Multiprocessor Task Executor using Work Queues

Jagan Savanth Reddy Munnangi
Department of Computer Science
Georgia State University
Atlanta, Georgia 30302
jmunnangi1@student.gsu.edu

Shyamsunder Talacheeru
Department of Computer Science
Georgia State University
Atlanta, Georgia 30302
stalacheeru1@student.gsu.edu

Abstract—We try to present a multi thread safe practical implementations of a concurrent array-based FIFO queue that are suitable for both multiprocessor and also preemptive multi-threaded systems. It is a well-known fact that FIFO queues which are based on mutual exclusion will cause blocking the execution of threads due to the concurrent execution, which has technically several drawbacks and can result in slowing down the process of execution. There are also many queuing algorithms which are Link based but does not suite to solve this problem because of the memory management problem which cause a great problem in the overall performance. This is because a removed node from the queue can neither be freed nor reused because other threads may still be accessing the node. Existing solutions to this problem introduce a fair amount of over-head and, when the number of threads that can access the FIFO queue is moderate to high, are shown to be less efficient compared to array-based algorithms, which inherently do not suffer from this problem. We are implementing the thread safe FIFO queues using mutual exclusion mechanism by making use of monitors or semaphores. The FIFO queue used to maintain the queued Tasks will be thread-safe without the data structure being affected and prevents problems like deadlock conditions, resource thrashing.

Keywords— task executor, thread pool, queue,

I. INTRODUCTION

In this paper, we have decided to create a task executor service to handle a safe thread implementation. A task executor service can be implemented primarily in any programming language which has object oriented features. Whether it is a linked list, an array or a queue they can be chosen based on the number of tasks and in the way they are executed. We have chosen Java to implement the Task Executor service as we felt implementation of queues in Java is quiet easier comparatively. The concept of thread pools is implemented as it offers a solution to both the problem of thread life-cycle overhead and the problem of resource thrashing. By reusing threads for multiple tasks, the thread-creation overhead is spread over many tasks. The thread already exists when a request arrives so the delay introduced by thread creation is eliminated. Thus, the request can be serviced immediately, rendering the application more responsive. Tuning the size of thread pool will also be discussed in this paper.

II. LITERATURE SURVEY

There have been many methods published in the similar domain. There is a paper from Institute of Control and Systems Engineering where scheduling tasks on moving executors in complex arithmetic systems is considered. They have assumed work station similar to a queue where the executors are assigned for each task from the station. Algorithms based on the branch and bound approach are implemented. Where as in Miroslav Popovic Task Tree Executor architecture is proposed based on Intel thread blocks. The ultimate goal is to build a parallel processing method thus implementing a multicore CPU utilization. It is given that they achieved a statistical reliability and speed up to 8 times on an average over the previous one. Although in our paper we may have not shown the exact time differences between the thread execution because it is a much complex process, but we might solve the problem of blocking during the task execution by executor threads.

III. COMMON TERMS

A. THREAD POOL

We can define a thread pool as a collection of worker threads that efficiently execute asynchronous callbacks on behalf of the application. The thread pool is primarily used to reduce the number of application threads and provide management of the worker threads. A thread pool consists of a number m of threads, created to perform a number n of tasks concurrently, typically m is not equal to n. The number of threads is tuned to the computing resources available to handle tasks in parallel (processors, cores, memory) while the number of tasks dependents on the problem and may not be known upfront. Reasons for using a thread pool, rather than the obvious alternative of spawning one thread per task, are to prevent the time and memory overhead inherent in thread creation, and to avoid running out of resources such as open files or network connections. A very common way of distributing the tasks to threads is by means of a synchronized queue. The threads in the pool take tasks off the queue, perform then, then return to the queue for their next task.

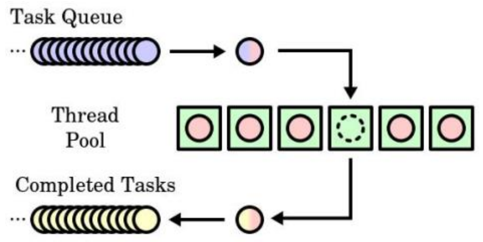


Fig 1: A thread pool with waiting tasks queue and completed tasks.

B. BLOCKING QUEUE

A blocking queue is a queue that blocks when you try to dequeue from it if the queue is empty, or if you try to enqueue items to it and the queue is already full. A thread trying to dequeue from an empty queue is blocked until some other thread inserts an item into the queue. A thread trying to enqueue an item in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more items or clearing the queue completely.

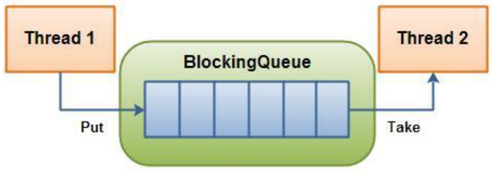


Fig 2: A Blocking Queue with one thread putting into it, and another thread taking from it

C. TUNING

Tuning the size of a thread pool is mainly to avoid having too few threads or too many threads. Fortunately, for most applications the middle ground between too few and too many is fairly wide. Advantages to using threading in applications is to allow processing to continue while waiting for slow operations such as I/O, and exploiting the availability of multiple processors. In a computer bound application running on an N-processor machine, adding additional threads may improve throughput as the number of threads approaches N, but adding additional threads beyond N will do no good. Eventually, too many threads will degrade the performance because of the additional context switching overhead.

The optimum size of a thread pool depends on the number of processors available and the nature of the tasks that are present on the work queue. On an N-processor system for a work queue that will hold entirely compute-bound tasks, you will generally achieve maximum CPU utilization with a thread pool of N or N+1 threads.

IV. PROBLEMS

There are few problems associated with implementing thread pools. Problems like Deadlocks can cause major inconvenience while executing the tasks. With any multithreaded application, there is a risk of deadlock. A set of processes or threads is said to be deadlocked when each is waiting for an event that only another process in the set can cause. Unless there is some way to break out of waiting for the lock , the deadlocked threads will wait forever.While deadlock is a risk in any multithreaded program, thread pools introduce another opportunity for deadlock, where all pool threads are executing tasks that are blocked waiting for the results of another task on the queue, but the other task cannot run because there is no unoccupied thread available.

Resource thrashing is an other problem with thread pools. They generally perform well only if the thread pool size is tuned properly. Threads consume numerous resources, includ-ing memory and other system resources. Besides the memory required for the Thread object, each thread requires two execution call stacks, which can be large. In addition, the JVM will likely create a native thread for each Java thread, which will consume additional system resources. Finally, while the scheduling overhead of switching between threads is small, with many threads context switching can become a significant drag on your program’s performance. If a thread pool is too large, the resources consumed by those threads could have a significant impact on system performance.

There can be Concurrency errors which are to be handled. Thread pools rely on the use of wait() and notify() methods, which can be tricky. If coded incorrectly, it is possible for notifications to be lost, resulting in threads remaining in an idle state even though there is work in the queue to be processed.

V. IMPLEMENTATION

Initially we present details of the implementation method-ology which we used to develop the Task Executor service.

In this project we will produce library JAR file con-taining a TaskExecutor service.

Task Executor is a service class that maintains a pool of N threads that are used to execute instances of Tasks provided by the TaskExecutors clients.

We created an application and Task implementation (TaskExecutorTest.java and SimpleTestTask.java) that is used to design, debug, and test the project.

A. TaskExecutor Service

The TaskExecutor is a service that accepts instances of tasks i.e., classes implementing the Task interface and executes each task in one of the multiple threads maintained by the service.The service maintains a pool of pre-spawned threads that are used to execute Tasks. Typically, a single instance of the TaskExecutor implementation would be installed on the application utilizing the service. The application would avoid creating multiple instances of the TaskExecutor service as that defeats the reason for building service and utilizing pooled threads.

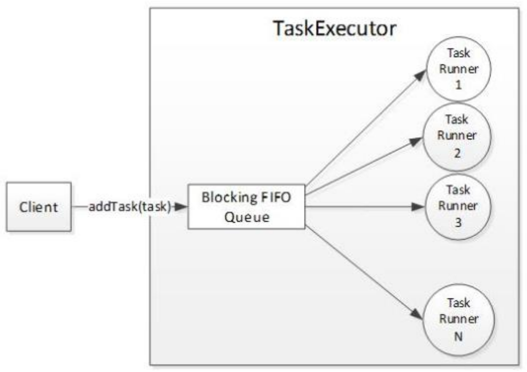
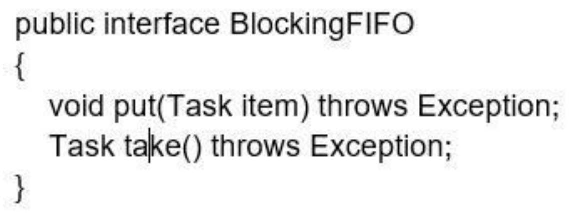


Fig 3: Task Executor Overview.

The above gives an overview of the structure and possible design of the TaskExecutor service. We design Clients to provide implementations of the Task interface which performs some application-specific operation. Clients utilize the TaskExecutor.addTask() method to add these tasks to the FIFO queue. Pooled threads remove the tasks from the queue and execute the Tasks execute() method. The application-specific Task executes for some amount of time before completing by returning from the execute() method. At this point the thread attempts to obtain a new Task from the queue. If the FIFO queue is empty the threads execution must be blocked until a new task is added to the queue. If the FIFO queue is full the clients threads execution must be blocked until a task is been removed from the queue.

B. Implementing the Blocking FIFO Queue

This is a normal FIFO queue with both thread safe and blocking. The queues interface is like below



```
public interface BlockingFIFO
{
   void put(Task item) throws Exception;
   Task take() throws Exception;
}
```

By blocking we mean that the

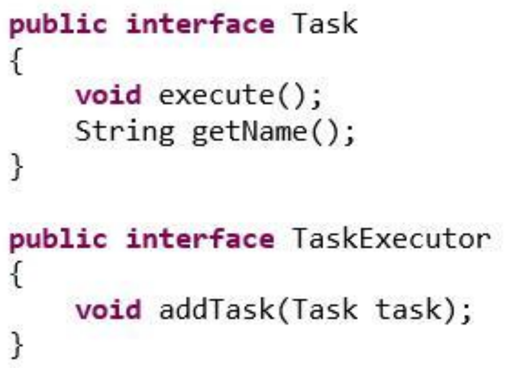
1. The put(task) method places the given task into the queue. If the queue is full, the put() method must blocking the clients thread until space is available when a Task is removed from the queue through the take() method.

2. The take() method removes and returns a task from the queue. If the queue is empty, the thread calling take() will block until a Task has been placed in the queue though the put() method.

In this project we used the class java.util.concurrent.ArrayBlockingQueue which is provided by the JDK runtime library. ArrayBlockingQueue implements the needed blocking behavior as described above. We implemented our own BlockingFIFO queue using semaphores and monitors. The implementation is based on using an array of Task as its container. That is, the size of the queue must be fixed when the container is created. The FIFO implementation size has no more than 100 elements. The reason to choose this size is that an array larger that the number of tasks injected during testing will not exercise the blocking nature of FIFO put() operations.

C. Project Interfaces

The following are the interfaces implemented for TaskExecutor.java and Task.java. These interfaces are followed and the code compiles and executes against the test code created as Testing Source (SimpleTestTask.java and TaskExecutorTest.java).

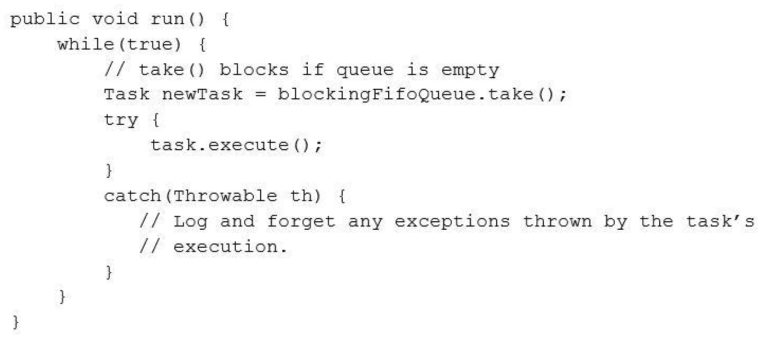


```
public interface Task
{
    void execute();
    String getName();
}

public interface TaskExecutor
{
    void addTask(Task task);
}
```

D. TaskRunner Design

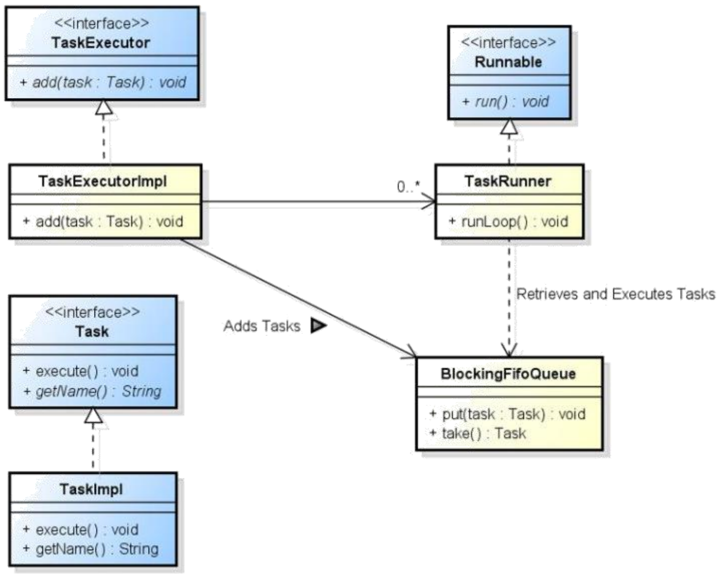
This is the design for the threads responsible for executing Tasks is to create a Runnable that 1) obtains a Task from the FIFO 2) executes the TASK by calling the execute() method. This Runnable is used to instantiate each of the Treads in the pool. The following provides an example of the TaskRunners implication of its run() method.



```
public void run() {
    while(true) {
        // take() blocks if queue is empty
        Task newTask = blockingFifoQueue.take();
        try {
            task.execute();
        }
        catch(Throwable th) {
           // Log and forget any exceptions thrown by the task's
           // execution.
        }
    }
}
```

We can clearly notice that the execution of the task is wrapped in an exception handler that will consume any Throwable generated during the execution of the Tasks execute method. This is needed to keep the Throwable (exception) from killing the TaskRunners thread.

The below is the design of this service. The interfaces provided by Java RT or by the project are marked in blue. The Task implementation is also marked in blue.



E. Testing

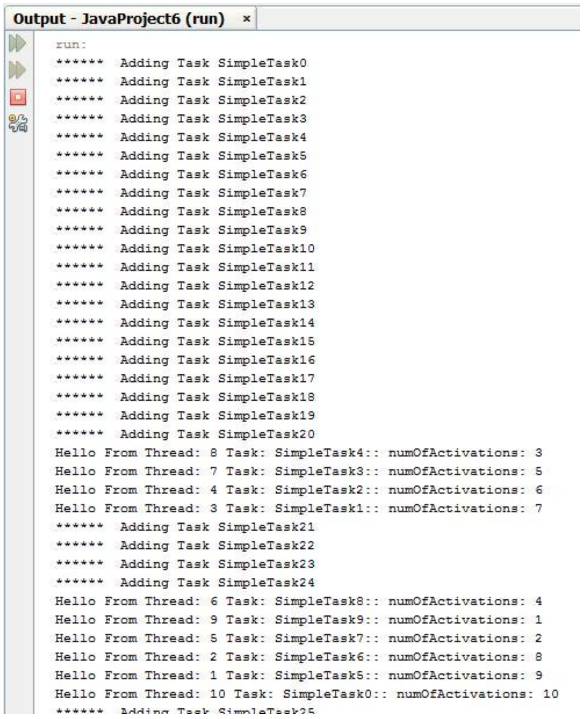
The implementation of the TaskExecutor and the interfaces are packaged and delivered in a JAR file. The implementation is evaluated by executing a prewritten test application using the provided library jar. SimpleTestTask.java and TaskExecutorTest.java are used to test the TaskExecutor implementation. In Eclipse or Net Beans we generate the jar in a development project and executing the test application using the imported JAR onto its classpath.

During the testing we find that each task is executed by following a queue pattern and randomly choosing the jobs. The completed task is removed from the queue and a new task is added. In similar way the loop is iterated 500 times indicating 500 new tasks and each task is executed one after the other.

F. Results

The below two images are the screen shots of jobs 1 to 20 and also the last 20 i.e from 481 to 500.

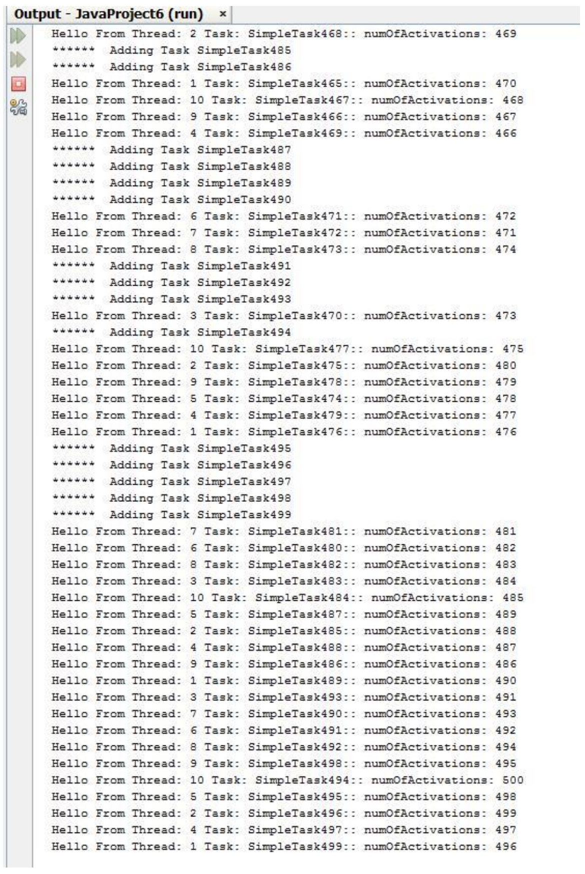
The first image shows that Tasks 1 to 20 are added to the queue and later each task is executed in random. So after Execution of Tasks 4,3,2 and 1 the later tasks of queue are taken and then executed and the process repeats.


Output - JavaProject6 (run)

```
run:
******  Adding Task SimpleTask0
******  Adding Task SimpleTask1
******  Adding Task SimpleTask2
******  Adding Task SimpleTask3
******  Adding Task SimpleTask4
******  Adding Task SimpleTask5
******  Adding Task SimpleTask6
******  Adding Task SimpleTask7
******  Adding Task SimpleTask8
******  Adding Task SimpleTask9
******  Adding Task SimpleTask10
******  Adding Task SimpleTask11
******  Adding Task SimpleTask12
******  Adding Task SimpleTask13
******  Adding Task SimpleTask14
******  Adding Task SimpleTask15
******  Adding Task SimpleTask16
******  Adding Task SimpleTask17
******  Adding Task SimpleTask18
******  Adding Task SimpleTask19
******  Adding Task SimpleTask20
Hello From Thread: 8 Task: SimpleTask4:: numOfActivations: 3
Hello From Thread: 7 Task: SimpleTask3:: numOfActivations: 5
Hello From Thread: 4 Task: SimpleTask2:: numOfActivations: 6
Hello From Thread: 3 Task: SimpleTask1:: numOfActivations: 7
******  Adding Task SimpleTask21
******  Adding Task SimpleTask22
******  Adding Task SimpleTask23
******  Adding Task SimpleTask24
Hello From Thread: 6 Task: SimpleTask8:: numOfActivations: 4
Hello From Thread: 9 Task: SimpleTask9:: numOfActivations: 1
Hello From Thread: 5 Task: SimpleTask7:: numOfActivations: 2
Hello From Thread: 2 Task: SimpleTask6:: numOfActivations: 8
Hello From Thread: 1 Task: SimpleTask5:: numOfActivations: 9
Hello From Thread: 10 Task: SimpleTask0:: numOfActivations: 10
******  Adding Task SimpleTask25
```

Fig 5: Adding Task-1 to Task-20

The below image clearly shows that all the last 20 Tasks are executed and now the queue is empty. Running the code will give an better idea of our work.


Output - JavaProject6 (run)

```
Hello From Thread: 2 Task: SimpleTask468:: numOfActivations: 469
******  Adding Task SimpleTask485
******  Adding Task SimpleTask486
Hello From Thread: 1 Task: SimpleTask465:: numOfActivations: 470
Hello From Thread: 10 Task: SimpleTask467:: numOfActivations: 468
Hello From Thread: 9 Task: SimpleTask466:: numOfActivations: 467
Hello From Thread: 4 Task: SimpleTask469:: numOfActivations: 466
******  Adding Task SimpleTask487
******  Adding Task SimpleTask488
******  Adding Task SimpleTask489
******  Adding Task SimpleTask490
Hello From Thread: 6 Task: SimpleTask471:: numOfActivations: 472
Hello From Thread: 7 Task: SimpleTask472:: numOfActivations: 471
Hello From Thread: 8 Task: SimpleTask473:: numOfActivations: 474
******  Adding Task SimpleTask491
******  Adding Task SimpleTask492
******  Adding Task SimpleTask493
Hello From Thread: 3 Task: SimpleTask470:: numOfActivations: 473
******  Adding Task SimpleTask494
Hello From Thread: 10 Task: SimpleTask477:: numOfActivations: 475
Hello From Thread: 2 Task: SimpleTask475:: numOfActivations: 480
Hello From Thread: 9 Task: SimpleTask478:: numOfActivations: 479
Hello From Thread: 5 Task: SimpleTask474:: numOfActivations: 478
Hello From Thread: 4 Task: SimpleTask479:: numOfActivations: 477
Hello From Thread: 1 Task: SimpleTask476:: numOfActivations: 476
******  Adding Task SimpleTask495
******  Adding Task SimpleTask496
******  Adding Task SimpleTask497
******  Adding Task SimpleTask498
******  Adding Task SimpleTask499
Hello From Thread: 7 Task: SimpleTask481:: numOfActivations: 481
Hello From Thread: 6 Task: SimpleTask480:: numOfActivations: 482
Hello From Thread: 8 Task: SimpleTask482:: numOfActivations: 483
Hello From Thread: 3 Task: SimpleTask483:: numOfActivations: 484
Hello From Thread: 10 Task: SimpleTask484:: numOfActivations: 485
Hello From Thread: 5 Task: SimpleTask487:: numOfActivations: 489
Hello From Thread: 2 Task: SimpleTask485:: numOfActivations: 488
Hello From Thread: 4 Task: SimpleTask488:: numOfActivations: 487
Hello From Thread: 9 Task: SimpleTask486:: numOfActivations: 486
Hello From Thread: 1 Task: SimpleTask489:: numOfActivations: 490
Hello From Thread: 3 Task: SimpleTask493:: numOfActivations: 491
Hello From Thread: 7 Task: SimpleTask490:: numOfActivations: 493
Hello From Thread: 6 Task: SimpleTask491:: numOfActivations: 492
Hello From Thread: 8 Task: SimpleTask492:: numOfActivations: 494
Hello From Thread: 9 Task: SimpleTask498:: numOfActivations: 495
Hello From Thread: 10 Task: SimpleTask494:: numOfActivations: 500
Hello From Thread: 5 Task: SimpleTask495:: numOfActivations: 498
Hello From Thread: 2 Task: SimpleTask496:: numOfActivations: 499
Hello From Thread: 4 Task: SimpleTask497:: numOfActivations: 497
Hello From Thread: 1 Task: SimpleTask499:: numOfActivations: 496
```

Fig 6: Executing Task-481 to Task-500

VI. CONCLUSION

Implementing thread pool gives a solution to both the problem of thread life-cycle overhead and the problem of resource thrashing. By reusing threads for multiple tasks, the thread-creation overhead is spread over many tasks and thus minimized. Thread already exists when a request arrives so the delay introduced by thread creation is eliminated. Thus, the request is serviced mostly immediate and rendering the application more responsive. By properly tuning the number of threads in the thread pool we can prevent resource thrashing by forcing any requests in excess of a certain threshold to wait until a thread is available to process it.By using thread safe implementations of FIFO we avoid the problems occurred in multi threaded systems

VII. FUTURE WORK

Futher investigation on the methods to improve the speed of concurrent execution of tasks might be preferred. Although there are many algorithms which works the best on the parallel task execution, implementing the one which is least prone to deadlocks and recourse thrashing is challenging till now. So developing an architecture from the scratch which utilizes parallel core CPU utilization and which deals with all the above mentioned problems should be the future task.

REFERENCES

[1] A Task Tree Executor: New Runtime for Parallelized Legacy Software By Miroslav Popovic1, Ilija Basicevic1, and Vladislav Vrtunski2 1 University of Novi Sad, Department for Computing Automation, 2 DMS Grupa, Ltd, email: vrtunski@neobee.net

[2] A Task Tree Executor Architecture Based on Intel Threading Building Blocks Miroslav Popovic, Miodrag Djukic, Vladimir Marinkovic

[3] Algorithm for scheduling of Tasks on Moving Executors with uncertain processing times Institute of Control and Systems Engineering, St. 11/17, 50-370 Wroclaw, Poland

[4] M. Popovic, I. Basicevic, and V. Vrtunski, A Task Tree Executor: New Runtime for Parallelized Legacy Software, 16th IEEE International Conference

[5] https://en.wikipedia.org/wiki/Threadpool

[6] https://msdn.microsoft.com/en-us/library/windows/desktop/ms686760 (v=vs.85).aspx

[7] http://www.ibm.com/developerworks/library/j-jtp0730/

[8] http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executor.html