Table of Contents

1. Scaling and Parallel Processing . . . 1
1.1. Multi-threaded Step . . . 1
1.2. Parallel Steps . . . 3
1.3. Remote Chunking . . . 5
1.4. Partitioning . . . 6
1.4.1. PartitionHandler . . . 8
1.4.2. Partitioner . . . 9
1.4.3. Binding Input Data to Steps . . . 10

Chapter 1. Scaling and Parallel Processing

Many batch processing problems can be solved with single threaded, single process jobs, so it is always a good idea to properly check if that meets your needs before thinking about more complex implementations. Measure the performance of a realistic job and see if the simplest implementation meets your needs first. You can read and write a file of several hundred megabytes in well under a minute, even with standard hardware.

When you are ready to start implementing a job with some parallel processing, Spring Batch offers a range of options, which are described in this chapter, although some features are covered elsewhere. At a high level, there are two modes of parallel processing:

- Single process, multi-threaded
- Multi-process

These break down into categories as well, as follows:

- Multi-threaded Step (single process)
- Parallel Steps (single process)
- Remote Chunking of Step (multi process)
- Partitioning a Step (single or multi process)

First, we review the single-process options. Then we review the multi-process options.

1.1. Multi-threaded Step

The simplest way to start parallel processing is to add a `TaskExecutor` to your Step configuration.

For example, you might add an attribute of the `tasklet`, as shown in the following example:

```
<step id="loading">
    <tasklet task-executor="taskExecutor">...</tasklet>
</step>
```

When using java configuration, a `TaskExecutor` can be added to the step as shown in the following example:

Java Configuration

```
@Bean
public TaskExecutor taskExecutor(){
    return new SimpleAsyncTaskExecutor("spring_batch");
}

@Bean
public Step sampleStep(TaskExecutor taskExecutor) {
    return this.stepBuilderFactory.get("sampleStep")
                .<String, String>chunk(10)
                .reader(itemReader())
                .writer(itemWriter())
                .taskExecutor(taskExecutor)
                .build();
}
```

In this example, the `taskExecutor` is a reference to another bean definition that implements the `TaskExecutor` interface. `TaskExecutor` is a standard Spring interface, so consult the Spring User Guide for details of available implementations. The simplest multi-threaded `TaskExecutor` is a `SimpleAsyncTaskExecutor`.

The result of the above configuration is that the `Step` executes by reading, processing, and writing each chunk of items (each commit interval) in a separate thread of execution. Note that this means there is no fixed order for the items to be processed, and a chunk might contain items that are non-consecutive compared to the single-threaded case. In addition to any limits placed by the task executor (such as whether it is backed by a thread pool), there is a throttle limit in the tasklet configuration which defaults to 4. You may need to increase this to ensure that a thread pool is fully utilized.

For example you might increase the throttle-limit, as shown in the following example:

```
<step id="loading"> <tasklet
    task-executor="taskExecutor"
    throttle-limit="20">...</tasklet>
</step>
```

When using java configuration, the builders provide access to the throttle limit:

Java Configuration

```
@Bean
public Step sampleStep(TaskExecutor taskExecutor) {
    return this.stepBuilderFactory.get("sampleStep")
                .<String, String>chunk(10)
                .reader(itemReader())
                .writer(itemWriter())
                .taskExecutor(taskExecutor)
                .throttleLimit(20)
                .build();
}
```

Note also that there may be limits placed on concurrency by any pooled resources used in your step, such as a `DataSource`. Be sure to make the pool in those resources at least as large as the desired number of concurrent threads in the step.

There are some practical limitations of using multi-threaded `Step` implementations for some common batch use cases. Many participants in a `Step` (such as readers and writers) are stateful. If the state is not segregated by thread, then those components are not usable in a multi-threaded `Step`. In particular, most of the off-the-shelf readers and writers from Spring Batch are not designed for multi-threaded use. It is, however, possible to work with stateless or thread safe readers and writers, and there is a sample (called `parallelJob`) in the Spring Batch Samples that shows the use of a process indicator (see Preventing State Persistence) to keep track of items that have been processed in a database input table.

Spring Batch provides some implementations of `ItemWriter` and `ItemReader`. Usually, they say in the Javadoc if they are thread safe or not or what you have to do to avoid problems in a concurrent environment. If there is no information in the Javadoc, you can check the implementation to see if there is any state. If a reader is not thread safe, you can decorate it with the provided `SynchronizedItemStreamReader` or use it in your own synchronizing delegator. You can synchronize the call to `read()` and as long as the processing and writing is the most expensive part of the chunk, your step may still complete much faster than it would in a single threaded configuration.

1.2. Parallel Steps

As long as the application logic that needs to be parallelized can be split into distinct responsibilities and assigned to individual steps, then it can be parallelized in a single process. Parallel Step execution is easy to configure and use.

For example, executing steps `(step1,step2)` in parallel with `step3` is straightforward, as shown in the following example:

```
<job id="job1">
    <split id="split1" task-executor="taskExecutor" next="step4">
        <flow>
            <step id="step1" parent="s1" next="step2"/>
            <step id="step2" parent="s2"/>
        </flow>
        <flow>
            <step id="step3" parent="s3"/>
        </flow>
    </split>
    <step id="step4" parent="s4"/>
</job>

<beans:bean id="taskExecutor" class="org.spr...SimpleAsyncTaskExecutor"/>
```

When using java configuration, executing steps `(step1,step2)` in parallel with `step3` is straightforward, as shown in the following example:

Java Configuration

```
@Bean
public Job job() {
    return jobBuilderFactory.get("job")
        .start(splitFlow())
        .next(step4())
        .build()        //builds FlowJobBuilder instance
        .build();       //builds Job instance
}

@Bean
public Flow splitFlow() {
    return new FlowBuilder<SimpleFlow>("splitFlow")
        .split(taskExecutor())
        .add(flow1(), flow2())
        .build();
}

@Bean
public Flow flow1() {
    return new FlowBuilder<SimpleFlow>("flow1")
        .start(step1())
        .next(step2())
        .build();
}

@Bean
public Flow flow2() {
    return new FlowBuilder<SimpleFlow>("flow2")
        .start(step3())
        .build();
}

@Bean
public TaskExecutor taskExecutor(){
    return new SimpleAsyncTaskExecutor("spring_batch");
}
```

The configurable task executor is used to specify which `TaskExecutor` implementation should be used to execute the individual flows. The default is `SyncTaskExecutor`, but an asynchronous `TaskExecutor` is required to run the steps in parallel. Note that the job ensures that every flow in the split completes before aggregating the exit statuses and transitioning.

See the section on Split Flows for more detail.

1.3. Remote Chunking

In remote chunking, the `Step` processing is split across multiple processes, communicating with each other through some middleware. The following image shows the pattern:

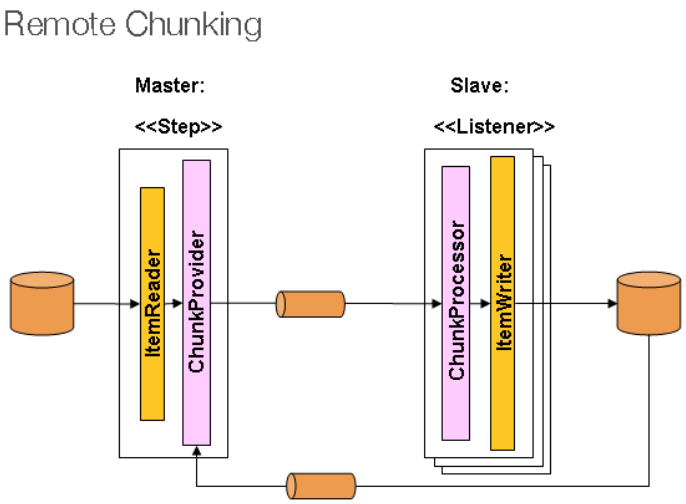

Figure 1. Remote Chunking

The master component is a single process, and the slaves are multiple remote processes. This pattern works best if the master is not a bottleneck, so the processing must be more expensive than the reading of items (as is often the case in practice).

The master is an implementation of a Spring Batch `Step` with the `ItemWriter` replaced by a generic version that knows how to send chunks of items to the middleware as messages. The slaves are standard listeners for whatever middleware is being used (for example, with JMS, they would be `MesssageListener` implementations), and their role is to process the chunks of items using a standard `ItemWriter` or `ItemProcessor` plus `ItemWriter`, through the `ChunkProcessor` interface. One of the advantages of using this pattern is that the reader, processor, and writer components are off-the-shelf (the same as would be used for a local execution of the step). The items are divided up dynamically and work is shared through the middleware, so that, if the listeners are all eager consumers, then load balancing is automatic.

The middleware has to be durable, with guaranteed delivery and a single consumer for each message. JMS is the obvious candidate, but other options (such as JavaSpaces) exist in the grid computing and shared memory product space.

See the section on Spring Batch Integration - Remote Chunking for more detail.

1.4. Partitioning

Spring Batch also provides an SPI for partitioning a `Step` execution and executing it remotely. In this case, the remote participants are `Step` instances that could just as easily have been configured and used for local processing. The following image shows the pattern:

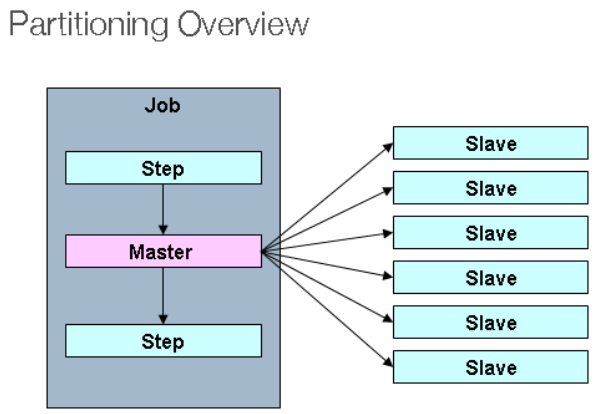

Figure 2. Partitioning

The `Job` runs on the left-hand side as a sequence of `Step` instances, and one of the `Step` instances is labeled as a master. The slaves in this picture are all identical instances of a `Step`, which could in fact take the place of the master, resulting in the same outcome for the `Job`. The slaves are typically going to be remote services but could also be local threads of execution. The messages sent by the master to the slaves in this pattern do not need to be durable or have guaranteed delivery. Spring Batch metadata in the `JobRepository` ensures that each slave is executed once and only once for each `Job` execution.

The SPI in Spring Batch consists of a special implementation of `Step` (called the `PartitionStep`) and two strategy interfaces that need to be implemented for the specific environment. The strategy interfaces are `PartitionHandler` and `StepExecutionSplitter`, and their role is shown in the following sequence diagram:

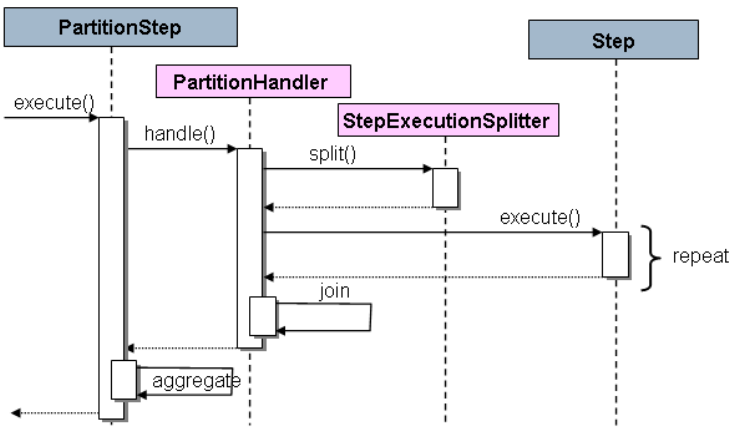

Figure 3. Partitioning SPI

The `Step` on the right in this case is the "remote" slave, so, potentially, there are many objects and or processes playing this role, and the `PartitionStep` is shown driving the execution.

The following example shows the PartitionStep configuration:

```
<step id="step1.master">
    <partition step="step1" partitioner="partitioner">
        <handler grid-size="10" task-executor="taskExecutor"/>
    </partition>
</step>
```

The following example shows the PartitionStep configuration using java configuration:

Java Configuration

```
@Bean
public Step step1Master() {
    return stepBuilderFactory.get("step1.master")
        .<String, String>partitioner("step1", partitioner())
        .step(step1())
        .gridSize(10)
        .taskExecutor(taskExecutor())
        .build();
}
```

Similar to the multi-threaded step's throttle-limit attribute, the grid-size attribute prevents the task executor from being saturated with requests from a single step.

There is a simple example that can be copied and extended in the unit test suite for Spring Batch Samples (see Partition*Job.xml configuration).

Spring Batch creates step executions for the partitions called "step1:partition0", and so on. Many people prefer to call the master step "step1:master" for consistency. You can use an alias for the step (by specifying the name attribute instead of the id attribute).

1.4.1. PartitionHandler

The PartitionHandler is the component that knows about the fabric of the remoting or grid environment. It is able to send StepExecution requests to the remote Step instances, wrapped in some fabric-specific format, like a DTO. It does not have to know how to split the input data or how to aggregate the result of multiple Step executions. Generally speaking, it probably also does not need to know about resilience or failover, since those are features of the fabric in many cases. In any case, Spring Batch always provides restartability independent of the fabric. A failed Job can always be restarted and only the failed Steps are re-executed.

The PartitionHandler interface can have specialized implementations for a variety of fabric types, including simple RMI remoting, EJB remoting, custom web service, JMS, Java Spaces, shared memory grids (like Terracotta or Coherence), and grid execution fabrics (like GridGain). Spring Batch does not contain implementations for any proprietary grid or remoting fabrics.

Spring Batch does, however, provide a useful implementation of PartitionHandler that executes Step instances locally in separate threads of execution, using the TaskExecutor strategy from Spring.

The implementation is called `TaskExecutorPartitionHandler`.

The `TaskExecutorPartitionHandler` is the default for a step configured with the XML namespace shown previously. It can also be configured explicitly, as shown in the following example:

```
<step id="step1.master">
    <partition step="step1" handler="handler"/>
</step>

<bean class="org.spr...TaskExecutorPartitionHandler">
    <property name="taskExecutor" ref="taskExecutor"/>
    <property name="step" ref="step1" />
    <property name="gridSize" value="10" />
</bean>
```

The `TaskExecutorPartitionHandler` can be configured explicitly within java configuration, as shown in the following example:

Java Configuration

```
@Bean
public Step step1Master() {
    return stepBuilderFactory.get("step1.master")
        .partitioner("step1", partitioner())
        .partitionHandler(partitionHandler())
        .build();
}

@Bean
public PartitionHandler partitionHandler() {
    TaskExecutorPartitionHandler retVal = new TaskExecutorPartitionHandler();
    retVal.setTaskExecutor(taskExecutor());
    retVal.setStep(step1());
    retVal.setGridSize(10);
    return retVal;
}
```

The `gridSize` attribute determines the number of separate step executions to create, so it can be matched to the size of the thread pool in the `TaskExecutor`. Alternatively, it can be set to be larger than the number of threads available, which makes the blocks of work smaller.

The `TaskExecutorPartitionHandler` is useful for IO-intensive `Step` instances, such as copying large numbers of files or replicating filesystems into content management systems. It can also be used for remote execution by providing a `Step` implementation that is a proxy for a remote invocation (such as using Spring Remoting).

1.4.2. Partitioner

The `Partitioner` has a simpler responsibility: to generate execution contexts as input parameters

for new step executions only (no need to worry about restarts). It has a single method, as shown in the following interface definition:

```
public interface Partitioner {
    Map<String, ExecutionContext> partition(int gridSize);
}
```

The return value from this method associates a unique name for each step execution (the `String`) with input parameters in the form of an `ExecutionContext`. The names show up later in the Batch metadata as the step name in the partitioned `StepExecutions`. The `ExecutionContext` is just a bag of name-value pairs, so it might contain a range of primary keys, line numbers, or the location of an input file. The remote `Step` then normally binds to the context input using `#{…}` placeholders (late binding in step scope), as illustrated in the next section.

The names of the step executions (the keys in the `Map` returned by `Partitioner`) need to be unique amongst the step executions of a `Job` but do not have any other specific requirements. The easiest way to do this (and to make the names meaningful for users) is to use a prefix+suffix naming convention, where the prefix is the name of the step that is being executed (which itself is unique in the `Job`), and the suffix is just a counter. There is a `SimplePartitioner` in the framework that uses this convention.

An optional interface called `PartitionNameProvider` can be used to provide the partition names separately from the partitions themselves. If a `Partitioner` implements this interface, then, on a restart, only the names are queried. If partitioning is expensive, this can be a useful optimization. The names provided by the `PartitionNameProvider` must match those provided by the `Partitioner`.

1.4.3. Binding Input Data to Steps

It is very efficient for the steps that are executed by the `PartitionHandler` to have identical configuration and for their input parameters to be bound at runtime from the `ExecutionContext`. This is easy to do with the StepScope feature of Spring Batch (covered in more detail in the section on Late Binding). For example, if the `Partitioner` creates `ExecutionContext` instances with an attribute key called `fileName`, pointing to a different file (or directory) for each step invocation, the `Partitioner` output might resemble the content of the following table:

Table 1. Example step execution name to execution context provided by `Partitioner` targeting directory processing

Step Execution Name (key)	*ExecutionContext (value)*
filecopy:partition0	fileName=/home/data/one
filecopy:partition1	fileName=/home/data/two
filecopy:partition2	fileName=/home/data/three

Then the file name can be bound to a step using late binding to the execution context, as shown in the following example:

XML Configuration

```
<bean id="itemReader" scope="step"
      class="org.spr...MultiResourceItemReader">
    <property name="resources" value="#{stepExecutionContext[fileName]}/*"/>
</bean>
```

Java Configuration

```
@Bean
public MultiResourceItemReader itemReader(
    @Value("#{stepExecutionContext['fileName']}/*") Resource [] resources) {
    return new MultiResourceItemReaderBuilder<String>()
            .delegate(fileReader())
            .name("itemReader")
            .resources(resources)
            .build();
}
```