is the one expressed in Equation 12.2:

$$L(m) = O(\text{MAXWT}/m) + O(\log m) \tag{12.2}$$

In terms of throughput of the multi-threaded volatility aggregator and stream matcher, given that the volatility aggregation overhead is equally distributed among the m parallel WPT threads, and that the synchronization overhead derived from using the input and output *ScaleGate* instances and executing the newly introduced control logic with respect to the previous iteration is orders of magnitude smaller than the multi-threaded operator overhead ($O(\log m) << O(WS/WA/m)$), the throughput is expected to grow linearly with the number of threads as expressed in Equation 12.3.

$$T(m) = m \cdot T(1) \tag{12.3}$$

Altogether, the expected latency and throughput of the multi-threaded volatility aggregator and stream matcher having H greater than 0 and close to 1 can be plotted as a function of the number of parallel WPT threads used.

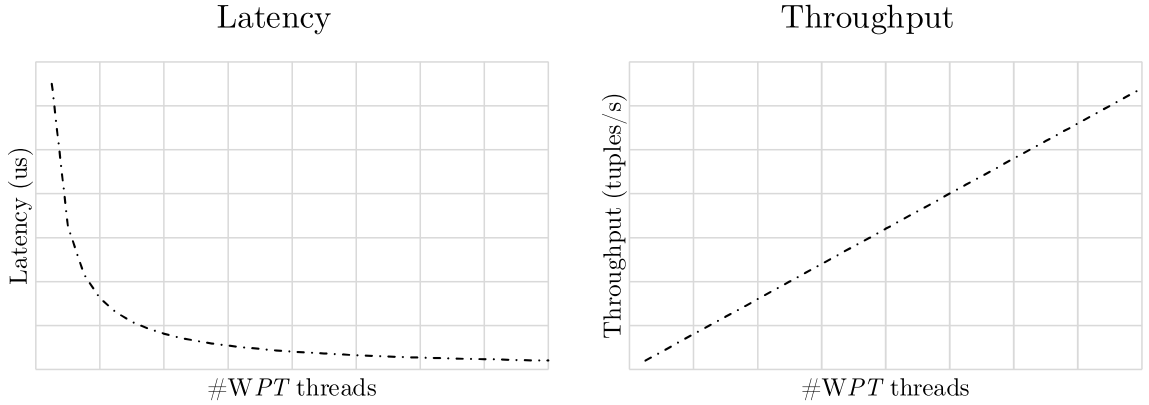


Figure 12.4: Latency and throughput as a function of the number of parallel WPT threads

Figure 12.4 plots the expected latency and throughput of the parallel volatility aggregation and stream matching operator introduced in this section. The observed expected latency and throughput trends are the same as in the previous iteration for the parallel volatility aggregation operator.

In terms of memory, given the fact that the two new variables added to the `pVolatilityAggregatorM` data structure, *lasttv*, and *lasttp*, occupy O(1) space in memory each, the cost of the operator in terms of memory per thread is, as in the case of the operator introduced in the previous chapter, the one expressed in Equation 12.4, and the cost of the operator at a global level is the one expressed

in Equation 12.5 which matches exactly the cost of the single-threaded operator introduced in Chapter 10:

$$O(\text{MAXSH} \cdot \text{MAXWT}) = O(\text{MAXSH} \cdot \text{MAXW}/m) = O(\text{MAXSH} \cdot WS/WA/m) \tag{12.4}$$

$$O(\text{MAXSH} \cdot \text{MAXW}) = O(\text{MAXSH} \cdot WS/WA) \tag{12.5}$$

12.4.2 Integrating the Multi-Threaded Volatility Aggregation and Stream Matching Operator

Getting back to the stream processing engine diagram from Section 12.3, it is worth noticing that, as anticipated in Section 12.4.1, and the same way as in the previous iteration of the stream processing engine, the two main values that the WPT threads need to be provided with in order to initialize each of the `pVolatilityAggregatorM` instances maintained by the `pVolatilityAggregatorMs` instance it uses to support the sliding-windows model discussed in detail in the previous section are WS and WA. Out of the six values that the binomial options pricing operator uses as an input, tp, os, om, rli, v, and N, five of them, tp, os, om, rli and v, can be retrieved in this seventh iteration of the stream processing engine by the PPT threads from the tuples served by the WPT threads to the *ScaleGate* instance they share with them as described in Section 12.3. The remaining N value is simply taken as constant with the default value introduced in Section 6.4.2 as it is understood to be an algorithm performance and result quality configuration value independent from the behavior of the market and the desires of the stream processing engine users. In addition to this, as in the sixth iteration of the stream processing engine, the H parameter needs to be specified and provided to both the parallel WPT threads and the PPT threads to determine whether to use or not the heartbeat mechanism and how exhaustively to use it.

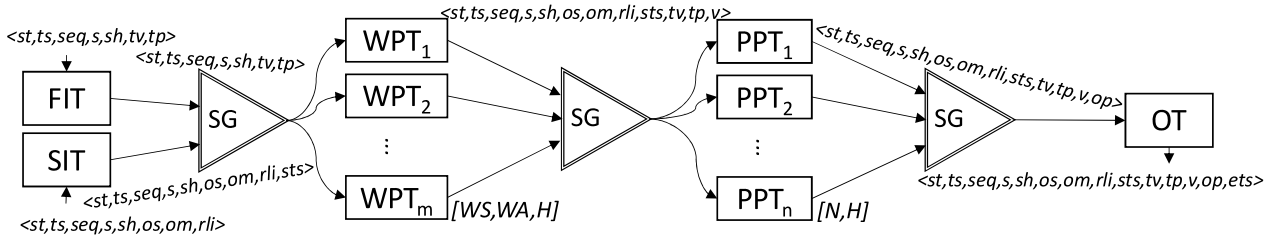


Figure 12.5: Operators and used constants

Figure 12.5 extends the data-structures diagram introduced in Figure 12.3 to represent the aforementioned constant values between square brackets close to the boxes representing the WPT and PPT threads, which execute respectively the multi-threaded sliding-window-based volatility aggregation and stream matching operator introduced in the previous section and the multi-threaded binomial options pricing operator introduced in Chapter 9 retrieving in addition to the trade price, tp and volatility, v, from the tuples, also the option strike, os, option maturity, om,

and risk-less interest rate, *rli*, this instead of taking them as a fixed not-very-useful constants.

Overall, this seventh iteration of the stream processing engine, as anticipated in Section 9.4.2, responds to the second line of improvement outlined in the last paragraph in Section 6.4.2 concluding Chapter 6.

13

Experimental Results and Analysis

Having formally described in the previous seven chapters, Chapters 6-12, the financial stream processing engine developed in the context of this Thesis, this chapter reports the throughput and latency median metrics obtained when executing in the machines introduced in Section 5.4.2 the different financial stream processing engine configurations introduced in Chapters 6-12.

Section 13.1 below briefly summarizes the experimental setup describing how the experiments were performed and introducing the abbreviations used in Section 13.2 to refer to the different financial stream processing engine configurations used in the different experiments. Section 13.2 reports the obtained throughput and latency metrics and discusses them comparing them whenever it is possible with the throughput and latency expectations introduced in Chapters 6-12 when theoretically analyzing the cost of the different algorithms in terms of execution time.

13.1 Experimental setup

The experimental framework introduced in Chapter 5 has been used to quantitatively assess the quality of the different designs introduced in Chapters 6-12 in terms of achieved throughput and latency.

All the iterations of operators introduced in Chapters 6-12 have been tested both in an isolated manner and together with the other operators in the pipeline in order to asses both their individual performance and scalability and the way they perform and scale in conjunction with other operators in case the corresponding iteration involves more than one operator as it happens in Chapters 10-12.

In order to refer to the different stream processing engine configurations a mnemonic notation has been established with three characters to identify each experimental configuration as follows:

- **Prefix**: The first character is always the prefix 'E' from *experimental* configuration.
- **Iteration**: The second character is a single digit specifying which iteration of the stream processing engine the configuration corresponds to, 1 referring to the first iteration introduced in Chapter 6, 2 referring to the second iteration introduced in Chapter 7, and so on.

- **Suffix**: The last character specifies whether the configuration corresponds to the full configuration introduced in the corresponding chapter or to a subset or variation of it in order to asses the individual performance of a newly introduced operator or optimization of an operator in the corresponding iteration of the stream processing engine:
  - **Full configuration (F)**: The 'F' suffix indicates that the configuration corresponds to the full configuration illustrated in the last figure in the corresponding chapter.
  - **Partial configuration (P)**: The 'P' suffix indicates that the configuration corresponds to a subset of the full configuration illustrated in the last figure in the corresponding chapter in order to test only the operator which is described in detail in the operators description section in that chapter.
  - **Combined configuration (C)**: The 'C' suffix indicates for the last two iterations of the stream processing engine introduced in Chapters 11 and 12 that the two kinds of process thread have been merged together letting each process thread price an option contract right after the same thread has assigned the corresponding tuple an aggregated volatility value.

Below all the configurations whose throughput and latency median metrics are reported and analyzed in the next section are summarized according to the mnemonic notation introduced above.

- **E1F**: first iteration full configuration corresponding to the stream processing engine configuration introduced in Figure 6.7 in Section 6.4.2:
  - One options pricing thread, PT, executes the single-threaded binomial options pricing operator.
  - It communicates with the input thread, IT, and the output thread, OT, through concurrent queues.
- **E2F**: second iteration full configuration corresponding to the stream processing engine configuration introduced in Figure 7.5 in Section 7.4.2:
  - n OpenMP options pricing threads, the PT threads, are synchronized by the batching helper thread, BPT, to execute the single-threaded binomial options pricing operator in parallel for batches of tuples.
  - The BPT thread communicates with the input thread, IT, and the output thread, OT, through concurrent queues, and with the OpenMP PT threads through a shared array.
- **E3F**: third iteration full configuration corresponding to the stream processing engine configuration introduced in Figure 8.5 in Section 8.4.2:
  - n options pricing threads, the PT threads, execute the single-threaded binomial options pricing operator in parallel.
  - The PT threads compete to retrieve financial tuples from the queue to which the input thread IT serves them, and share a *ScaleGate* instance with the output thread, OT.
- **E4F**: fourth iteration full configuration corresponding to the stream processing engine configuration introduced in Figure 9.5 in Section 9.4.2:

  - n options pricing threads, the PT threads, execute the single-threaded binomial options pricing operator in parallel.
  - The PT threads share a *ScaleGate* instance with the input thread, IT and another *ScaleGate* instance with the output thread, OT. They are able to expedite the behavior of the second *ScaleGate* instance using the heartbeat mechanism.
- **E5P**: fifth iteration partial configuration corresponding to the stream processing engine configuration introduced in Figure 10.8 in Section 10.4.2 excluding the options pricing threads, PPT threads, and letting the volatility aggregation thread, VPT serve tuples to the output thread, OT, through a second queue instance:
  - One volatility aggregation thread, VPT, executes the single-threaded sliding-windows-based volatility aggregation operator.
  - It communicates with the input thread, IT, and the output thread, OT, through concurrent queues.
- **E5F**: fifth iteration full configuration corresponding to the stream processing engine configuration introduced in Figure 10.8 in Section 10.4.2:
  - One volatility aggregation thread, VPT, executes the single-threaded sliding-windows-based volatility aggregation operator.
  - n options pricing threads, the PPT threads, execute the single-threaded binomial options pricing operator in parallel using the volatility values added to the tuples by VPT.
  - The VPT thread communicates with the input thread, IT, through a concurrent queue. It shares with the PPT threads a *ScaleGate* instance and the PPT threads share with the output thread, OT, a second *ScaleGate* instance whose behavior they can expedite using the heartbeat mechanism.
- **E6P**: sixth iteration partial configuration corresponding to the stream processing engine configuration introduced in Figure 11.6 in Section 11.4.2 excluding the options pricing threads, PPT threads, and letting the volatility aggregation threads, VPT threads, serve tuples to the output thread, OT, through a second *ScaleGate* instance:
  - m parallel volatility aggregation threads, the VPT threads, execute the multi-threaded sliding-windows-based volatility aggregation operator.
  - The VPT threads share with the input thread, IT, a *ScaleGate* instance, and with the output thread, OT, a second *ScaleGate* instance whose behavior they can expedite using the heartbeat mechanism.
- **E6F**: sixth iteration full configuration corresponding to the stream processing engine configuration introduced in Figure 11.6 in Section 11.4.2:
  - m parallel volatility aggregation threads, the VPT threads, execute the multi-threaded sliding-windows-based volatility aggregation operator.
  - n options pricing threads, the PPT threads, execute the single-threaded binomial options pricing operator in parallel using the volatility values added to the tuples by the VPT threads.
  - The VPT threads share with the input thread, IT, a *ScaleGate* instance, and with the PPT threads, a second *ScaleGate* instance whose behavior

they can expedite using the heartbeat mechanism. The PPT threads share with the output thread, OT, a third *ScaleGate* instance whose behavior they can also expedite using the heartbeat mechanism.

- **E6C**: sixth iteration combined configuration corresponding to the stream processing engine configuration introduced in Figure 11.6 in Section 11.4.2 but combining the VPT and PPT threads together to form $VPPT$ threads which execute both the operators the VPT and PPT threads executed:
  - r parallel volatility aggregation and options pricing threads, the $VPPT$ threads, execute the multi-threaded sliding-windows-based volatility aggregation operator. For each tuple they assign a volatility value to, they also execute the single-threaded binomial options pricing operator in parallel.
  - The $VPPT$ threads share with the input thread, IT, a *ScaleGate* instance, and with the output thread, OT, a second *ScaleGate* instance whose behavior they can expedite using the heartbeat mechanism.
- **E7P**: seventh iteration partial configuration corresponding to the stream processing engine configuration introduced in Figure 12.5 in Section 12.4.2 excluding the options pricing threads, PPT threads, and letting the volatility aggregation and stream matching threads, WPT threads, serve tuples to the output thread, OT, through a second *ScaleGate* instance:
  - m parallel volatility aggregation and stream matching threads, the WPT threads, execute the multi-threaded sliding-windows-based volatility aggregation and stream matching operator.
  - The WPT threads share with the financial and options settings input threads, FIT and SIT, a *ScaleGate* instance, and with the output thread, OT, a second *ScaleGate* instance whose behavior they can expedite using the heartbeat mechanism.
- **E7F**: seventh iteration full configuration corresponding to the stream processing engine configuration introduced in Figure 12.5 in Section 12.4.2:
  - m parallel volatility aggregation and stream matching threads, the WPT threads, execute the multi-threaded sliding-windows-based volatility aggregation and stream matching operator.
  - n options pricing threads, the PPT threads, execute the single-threaded binomial options pricing operator in parallel using the trade price, volatility, option strike, option maturity and risk-less interest rate values added to the tuples by the WPT threads.
  - The WPT threads share with the financial and options settings input threads, FIT and SIT, a *ScaleGate* instance, and with the PPT threads, a second *ScaleGate* instance whose behavior they can expedite using the heartbeat mechanism. The PPT threads share with the output thread, OT, a third *ScaleGate* instance whose behavior they can also expedite using the heartbeat mechanism.
- **E7C**: seventh iteration combined configuration corresponding to the stream processing engine configuration introduced in Figure 12.5 in Section 12.4.2 but combining the WPT and PPT threads together to form $WPPT$ threads which execute both the operators the WPT and PPT threads executed:

  - r parallel volatility aggregation, stream matching and options pricing threads, the $WPPT$ threads, execute the multi-threaded sliding-windows-based volatility aggregation and stream matching operator. For each tuple they assign a volatility value to, they also execute the single-threaded binomial options pricing operator in parallel.
  - The $WPPT$ threads share with the input thread, IT, a *ScaleGate* instance, and with the output thread, OT, a second *ScaleGate* instance whose behavior they can expedite using the heartbeat mechanism.

All the aforementioned configurations in which one or more concurrent queues have been used to let different involved threads in the pipeline communicate, namely E1F, E2F, E3F, E5P, and E5F, have been tested with three different NOBLE [42, 43] queue implementations. Similarly as done above with the different stream processing engine configurations, each of the three concurrent queue implementations are henceforth referred to using the following three mnemonics, LB, LF_BB, and LF_DB introduced in [42]:

- **Standard spin-lock based queue (LB)**: the standard spin-lock based queue is assigned the mnemonic LB (lock-based) according to [42] as anticipated in Section 3.1.
- **Lock-free block structure bounded memory queue (LF_BB)**: the lock-free block structure bounded memory queue introduced in [16] is assigned the mnemonic LF_BB (lock-free block-bounded) according to [42] as anticipated in Section 3.1.
- **Lock-free dynamic structure bounded memory queue (LF_DB)**: the lock-free dynamic structure bounded memory queue introduced in [35] is assigned the mnemonic LF_DB (lock-free dynamic-bounded) according to [42] as anticipated in Section 3.1.

In addition to this, the aforementioned configurations in which the behavior of one or more *ScaleGate* instances can be expedited using the heartbeat mechanism, namely E4F, E5F, E6P, E6F, E6C, E7P, E7F, and E7C, have been tested both without using the heartbeat mechanism at all and using it having each non-output-generating tuple triggering the addition of a heartbeat tuple to the corresponding *ScaleGate* instance. Similarly as done above with the different stream processing engine configurations and queue implementations, each of these two modes of operation are henceforth referred to using the following mnemonics, NO HB, and HB:

- **Heartbeat mechanism not used (NO HB)**: the experiments in which the heartbeat mechanism can be used but is not used at all are flagged with the mnemonic NO HB.
- **Heartbeat mechanism fully used (HB)**: the experiments in which the heartbeat mechanism can be used and is used at its full capacity are flagged with the mnemonic HB. Given the fact that the achieved throughput metrics recorded, as it can be verified in the next section, with and without using the heartbeat mechanism do not differ more than a couple of tuples per second given the proportion between the cost of the operators and the cost of using the different concurrent data-structures, experiments partially utilizing the

heartbeat mechanisms with H values greater than 1 are considered not relevant and consequently, not reported in the next sections.

As anticipated when formally describing the different iterations of the financial stream processing engine, the latency of each individual tuple is individually measured minimizing the impact of this measure in the performance of the operators by letting the corresponding input and output threads add the tuples a process start timestamp, *sts*, before adding them to the concurrent data-structure the input threads in the different iterations share with the process threads, and letting the corresponding output thread add the tuples a process end timestamp, *ets*, after retrieving them from the concurrent data-structure the output threads in the different iterations share with the process threads. A careful analysis of the latency plots for all the individual tuples processed by the different configurations of the stream processing engine produced by the output analyzer program introduced in Section 5.3.3 for each individual experimental execution led to the conclusion that the latency median of all the individual latencies measured for each individual processed tuple was the value which better summarized the behavior of the operators in terms of latency under a non-saturation situation, which is the reason why the latency results reported in the following sections express the median latency among all the tuples in the input dataset processed in each individual execution of the stream processing engine. In order to measure the throughput achieved in each execution, the difference between the last processed tuple *ets* timestamp and the first processed tuple *sts* timestamp has been divided by the number of processed tuples in each execution leading to the achieved throughput metrics reported in the next section.

All the throughput and latency median metrics reported in the next section were obtained controlling the input rate at which the input threads in the different iterations added tuples to the concurrent data-structures shared with the different process threads in order to ensure a non-saturation situation. Otherwise, the latency median metrics would have reflected more the time spent by the different tuples waiting inside the concurrent data-structures for the different process threads to deal with the excessive workload of receiving more tuples than the ones they can process at a time than the time actually spent being processed by the operators and profiting from the concurrent data-structures to meet the linearizability requirements of the stream processing engine. To achieve this, all the experimental configurations whose throughput and latency median metrics are reported in the next sections were executed twice:

- In the first execution all the tuples from the relatively small subset of the main input dataset, introduced in Section 5.3.2 as *tiny* dataset, with 190442 tuples were served by the different input threads at full speed rate in order to assess the maximum throughput the configuration could achieve.
  As anticipated in Section 5.3.2, the financial input stream contained a selection of the 190442 tuples associated to the trades registered for the three most traded symbols in the NASDAQ market from 11:40 to 13:50 the 5th of August 2015, and the options settings stream, when used, contained also 190442 tuples with the same physical and logical timestamps as the former ones.

- In the second execution all the tuples in the *main* dataset introduced in 5.3.2 with 1705386 tuples were served at a constant rate at 90% the tuples per second that the same configuration could process according to the achieved throughput in the first execution described above in order to assess the latency the different configurations could achieve under a non-saturation situation.

  As anticipated in Section 5.3.2, the financial input dataset contained a selection of the 1705386 tuples associated to the trades registered for the ten most traded symbols in the NASDAQ market from 09:25 to 16:05 the 5th of August 2015, and the options settings stream, when used, contained also 1705386 tuples with the same physical and logical timestamps as the former ones.

  The results obtained in this second iteration for each configuration are the ones reported and analyzed in the next section.

13.2 Experimental Results and Analysis

In the following sections, the throughput and latency median results obtained when executing each of the aforementioned stream processing engine configurations as described in the previous section are reported and analyzed.

Section 13.2.1 reports the results obtained for the E1F, and E5P configurations in both of which the single-threaded version of the two main operators in which this Thesis focuses are individually tested. Section 13.2.2 reports and compares the results obtained for the E2F, E3F, and E4F configurations analyzing the performance of the different ways of executing in parallel the single-threaded binomial options pricing operator. Section 13.2.3 reports the results obtained for the E5F configuration showing how the single-threaded volatility aggregation operator introduced in Chapter 10 performs together with the options pricing operator as it is used since Chapter 9 letting multiple options pricing threads price option contracts in parallel.

Section 13.2.4 reports the results obtained for the E6P configuration analyzing the performance of the parallel volatility aggregator introduced in Chapter 11 independently of the behavior of the options pricing threads to be used in the full configuration, E6F, whose results are reported in Section 13.2.5 together with the results obtained using the alternative E6C configuration.

Finally, Section 13.2.6 reports the results obtained for the E7P configuration analyzing the performance of the parallel volatility aggregator and stream matcher introduced in Chapter 12 independently of the behavior of the options pricing threads to be used in the full configuration, E7F, whose results are reported in Section 13.2.7 together with the results obtained using the alternative E7C configuration.

13.2.1 Single-Threaded Binomial Options Pricing, and Single-Threaded Volatility Aggregation (E1F, and E5P)

Tables 13.1, and 13.2 below report the throughput and latency median achieved respectively in the 31228 machine (Intel Xeon), and Hasgreen (Intel Core i7) when executing the E1F, and E5P configurations using the three different concurrent queue implementations LB, LF_BB, and LF_DB according to the naming conventions introduced in Section 13.1.

Table 13.1: 31228 (Intel Xeon): E1F and E5P throughput and latency median

	Throughput (tuples/s)			Latency Median (μs)		
	LB	LF_BB	LF_DB	LB	LF_BB	LF_DB
E1F	485.00	489.00	490.00	1015	939	939
E5P	2146.87	2148.99	2135.00	373	289	289

Table 13.2: Hasgreen (Intel Core i7): E1F and E5P throughput and latency median

	Throughput (tuples/s)			Latency Median (μs)		
	LB	LF_BB	LF_DB	LB	LF_BB	LF_DB
E1F	580.19	586.00	586.00	866	791	792
E5P	1212.00	1264.00	1262.00	493	478	477

As it can be seen, in both machines, the lock-based queue, LB, tends to perform slightly worse in terms of throughput and latency than the lock-free alternatives, LF_BB, and LF_DB, which could be expected in advance given the lock-free nature of the latter ones in contrast with the lock-based nature of the former.

As it can also be appreciated, the single-threaded volatility aggregation operator (E5P) records better results in terms of throughput and latency than the single-threaded binomial options pricing operator (E1F) in both machines. This means that given the configuration parameters chosen for the execution of the experiments here reported, namely $N = 2048$ steps in the underlying binomial model in E1F, as anticipated in Section 6.4.1 , and $WS = 1h$, $WA = 50ms$ for the underlying sliding-windows model in E5P, as anticipated in Section 10.4.2, make the single-threaded binomial options pricing operator more expensive in terms of execution time than the single-threaded sliding-window-based volatility aggregation operator.

It can also be seen that the aforementioned difference in performance is considerably greater in the Intel Xeon machine than in the Intel Core i7 machine. As it can be seen, while the E1F configuration reports in the Intel Core i7 machine better results in terms of throughput and latency than in the Intel Xeon machine, which could be expected given the faster clock rate of the former and the fact that in none of the machines all the cores need to be used by any of the configurations here analyzed, the E5P configuration behaves completely the opposite way reporting in the Intel Xeon machine considerably better results in terms of throughput and

latency than in the Intel Core i7 machine. A possible explanation for this is the fact that, as analyzed in Section 10.4.1, the sliding-window-based volatility aggregation operator maintains in memory a considerably big data-structure, the underlying sliding-windows model, which may give the Intel Xeon machine a competitive advantage in comparison to the Intel Core i7 machine given the bigger cache the former machine makes use of, as anticipated in Section 5.4.2.

The last observation above helps understanding the different nature of the two machines used to produce the experimental results discussed in this chapter, which go beyond the difference in the number of threads or the clock rates.

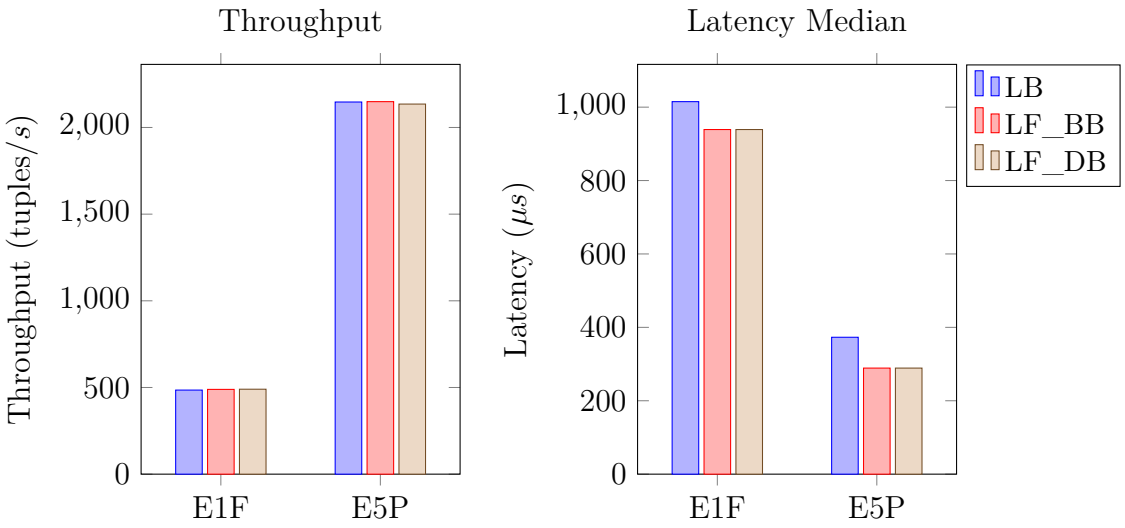


Figure 13.1: 31228 (Intel Xeon): E1F and E5P throughput and latency median

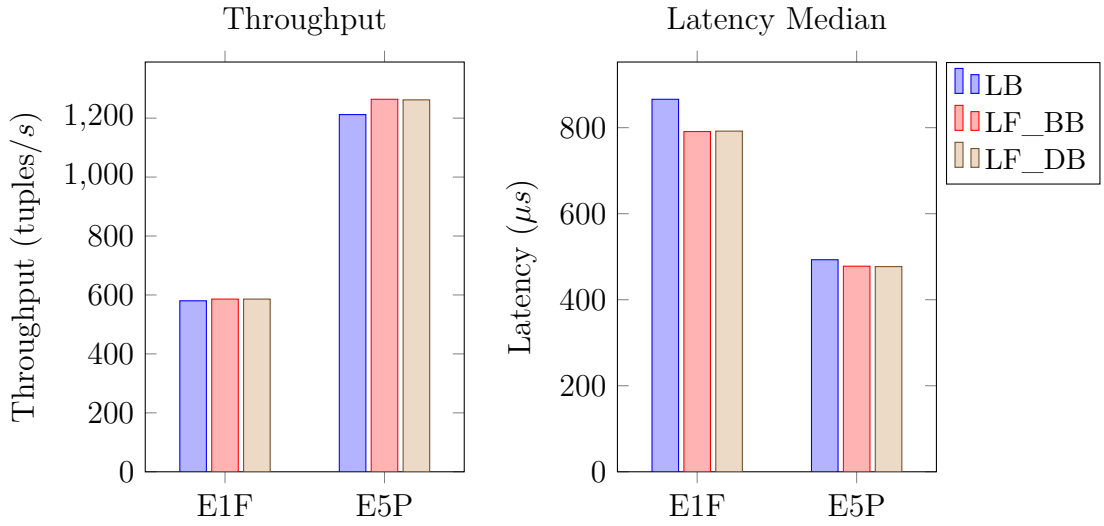


Figure 13.2: Hasgreen (Intel Core i7): E1F and E5P throughput and latency median

To help better appreciating the different behavior of the operators in both machines according to the results reported and analyzed above, Figures 13.1, and 13.2 above plot all the results reported in Tables 13.1, and 13.2.

13.2.2 Multi-Threaded Binomial Options Pricing Operators (E2F, E3F, and E4F)

Tables 13.3, 13.4, 13.5, and 13.6 below report the throughput and latency median achieved respectively in the 31228 machine (Intel Xeon), and Hasgreen (Intel Core i7) when executing the E2F, E3F, and E4F configurations with the number of options pricing threads, *PT* threads, reported in the left-most column, using in E2F and E3F the three different concurrent queue implementations, LB, LF_BB, and LF_DB, and in E4F both using and not the heartbeat mechanism, according to the naming conventions introduced in Section 13.1.

Table 13.3: 31228 (Intel Xeon): E2F, E3F, and E4F throughput

	Throughput (tuples/s)							
	E2F			E3F			E4F	
	LB	LF_BB	LF_DB	LB	LF_BB	LF_DB	NO HB	HB
1	457.00	490.00	491.00	488.00	492.00	491.00	488.00	490.00
4	1670.00	1727.00	1735.00	1729.97	1737.00	1739.00	1738.00	1728.00
8	3244.35	3345.99	3342.99	3366.35	3370.00	3371.00	3369.00	3362.00
12	4815.42	5003.98	4995.97	5050.14	5055.00	5056.00	5058.00	5033.00
16	3627.99	3599.00	3622.99	5758.63	5772.95	5773.95	3718.00	3710.00
20	4525.86	4526.98	4510.96	5917.58	5918.98	5924.95	4531.98	4503.99
24	5414.93	5351.95	5410.97	6057.85	6063.98	6061.96	5437.00	5395.00
27	5918.43	6099.34	6061.94	6165.78	6170.70	6171.94	6114.96	6076.49

Table 13.4: 31228 (Intel Xeon): E2F, E3F, and E4F latency median

	Latency Median (μs)							
	E2F			E3F			E4F	
	LB	LF_BB	LF_DB	LB	LF_BB	LF_DB	NO HB	HB
1	939	938	938	959	938	939	939	940
4	2262	2110	2140	3957	3944	3940	2792	1646
8	2336	2297	2299	4698	4691	4963	3181	1402
12	2352	2303	2305	5258	5095	5292	3278	1305
16	4263	4278	4266	7032	7175	7156	5256	1499
20	4265	4259	4272	9957	9596	9316	6232	2268
24	4281	4300	4279	11140	11152	11111	6275	2241
27	4410	4235	4253	12245	12193	12472	6297	2224

Table 13.5: Hasgreen (Intel Core i7): E2F, E3F, and E4F throughput

	Throughput (tuples/s)							
	E2F			E3F			E4F	
	LB	LF_BB	LF_DB	LB	LF_BB	LF_DB	NO HB	HB
1	575.47	587.00	586.00	585.20	586.00	586.00	586.00	586.00
2	1113.88	1141.00	1140.00	1141.80	1142.00	1142.00	1143.00	1142.00
3	1645.00	1664.00	1662.00	1567.68	1669.00	1669.00	1669.00	1668.00
4	1405.00	1337.00	1399.00	1652.40	1968.00	1968.00	1225.00	1223.00
5	1508.00	1508.00	1507.00	1749.88	2016.00	2017.00	1530.00	1529.00
6	1803.99	1807.00	1804.00	1714.73	2065.00	2065.00	1814.91	1813.00
7	2089.99	2104.99	2101.00	2112.91	2113.00	2112.98	2093.00	2090.00

Table 13.6: Hasgreen (Intel Core i7): E2F, E3F, and E4F latency median

	Latency Median (μs)							
	E2F			E3F			E4F	
	LB	LF_BB	LF_DB	LB	LF_BB	LF_DB	NO HB	HB
1	900	790	791	1123	792	792	792	792
2	1708	1590	1457	1802	2562	1760	1688	1689
3	1448	1438	1440	2634	2630	2632	2033	1435
4	2762	2828	2766	3615	3375	3376	3288	1658
5	2921	2898	2901	5442	4380	4530	4155	2197
6	3238	3225	3233	7820	5698	5757	4302	2103
7	3050	3022	3027	7190	6683	6473	4418	2034

As it can be seen, similarly as in the results reported in the previous section, in both machines when executing the E2F and E3F configurations, the lock-based queue, LB, tends to perform slightly worse in terms of throughput and latency than the lock-free alternatives, LF_BB, and LF_DB, which could be expected in advance given the lock-free nature of the latter ones in contrast with the lock-based nature of the former. In the particular case of E3F executed in Hasgreen (Intel Core i7), the the difference of performance between the lock-based queue, LB, and the lock-free alternatives, LF_BB, and LF_DB, is especially noticeable when using from 4 to 6 *PT* threads.

As it can also be seen when briefly analyzing the throughput metrics reported when executing the E4F configuration in both machines, the heartbeat mechanism, as expected in Section 9.4.1 when estimating the expected throughput, does not represent a significant overhead in terms of throughput due to the small cost of adding tuples to, and getting tuples from the *ScaleGate* instances in comparison to the cost of executing the options pricing procedure.

To help better appreciating the way throughput and latency scale as more *PT* threads are used, Figures 13.3, and 13.4 below plot all the results reported in Tables

13.3, 13.4, 13.5, and 13.6.

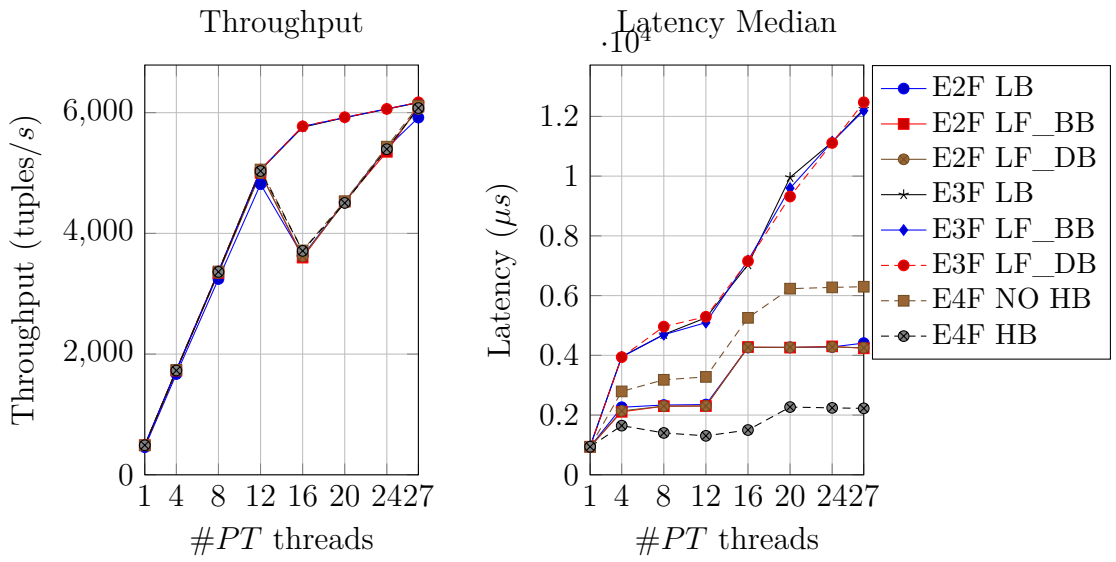


Figure 13.3: 31228 (Intel Xeon): E2F, E3F, and E4F throughput and latency median

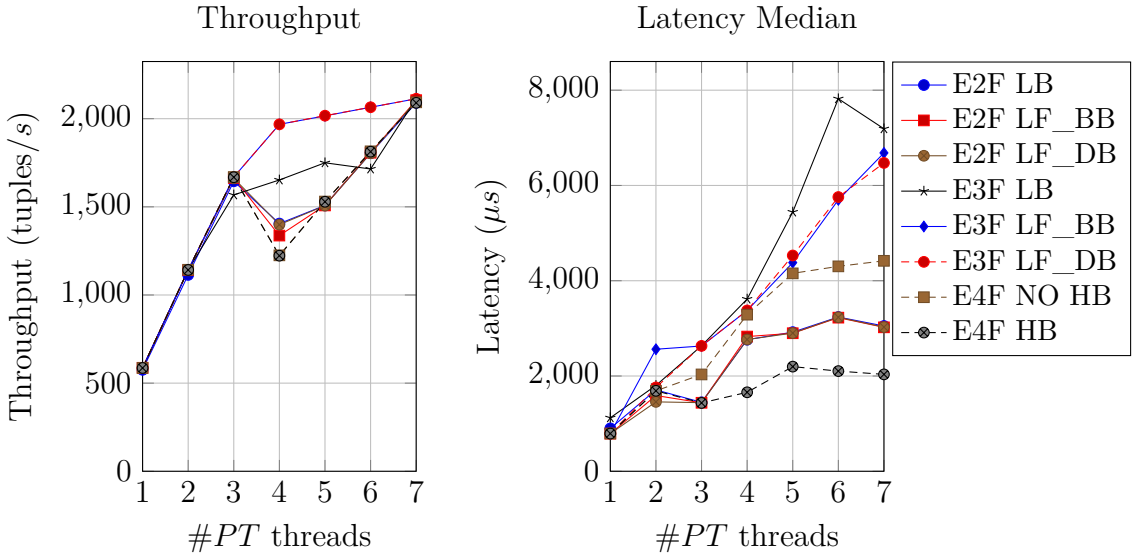


Figure 13.4: Hasgreen (Intel Core i7): E2F, E3F, and E4F throughput and latency median

As it can be appreciated in all plots, there is a clear change of tendencies in terms of throughput and latency when more than respectively 12 and 3 *PT* threads are used in 31228 (Intel Xeon) and Hasgreen (Intel Core i7). The reason for this is hyper-threading. On the one hand, when needing no more than the total number of physical processors available, equivalent in both machines to half the number of virtual processors available, each of the threads have been assigned to a separate

physical processor minimizing contention and scheduling overhead. On the other hand, when needing more than the total number total number of physical processors available, some of the threads have had to share processor with other threads increasing contention, scheduling overhead in the affected physical processors.

Focusing on the reported throughput, as it can be seen, the expected linear scaling tendencies for the three configurations have been met before having to start assigning more than one thread to some or all of the processors. Reached that point, both the E2F, and E4F configurations show a sudden drop in throughput in both machines before getting back to a linear increasing tendency with less slope while in the E3F configuration, the slope changes even more but no drop is appreciated.

One possible explanation for this two different responses to hyper-threading is the load balancing flexibility of the different configurations. Given the behavior of the operators and data-structures in the second and fourth iterations of the stream processing engine, each of the PT threads has to process exactly one tuple out of every n consecutive tuples, n being the number of PT threads, whereas in the third iteration, the usage of the concurrent queue with multiple readers allows the PT threads to compete to retrieve tuples to be processed, resulting in the threads assigned alone to their corresponding processor being able to process more tuples than the threads assigned to a processor together with other thread. This hypothesis is reinforced when observing the throughput reported when using all the available logical cores, which is the same for the three configurations, as each of the processors is assigned two threads getting back to a situation in which all the threads are assigned to equally busy processors.

Focusing on the reported latency, the E4F configuration making use of the heartbeat mechanism is clearly the best configuration being able to produce option prices for all tuples in both machines using any number of threads in less than twice the time it takes to produce options prices using the only one options pricing thread. It is also worth highlighting how if the heartbeat mechanism is not used, the E2F configuration becomes better than the E4F configuration in terms of latency. This last observation helps appreciating the overhead of using *ScaleGate* without expediting its behavior with heartbeat tuples as reasoned in detail when introducing the different iterations of the stream processing engine. Finally, the E3F configuration is the one which behaves the worst in terms of latency, especially when the lock-based queue, LB, is used in Hasgreen (Intel Core i7). With this configuration, the effect of the *ScaleGate* semantics on latency is inflated due to the indeterminism added by the competition of the PT threads to process tuples.

All the results reported and analyzed in this section motivated the election of the fourth iteration of the stream processing engine as the one on top of which the fifth, sixth, and seventh iterations are built.

13.2.3 Single-Threaded Volatility Aggregation, and Multi-Threaded Binomial Options Pricing (E5F)

Tables 13.7, 13.8, 13.9, and 13.10 below report the throughput and latency median achieved respectively in the 31228 machine (Intel Xeon), and Hasgreen (Intel Core i7) when executing the E5F configuration with one volatility aggregation thread, VPT, and the number of options pricing threads, PPT threads, reported in the left-most column, using the three different concurrent queue implementations, LB, LF_BB, and LF_DB, and both using and not the heartbeat mechanism, according to the naming conventions introduced in Section 13.1.

Table 13.7: 31228 (Intel Xeon): E5F throughput

	Throughput (tuples/s)					
	LB		LF_DB		LF_BB	
	NO HB	HB	NO HB	HB	NO HB	HB
1	480.00	479.00	480.00	481.00	478.00	480.00
4	1677.92	1655.75	1674.00	1677.00	1683.00	1674.00
8	1951.25	1962.98	1962.00	1964.00	1954.00	1954.00
12	1941.57	1921.15	1962.00	1963.00	1964.00	1961.00
15	1332.00	1332.00	1334.00	1334.00	1333.00	1332.00
19	1334.00	1333.00	1335.00	1334.00	1335.00	1333.00
23	1335.00	1333.00	1336.99	1334.97	1336.00	1334.00
26	1334.41	1333.89	1336.97	1332.94	1334.95	1333.95

Table 13.8: 31228 (Intel Xeon): E5F latency median

	Latency Median (μs)					
	LB		LF_DB		LF_BB	
	NO HB	HB	NO HB	HB	NO HB	HB
1	1308	1304	1300	1302	1300	1302
4	3247	2031	3213	2018	3204	2020
8	5106	1948	4990	1932	5004	1935
12	7185	2003	7029	1934	7024	1935
15	12118	2359	12095	2353	12109	2355
19	16002	3269	15992	3268	15992	3269
23	18994	3277	18969	3275	18981	3275
26	21249	3280	21215	3280	21244	3280

Table 13.9: Hasgreen (Intel Core i7): E5F throughput

	Throughput (tuples/s)					
	LB		LF_DB		LF_BB	
	NO HB	HB	NO HB	HB	NO HB	HB
1	575.00	574.00	574.00	575.00	575.00	574.00
2	1106.00	1104.00	1103.00	1103.00	1106.00	1104.00
3	1004.00	1004.00	1005.00	1004.00	1005.00	1004.00
4	1050.00	1049.00	1049.00	1051.00	1046.00	1050.00
5	1050.00	1050.00	1050.00	1051.00	1051.00	1050.00
6	1046.00	1048.98	1051.00	1051.99	1049.00	1050.00

Table 13.10: Hasgreen (Intel Core i7): E5F latency median

	Latency Median (μs)					
	LB		LF_DB		LF_BB	
	NO HB	HB	NO HB	HB	NO HB	HB
1	1316	1317	1315	1315	1318	1317
2	2245	2246	2265	2247	2244	2246
3	3576	2577	3574	2578	3569	2579
4	4786	2882	4825	2917	4809	2894
5	5934	3079	5934	3078	5930	3080
6	6925	3102	6906	3098	6910	3101

As it can be seen, similarly as in the results reported in the previous sections, in both machines when executing the E5F configuration, the lock-based queue, LB, tends to perform slightly worse in terms of throughput and latency than the lock-free alternatives, LF_BB, and LF_DB.

As it can also be seen when briefly analyzing the throughput metrics reported when executing the E5F configuration in both machines, the heartbeat mechanism, as observed in the results reported in the previous section for the E4F configuration, does not represent a significant overhead in terms of throughput.

To help better appreciating the way throughput and latency scale as more PPT threads are used, Figures 13.5, and 13.6 below plot all the results reported in Tables 13.7, 13.8, 13.9, and 13.10.

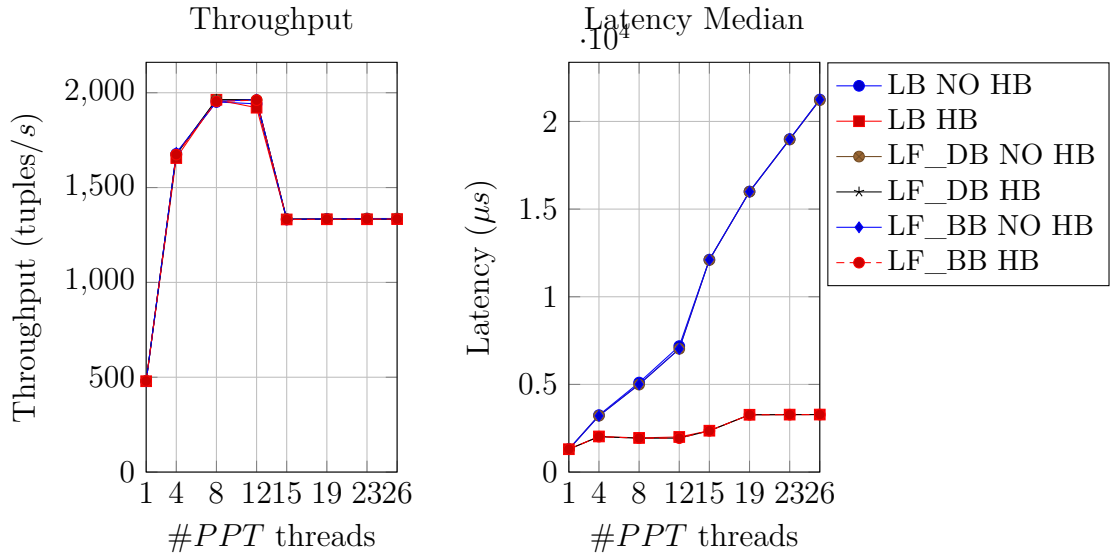


Figure 13.5: 31228 (Intel Xeon): E5F throughput and latency median

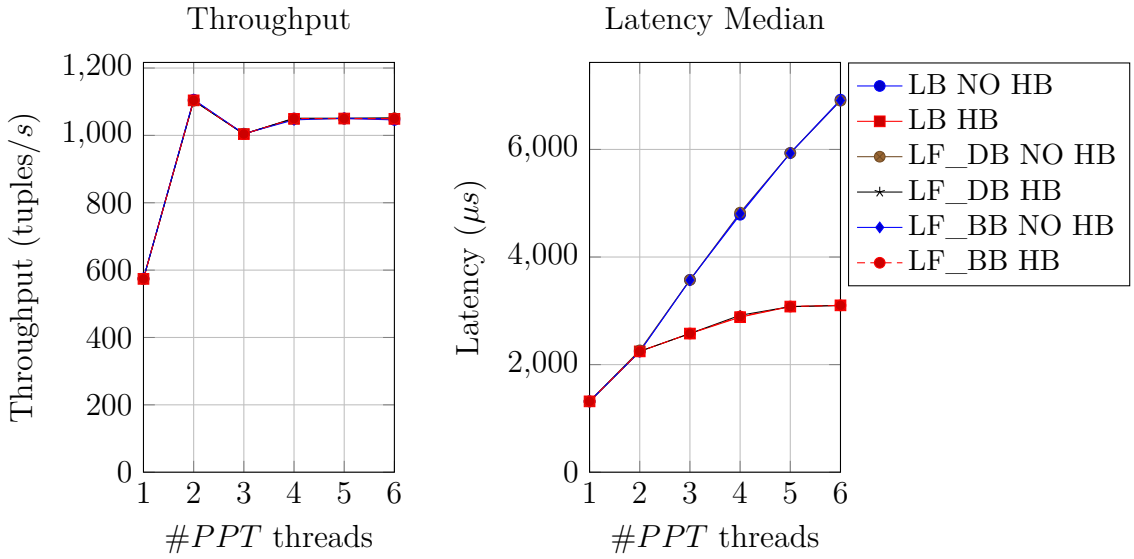


Figure 13.6: Hasgreen (Intel Core i7): E5F throughput and latency median

Focusing on the reported throughput, the reported results help very well understanding the impact of performance bottlenecks when different operators precede each other as it happens in the E5F configuration. When using only one PT thread, the reported throughput is the same as the one reported in Section 13.2.1 for the E1F configuration. This means that the options pricing operator, working at its full capacity, is the one upper bounding the achieved throughput, in other words, acting as a bottleneck. The volatility aggregation thread, even though it is able to process more tuples per second as shown in Section 13.2.1, cannot be used at its full capacity when having only one PPT thread without negatively affecting the reported latency because the output stream of tuples served by the VPT threads

to the data-structure it shares with the PPT thread, cannot be assimilated by the PPT thread, forcing the tuples to spend more time the more tuples served by VPT waiting in the data-structure from which the PPT thread retrieves tuples more slowly than they are added to it.

When using more PPT threads, the options pricing stage of the stream processing engine becomes able to process more tuples per second than the volatility aggregation stage with only one VPT thread as it can be anticipated when analyzing the results reported in Section 13.2.2. In this situation, and before reaching a hyper-threading situation, the reported throughput is the same as the one reported in Section 13.2.1 for the E5P configuration. This means that it is now the volatility aggregation operator, working at its full capacity, is the one upper bounding the achieved throughput, in other words, acting as a bottleneck. The options pricing threads, even though they are able to process more tuples per second as shown in Section 13.2.2, cannot be used at their full capacity when having only one VPT thread without negatively affecting the reported latency because the input stream of tuples that the input thread would have to serve to the concurrent data-structure it shares with VPT would not be able to be assimilated by the VPT thread, forcing the tuples to spend more time the more tuples served by the input thread waiting in the data-structure from which VPT retrieves tuples more slowly than they are added to it. When hyper-threading appears, the VPT thread performs slightly worse having to share the physical processor with a PPT thread leading to the drop in throughput which can be observed especially in the results reported for the 31228 machine (Intel Xeon).

Focusing on the reported latency, similarly as it was observed when analyzing the latency results reported for the E4F configuration in the previous section, the heartbeat mechanism helps keeping the latency considerably lower than the absence of it. When hyper-threading appears, a step can be appreciated especially in the 31228 machine (Intel Xeon) latency plot which can be understood considering that both the volatility aggregation operator and the options pricing operator need more time to produce a result for each single tuple when sharing a physical core with other thread than when not having to do so.

13.2.4 Multi-Threaded Volatility Aggregation (E6P)

Tables 13.11, and 13.12 below report the throughput and latency median achieved respectively in the 31228 machine (Intel Xeon), and Hasgreen (Intel Core i7) when executing the E6P configuration with the number of volatility aggregation threads, VPT threads, reported in the left-most column, using and not the heartbeat mechanism, according to the naming conventions introduced in Section 13.1.

Table 13.11: 31228 (Intel Xeon): E6P throughput and latency median

	Throughput (tuples/s)		Latency Median (μs)	
	NO HB	HB	NO HB	HB
1	2114.00	2102.00	294	294
4	4816.00	4793.01	6195	338
8	9335.86	9207.86	8688	177
12	11639.78	11634.76	12105	143
16	9876.86	9717.76	20921	143
20	9229.84	9145.88	31063	181
24	11075.74	10938.59	31604	154
27	15316.36	14987.41	26396	116

Table 13.12: Hasgreen (Intel Core i7): E6P throughput and latency median

	Throughput (tuples/s)		Latency Median (μs)	
	NO HB	HB	NO HB	HB
1	1250.00	1244.00	482	482
2	2036.00	2035.00	4434	786
3	2752.00	2754.00	7011	581
4	2803.00	2777.00	10883	535
5	2690.00	2672.00	15324	558
6	3119.00	3089.00	17381	525
7	3398.99	3372.00	19903	481

As it can be seen when briefly analyzing the throughput metrics reported when executing the E6P configuration in both machines, the heartbeat mechanism, as observed in the results reported in the previous sections, does not represent a significant overhead in terms of throughput.

To help better appreciating the way throughput and latency scale as more VPT threads are used, Figures 13.7, and 13.8 below plot all the results reported in Tables 13.11, and 13.12.

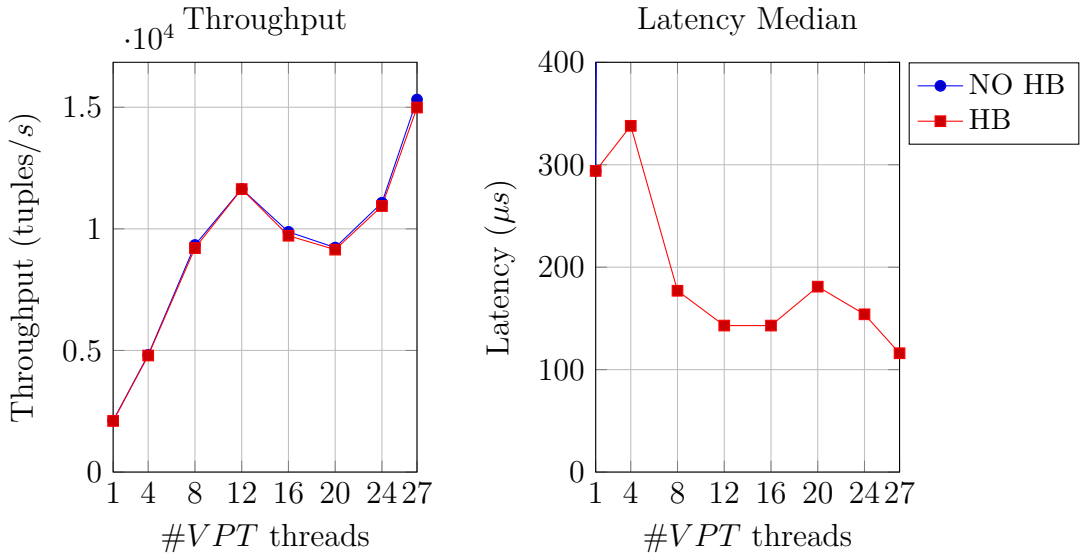


Figure 13.7: 31228 (Intel Xeon): E6P throughput and latency median

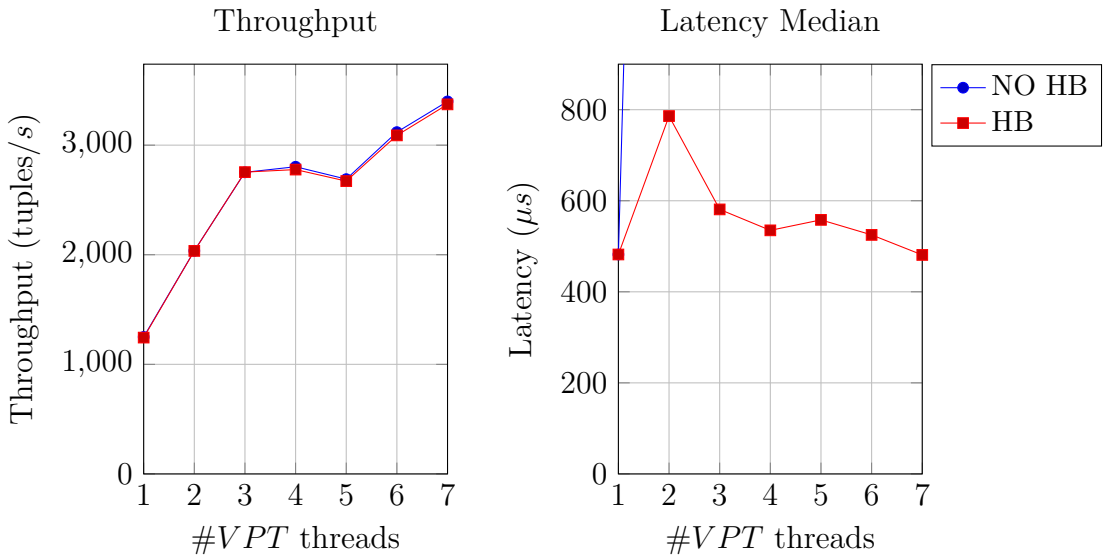


Figure 13.8: Hasgreen (Intel Core i7): E6P throughput and latency median

Focusing on the reported throughput, as it can be seen, the expected linear scaling tendency has been met before having to start assigning more than one thread to some or all of the processors. Reached that point, as it happened with the E2F, and E4F configurations as reported in Section 13.2.2, a drop in throughput can be appreciated before getting back to an increasing tendency.

Focusing on the reported latency, as anticipated in Section 11.4.1 when estimating the expected latency, the heartbeat mechanism plays an even more relevant role than when executing in parallel the options pricing operator given the longer cycles followed by the volatility aggregation threads adding tuples to the *ScaleGate* data-structure. The difference is so big that if the latency reported for both modes

of operations were plotted in the figures above together, the results reported for the heartbeat mechanism would be perceived as a red line attached to the x axis no matter the number of VPT threads used. For this reason, the y axis in the latency plots has been adapted to the order of magnitude of the latency results reported when using the heartbeat mechanism making it possible to appreciate the evolution with the number of VPT threads of the latency reported using the heartbeat mechanism.

As it can be seen in the latency results reported for both machines using the hear beat mechanism, even though the reported latency increases when moving forward from one to more than one VPT threads, as soon as the number of threads starts growing more, a decreasing tendency inversely proportional to the number of threads, as expected in Section 11.4.1, can be clearly appreciated. This tendency is slightly broken when different threads start having to share physical processor with another thread but even with this, the lowest reported latency results are the ones recorded when using all the available virtual processors.

13.2.5 Multi-Threaded Volatility Aggregation and Multi-Threaded Binomial Options Pricing (E6F, and E6C)

Tables 13.13, and 13.14 below report the throughput and latency median achieved respectively in the 31228 machine (Intel Xeon), and Hasgreen (Intel Core i7) when executing the E6F configuration using all the available logical processors with the number of volatility aggregation threads, VPT threads, reported in the left-most column, and the number of options pricing threads, PPT threads, necessary to utilize all the available logical processors, namely the left-most column read bottom-up, using and not the heartbeat mechanism, according to the naming conventions introduced in Section 13.1.

Table 13.13: 31228 (Intel Xeon): E6F throughput and latency median

	Throughput (tuples/s)		Latency Median (μs)	
	NO HB	HB	NO HB	HB
1	1275.94	1277.94	22134	3337
4	2490.87	2481.96	25166	3124
8	4291.92	4264.68	31906	2651
12	3450.89	3386.88	59829	2749
15	3885.41	3829.55	79209	2668
19	2947.21	2910.80	148444	2807
23	1613.48	1598.91	367883	3376
26	473.58	471.66	4831458	6311926

Table 13.14: Hasgreen (Intel Core i7): E6F throughput and latency median

	Throughput (tuples/s)		Latency Median (μs)	
	NO HB	HB	NO HB	HB
1	1036.00	1037.00	6984	3127
2	1484.00	1480.00	16894	3259
3	1199.00	1195.00	31355	3512
4	1319.98	1319.00	53526	3274
5	983.98	980.00	103553	3798
6	542.98	542.00	279910	3610

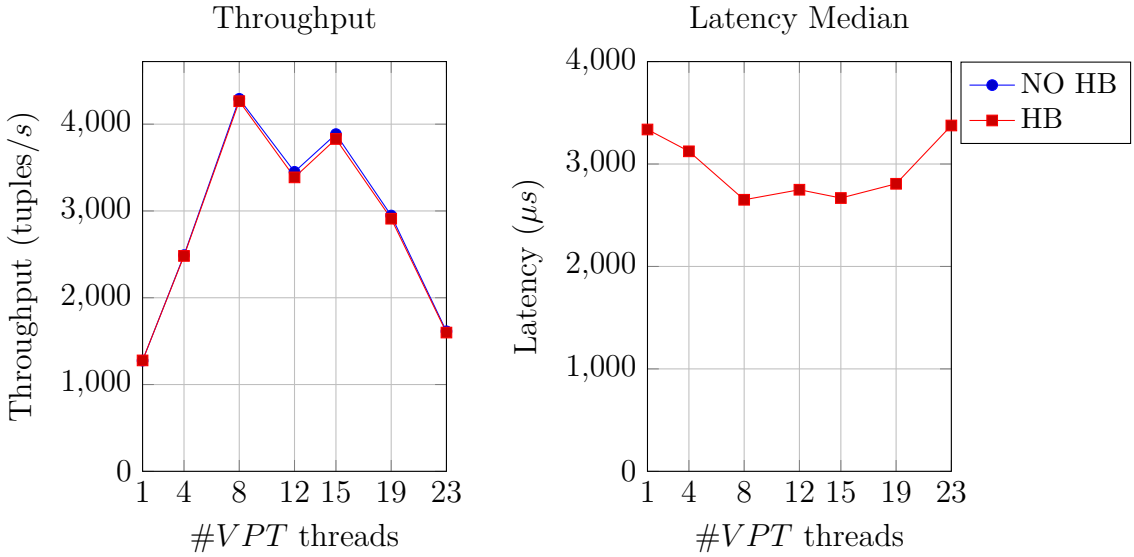


Figure 13.9: 31228 (Intel Xeon): E6F throughput and latency median

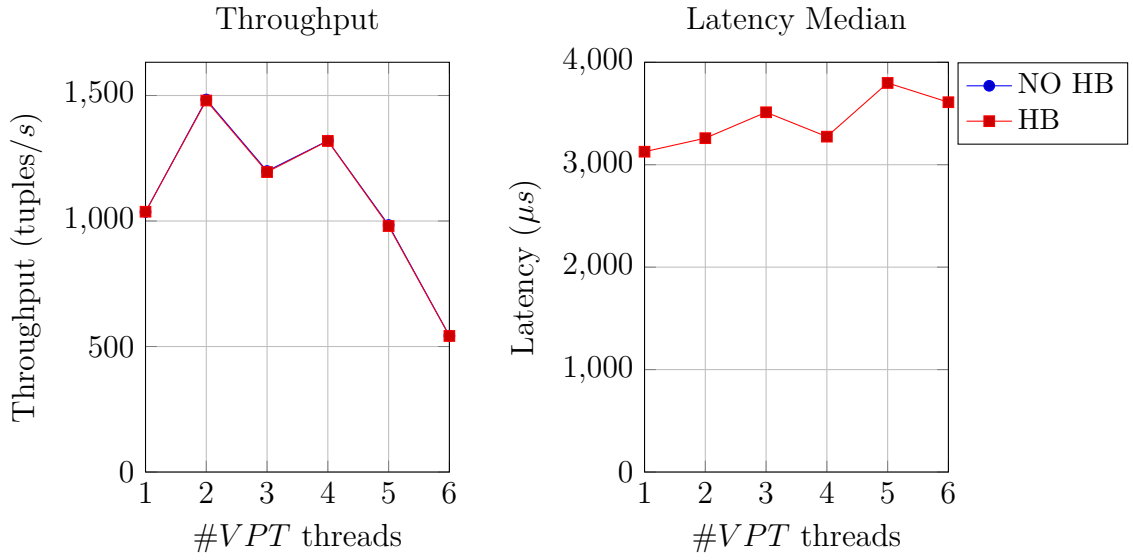


Figure 13.10: Hasgreen (Intel Core i7): E6F throughput and latency median

To help better appreciating the way throughput and latency scale as more VPT threads and less PPT threads are used, Figures 13.9, and 13.10 above plot all the results reported in Tables 13.13, and 13.14.

Focusing on the throughput results, in both edges of the plots for both machines it can be appreciated how respectively the single volatility aggregation thread and the single options pricing thread act as bottlenecks. Getting more centric, the best tradeoffs when distributing the available logical processors among VPT and PPT threads are found closer to the left side having slightly more options pricing threads than volatility aggregation threads. This seems reasonable given the fact that, as seen in Section 13.2.1, the options pricing workload for each individual tuple is higher than the volatility aggregation workload.

Focusing on the latency results, similarly as in the previous section, the results produced using the heartbeat mechanism are orders of magnitude better leaving the results produced without using the heartbeat mechanism out of the latency plots. When using the heartbeat mechanism, the latency results, especially in the 31228 machine (Intel Xeon) are better the better the throughput is. This can be seen as a positive feature of the E6F configuration avoiding the choice between better throughput or better latency which would have had to be done if this correlation did not hold.

Tables 13.15, and 13.16 below report the throughput and latency median achieved respectively in the 31228 machine (Intel Xeon), and Hasgreen (Intel Core i7) when executing the E6C configuration with the number of volatility aggregation and options pricing threads, $VPPT$ threads, reported in the left-most column, using and not the heartbeat mechanism, according to the naming conventions introduced in Section 13.1.

Table 13.15: 31228 (Intel Xeon): E6C throughput and latency median

	Throughput (tuples/s)		Latency Median (μs)	
	NO HB	HB	NO HB	HB
1	394.00	395.00	1239	1239
4	1188.00	1180.00	29823	7873
8	2267.95	2260.95	52321	31541
12	3263.95	3253.94	70308	51928
16	2662.95	2654.96	108239	67617
20	3028.72	3005.72	156331	124628
24	3562.68	3537.68	178640	156961
27	4164.82	4122.23	202078	186341

Table 13.16: Hasgreen (Intel Core i7): E6C throughput and latency median

	Throughput (tuples/s)		Latency Median (μs)	
	NO HB	HB	NO HB	HB
1	396.00	396.00	1298	1309
2	711.00	712.00	12862	1706
3	1050.00	1044.00	19257	2339
4	906.00	901.00	36371	2150
5	1079.00	1074.00	47287	13536
6	1212.72	1209.00	57698	24981
7	1407.00	1401.00	67732	38441

To help better appreciating the way throughput and latency scale as more $VPPT$ threads are used, Figures 13.11, and 13.12 below plot all the results reported in Tables 13.15, and 13.16.

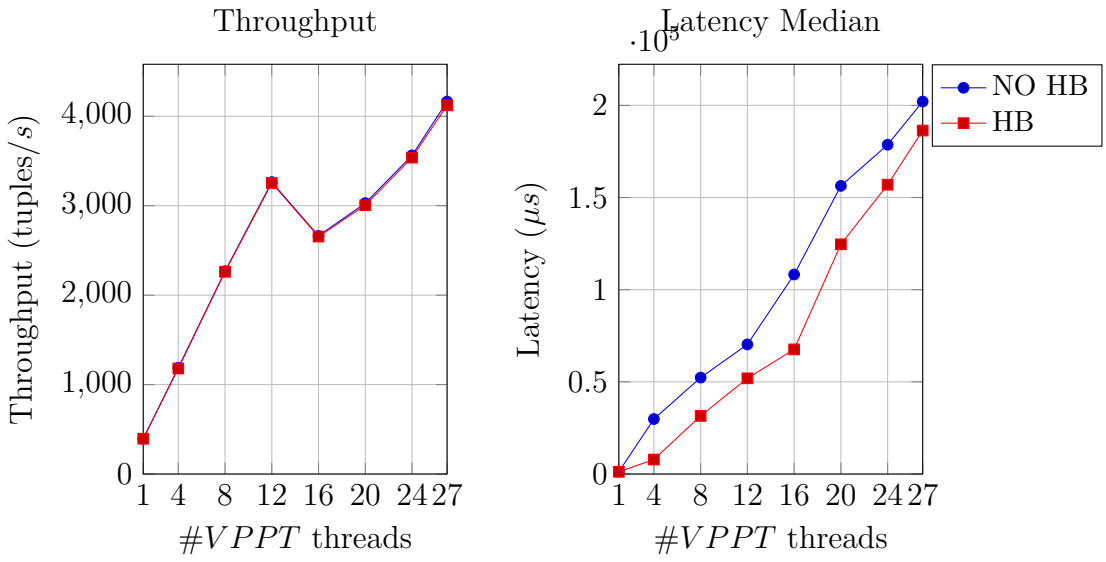


Figure 13.11: 31228 (Intel Xeon): E6C throughput and latency median

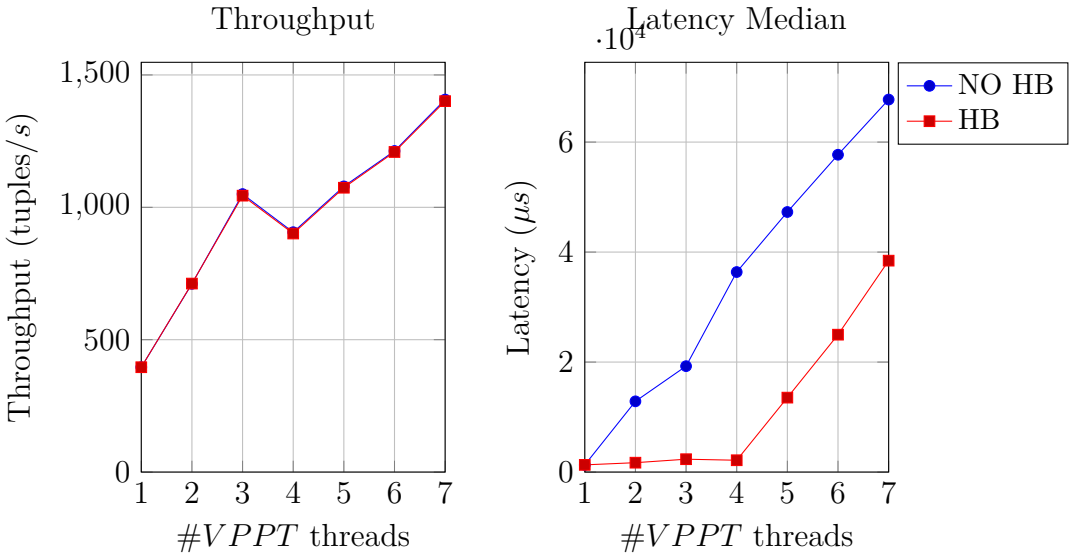


Figure 13.12: Hasgreen (Intel Core i7): E6C throughput and latency median

Focusing on the throughput results, a similar tendency as in previous occasions can be observed, having a linear increasing tendency before hyper-threading, a drop, and a second linear increasing tendency which achieves when all the available logical processors are used in both machines a similar throughput as the one achieved with the E6F configuration optimally distributing the available logical cores among VPT and PPT cores. This means that letting the threads in the sixth iteration of the stream processing engine execute both the volatility aggregation logic and the options pricing logic in order to use two *ScaleGate* instances instead of three does not result in appreciable performance gains in terms of throughput.

Focusing on the latency results, even when using the heartbeat mechanism the reported latency linearly grows becoming orders of magnitude higher than in the E6F configuration even when using the same number of virtual processors in the system or even less virtual processors. The reason for this is that given the different cycles the parallel volatility aggregation operator and the options pricing operator followed when outputting tuples, the heartbeat mechanism cannot be used to expedite the behavior of the *ScaleGate* instance as effectively as in the previous occasions when executing both workloads by the same thread. This together with the discussed throughput results represent a very strong argument towards the usage of *ScaleGate* as an articulation point in stream processing engines. The E6F configuration preforms clearly better than the E6C configuration thanks to the intermediate *ScaleGate* instance.

13.2.6 Multi-Threaded Volatility Aggregation and Stream Matching (E7P)

Tables 13.17, and 13.18 below report the throughput and latency median achieved respectively in the 31228 machine (Intel Xeon), and Hasgreen (Intel Core i7) when ex-

ecuting the E7P configuration with the number of volatility aggregation and stream matching threads, WPT threads, reported in the left-most column, using and not the heartbeat mechanism, according to the naming conventions introduced in Section 13.1.

Table 13.17: 31228 (Intel Xeon): E7P throughput and latency median

	Throughput (tuples/s)		Latency Median (μs)	
	NO HB	HB	NO HB	HB
1	748.94	747.00	1336	1337
4	4629.00	4579.01	6521	354
8	8955.94	8812.91	9137	189
12	10936.82	10666.75	12908	158
16	9426.80	9259.83	22285	308
20	9221.84	9096.87	31137	182
24	11013.69	10888.80	31782	156
27	14024.34	13751.34	35252	178

Table 13.18: Hasgreen (Intel Core i7): E7P throughput and latency median

	Throughput (tuples/s)		Latency Median (μs)	
	NO HB	HB	NO HB	HB
1	827.00	825.00	738	1211
2	2018.00	2004.00	4479	798
3	2780.00	2733.00	6909	585
4	2737.99	2707.00	11127	598
5	2671.00	2650.00	15461	611
6	3099.90	3074.00	17505	527
7	3392.88	3356.99	19927	487

As it can be seen when briefly analyzing the throughput metrics reported when executing the E7P configuration in both machines, the heartbeat mechanism, as observed in the results reported in the previous sections, does not represent a significant overhead in terms of throughput.

To help better appreciating the way throughput and latency scale as more WPT threads are used, Figures 13.13, and 13.14 below plot all the results reported in Tables 13.17, and 13.18.

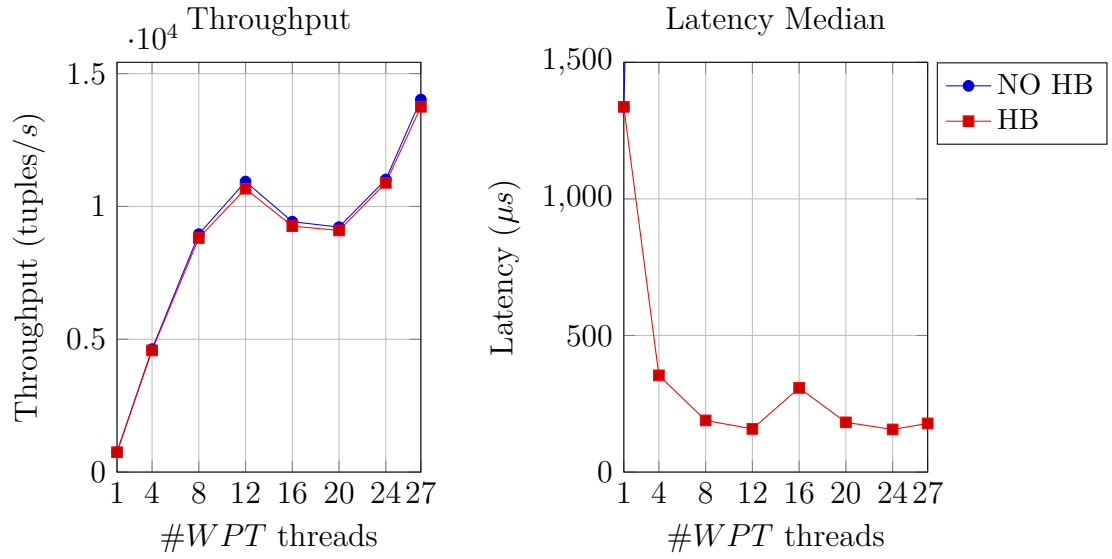


Figure 13.13: 31228 (Intel Xeon): E7P throughput and latency median

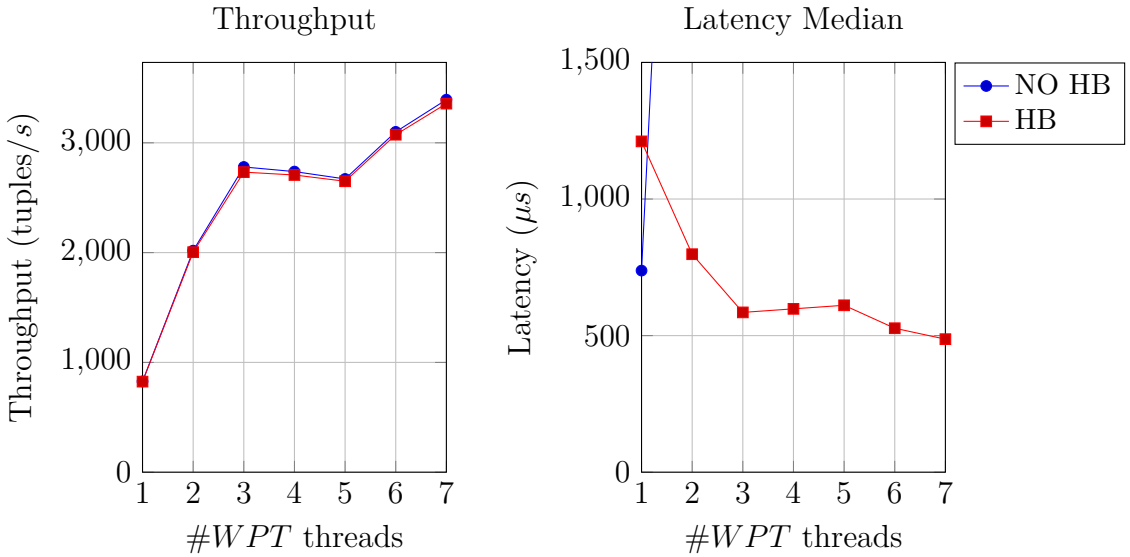


Figure 13.14: Hasgreen (Intel Core i7): E7P throughput and latency median

Focusing on the reported throughput, as it can be seen, the expected linear scaling tendency has been met before having to start assigning more than one thread to some or all of the processors. Reached that point, as it happened with the E6P configuration as reported in Section 13.2.4, a drop in throughput can be appreciated before getting back to an increasing tendency. Actually, if the throughput reported in this section is compared to the throughput reported in Section 13.2.4, it is difficult to appreciate big differences. The reason for this is that the overhead of matching the streams is, as anticipated in Section 12.4.1 when analyzing the cost of the multi-threaded volatility aggregation and stream matching operator, almost negligible compared to the volatility aggregation workload.

Focusing on the reported latency, the results obtained with the E7P configuration are also very similar to the results reported in Section 13.2.4 for the E6P configuration for the same reason as explained above when analyzing the throughput results.

13.2.7 Multi-Threaded Volatility Aggregation and Stream Matching, and Multi-Threaded Binomial Options Pricing (E7F, and E7C)

Tables 13.19, and 13.20 below report the throughput and latency median achieved respectively in the 31228 machine (Intel Xeon), and Hasgreen (Intel Core i7) when executing the E7F configuration using all the available logical processors with the number of volatility aggregation and stream matching threads, WPT threads, reported in the left-most column, and the number of options pricing threads, PPT threads, necessary to utilize all the available logical processors, namely the left-most column read bottom-up, using and not the heartbeat mechanism, according to the naming conventions introduced in Section 13.1.

Table 13.19: 31228 (Intel Xeon): E7F throughput and latency median

	Throughput (tuples/s)		Latency Median (μs)	
	NO HB	HB	NO HB	HB
1	450.00	450.00	58996	5674
4	2397.89	2387.81	26683	3161
8	4337.86	4290.88	31102	2642
12	3484.59	3418.81	58461	2741
15	3881.62	3777.61	76720	2668
19	2962.59	2896.00	142175	2809
23	1624.46	1600.90	349693	3371
26	481.75	477.00	2300212	4216

Table 13.20: Hasgreen (Intel Core i7): E7F throughput and latency median

	Throughput (tuples/s)		Latency Median (μs)	
	NO HB	HB	NO HB	HB
1	583.00	583.00	11202	4987
2	1492.99	1487.00	16303	3189
3	1206.00	1202.00	31412	3508
4	1326.99	1323.00	50850	4787
5	985.99	984.00	96972	3816
6	547.99	547.00	256673	3587

To help better appreciating the way throughput and latency scale as more VPT threads and less PPT threads are used, Figures 13.15, and 13.16 below plot all the results reported in Tables 13.19, and 13.20.

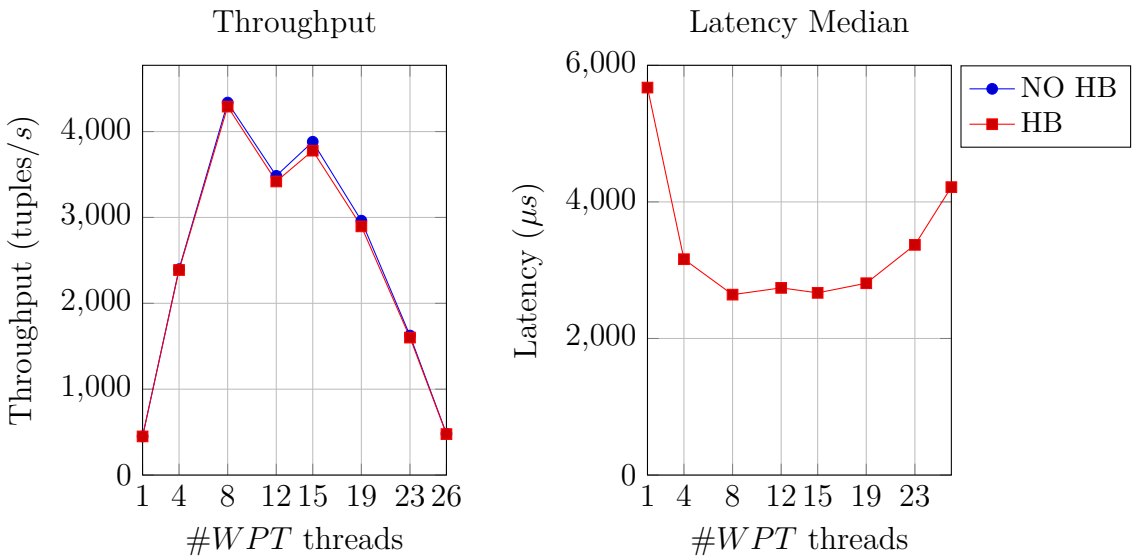


Figure 13.15: 31228 (Intel Xeon): E7F throughput and latency median

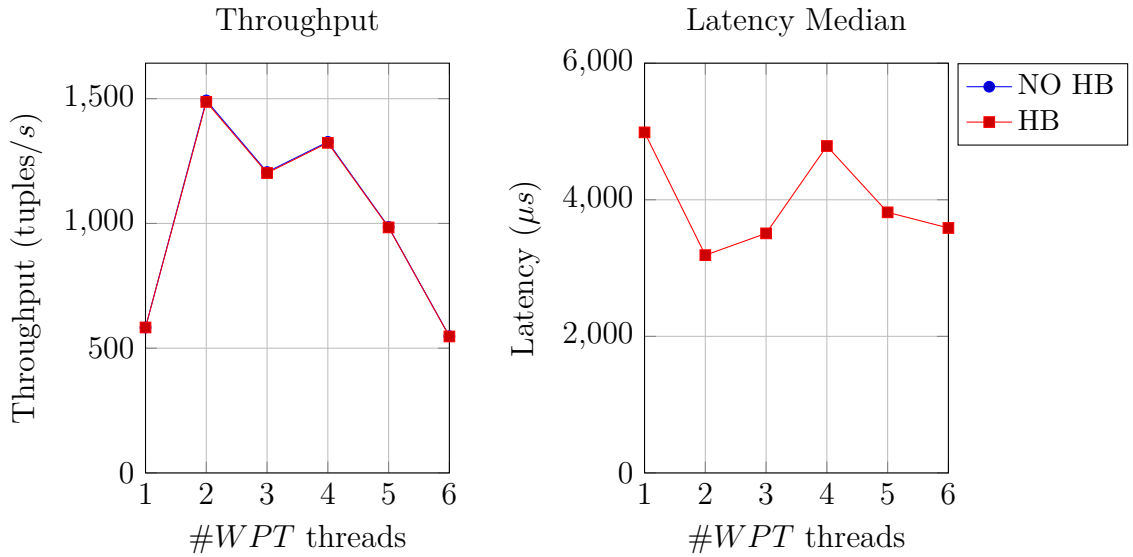


Figure 13.16: Hasgreen (Intel Core i7): E7F throughput and latency median

Focusing on both the throughput and latency results, similarly as it happened when comparing the results reported for the E7P, and E6P configurations in the previous section, and for the same reasons, the throughput and latency results reported for the E7F configuration are very similar to the throughput and latency results reported for the E6F configuration.

Tables 13.21, and 13.22 below report the throughput and latency median achieved respectively in the 31228 machine (Intel Xeon), and Hasgreen (Intel Core i7) when executing the E7C configuration with the number of volatility aggregation, stream matching and options pricing threads, $WPPT$ threads, reported in the left-most col-

umn, using and not the heartbeat mechanism, according to the naming conventions introduced in Section 13.1.

Table 13.21: 31228 (Intel Xeon): E7C throughput and latency median

	Throughput (tuples/s)		Latency Median (μs)	
	NO HB	HB	NO HB	HB
1	296.00	297.00	1743	4304
4	1185.00	1179.00	27712	28250
8	2258.95	2255.95	48191	47797
12	3221.94	3218.95	63808	63856
16	2656.96	2672.95	103034	103585
20	3024.79	3024.79	144382	145165
24	3557.76	3559.76	164811	165661
27	4125.71	4110.24	185902	185969

Table 13.22: Hasgreen (Intel Core i7): E7C throughput and latency median

	Throughput (tuples/s)		Latency Median (μs)	
	NO HB	HB	NO HB	HB
1	343.00	343.00	1896	1532
2	714.00	713.00	12505	13466
3	1054.00	1054.00	18889	18888
4	900.00	902.00	36418	35533
5	1073.00	1073.00	45405	46263
6	1208.98	1210.00	54726	54741
7	1403.00	1402.00	64081	64515

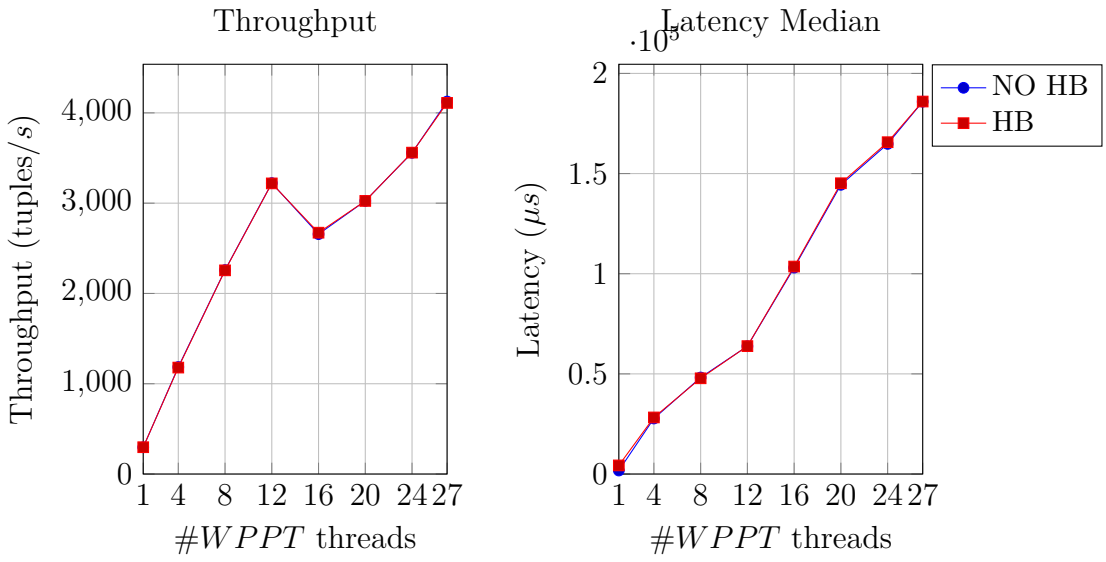


Figure 13.17: 31228 (Intel Xeon): E7C throughput and latency median

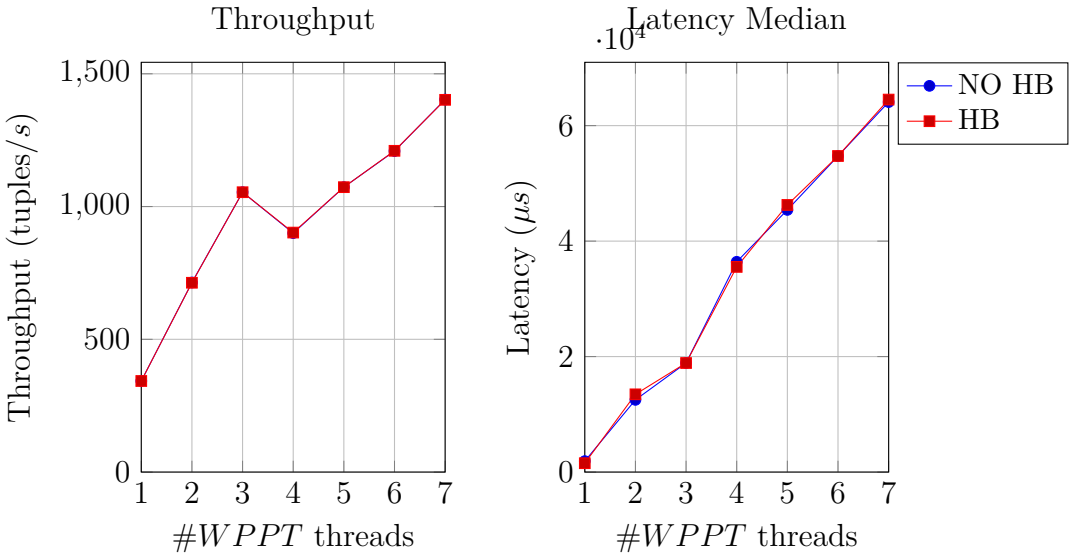


Figure 13.18: Hasgreen (Intel Core i7): E7C throughput and latency median

To help better appreciating the way throughput and latency scale as more $WPPT$ threads are used, Figures 13.17, and 13.18 above plot all the results reported in Tables 13.21, and 13.22.

Focusing on both the throughput and latency results, similarly as it happened when comparing the results reported for the E7F, and E6F configurations earlier in this section, and for the same reasons, the throughput and latency results reported for the E7C configuration are very similar to the throughput and latency results reported for the E6C configuration.

14
Related Work

In this chapter, state of the art work related to the research performed in the context of this Thesis is briefly introduced and analyzed in comparison to the contributions introduced in the previous chapters.

In the scope of shared-memory parallelization, lock-free synchronization, [5], has progressively gained popularity motivating the research towards lock-free synchronizing concurrent data-structures such as the queues proposed in [35, 32, 16] used in the scope of this thesis thanks to the NOBLE library [42, 43]. A good compendium of some of the most relevant concurrent data-structures to date can be found in [19, 20] which served as a great source of reference in the context of this Thesis together with [25], which elaborate on the implementation of the most common concurrent data-structures with a strong penchant towards lock-freedom.

In the scope of data-streaming, the *ScaleGate* data-structure introduced in Section 3.2, has been proposed by the Division of Networks and Systems of the Department of Computer Science and Engineering at Chalmers as a key building block for parallel efficient stream processing solutions [7, 22].

This data-structure, which actually represents a key building block in the last iterations of the stream processing engine introduced in the previous chapters, has been used to address generic data-streaming problems such as multiway aggregation [6], and stream join [21, 23], as well as scope-specific data-streaming problems such as the production of deterministic real-time analytics of geospatial data streams [24]. It is worth mentioning that the multiway aggregation solution [6], and the stream join solution [21, 23] strongly inspired respectively the volatility aggregation solutions introduced in Chapters 10 and 11, and the stream matching solution introduced in Chapter 12.

In the scope of computational finance, several papers highlight the continuously increasing high throughput and low latency and energy consumption requirements demanded to process financial streams of data [17, 14, 30]. The first two cited papers also elaborate on the definition of platform independent energy and performance metrics.

One of the most widely approached financial computational problems approached in the literature is the pricing of equities and in particular options in its many

flavors. Two main trends to approach options pricing problems are identified in [17, 10], either based on Monte Carlo models, or grid-based or binomial models.

As anticipated in Chapter 6, in [29] the single-asset European options pricing problem is approached relying on an underlying binomial tree model strongly inspired by the classical options pricing literature [4, 8]. Even though the operator presented in this paper is the one used to price options in all the different iterations of the options pricing stream processing engine introduced in this Thesis, the approach followed in this paper strongly differs from the approach followed in this Thesis. The former focuses on parallelizing the operator itself as much as possible in order to maximize throughput profiting from architecture specific optimizations such as vectorization whereas the latter focuses on the integration of the operator in a stream processing engine pursuing the increase of throughput as well as the smallest increase as possible of latency and ensuring that options are output in the same order as their corresponding settings are provided to the stream processing engine. Another architecture specific approach to price, in this case, American options on GPUs implementing the Least Squares Monte Carlo method [31], is introduced in [12].

More complex options pricing and equity pricing problems are approached in [10, 3, 38] which respectively approaches the problems of pricing multi-asset American options, pricing more elaborate equity products and parallelizing the pricing operators through the usage of GPUs, and pricing swing options on GPGPUs.

Other widely approached problems concern market-risk assessment. In [9], the problem of calculating of the correlation matrix is approached targeting low latency and profiting from the specific architecture features of the Blue Gene supercomputer. In [2], the problem of computing the diagonal of inverse covariance matrices for uncertainty quantification in risk analysis is approached reducing its complexity from cubic to quadratic. The assessment of the underlying market volatility has also been approached from different theoretical points of view in [1, 39, 18]. In the scope of this thesis, a more conservative approach to the assessment of volatility [41], has been integrated in the window size and window advance sliding-window model introduced in [6].

15
Future Work

The research effort performed in the context of this Thesis inspired the proliferation of many potential research lines which could not be covered in the scope of the Thesis here reported. This chapter briefly outlines the most interesting open research lines which can take as a starting point the research contributions reported in the scope of this Thesis.

In order to better understand the mathematical nature of the underlying operators introduced in this Thesis, one interesting line of research would involve approaching the visualization of the values assigned to the tuples the by the different operators in conjunction with the input data used by these operators. This could lead to interesting insights towards the definition of computationally-cheaper estimation operators leading to further increases in throughput and decreases in latency.

Given the fact that the binomial options pricing operator has not been internally parallelized but instead used in parallel to price independent options contracts, it should be straight forward to try different single-threaded options pricing operators targeted for more complex options pricing tasks such as Bermudan options pricing, multi-asset options pricing or stochastic volatility model based options pricing. Another interesting line of research would involve trying to parallelize either the binomial options pricing operator or a different options pricing operator in order to profit from the *ScaleGate* data-structure increasing throughput and reducing latency in a similar manner as the parallel sliding-window based volatility aggregation operator introduced in Chapter 11 does. One intermediate approach between the two previous research lines would be to profit from the vectorialization capabilities of the Xeon Phi coprocessor in the 31228 machine in a similar manner as done in [29] to expedite the behavior of the binomial option pricing operator as used in the different experiments performed in the scope of this thesis.

Given the fact that the parallelization approach followed in Chapter 11 to distribute the workload derived from the maintenance of the underlying sliding-window model is actually inherent to the underlying sliding-window model independently of the internal behavior of the volatility aggregation sliding windows, a potentially fruitful research line would involve profiting from the parallelization logic introduced in Chapter 11 in order to parallelize different stateless or stateful aggregation operators. This can be achieved by simply modifying the internal data-structure of the volatility aggregation windows, the set of values retrieved from the tuples to be

provided in the different procedures described in Chapter 11, and the `update` and `consume` procedures introduced in Listing 10.1.

One example of aggregation operator which might be interesting testing as described above would be a tendency line aggregator whose output can be potentially used to determine whether to trigger or not a bid or ask order according to the expected behavior of the underlying asset as summarized by the tendency line aggregator. An interesting way to assess the quality of the resulting stream of bid and ask orders would be to model given the tuples belonging to the financial stream of tuples introduced in Section 5.1.1 the achieved earnings or losses given an initial simulated budget. In addition to this example, other non-necessarily financial aggregation operators can also profit from the aforementioned sliding-window model based parallelization.

Apart from the approached options pricing, volatility aggregation and stream matching problems, other interesting financial problems such as the ones approached in [10, 3, 38] briefly described in the previous chapter can be approached on a streaming fashion profiting from the *ScaleGate* data-structure similarly as the aforementioned problems have done in the context of this thesis.

Another interesting research line would involve carefully profiling the behavior of the different solutions presented in the previous chapters to identify potential performance bottlenecks derived from the specific implementation of the operators and concurrent data-structures. In this research line, carefully configuring the CPU affinity of the different process threads in the stream processing engine could lead to a reduction in the cache access contention potentially improving both the reported throughput and latency for all the experiments.

A natural future research line would be the one following the path already outlined by the main research lines followed in the context of this Thesis. Further iterations beyond the seven ones reported in this Thesis can be introduced further introducing new functionality to the stream processing engine or trying different configurations, such as letting the *SIT* thread introduced in Chapter 12 directly serve tuples to the second *ScaleGate* instance and letting the stream matching logic be executed by a single stream matching process thread before serving the tuples this thread would share with the options pricing threads.

Given the *ScaleGate* data-structure ability to deal with multiple input physical and logical streams of data, an interesting research line would involve letting the output thread monitor the latencies registered by each tuple allowing it to add in case of need control tuples to the input *ScaleGate* instance to dynamically rearrange the load balance dedicating more threads to volatility aggregation or options pricing according to the needs deducted by the output thread.

Finally, a very interesting research line would involve executing in a wider variety of machines with different architectures such as more than one socket, all the iterations of the stream processing engine as done in the 31228 and Hasgreen machines in order to extend the experimental results reported and analyzed in Chapter 13.

16
Discussion and Conclusion

In this Thesis the *ScaleGate* concurrent data-structure introduced by the Division of Networks and Systems of the Department of Computer Science and Engineering at Chalmers has been implemented in C and used to approach three main computational finance problems, namely options pricing, volatility aggregation and stream matching, in the scope of data streaming. As a result, the experimental financial stream processing engine described in Chapter 12 has been produced. This stream processing engine is able to efficiently process two different financial streams producing aggregated market volatility measurements on a sliding window basis and pricing options contracts based on the information resulting from matching the two streams of data.

In the chapters describing the different iterations of the aforementioned stream processing engine, efficient ways to either parallelize an operator or let it process independent tuples in parallel yet ensuring that tuples are output by the stream processing engine in the same order as they are input to it, have been introduced. Worth mentioning is the heartbeat mechanism introduced in Chapter 9 to expedite the behavior of *ScaleGate* yet meeting the aforementioned ordering constraint, which achieves when used in most the stream processing engine configurations orders of magnitude lower latency metrics than the ones achieved without making use of it yet achieving the same performance in terms of throughput.

Bibliography

[1] T. G. Andersen, T. Bollerslev, F. X. Diebold, and H. Ebens. The distribution of realized stock return volatility. *Journal of Financial Economics*, 61(1):43–76, 2001.

[2] C. Bekas, A. Curioni, and I. Fedulova. Low Cost High Performance Uncertainty Quantification. In *Proceedings of the 2Nd Workshop on High Performance Computational Finance*, WHPCF '09, pages 8:1–8:8, New York, NY, USA, 2009. ACM.

[3] A. Bernemann, R. Schreyer, and K. Spanderen. Pricing structured equity products on GPUs. In *High Performance Computational Finance (WHPCF), 2010 IEEE Workshop on*, pages 1–7, Nov 2010.

[4] F. Black and M. Scholes. The Pricing of Options and Corporate Liabilities. *Journal of Political Economy*, 81(3):637–654, 1973.

[5] D. Cederman, A. Gidenstam, P. H. Ha, H. Sundell, M. Papatriantafilou, and P. Tsigas. Lock-free Concurrent Data Structures. *CoRR*, abs/1302.2757, 2013.

[6] D. Cederman, V. Gulisano, I. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas. Concurrent Data Structures for Efficient Streaming Aggregation. Report, Chalmers University of Technology, 2013.

[7] D. Cederman, V. Gulisano, I. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas. Brief Announcement: Concurrent Data Structures for Efficient Streaming Aggregation. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 76–78, New York, NY, USA, 2014. ACM.

[8] J. C. Cox, S. A. Ross, and M. Rubinstein. Option pricing: A simplified approach. *Journal of Financial Economics*, 7(3):229–263, 1979.

[9] D. Daly, K. D. Ryu, and J. E. Moreira. Multi-variate finance kernels in the Blue Gene supercomputer. In *High Performance Computational Finance, 2008. WHPCF 2008. Workshop on*, pages 1–7, Nov 2008.

[10] D. M. Dang, C. C. Christara, and K. R. Jackson. Pricing multi-asset American options on Graphics Processing Units using a PDE approach. In *High Performance Computational Finance (WHPCF), 2010 IEEE Workshop on*, pages 1–8, Nov 2010.

[11] P. M. Dubois, M. Annavaram, and P. Stenström. *Parallel Computer Organization and Design*. Cambridge University Press, 2012.

[12] M. Fatica and E. Phillips. Pricing American Options with Least Squares Monte Carlo on GPUs. In *Proceedings of the 6th Workshop on High Performance Computational Finance*, WHPCF '13, pages 5:1–5:6, New York, NY, USA, 2013. ACM.

[13] B. Gedik, R. R. Bordawekar, and P. S. Yu. Celljoin: a parallel stream join operator for the cell processor. *The VLDB Journal*, 18(2):501–519, 2009.

[14] G. Georgakoudis, C. J. Gillan, A. Sayed, I. T. A. Spence, R. Faloon, and D. S. Nikolopoulos. Methods and Metrics for Fair Server Assessment under Real-Time Financial Workloads. *CoRR*, abs/1501.00048, 2015.

[15] A. Gidenstam, M. Papatriantafilou, H. Sundell, and P. Tsigas. Efficient and Reliable Lock-Free Memory Reclamation Based on Reference Counting. *IEEE Transactions on Parallel and Distributed Systems*, 20(8):1173–1187, Aug 2009.

[16] A. Gidenstam, H. Sundell, and P. Tsigas. *Principles of Distributed Systems: 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings*, chapter Cache-Aware Lock-Free Queues for Multiple Producers/Consumers and Weak Memory Consistency, pages 302–317. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[17] C. J. Gillan, D. S. Nikolopoulos, G. Georgakoudis, R. Faloon, G. Tzenakis, and I. Spence. On the Viability of Microservers for Financial Analytics. In *High Performance Computational Finance (WHPCF), 2014 Seventh Workshop on*, pages 29–36, Nov 2014.

[18] V. Golosnoy, B. Gribisch, and R. Liesenfeld. The conditional autoregressive Wishart model for multivariate stock market volatility. *Journal of Econometrics*, 167(1):211–223, 2012.

[19] V. Gramoli. More Than You Ever Wanted to Know About Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 1–10, New York, NY, USA, 2015. ACM.

[20] V. Gramoli. More Than You Ever Wanted to Know About Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms. *SIGPLAN Not.*, 50(8):1–10, Jan. 2015.

[21] V. Gulisano, I. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas. ScaleJoin: a deterministic, disjoint-parallel and skew-resilient stream join enabled by concurrent data structures. Technical report, Chalmers University of Technology, 2014. 12.

[22] V. Gulisano, I. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas. Data-Streaming and Concurrent Data-Object Co-design: Overview and Algorithmic Challenges. In *Lecture Notes in Computer Science. European Symposium on Algorithms, ESA 2015, Patras, Greece, 16 September 2015*, pages 242–260, 2015.

[23] V. Gulisano, I. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas. Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 144–153, Oct 2015.

[24] V. Gulisano, I. Nikolakopoulos, I. Walulya, M. Papatriantafilou, and P. Tsigas. Deterministic Real-time Analytics of Geospatial Data Streams Through ScaleGate Objects. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS '15, pages 316–317, New York, NY, USA, 2015. ACM.

[25] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

[26] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[27] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[28] B. Klemens. *21st Century C: C Tips from the New School*. O'Reilly Media, 2012.

[29] S. Li. Binomial Options Pricing Model Code for Intel© Xeon Phi™ Coprocessor. Technical report, Intel Developer Zone, May 2014. `https://software.intel.com/en-us/articles/binomial-options-pricing-model-code-for-intel-xeon-phi-coprocessor`.

[30] A. Lindeman. Opportunities for shared memory parallelism in financial modeling. In *High Performance Computational Finance (WHPCF), 2010 IEEE Workshop on*, pages 1–6, Nov 2010.

[31] F. A. Longstaff and E. S. Schwartz. Valuing American options by simulation: A simple least-squares approach. *Review of Financial Studies*, pages 113–147, 2001.

[32] C. Lu, T. Masuzawa, and M. Mosbah, editors. *Principles of Distributed Systems: 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[33] R. W. Melicher and E. A. Norton. *Introduction to Finance: Markets, Investments, and Financial Management*. Wiley, 2011.

[34] M. M. Michael. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004.

[35] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[36] Nyxdata. *Daily TAQ Client Specification*, 2.1. edition, June 2015. `http://www.nyxdata.com/doc/243156`.

[37] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*, v 4.5 edition, November 2015. `http://www.openmp.org/mp-documents/openmp-4.5.pdf`.

[38] G. Pagès and B. Wilbertz. Parallel implementation of Quantization methods for the valuation of swing options on GPGPU. In *High Performance Computational Finance (WHPCF), 2010 IEEE Workshop on*, pages 1–5, Nov 2010.

[39] Z. Qu and P. Perron. A stochastic volatility model with random level shifts and its applications to S&P 500 and NASDAQ return indices. *The Econometrics Journal*, 16(3):309–339, 2013.

[40] J. Reinders. *An Overview of Programming for Intel© Xeon© processors and Intel Xeon Phi™ coprocessors*. © 2012, Intel Corporation, rev 20120131 edition, October 2012.

[41] S. M. Ross. *Introductory Statistics*. Elsevier Sciences, 3rd edition, 2010.

[42] H. Sundell and P. Tsigas. *NOBLE Professional Edition Application Programmers Interface (API)*. © 2008 Parallel Scalable Solutions AB, May 2008.

[43] H. Sundell and P. Tsigas. *NOBLE Professional Edition v2.2 Developers Manual*. © 2009 Parallel Scalable Solutions AB, March 2009.

[44] A. S. Tanenbaum and H. Bos. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 4th edition, 2014.

[45] A. Williams. *C++ Concurrency in Action: Practical Multithreading*. Manning Publications, 2012.