

# ProActive Parallel Suite

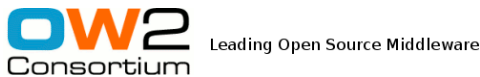


An Open Source Middleware For Parallel, Distributed, Multicore Computing

## ProActive MapReduce

Version 3.1.0

The OASIS Research Team and ActiveEon Company





# ProActive Scheduling v3.1.0 Documentation

## Legal Notice

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Affero General Public License as published by the Free Software Foundation; version 3 of the License.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Affero General Public License for more details.

You should have received a copy of the GNU Affero General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

If needed, contact us to obtain a release under GPL Version 2 or 3 or a different license than the AGPL.

Contact: [proactive@ow2.org](mailto:proactive@ow2.org) or [contact@activeeon.com](mailto:contact@activeeon.com)

Copyright 1997-2011 INRIA/University of Nice-Sophia Antipolis/ActiveEon.

## Mailing List

[proactive@ow2.org](mailto:proactive@ow2.org)

## Mailing List Archive

<http://www.objectweb.org/wws/arc/proactive>

## Bug-Traking System

<http://bugs.activeeon.com/browse/PROACTIVE>

---

# Contributors and Contact Information

---

## Team Leader

Denis Caromel  
INRIA 2004, Route des Lucioles, BP 93  
06902 Sophia Antipolis Cedex  
France  
phone: +33 492 387 631  
fax: +33 492 387 971  
e-mail: [Denis.Caromel@inria.fr](mailto:Denis.Caromel@inria.fr)

---

### Contributors from OASIS Team

Brian Amedro  
Francoise Baude  
Francesco Bongiovanni  
Borelli Elvio  
Yu Feng  
Ludovic Henrio  
Fabrice Huet  
Virginie Legrand Contes  
Eric Madelaine  
Laurent Pellegrino  
Guilherme Peretti-Pezzi  
Franca Perrina  
Marcela Rivera  
Christian Ruz  
Bastien Sauvan  
Mathieu Schnoor  
Doglov Sergei  
Oleg Smirnov  
Marc Valdener  
Fabien Viale

---

### Contributors from ActiveEon Company

Vladimir Bodnartchouk  
Arnaud Contes  
Cédric Dalmasso  
Christian Delbé  
Jean-Michel Guillaume  
Clément Mathieu  
Emil Salageanu  
Jean-Luc Scheefer

---

### Former Important Contributors

Laurent Baduel (Group Communications)  
Vincent Cave (Legacy Wrapping)  
Alexandre di Costanzo (P2P, B&B)  
Abhijeet Gaikwad (Option Pricing)  
Mario Leyton (Skeleton)  
Matthieu Morel (Initial Component Work)  
Romain Quilici  
Germain Sigety (Scheduling)  
Julien Vayssiere (MOP, Active Objects)

# Table of Contents

List of figures .....	iii
List of tables .....	iv
Chapter 1. Introduction to MapReduce .....	1
1.1. ProActive MapReduce .....	3
Chapter 2. Tutorial: running Hadoop job using ProActive MapReduce .....	4
2.1. Hadoop example .....	4
2.2. Paths used in this tutorial .....	5
2.3. Modifications to the Hadoop example .....	5
2.4. Running the exmaple .....	8
Chapter 3. ProActive MapReduce Reference .....	11
3.1. Internal structure of a ProActive MapReduce job .....	11
3.2. ProActive MapReduce configuration .....	12
3.2.1. Required configuration properties .....	12
3.2.2. Optional configuration properties .....	12
3.3. API limitations .....	14
Chapter 4. ProActive MapReduce performance .....	15
4.1. Execution environment .....	15
4.2. Input data .....	15
4.3. ProActive MapReduce configuration .....	15
4.4. Hadoop configuration .....	15
4.5. Results .....	16

## List of Figures

1.1. MapReduce execution .....	1
1.2. Hadoop MapReduce execution .....	2
3.1. ProActive MapReduce workflow .....	11

## List of Tables

4.1. ProActive MapReduce performance .....	16
--	----

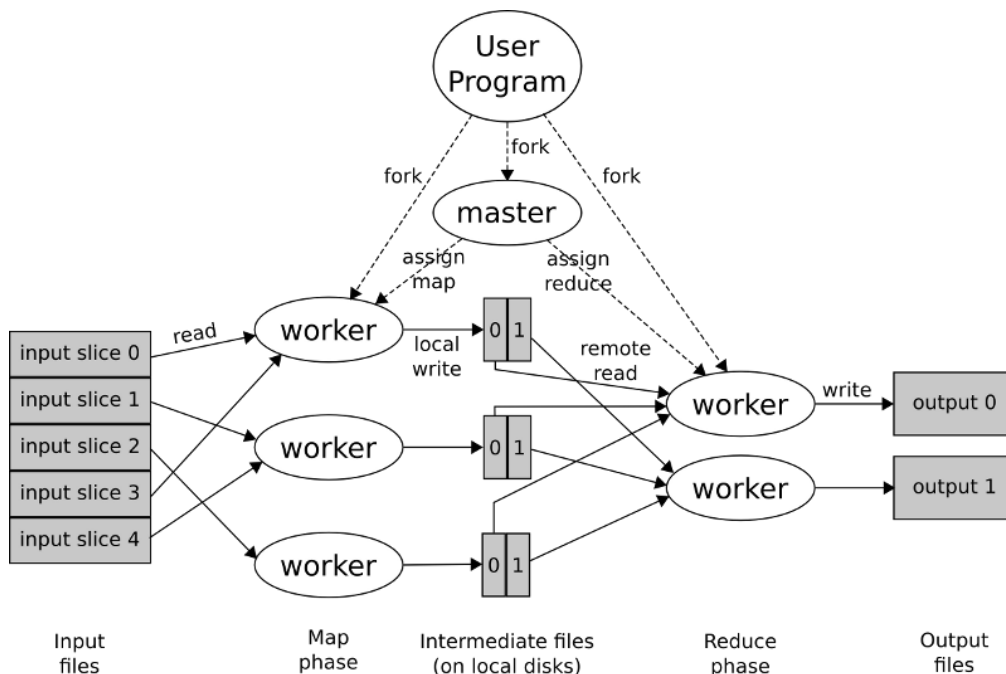
# Chapter 1. Introduction to MapReduce

MapReduce is a parallel programming model that derives from the **map** and **reduce** combinators present in functional languages like Lisp. In Lisp, the **map** takes as input a function and a sequence of values. It then applies the function to each value in the sequence. The **reduce** combines all the elements of a sequence using a binary operation. For example, it can use the sum function to add up all the elements in a sequence. MapReduce is inspired by these concepts.

MapReduce was first introduced by Google® as a framework for processing large amounts of raw data, for example, web pages collected by a web crawler or web request logs, using clusters of commodity hardware. The amount of data is so large that it must be distributed across thousands of machines in order to be processed in a reasonable time. The distribution implies parallel computing since the same computations are performed on each CPU, but with a different dataset.

MapReduce is an abstraction that allows Google engineers to perform simple computations while hiding the details of parallelization, data distribution, load balancing and fault tolerance. MapReduce programs are automatically distributed and executed on a large cluster of commodity machines. The run-time system takes care of the details of splitting the input data, scheduling the program execution across the set of machines, handling machine failures and managing the required inter-machine communication. This allows programmers without any experience in parallel and distributed systems to easily utilize the resources of a large distributed system. The only requirement is that the application should be data parallel in nature. To utilize the MapReduce framework, a programmer must define 2 functions: **map** and **reduce**. Each function has key-value pairs as its input and output. The type of the key and value can be defined by the user. Input data can be in any format as long as it can be loaded into key-value pairs by a user defined function. The map function takes the input key-value pairs and produces a bag of intermediate key-value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key and passes them to the reduce function. The reduce function, also written by the programmer, accepts an intermediate key and the list of values for that key and processes the values sharing the same key.

The [Figure 1.1, “MapReduce execution”](#) shows the execution of the Google MapReduce application.



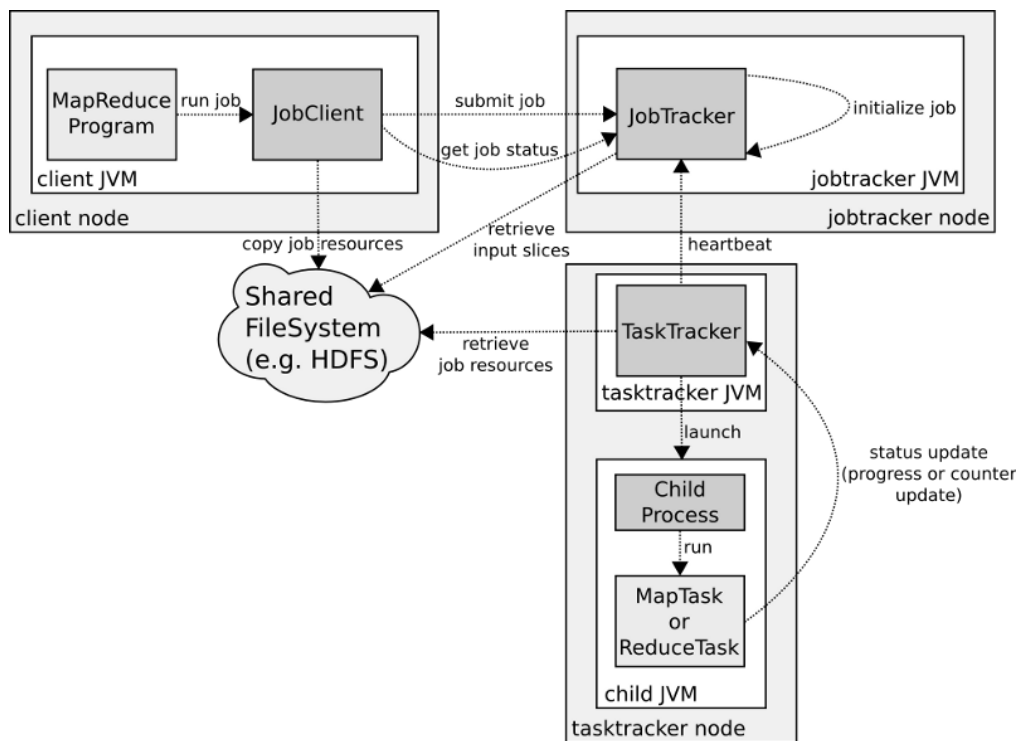
**Figure 1.1. MapReduce execution**

Notice that the reduce phase can begin only after all the map tasks are done and intermediate files are produced. However, this is the only needed synchronization point and the only inter-process communication.

The most important features of the MapReduce parallel programming model are: fault tolerance and data locality. The target architecture of the Google MapReduce is a cluster of thousands of commodity machines (e.g., 2-4 GB of main memory while networking hardware bandwidth is 100 Mbit/s). This means that machine failures are common and re-execution of map and/or reduce tasks is the primary mechanism for fault tolerance. The data to process that is stored on the disks is managed by a distributed file system, GFS® (Google File System), that uses replication to provide availability and reliability on top of unreliable hardware. Moreover, the bandwidth is conserved because GFS file blocks are stored on the local disks of the machines that make up the Google cluster. The processing is data local because each user defined map function reads the data from the replica stored locally on the same machine.

Google MapReduce is a C++ library, but an open source implementation in Java also exists: Apache® Hadoop® MapReduce. Hadoop MapReduce follows the same approach taken by Google: the system is in charge of the communication between machines. The user must only implement the **map** and **reduce** functions and run the job specifying the input data and the output directory for the results. Like Google, Hadoop uses its own distributed file system, HDFS® (Hadoop Distributed File System).

The architecture and the execution of Hadoop MapReduce job is presented in the [Figure 1.2, “Hadoop MapReduce execution”](#).



**Figure 1.2. Hadoop® MapReduce execution**

At the highest level there are four independent entities:

- The **client** which submits the MapReduce job;
- The **JobTracker**, a Java daemon running on a particular Hadoop cluster node. It coordinates the job execution;
- The **TaskTracker**, a java daemon running on each Hadoop cluster node. It runs the tasks (map and/or reduce) the MapReduce job is composed of;
- The shared file system, usually **HDFS** (Hadoop Distributed File System) which is used to store input and output files.

The architecture shown in the [Figure 1.2, “Hadoop MapReduce execution”](#) is quite similar to the one used by the ProActive Scheduler to execute ProActive workflows: the job is submitted to the ProActive Scheduler, the ProActive Scheduler and the ProActive Resource Manager coordinate the execution of the ProActive workflow, the ProActive node runs the tasks belonging to the ProActive workflow. This similarity made the implementation of ProActive MapReduce possible.

## 1.1. ProActive MapReduce

ProActive MapReduce is a framework for execution of MapReduce jobs using the infrastructure provided by ProActive Scheduler and ProActive Resource Manager. The scheduler and resource manager provide fault tolerance, easy deployment and advanced scheduling capabilities not found in Hadoop. There is no distributed file system in ProActive though, so the **DataSpaces** mechanism is used for accessing input and writing output data.

ProActive MapReduce API is **Hadoop-like**. The user can define the ProActive MapReduce workflow as in Hadoop, but, internally, the ProActive MapReduce builds a ProActive job and submits it to ProActive Scheduler. Adapting existing Hadoop jobs to execute them using ProActive MapReduce framework is easy. No changes are required to the Mapper and Reducer classes; and only minimal amount of additional configuration is needed.

# Chapter 2. Tutorial: running Hadoop job using ProActive MapReduce

This tutorial describes the steps necessary to port existing Hadoop job to Proactive MapReduce.

## 2.1. Hadoop example

Let's start with a working Hadoop example (taken from [Hadoop wiki](http://wiki.apache.org/hadoop/WordCount)<sup>1</sup>):

```
import java.io.*;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class WordCount {

    public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
            String line = value.toString();
            StringTokenizer tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
                word.set(tokenizer.nextToken());
                context.write(word, one);
            }
        }
    }

    public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

        public void reduce(Text key, Iterable<IntWritable> values, Context context)
            throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            context.write(key, new IntWritable(sum));
        }
    }

    public static void main(String[] args) throws Exception {
```

<sup>1</sup> <http://wiki.apache.org/hadoop/WordCount>

```
// 1. instantiate Configuration and Job classes
Configuration conf = new Configuration();
Job job = new Job(conf, "wordcount");

// 2. set the types of output keys and values produced by reducer
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);

// 3. tell the job to use the previously defined Map and Reduce classes
job.setMapperClass(Map.class);
job.setReducerClass(Reduce.class);

// 4. specify the formats of job input and output files
job.setInputFormatClass(TextInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);

// 5. specify the paths to input file and output directory; the
// paths here are absolute HDFS paths
FileInputFormat.addInputPath(job, new Path("/data/input/input_file"));
FileOutputFormat.setOutputPath(job, new Path("/data/output/output_dir"));

// 6. submit the job and wait for completion
job.waitForCompletion(true);
}
```

At the beginning of the WordCount class above, the **Map** and **Reduce** classes are defined, with **map** and **reduce** methods that do the actual work. The **main** method contains job creation, configuration and submission code.

## 2.2. Paths used in this tutorial

In the following it is assumed that the environment variables `SCHEDULER_HOME` and `EXAMPLE_DIR` are set. `$SCHEDULER_HOME` should point to the directory where the scheduler is installed, and `$EXAMPLE_DIR` will be used for the code of the example, as well as job input and output files. The assumed values for those variables are listed below, feel free to adjust them to match your setup:

```
SCHEDULER_HOME=/home/user/proactive/scheduler
EXAMPLE_DIR=/home/user/mapreduce_example
```

We'll use variable references in the shell commands, and the corresponding values in java code and configuration files. We'll also use `$EXAMPLE_DIR/data/input/input_file` as an input path, and `$EXAMPLE_DIR/data/output/output_dir` as an output directory.

## 2.3. Modifications to the Hadoop example

Here are the modifications necessary to be able to execute the above example in Proactive MapReduce:

- The **Map** and **Reduce** classes require no modification.
- Steps 1-4 of job creation and configuration from the listing above also require no modification.
- Step 5 requires the following modification: input and output paths should be **relative to the input and output dataspaces**. Assuming we would like to use `$EXAMPLE_DIR/data/input` as an input dataspaces, and `$EXAMPLE_DIR/data/output` as an output dataspaces (we'll configure the dataspaces in a minute), the input and output paths should look like this:

```
FileInputFormat.addInputPath(job, new Path("input_file"));
```

```
FileOutputFormat.setOutputPath(job, new Path("output_dir"));
```

- Apart from that, some ProActive MapReduce-specific configuration is necessary. First, we need to have a configuration file. An example of a configuration file with all possible configuration properties can be found in `$SCHEDULER_HOME/src/scheduler/src/org/ow2/proactive/scheduler/ext/mapreduce/examples/paMapReduceConfigurationProperties.property`. Here we'll use a minimal configuration file sufficient to run the example:

```
# filename: $MAPREDUCE_EXAMPLE/paMapReduceConfigurationProperties.property

# scheduler URL
org.ow2.proactive.scheduler.ext.mapreduce.schedulerUrl=rmi://localhost:55855/

# credentials to use when submitting the job
org.ow2.proactive.scheduler.ext.mapreduce.username=admin
org.ow2.proactive.scheduler.ext.mapreduce.password=admin

# dataspace configuration
org.ow2.proactive.scheduler.ext.mapreduce.workflow.inputSpace=file:///home/user/mapreduce_example/data/input
org.ow2.proactive.scheduler.ext.mapreduce.workflow.outputSpace=file:///home/user/mapreduce_example/data/output

# input split size: the size in bytes of the chunk of data that each
# mapper will get as its input
org.ow2.proactive.scheduler.ext.mapreduce.workflow.splitterPATask.inputSplitSize=10

# additional classpath: here it contains the path to the directory
# where the class files of this example will be put
org.ow2.proactive.scheduler.ext.mapreduce.workflow.classpath=/home/user/mapreduce_example/classes
```

The settings for `inputSpace` and `outputSpace` should contain the URIs supported by the `DataSpaces` mechanism and accessible from all nodes that the job will execute on. In this example, all nodes will be started locally, so we are using the `file://` protocol and the local paths.

The `inputSplitSize` setting determines the size in bytes of the input chunk that each mapper will process, and thus it also determines the total number of mapper tasks. In Hadoop, setting this parameter is usually unnecessary, because by default the split size is equal to the size of an HDFS block, and there is usually no reason to change it. In ProActive MapReduce, any value can be used. Here we set it to a small value, because the size of an input file in this example is going to be small.

Setting the number of reducer tasks is done in exactly the same way as it is done in Hadoop: by calling the `setNumReduceTasks(int)` method of the `Hadoop Job` class. In this example the number of reducers is not defined, so the default value is used: the number of reducer tasks is one.

Note: most of the Proactive MapReduce configuration options can be specified using the Java API instead of configuration files.

- In the Java code, an instance of `org.ow2.proactive.scheduler.ext.mapreduce.PAMapReduceJobConfiguration` must be created, using the above file as an argument:

```
File f = new File("/home/user/mapreduce_example/paMapReduceConfigurationProperties.property");
PAMapReduceJobConfiguration pamrjc = new PAMapReduceJobConfiguration(f);
```

- The next step (performed instead of the step 6 in the original example) is the creation and submission of a `PAMapReduceJob`. `Hadoop Job` instance and `PAMapReduceJobConfiguration` instance are passed as arguments to the constructor:

```
PAMapReduceJob pamrj = null;
try {
    pamrj = new PAMapReduceJob(job, pamrjc);
} catch (PAJobConfigurationException e) {
```

```
e.printStackTrace();
}

if (pamrj.run())
    System.out.println("Submitted " + pamrj.getJobId());
else
    System.out.println("Not submitted");
```

Here is the full result of the modifications described above:

```
// filename: $MAPREDUCE_EXAMPLE/WordCount.java
```

```
import java.io.*;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

import org.ow2.proactive.scheduler.ext.mapreduce.PAMapReduceJob;
import org.ow2.proactive.scheduler.ext.mapreduce.PAMapReduceJobConfiguration;
import org.ow2.proactive.scheduler.ext.mapreduce.exception.PAJobConfigurationException;

public class WordCount {

    public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
            String line = value.toString();
            StringTokenizer tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
                word.set(tokenizer.nextToken());
                context.write(word, one);
            }
        }
    }

    public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

        public void reduce(Text key, Iterable<IntWritable> values, Context context)
            throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            context.write(key, new IntWritable(sum));
        }
    }
}
```

```
}  
  
public static void main(String[] args) throws Exception {  
  
    // 1. instantiate Configuration and Job classes  
    Configuration conf = new Configuration();  
    Job job = new Job(conf, "wordcount");  
  
    // 2. set the types of output keys and values produced by reducer  
    job.setOutputKeyClass(Text.class);  
    job.setOutputValueClass(IntWritable.class);  
  
    // 3. tell the job to use the previously defined Map and Reduce classes  
    job.setMapperClass(Map.class);  
    job.setReducerClass(Reduce.class);  
  
    // 4. specify the formats of job input and output files  
    job.setInputFormatClass(TextInputFormat.class);  
    job.setOutputFormatClass(TextOutputFormat.class);  
  
    // 5. specify the paths to input file and output directory;  
    // the paths here are relative to the dataspace declared in a  
    // configuration file below  
    FileInputFormat.addInputPath(job, new Path("input_file"));  
    FileOutputFormat.setOutputPath(job, new Path("output_dir"));  
  
    // 6. Instantiate PAMapReduceJobConfiguration using a path to  
    // the configuration file  
    File f = new File("/home/user/mapreduce_example/paMapReduceConfigurationProperties.property");  
    PAMapReduceJobConfiguration pamrjc = new PAMapReduceJobConfiguration(f);  
  
    // 7. Instantiate PAMapReduceJob using instances of Job and  
    // PAMapReduceJobConfiguration as arguments  
    PAMapReduceJob pamrj = null;  
    try {  
        pamrj = new PAMapReduceJob(job, pamrjc);  
    } catch (PAJobConfigurationException e) {  
        e.printStackTrace();  
        System.exit(1);  
    }  
  
    // 8. submit the job to the scheduler  
    if (pamrj.run())  
        System.out.println("Submitted " + pamrj.getJobId());  
    else  
  
        System.out.println("Not submitted");  
}  
}
```

## 2.4. Running the example

To compile and run the example, take the following steps:

- Check that the path to configuration file is specified correctly in the code of example, and that the setting `org.ow2.proactive.scheduler.ext.mapreduce.workflow.classpath` in the configuration file points to `$EXAMPLE_DIR/classes`.
- Compile:

```
# for compilation, we need 3 jars: Hadoop core, ProActive core and ProActive MapReduce addon
$ COMPILE_CLASSPATH=$SCHEDULER_HOME/dist/lib/mapreduce/hadoop-0.20.2-core.jar:
$SCHEDULER_HOME/dist/lib/ProActive_Scheduler-core.jar:$SCHEDULER_HOME/dist/lib/ProActive_Scheduler-
mapreduce.jar

$ mkdir $EXAMPLE_DIR/classes
$ javac -cp $COMPILE_CLASSPATH -d $EXAMPLE_DIR/classes $EXAMPLE_DIR/WordCount.java
```

- Create an input file:

```
$ mkdir -p $EXAMPLE_DIR/data/input
$ mkdir -p $EXAMPLE_DIR/data/output
$ cat > $EXAMPLE_DIR/data/input/input_file << EOF
this is a test
of word count example
test
EOF
```

- In a separate terminal, start the scheduler in default configuration (this also starts the Resource Manager and 4 local nodes):

```
$ cd $SCHEDULER_HOME
$ ./bin/unix/scheduler-start-clean
Starting Scheduler, Please wait...
Resource Manager doesn't exist on the local host
Trying to start a local Resource Manager
Resource Manager created on rmi://deephought.inria.fr:55855/
Starting scheduler...
Scheduler successfully created on rmi://deephought.inria.fr:55855/
```

Make sure that the value of `org.ow2.proactive.scheduler.ext.mapreduce.schedulerUrl` property in the configuration file matches the URL in the output of the command.

- Run:

```
java -cp $SCHEDULER_HOME/dist/lib/*:$EXAMPLE_DIR/classes WordCount
```

- Check job status with `scheduler-client`. Once the job is finished, the output should look like this:

```
$ $SCHEDULER_HOME/bin/unix/scheduler-client -lj
```

ID	NAME	OWNER	PRIORITY	PROJECT	STATUS	START AT	DURATION
1	wordcount	admin	Normal	ProActiveMapReduce	Finished	20:16:41 07/05/11	29s 235ms

Refer to the scheduler documentation for mor information about the usage of `scheduler-client`.

- The output of the reducer tasks is inside the `$EXAMPLE_DIR/data/output/output_dir/` directory. There should be one file per each reducer task, named `reduced_<reducer task id>`. Since in this example we have only one reducer, there should be only one file. Verify its contents:

```
$ cat $EXAMPLE_DIR/data/output/output_dir/reduced_100*
a 1
count 1
example 1
is 1
of 1
```

```
test 2  
this 1  
word 1
```

# Chapter 3. ProActive MapReduce Reference

## 3.1. Internal structure of a ProActive MapReduce job

The ProActive MapReduce, when the user creates the ProActive MapReduce job and before that job is submitted to the ProActive Scheduler, internally builds a ProActive workflow made up of five tasks as the [Figure 3.1, “ProActive MapReduce workflow”](#) shows: **SplitterPATask**, **MapperPATask**, **MapperJoinPATask**, **ReducerPATask** and **ReducerJoinPATask**. The execution order of those tasks and their behavior is the following: the **SplitterPATask** creates the input splits from the input file. Each **MapperPATask** processes one and only one input split. To achieve that, MapperPATask is replicated with the replication factor equal to the number of created input splits. The **MapperJoinPATask** implements the join of the execution of the MapperPATask replicas. **ReducerPATask** starts its execution only after the MapperJoinPATask (and so all the MapperPATask replicas) ends. The ReducerPATask is replicated too. The number of replicas of the ReducerPATask is defined by the user. If the user does not define it, only one ReducerPATask is executed. The last executed task is the **ReducerJoinPATask** that implements the join of the ReducerPATask replicas.

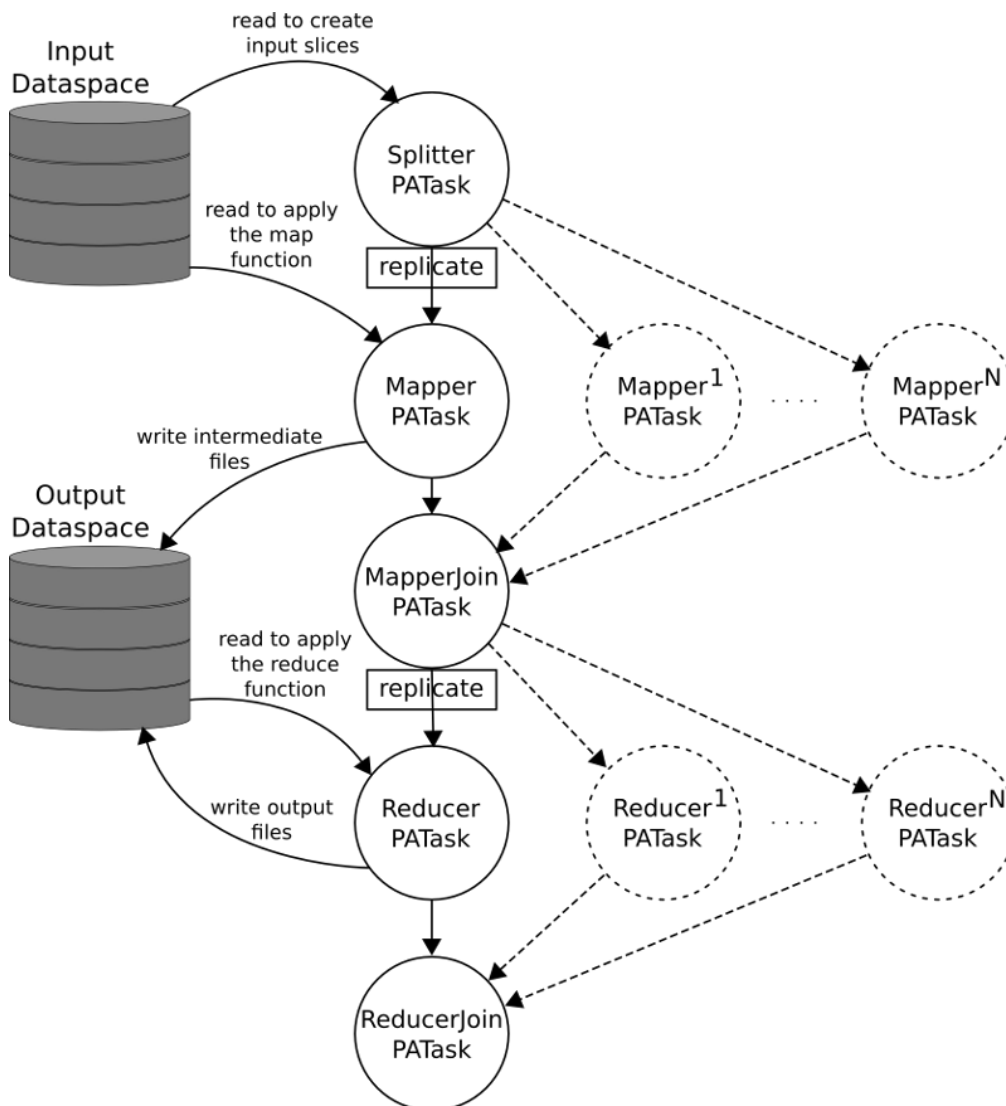


Figure 3.1. ProActive MapReduce workflow

## 3.2. ProActive MapReduce configuration

As was already mentioned in the [Chapter 2, Tutorial: running Hadoop job using ProActive MapReduce](#), ProActive MapReduce configuration consists of 2 major steps:

- Hadoop-compatible configuration: done in the same way as the configuration of a real Hadoop job.
- ProActive MapReduce-specific configuration: done by creating a property file and passing it as an argument to PAMapReduceJobConfiguration constructor.

As an alternative to using the property file, most of the properties can be set by calling the methods of the PAMapReduceJobConfiguration class.

The full list of supported configuration properties, together with the corresponding PAMapReduceJobConfiguration methods and default values is given in this section.

Note: the ProActive MapReduce job is actually a ProActive workflow. Thus, many of its configuration properties are directly related to the ProActive workflow and to the tasks belonging to it, for instance, the **cancelJobOnError** attribute of a task.

### 3.2.1. Required configuration properties

The ProActive MapReduce requires that the user specifies the following properties:

- **org.ow2.proactive.scheduler.ext.mapreduce.schedulerUrl**: the URL of the ProActive Scheduler.
- **org.ow2.proactive.scheduler.ext.mapreduce.username**: the username to use to establish the connection to the ProActive Scheduler.
- **org.ow2.proactive.scheduler.ext.mapreduce.password**: the password to use to establish the connection to the ProActive Scheduler.
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.inputSpace**: the INPUT space of the job. The input files should reside in the root of that specified INPUT space or in a sub-folder of it. Corresponding PAMapReduceJobConfiguration method: **setInputSpace()**.
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.outputSpace**: the OUTPUT space of the job. The output files will be stored in a sub-folder of the OUTPUT space; they cannot be stored in the root of the OUTPUT space. Corresponding PAMapReduceJobConfiguration method: **setOutputSpace()**.

### 3.2.2. Optional configuration properties

All the other configuration properties are optional as the ProActive MapReduce provides default values for them.

- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.projectName**: the "projectName" attribute of the ProActive MapReduce workflow. Default: **"ProActiveMapReduce"**. Corresponding PAMapReduceJobConfiguration method: **setProjectName()**.
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.description**: the "description" attribute of the ProActive MapReduce workflow. Default: **"ProActive MapReduce"**. Corresponding PAMapReduceJobConfiguration method: **setDescription()**.
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.maxNumberOfExecutions**: in case if a task fails, an attempt to restart it will be made until the total number of executions reaches the value of this property. Default: **1**, which means that no restart attempts will be made. Corresponding PAMapReduceJobConfiguration method: **setMaxNumberOfExecutions()**.
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.restartTaskOnError**: determines **where** the restart of a failed task will be performed, the possible values for this property are `org.ow2.proactive.scheduler.ext.mapreduce.workflow.restartAnywhere` (restart on any node) and `org.ow2.proactive.scheduler.ext.mapreduce.workflow.restartElsewhere` (restart on any node except the one where the task failed). Default: **"org.ow2.proactive.scheduler.ext.mapreduce.workflow.restartAnywhere"**. Corresponding PAMapReduceJobConfiguration method: **setRestartTaskOnError()**.
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.cancelJobOnError**: whether to cancel the job in the event of one of the tasks reaching its `maxNumberOfExecutions` without a successful result. Default: **true**. Corresponding PAMapReduceJobConfiguration method: **setJobCancelJobOnError()**.

The above properties also have task-specific counterparts, which, if specified, take precedence for a particular type of task over the general ones:

- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.[splitter|mapper|mapperJoin|reducer|reducerJoin]PATask.name:** the "name" attribute for the corresponding task;
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.[splitter|mapper|mapperJoin|reducer|reducerJoin]PATask.description:** the "description" attribute for the corresponding task;
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.[splitter|mapper|mapperJoin|reducer|reducerJoin]PATask.maxNumberOfExecutions:** the "maxNumberOfExecutions" attribute for the corresponding task;
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.[splitter|mapper|mapperJoin|reducer|reducerJoin]PATask.restartTaskOnError:** the "restartTaskOnError" attribute for the corresponding task.
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.[splitter|mapper|mapperJoin|reducer|reducerJoin]PATask.cancelJobOnError:** the "cancelJobOnError" attribute for the corresponding task.

The following properties, while optional, are still very important:

- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.classpath:** a comma-separated list of the additional classpath entries for the ProActive MapReduce tasks. This classpath is meant to point to the user implementations of the Hadoop Mapper, Reducer, InputFormat etc... Default: **empty**. Corresponding PAMapReduceJobConfiguration method: **setClasspath()**.
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.splitterPATask.inputSplitSize:** it defines the size, in bytes, of the input split (fragment of the input data) each mapper will get as an input. This configuration property is optional but no default value is defined by the ProActive MapReduce. When the user does not specify the size of the input split then the default value is given by the user-provided Hadoop InputFormat. ProActive MapReduce uses the user specified Hadoop InputFormat class to create input splits and that the size of those input split is equal to a default value computed by the user specified Hadoop InputFormat class. When the user defines a value of 0 (zero) bytes for the size of the input split, then input splits with a size equal to the minimum possible value forecast by the Hadoop InputFormat class used are created. On the other hand, when the user defines a size greater than the size of the input file then only one input split is created and the size of that input split is equal to the size of the input file. Corresponding PAMapReduceJobConfiguration method: **setInputSplitSize()**.
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.splitterPATask.readMode:** it defines the read mode of the SplitterPATask. The two possible values are: "fullLocalRead" and "remoteRead". The former means that the input file is transferred to the node the SplitterPATask executes on before input splits are created. The latter means that the input file is left in the ProActive MapReduce workflow INPUT space so that data used to create input splits are read directly from there. Default: **"remoteRead"**.
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.mapperPATask.readMode:** it defines the read mode of the MapperPATask. There are three possible values:
  - **fullLocalRead:** the input file of the ProActive MapReduce workflow is transferred entirely to the node the MapperPATask executes on and then the MapperPATask reads from it and processes only the data of its own input split;
  - **partialLocalRead:** only the data the MapperPATask must process are copied from the input file and transferred to the node the MapperPATask executes on;
  - **remoteRead:** the data the MapperPATask must process are read remotely from the ProActive MapReduce workflow INPUT space.

The default value is **"remoteRead"** but if the input file is not randomly accessible then **"fullLocalRead"** is used.

- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.mapperPATask.writeMode:** it defines the write mode of the MapperPATask. The two possible values are: "localWrite" and "remoteWrite". The former implies that the output data of the MapperPATask are first stored on the node it executes on and then the ProActive DataSpaces mechanism transfers them to the user defined OUTPUT space. The latter indicates that the output data are stored directly in the user defined OUTPUT space. Default: **"localWrite"**.
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.reducerPATask.readMode:** it defines the read mode of the ReducerPATask. The two possible values are: "fullLocalRead" and "remoteRead". The former means that the intermediate data (the MapperPATask output data) are transferred from the OUTPUT space to the node the ReducerPATask will execute on while "remoteRead" means that the intermediate data are left in the OUTPUT space so that they are read remotely by the ReducerPATask. Default: **"remoteRead"**.
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.reducerPATask.writeMode:** it defines the write mode of the ReducerPATask. The two possible values are: "localWrite" and "remoteWrite". The former implies that the output data of the ReducerPATask are first stored on the node it executed on and then the ProActive DataSpaces mechanism transfers them to the

user defined OUTPUT space. While the latter indicates that the output data are stored directly in the user defined OUTPUT space. Default: "**localWrite**".

- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.reducerPATask.outputFileNamePrefix**: it defines the prefix the ProActive MapReduce framework/API must use to build the name of the output file. The number of output files of the ProActive MapReduce job is equal to the number of executed ReducerPATask and each file has a name compliant to the following format: `<outputFileNamePrefix>_<reducerPATaskId>`. Default: "**reduced**". Corresponding PAMapReduceJobConfiguration method: **setReducerOutputFileNamePrefix()**.

### 3.3. API limitations

There are two main restrictions related to the ProActive MapReduce: it requires Java 6 and it provides the support only for the **Hadoop MapReduce 0.20.2** release and only for the *Hadoop new MapReduce API* (i.e., the Hadoop MapReduce job must be built using the classes defined in the package **org.apache.hadoop.mapreduce**, not **org.apache.hadoop.mapred**).

Moreover, the ProActive MapReduce Hadoop-like API implementation does not support many of the advanced Hadoop features, such as reporters, counters, compression, distributed cache, debug scripts and speculative execution. Those features, if specified during the configuration step, are ignored by the ProActive MapReduce.

# Chapter 4. ProActive MapReduce performance

## 4.1. Execution environment

To perform the tests we used the same group of hosts to run both Hadoop and ProActive MapReduce. The execution environment in which the benchmarks were performed is a cluster of 20 machines. The machines are identical and their characteristics are the following:

- Bi-processor Intel® Xeon E5335 2.00GHz quad core;
- 16 GB of RAM;
- 30 GB per node of disk space;
- Fedora® Core 7 with 2.6.23.17-88 kernel;
- Network connection between the cluster nodes is Gigabit Ethernet.

2 machines were reserved for running Hadoop JobTracker, HDFS NameNode, ProActive Scheduler and ProActive Resource Manager; the remaining 18 were used for dataNodes of the HDFS and execution of map and reduce tasks.

The version of JDK used for benchmarks is JDK 6 update 14.

## 4.2. Input data

Input files were generated using [TPC-H dbgen utility](#)<sup>1</sup>, a database population program that generates files to be loaded into database tables. They consist of lines containing strings separated by "|" symbols. Of the several files generated by dbgen, we used only the file "lineitem.tbl".

## 4.3. ProActive MapReduce configuration

On each of 18 machines 2 ProActive nodes were deployed for the executions of the ProActive MapReduce workflow. The main configuration parameters of the job were set as follows:

- the INPUT and OUTPUT spaces are directories on an NFS server NFS; the link to the NFS server is Gigabit Ethernet;
- the input split size is chosen so that the number of mappers is 36, equal to the number of available nodes;
- the number of reducer tasks is 36, equal to the number of mappers;
- the read mode of the SplitterPATask is *remoteRead*;
- the read mode of the MapperPATask is *remoteRead*;
- the write mode of the MapperPATask is *localWrite*;
- the read mode of the ReducerPATask is *remoteRead*;
- the write mode of the ReducerPATask is *localWrite*.

## 4.4. Hadoop configuration

Each TaskTracker<sup>2</sup> was configured to run at most 6 map tasks during the map phase and 6 reduce tasks during the reduce phase. The size of input split was not configured and thus was chosen by Hadoop based on the HDFS block size (64 MB). The number of reducers was set to 54. Input files were uploaded to the HDFS with a replication factor of 3. We decided to include into the total execution time the time to upload the file to the HDFS to account for the fact that, unlike NFS used in ProActive MapReduce, HDFS is not a general purpose file system and thus a separate upload step is often required.

<sup>1</sup> <http://www.tpc.org/tpch/>

<sup>2</sup> In the Hadoop cluster there is a TaskTracker for each machine

## 4.5. Results

The results of the benchmark are given in the table below:

File Size	Sequential	Hadoop + upload	PA MapReduce	Speedup
0.7 GB	5m 04s	1m 17s	1m 05s	4.6
4.3 GB	25m 31s	2m 30s	2m 20s	10.9
7.3 GB	46m 00s	3m 31s	3m 30s	13.1
20 GB	2h 07m 00s	8m 30s	7m 09s	17.8
50 GB	5h 19m 00s	21m 05s	25m 11s	12.7
100 GB	10h 38m 00s	43m 23s	1h 07m 00s	9.4

**Table 4.1. ProActive MapReduce performance**

Legend:

- **File Size:** size of the input file.
- **Sequential:** average total execution time of PA MapReduce job configured to use one Mapper and one Reducer, input and output files stored on the local disk.
- **Hadoop + upload:** average execution time of a Hadoop job plus the upload time of the input file to the HDFS with the replication factor 3.
- **PA MapReduce:** average total execution time of parallel PA MapReduce job.
- **Speedup:** the ratio Sequential / PA MapReduce.