Performance Analysis of Thread Synchronization Strategies in Concurrent Data Structures Based on Flat-Combining

Marsel Galimullin, Eugeny Kalishenko
St.Petersburg Electrotechnical University "LETI"
St.Petersburg, Russian Federation
mfgalimullin@yandex.ru, ydginster@gmail.com

Nikolay Rapotkin
PJSC Information Telecommunication Technologies
St.Petersburg, Russian Federation
rapotkinnik@gmail.com

***Abstract*—The article deals with the development of threads synchronizing strategies based on the creation of concurrent "flat-combining" data structures as well as research of their performance. The paper considers "flat-combining" approach and its implementation in the library libcds, the development of thread synchronization strategy and its possible implementations. The efficiency of synchronization strategies usage is researched on the example of the open source library libcds. The research revealed the strategy with the lowest operation execution time on a container and the lowest amount of CPU resources, and identifies use cases of the developed strategies. A mechanism with the developed synchronization strategy to build concurrent data structures was implemented. The implemented strategies were integrated in the cross-platform open source library libcds.**

I. INTRODUCTION

The development of high-performance systems caused the uprise of competitive data structures encapsulating the logic of threads' synchronization and targeting different usage scenarios of containers. Some of the scenarios represent the rise of performance in case of increase of one thread's operate time while other threads delegate their tasks to that one thread. This approach was called flat-combining (FC) [1].

FC is the most general approach for creation concurrent high performance data structures ased on sequential access data structures as deque, list, tree etc. High performance ensured by the fact that operating system scheduler allocates time quanta to threads proportional to their load.

Common FC which described in [1] implemented in the cross-platform open source library libcds [9]. It is C++ library wich contains wide set of c high performance oncurrent data structures based on lock-free and wait-free algorithms.

The main idea of flat-combining in case of stack is as follows: for the stack we use mutex and a publication list of size that is proportional to the number of threads working with the stack. Every thread first time accessing to the stack adds its record to the publication list. If a thread needs to execute an operation on the container, it publishes a request in its record and tries to acquire mutex. The request consists of an operation (in case of a stack, push or pop) and its arguments. If mutex is acquired, the thread becomes a combiner. So, it looks through the list, executes all requests from it, puts the result in the corresponding list item and, finally, releases mutex. Otherwise, if the attempt to acquire mutex failed, the thread is spinning on its record until a combiner executes its request and puts the result in its record in the publication list.

A publication list is built in a special way to reduce overhead costs on its control. The clue moment is rare changing of a publication list. Otherwise, apart from controlling access to a sequential structure it is needed to control access and the consistency of a lock-free publication list. A request for operation is placed in already existing record which is placed in TLS. To simplify lock-free list control the list head is never changed and used as a fake element, which doesn't belong to any thread of FC core. New records are placed strictly in the list's end.

Some records of the list can have «empty» status. That means the corresponding thread is not performing any actions through the sequential structure at the moment. From time to time a combiner excludes inactive records from the list choosing records that were inactive for a number of combiner's iterations. That reduces time needed to process «empty» annotations (long time inactive threads). Instead of physical removal a logical list record removal is used. So such a record is marked as «inactive» and that excludes it from the combiner-thread's processing. As physical removal is a rare occasion that reduces overhead costs on lock-free list control. Still, that helps to control list's size.

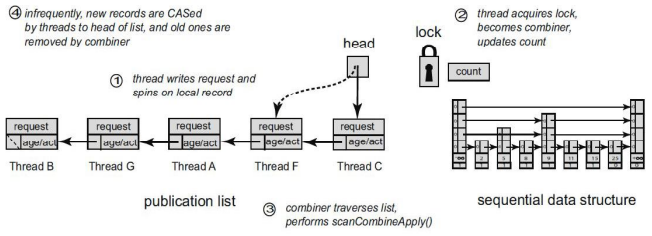

Fig. 1. The principal components of the data structure constructed within the framework of flat-combining: consecutive data structure protected by synchronization primitive and the publication list

II. SYNCHRONIZATION STRATEGIES

However, threads' busy waiting disturbs combiner-thread's work, aside from using processor resources. Different

synchronization strategies can be used to decrease the influence of waiting threads on the work of combiner-thread. Strategies had been selected for development and investigation are based on wait/notify technique and feature the configuration of mutex usage and a condition variable. The standard wait/notify algorithm is frequently used for Publish/Subscribe pattern realization and consists of the next steps:

- A consumer-thread is waiting some condition to become true, in this case such thread is the thread that announces its operation and waits the operation to complete. The waiting thread should acquire the synchronization primitive. This blocking is passed to the wait() method that releases mutex and suspends thread until the signal from the conditional variable is got. Then the thread awakes and the synchronization primitive is acquired again;
- A producer-thread indicates the condition has become true, in this case such thread is the combiner-thread which executes the announced operation and sets operation-completed flag.

Let's consider implemented strategies in details and compare them.

A. Back-off strategy

This strategy is used in the original realization of flat-combining in libcds library. When N threads compete for the critical resource that can be accessed with the help of CAS-operations, only one of them gets an access. Other N–1 threads interrupt each other and consume process time in vain. In case of such a situation detection in order to offload a processor threads can back off, stop execution of the main task and do something useful or simply wait. To do it the back-off strategies are used.

According to the report [3], using back-off strategy implemented as a 2 milliseconds delay in every iteration of cycle, increased FC productivity significantly and showed good results in high load condition.

B. Wait/notify strategy based on the global mutex and the condition variable

The strategy is based on the usage of synchronization primitives of the FC core which are shared by all threads.

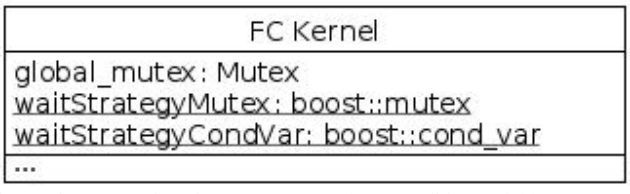

Fig. 2. The strategy based on one mutex and one condition variable

Algorithm 1 The threads interaction algorithm (Fig.3)

1: If a thread wants to execute an operation on the container, it announces a record and tries to acquire mutex of the FC core.

2: If a thread fails to acquire mutex of the FC core and become a combiner, it waits for notification of the operation completion by a combiner-thread on a condition variable shared by all threads.

3: If a thread successes to acquire the FC core mutex this thread becomes a combiner. The combiner processes all the records one by one. Having processed an announced record, the combiner-thread notifiers all awaiting threads.

4: The thread that have got a notification of an operation completion then transits into «Ready» state and starts being processed by a task scheduler. When it transits into «Execution» state and the work actually restarts, it examines its operation completion flag. If the operation is still not executed, the thread starts spinning again awaiting of the next notification.

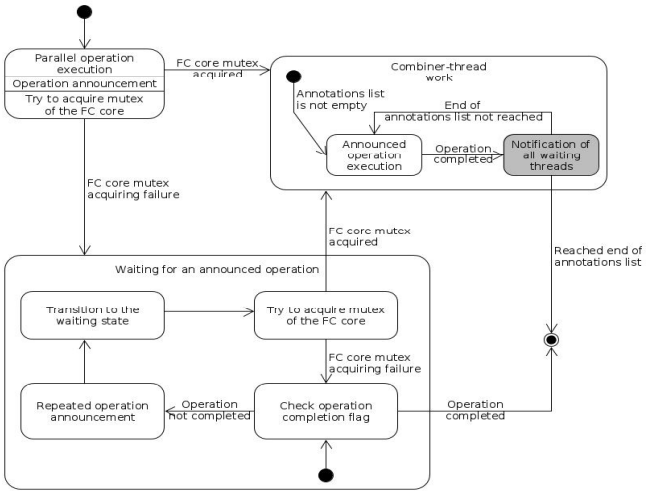

Fig. 3. The strategy algorithm based on one mutex and one condition variable

C. Wait/notify strategy based on the global mutex and the thread-local condition variable

This is a modification of the former strategy. It uses one extra mutex aggregated in the FC core and a condition variable for every thread aggregated in thread's publication record in the publication list (Fig. 4). This modification is dedicated to minimize needless iterations in the cycle of waiting for an announced operation execution in the worst case scenario. It excludes needless iterations at all.

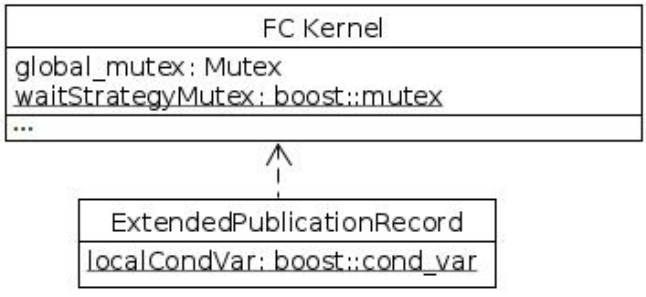

Fig. 4. The strategy based on the global mutex and thread-local cond. variables

Algorithm 2 The threads interaction algorithm (Fig.5)

1: If a thread wants to execute an operation on the container, it announces a record and tries to acquire mutex of the FC core.

2: If a thread fails to acquire mutex of the FC core and become a combiner, it waits for a combiner-thread notification on a condition variable aggregated in the announced record.

3: If a thread successes to acquire the FC core mutex this thread becomes a combiner. The combiner processes all announced records one by one. Having processed an announced record, the combiner-thread notifies its awaiting thread only with the help of the aggregated condition variable.

4: The thread that have got a notification of its operation completion restarts its work and examines the operation completion flag. If the operation is still not executed, the thread starts spinning again awaiting of the next notification.

Despite the fact that this strategy notifies the executed record's thread only, after the thread restart we need to check the operation completion flag because of possible spurious wake-ups. The point is that there is an OS mechanism of live and dead-lock prevention. If an operating system consider threads to be alive or dead-locked it can awake all the threads to rearrange them and to give other threads an opportunity to acquire the synchronization primitive.

Aside from that, a global mutex leads to threads' consecutive processing of the operation completion events what can cause a performance decrease.

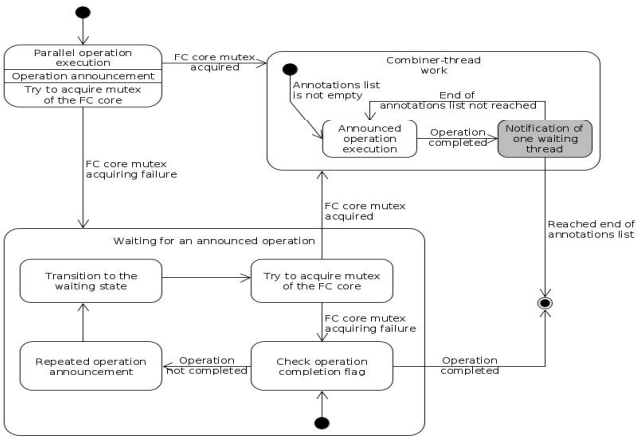

Fig. 5. The strategy algorithm based on condition variables, local for every thread

D. Wait/notify strategy based on thread-local mutex and thread-local condition variable

The last of investigated synchronization strategies is based on a mutex and a condition variable local for every thread. This strategy is alike the algorithms of tweak synchronization of competitive data structures [4]. It is dedicated to exclude needless iterations in the cycle of awaiting announced operations completion. It is also supposed to decrease the competitive struggle of threads for the synchronization primitive. However, this strategy needs more memory for an announced record then the previous strategies as the record needs to aggregate 2 synchronization primitives (Fig. 6).

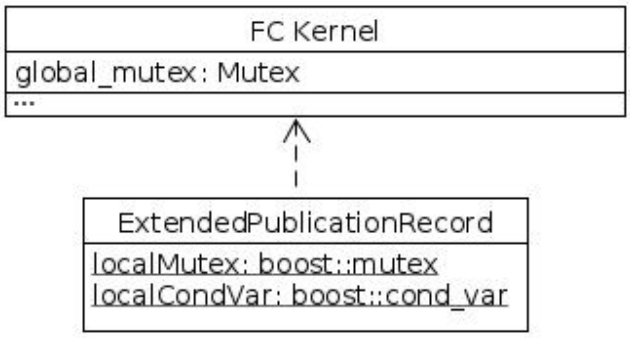

Fig. 6. The strategy based on mutexes and condition variables, local for every thread

Algorithm 3 The threads interaction algorithm

1: If a thread wants to execute an operation on the container, it announces a record and tries to acquire mutex of the FC core.

2: If a thread fails to acquire mutex of the FC core and become a combiner, it waits for a combiner-thread notification on a condition variable aggregated in the announced record.

3: If a thread successes to acquire the FC core mutex this thread becomes a combiner. The combiner processes all the records one by one. Having processed an announced record, the combiner-thread notifiers its awaiting thread only.

4: The thread that have got a notification of its operation completion restarts its work and examines the operation completion flag to avoid the former described situation with live and dead-locks.

The state diagram and the transition diagram of this synchronization strategy is shown on the Fig. 5.

III. Synchronization strategies implementation

A. General implementation

The considered synchronization strategies are implemented in “libcds“ library in the following way:

Algorithm 4 The synchronization algorithms description (Fig. 7)

1: When the FC core object created, an object of the selected strategy class is created.

2: The publication record is extended to ExtendedPublicationRecord by single public sub-classing.

3: The object of ExtendedPublicationRecord is used in the FC core.

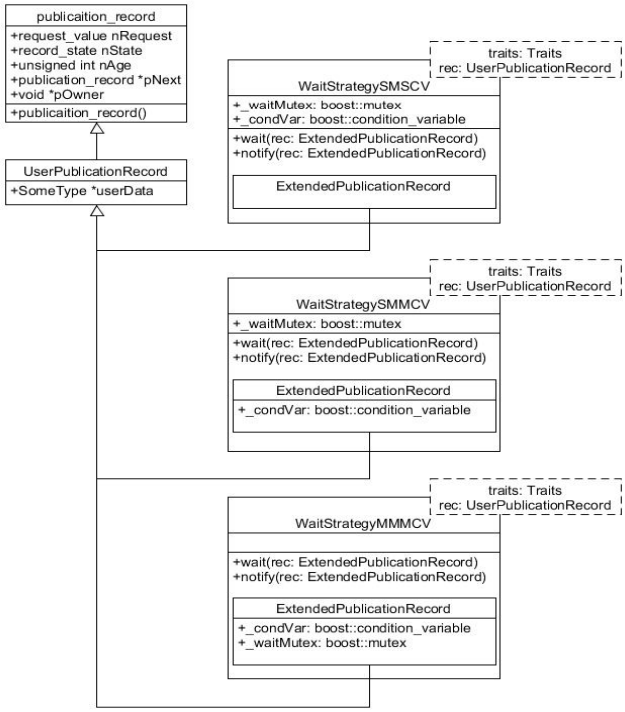

Fig. 7. The synchronization algorithms implementation on the base of "Strategy" design pattern[5][6]; SMSCV - Single Mutex Single Conditional Variable, SMMCV - Single Mutex Multiple Conditional Variable, MMMCV - Multiple Mutex Multiple Conditional Variable

B. Adaptive strategy

Test results shown in the testing section lead to one more type of strategies – the adaptive one. It works like “back-off”-strategy with “light” elements with small size and like MMMCV with “heavy” elements, it based on “Int2Type” metaprogramming pattern[8] and shown on the Fig. 8.

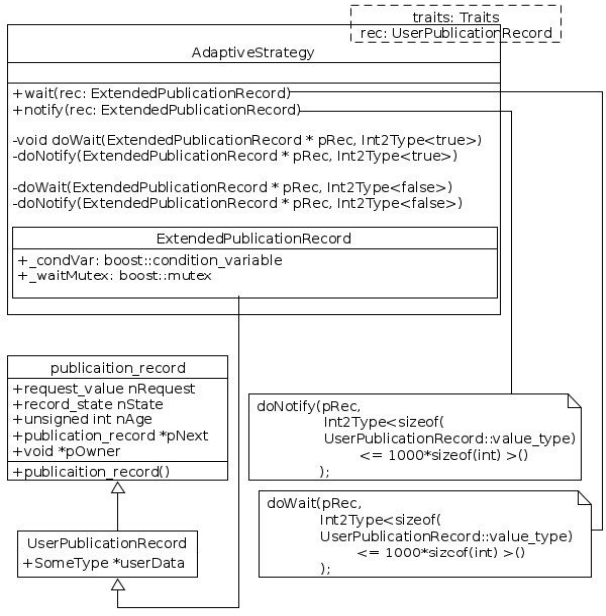

Fig. 8. Adaptive strategy

IV. SYNCHRONIZATION STRATEGIES TESTING

In order to test the developed synchronization strategies it is needed to build a competitive data structure in terms of FC method. For example, it can be a competitive queue (FIFO).

To do it we develop a descendant class of the class cds::algo::flat_combining::container in the FC core to implement interaction with the FC core and to aggregate the FC core object and a standard consecutive queue object (like std::queue). In this case an object of the FC core is instantiated by the user extended record fc_record and the one of the implemented synchronization strategies.

To test the implemented strategies we created the following tests:

- Reader/writer – a half of threads adds elements into the container, another half of threads only deletes elements
- Random – a random sequence of push/pop operations
- Pop – operations of adding elements into the container prevail
- Push – operations of deleting elements from the container prevail

The usage efficiency analysis of the implemented strategies is based on the following values:

- Duration – an average time of push/pop (ms/op) operations execution. This is the main usage efficiency factor as it shows total FC running speed.
- Combining factor – a relation of the number of executed operations to the number of the combiner methods calls. So the combining factor defines the efficiency of the FC in common: the more operations one combiner executes and the less times a combiner-thread changes, the more efficient processor resources usage is.
- Redundant iterations – an average number of redundant iterations in the cycle of waiting for the announced operation completion. The main aim of the developed strategies is this factor's minimization. As, at first, each iteration accesses the FC core's synchronization primitive (see "Flat-combining implementation in libcds library"). And secondly, as a thread is active and consumes the processor resources, therefore it interrupts a combiner-thread and increases overhead costs of threads management by the scheduler.

This factors set allows estimation of the developed synchronization strategies usage efficiency in a most complete way in the context of performance and resources consumption.

The developed synchronization strategies were tested on Intel Core i5-6402P processor with four physical cores and

four hardware threads. Tests were built on Windows 7 by MSVC 2013 compilers.

A. The results of testing a queue with "light" elements

The "light" element's structure:

```
struct SimpleValue {
    size_t    nNo;
    size_t    nThread;
}
```

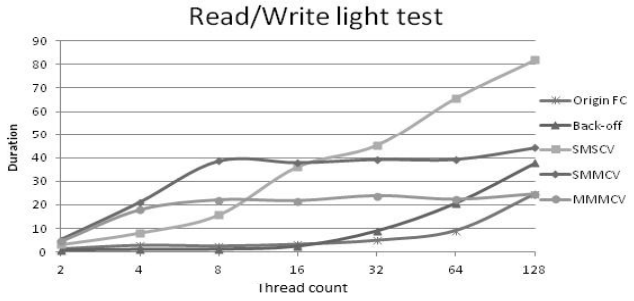

Fig. 9. Duration of the FC with each of implemented strategies

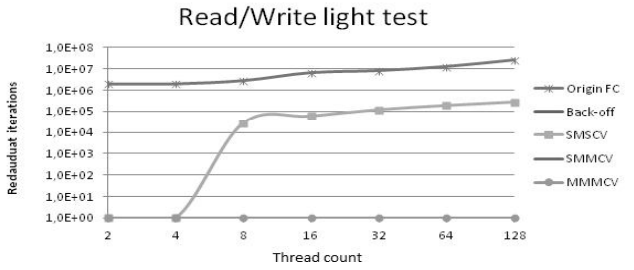

Fig. 10. Number of redundant iterations in tests of "light" element queue

As can be seen on Fig. 9, the developed strategies even with strict competition fall short of the efficiency in comparison with the original implementation and the back-off strategy implementation. Even despite the fact that they have 10 times less redundant iterations at the worst and 10^7 times less redundant iterations at the best case (Fig. 10).

Copying 8 bytes of memory takes negligibly little time so push/pop operations in such structure are executed much faster than the context switching and threads state management. Therefore wait/notify mechanism of the developed strategies only prevents FC's work.

Analyzing the combining factor value while running the FC with a back-off strategy (Fig. 11) explains that high results: an average combining factor for back-off strategy is no more than 1.1. That means almost every thread having announces its record becomes a combiner-thread and has time to execute his own operation only. It can also be confirmed by the absence of redundant iterations (redundant iterations = 0). So threads parallel work comes to consecutive execution of operations over the container.

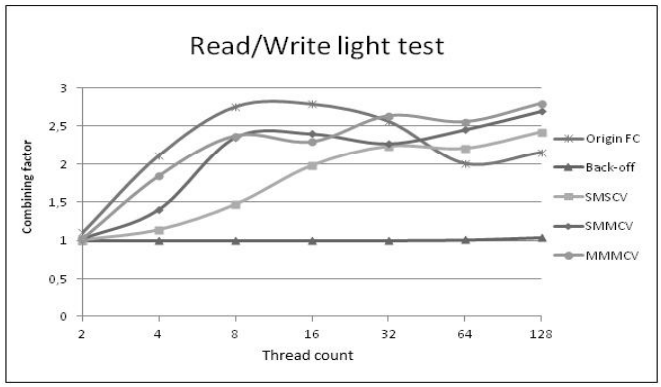

Fig. 11. A combiner-thread's work efficiency

B. The results of testing a queue with "heavy" elements

To avoid former situation we created test of a parallel queue with "heavy" structures as elements of the queue. And the "heavy" structure is:

```
struct HeavyValue {
    size_t    nNo;
    size_t    nWriterNo;

    static int pop_buff[1000];
    HeavyValue() :nNo(0), nWriterNo(0){}

    size_t getNo() const { return  nNo; }
};
```

Static array was used for simulating copy constructor that takes a lot of time by this way:

```
HeavyValue(const HeavyValue &object):
  nNo(object.nNo){
    for (int i = 0; i < array_size; ++i)
        this->pop_buff[i] =
     (int)std::sqrt(object.pop_buff[i]);
}
```

The developed strategies tested "heavily" show much better results (Fig. 12) especially with a big number of threads and hard competitive struggle for the resource. This occurs as in this case the context switch time is small in comparison with the time of operation execution and affects total performance weakly.

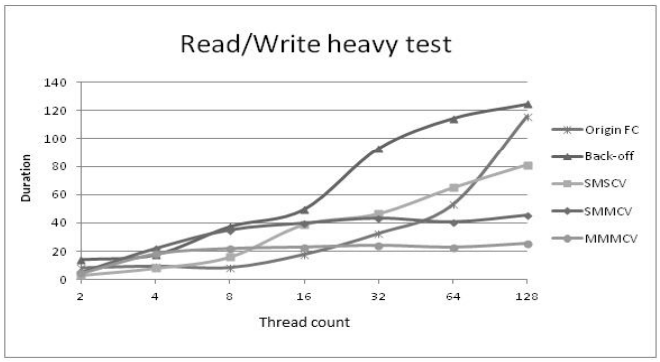

Fig. 12. Duration of "heavy" elements queue in the developed strategies

The original FC implementation and the back-off strategy implementation significantly fall short of the performance in comparison with the developed strategies, especially with a big number of threads. That is caused by the frequent accessing of a synchronization primitive and processor resources usage what leads to increase in overhead costs of a scheduler during threads managing.

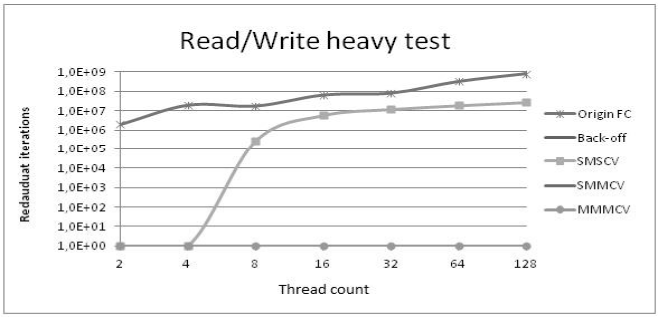

Fig. 13. Number of redundant iterations in the "heavy" elements queue tests

The number of redundant iterations in the original FC implementation and the back-off strategy implementation is 10 times larger at the worst and 10^9 times larger at the best case than in the developed strategies implementations.

C. Unsynchronized wait/notify mechanism problem

It is possible for a thread to not fall asleep before a combiner-thread starts processing operation announced by this thread. Then the following situation happens: a non-combiner-thread checks operation completion flag, finds out it is not executed yet and prepares to fall asleep. At this moment a combiner-thread processes the operation, sets the flag as completed and notifies the thread that announced the operation. But the thread has already begun falling asleep and it will never wake up as the operation has already completed and the notification has already been sent.

To solve the problem it is needed to synchronize access to an operation completion flag and to wait-notify mechanism's methods. That means only one operation can be executed at any moment:

- If operation completion flag is not true, a thread falls asleep, otherwise, it comes out of waiting cycle.
- Setting of completion flag and threads' notification is done according to the selected strategy's algorithm.

To synchronize access to the operation completion flag and to wait-notify mechanism's methods an extra synchronization primitive is used and it is a mutex associated with the selected strategy.

Certainly, the solution slows FC's work down due to the extra piece of code that is executed consequently. But that solves the live-locks problem.

VII. Conclusion

According to the tests we can conclude that the developed synchronization strategies based on the wait/notify mechanism are effective on containers with big size elements (where the size is comparable with **sizeof(int)***1000) and algorithms that need a lot of time for operation execution.

In contrary the usage of the developed strategies for "light"-elemented containers is quite noneffective due to large overhead costs on frequent context switching and threads managing that lasts much longer than a simple operation over a parallel data structure does.

The best testing results were gained for the strategy based on mutex and condition variable aggregation into each publication record. That is explained by the fact that the aggregated mutexes are accessed in a parallel way in contrast to other two strategies with one shared mutex. Multiple Mutex Multiple Conditional Variable Strategy excludes redundant completion notifications due to condition variable aggregation into each record and the determined notification, therefore. The next stable libcds version will include implemented synchronization strategies.

In this paper we presented some aspects of FC and synchronization strategies. We described some issues regarding the efficiency of developed strategies depending on container's element size. But some work has still to be done. First of all, the adaptive strategy should be analyzed more thoroughly in terms of taking into account some other aspects of working with concurrent containers apart from element size. Then it is interesting to analyze some existing systems which use FC techniques, for instance, partitioned global address space framework [7] and apply adaptive strategies implementation there.

Acknowledgment

Firstly, we would like to express our sincere gratitude to libcds maintainer Maxim Khiszinsky for the comprehensive answers about FC implementation and constructive discussions. His guidance helped us to integrate described strategies into the library and use existing library multithreading test framework.

References

[1] Danny Hendler, Itai Incze, Nir Shavit, Moran Tzafrir "Flat Combining and the Synchronization-Parallelism Tradeoff", *in Proc.of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures,* June 13-15, 2010

[2] "Lock-free data structures. Stack evolution", Web: http://habrahabr.ru/company/ifree/blog/216013/

[3] "C++ developers meeting" Web: http://video.yandex.ru/users/ya-events/view/2932

[4] Maurice Herlihy, Nir Shavit. "The Art of Multiprocessor Programming". Morgan Kaufmann, 2012

[5] Erich Gamma, Ralph Johnson, John Vlissides "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley Professional, 1994

[6] David Vandevoorde, Nicolai M. *Josuttis* "C++ Templates: The Complete Guide". Addison-Wesley Professional, 2002

[7] B Holt, J Nelson, B Myers, P Briggs, L Ceze, S Kahan, M Oskin "Flat Combining Synchronized Global Data Structures", *in Proc.of 7th International Conference on PGAS Programming Models*, October 2013, p. 76

[8] Andrei Alexandrescu "Modern C++ Design: Generic Programming and Design Patterns Applied". Addison-Wesley Professional, 2001

[9] libcds source on GitHub, Web: https://github.com/khizmax/libcds.