Saxon - Bug #3958

Scheduling algorithm for xsl:for-each/@saxon:threads=N

2018-10-09 19:53 - Michael Kay

Status:	Closed	**Start date:**	2018-10-09
Priority:	Low	**Due date:**	
Assignee:	Michael Kay	**% Done:**	100%
Category:	Multithreading	**Estimated time:**	0:00 hour
Sprint/Milestone:		**Spent time:**	0:00 hour
Legacy ID:		**Fix Committed on Branch:**	9.9, trunk
Applies to branch:	9.9, trunk	**Fixed in Maintenance Release:**	9.9.1.3

Description

The scheduling algorithm used by xsl:for-each/@saxon:threads="N" is not especially smart. In particular, it gives poor throughput when the amount of work required to process different items in the input sequence is highly variable. The following test case illustrates the problem:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet exclude-result-prefixes="#all" version="3.0" xmlns:saxon="http://saxon.sf.net/"
 xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

 <xsl:output method="text" />

 <xsl:template name="main">
  <xsl:for-each saxon:threads="4" select="1 to 10">
   <xsl:choose>
    <xsl:when test=". eq 1">
     <!-- Will take 10 seconds -->
     <xsl:sequence select="json-doc('https://httpbin.org/delay/10')?url" />
    </xsl:when>
    <xsl:when test=". eq 2">
     <!-- Will take 9 seconds -->
     <xsl:sequence select="json-doc('https://httpbin.org/delay/9')?url" />
    </xsl:when>
    <xsl:when test=". eq 3">
     <!-- Will take 8 seconds -->
     <xsl:sequence select="json-doc('https://httpbin.org/delay/8')?url" />
    </xsl:when>
   </xsl:choose>
  </xsl:for-each>
  <xsl:text>&#x0A;</xsl:text>
 </xsl:template>
</xsl:stylesheet>
```

Ideally the elapsed time for this workload should be hardly longer than the longest processing time for one item, that is, 10 seconds. In practice it is about 30 seconds, because the threads are not well utilised. The problem can be solved by allocating more threads, but we should be able to do better.

History

#1 - 2018-10-11 17:24 - Vladimir Nesterovsky

Not sure what implementation technique you're using to run in parallel.

Following may give you a hint.

```
/*
 * Creates a thread pool that maintains enough threads to support
 * the given parallelism level, and may use multiple queues to
 * reduce contention. The parallelism level corresponds to the
 * maximum number of threads actively engaged in, or available to
 * engage in, task processing. The actual number of threads may
```

```
 * grow and shrink dynamically. A work-stealing pool makes no
 * guarantees about the order in which submitted tasks are
 * executed.
 */
java.util.concurrent.Executors.newWorkStealingPool()
```

#2 - 2018-10-12 17:39 - Michael Kay

We're currently using Executors.newFixedThreadPool. Switching to newWorkStealingPool() would be very easy; it's measuring the effect across a range of workloads that's more difficult.

In 9.9 it's actually pluggable so you can change what kind of ExecutorService is used by registering your own MultithreadingFactory with the Configuration.

#3 - 2018-10-12 17:45 - Michael Kay

While I'm here, I won't bother raising a new bug for this, but ThreadManagerEE casts the ExecutorService returned by the MultithreadingFactory to a ThreadPoolExecutor which is unsafe in the case where there's a user-supplied factory class. It should protect this cast with an "instance of" test. Committing a patch for this.

#4 - 2019-03-21 00:18 - Michael Kay

- Status changed from New to In Progress

I have not been able to reproduce the 30second run-time; the test is taking just over 10 seconds, and this is true whether I use newFixedThreadPool() or newWorkStealingPool(). As a result, I don't think it is appropriate to make any change.

The way the MultithreadingFactory works could do with tidying up. The makeMultithreadedItemMappingIterator() method seems to create an ExecutorService without using the factory's makeExecutorService() method.

#5 - 2019-03-21 00:46 - T Hata

Here's my initial repro (when tests are different):

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet exclude-result-prefixes="#all" version="3.0" xmlns:saxon="http://saxon.sf.net/"
        xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

        <xsl:output method="text" />

        <xsl:template name="main">
                <xsl:for-each saxon:threads="4" select="1 to 10">
                        <xsl:choose>
                                <xsl:when test=". eq 1">
                                        <!-- Will take 10 seconds -->
                                        <xsl:sequence select="json-doc('https://httpbin.org/delay/10')?url" />
                                </xsl:when>
                                <xsl:when test=". eq 5">
                                        <!-- Will take 9 seconds -->
                                        <xsl:sequence select="json-doc('https://httpbin.org/delay/9')?url" />
                                </xsl:when>
                                <xsl:when test=". eq 10">
                                        <!-- Will take 8 seconds -->
                                        <xsl:sequence select="json-doc('https://httpbin.org/delay/8')?url" />
                                </xsl:when>
                        </xsl:choose>
                </xsl:for-each>
                <xsl:text>&#x0A;</xsl:text>
        </xsl:template>
</xsl:stylesheet>
```

9.9.1.2 takes 30 seconds.

#6 - 2019-03-21 12:37 - Michael Kay

Thanks. What platform are you running on?

#7 - 2019-03-21 13:11 - T Hata

On (Windows 10 + Java 8) and (Ubuntu 18 + Java 10).

#8 - 2019-03-21 14:04 - Michael Kay

Looking more carefully at the logic of the MultithreadedContextMappingIterator, it's clear that this has nothing to do with the nature of the

ExecutorPool and depends entirely on the structure of the BlockingQueue that we maintain. We prime the queue with (in this case) requests, then issue blockingQueue.poll(), followed by future.get() which blocks; so processing of the 5th item does not start until processing of the 1st item has finished. The main reason for this is that it's the simplest way to ensure that the result items are delivered in the correct order.

#9 - 2019-03-21 14:46 - Vladimir Nesterovsky

It might be worth to add attribute or in some other way to tell to processer, like with fn:unordered(), that order of result is not important. This would give an optimization freedom.

#10 - 2019-03-21 16:18 - Michael Kay

I can't help feeling that the answer to this must lie in using the Streams machinery, and Spliterators in particular. I've spent another hour or so reading all about Spliterators, and I have to confess I really don't understand the paradigm. If someone can enlighten me, please go ahead...

#11 - 2019-03-21 17:19 - Michael Kay

We can get the elapsed time for this task down to a little over 10 seconds simply by increasing the number of items we place on the queue when priming it. Currently if the number of threads is 4, then we put the first 4 items on the queue for processing, and then queue another item for processing only when something has finished. I think this means we are essentially processing a batch of 4 items in parallel, then another batch of 4, then the next batch, and so on.

I've changed the test (a) to show a message each time round the loop, and (b) to process 100 items with a random delay between 1 and 10 seconds. With the existing code, it's clearly that it's operating in "bursts" of 4 items, and takes a total of 199 seconds to finish. If I prime the queue with 20 items, the processing is far less bursty, and now takes 147s. In fact, priming with 8 items is enough to get it down to 145s; the optimum number is clearly going to depend on the randomness of the distribution of service times.

After some further experiments, I've decided to prime the queue with 3*threads items.

#12 - 2019-03-21 17:26 - Michael Kay

- Status changed from In Progress to Resolved

- Applies to branch 9.9, trunk added

- Fix Committed on Branch 9.9, trunk added

Fixed as described.

#13 - 2019-03-22 15:21 - Vladimir Nesterovsky

I think streaming example may give the good picture:

```
import java.util.stream.IntStream;
import java.util.stream.Stream;
import java.util.function.Consumer;
import java.util.function.Function;

public class Streams
{
  public static class Item<T>
  {
    public Item(int index, T data)
    {
      this.index = index;
      this.data = data;
    }

    int index;
    T data;
  }

  public static void main(String[] args)
  {
    run(
      "Sequential",
      input(),
      Streams::action,
      Streams::output,
      true);

    run(
      "Parallel ordered",
      input().parallel(),
      Streams::action,
      Streams::output,
      true);
```

```
    run(
      "Parallel unordered",
      input().parallel(),
      Streams::action,
      Streams::output,
      false);
  }

  private static void run(
    String description,
    Stream<Item<String>> input,
    Function<Item<String>, String[]> action,
    Consumer<String[]> output,
    boolean ordered)
  {
    System.out.println(description);

    Consumer<Item<String>> process = item -> output.accept(action.apply(item));

    long start = System.currentTimeMillis();

    if (ordered)
    {
      input.forEachOrdered(process);
    }
    else
    {
      input.unordered().forEach(process);
    }

    long end = System.currentTimeMillis();

    System.out.println("Execution time: " + (end - start) + "ms.");
    System.out.println();
  }

  private static Stream<Item<String>> input()
  {
    return IntStream.range(0, 10).
      mapToObj(i -> new Item<String>(i + 1, "Data " + (i + 1)));
  }

  private static String[] action(Item<String> item)
  {
    switch(item.index)
    {
      case 1:
      {
        sleep(10);

        break;
      }
      case 5:
      {
        sleep(9);

        break;
      }
      case 10:
      {
        sleep(8);

        break;
      }
    }

    String[] result = { "data:", item.data, "index:", item.index + "" };

    return result;
  }

  private synchronized static void output(String[] value)
  {
    boolean first = true;
```

```
    for(String item: value)
    {
      if (first)
      {
        first = false;
      }
      else
      {
        System.out.print(' ');
      }

      System.out.print(item);
    }

    System.out.println();
  }

  private static void sleep(int seconds)
  {
    try
    {
      Thread.sleep(seconds * 1000);
    }
    catch(InterruptedException e)
    {
      throw new IllegalStateException(e);
    }
  }
}
```

I have verified that with "Sequential" only one thread is used, with "Parallel ordered" multiple thread from common pool are used though it's slower than Sequential, with "Parallel unordered" multiple thread from common pool as used and performance is fastest.

#14 - 2019-03-23 22:33 - Vladimir Nesterovsky

Sorry for repeating mostly the same sample but here I have figured out how to make parallel ordered to be faster than sequential:

```
import java.util.stream.IntStream;
import java.util.stream.Stream;
import java.util.function.Consumer;
import java.util.function.Function;

public class Streams
{
  public static class Item<T>
  {
    public Item(int index, T data)
    {
      this.index = index;
      this.data = data;
    }

    int index;
    T data;
  }

  public static void main(String[] args)
  {
    run(
      "Sequential",
      input(),
      Streams::action,
      Streams::output,
      true);

    run(
      "Parallel ordered",
      input().parallel(),
      Streams::action,
      Streams::output,
      true);

    run(
```

```
      "Parallel unordered",
      input().parallel(),
      Streams::action,
      Streams::output,
      false);
  }

  private static void run(
    String description,
    Stream<Item<String>> input,
    Function<Item<String>, String[]> action,
    Consumer<String[]> output,
    boolean ordered)
  {
    System.out.println(description);

    long start = System.currentTimeMillis();

    if (ordered)
    {
      input.map(action).forEachOrdered(output);
    }
    else
    {
      input.map(action).forEach(output);
    }

    long end = System.currentTimeMillis();

    System.out.println("Execution time: " + (end - start) + "ms.");
    System.out.println();
  }

  private static Stream<Item<String>> input()
  {
    return IntStream.range(0, 10).
      mapToObj(i -> new Item<String>(i + 1, "Data " + (i + 1)));
  }

  private static String[] action(Item<String> item)
  {
    switch(item.index)
    {
      case 1:
      {
        sleep(10);

        break;
      }
      case 5:
      {
        sleep(9);

        break;
      }
      case 10:
      {
        sleep(8);

        break;
      }
//      default:
//      {
//        sleep(1);
//
//        break;
//      }
    }

    String[] result = { "data:", item.data, "index:", item.index + "" };

    return result;
  }

  private synchronized static void output(String[] value)
```

```
  {
    boolean first = true;

    for(String item: value)
    {
      if (first)
      {
        first = false;
      }
      else
      {
        System.out.print(' ');
      }

      System.out.print(item);
    }

    System.out.println();
  }

  private static void sleep(int seconds)
  {
    try
    {
      Thread.sleep(seconds * 1000);
    }
    catch(InterruptedException e)
    {
      throw new IllegalStateException(e);
    }
  }
}
```

Now we can see "Sequential" is slowest, "Parallel ordered" is much faster, but "Parallel unordered" is even faster as it was able to spill fast results earlier.

#15 - 2019-05-15 12:40 - O'Neil Delpratt

- Status changed from Resolved to Closed

- % Done changed from 0 to 100

- Fixed in Maintenance Release 9.9.1.3 added

Bug fix applied to the Saxon 9.9.1.3 maintenance release