Parallel computing in OpenDA using Threading and Java RMI

Nils van Velzen

November 30, 2012

1 Front-end models in OpenDA

Front-end models are a special kind of models in OpenDA. A front-end model implements the OpenDA Stochastic model interface but it does not implement any model equations. A front-end model adds additional functionality on top of an arbitrary OpenDA Stochastic model. This model is called the back end model. In general, all methods of the front-end model are implemented by use of methods of the back-end model.

The front-end models implement generic extension to OpenDA Stochastic models.

In this memo we will give a description of two of these kind of front-end models. The Thread Stochastic Model adds some parallelism to arbitrary OpenDA stochastic models using threading. The RMI Stochastic Model allows models to be run in a different process and optionally remotely on a remote computer.

2 Parallelism using Threading

2.1 introduction

Java has a good support for threading. Using threads, a program can make use of all the available computational cores in a computer. Threading is therefore in Java a simple and natural way to implement shared-memory parallelism.

Many data assimilation and calibration algorithms perform a (large) number of model simulation steps

$$x\left(t+\Delta t\right)=M\left(x\left(t\right),u\left(t\right),p\right) \tag{1}$$

for various state vectors $x\left(t\right)$ or parameters p. When these model simulations are independent, we can compute them in parallel.

2.2 Thread Stochastic Model

2.2.1 Basic usage and configuration

OpenDA provides a front-end model that parallelizes the compute method of a stochastic model using threading. This model can be used as a front-end to any OpenDA model that implements the compute method in a thread safe way. The maximal number of threads can be specified in order to limit the number of simultaneous compute invocations.

The model factory of the threaded front-end model is

`org.openda.models.threadModel.ThreadStochModelFactory`

The configuration file of the `ThreadStochModelFactory` specifies

- maxTheads; the max number of simultaneous compute threads. This is typically set to the max number of cores that are available to the user.
- stochModelFactory; the configuration of the the back-end stochastic model factory.

A typical example of the `ThreadStochModelFactory` configuration looks like

```
<threadConfigstoch>
   <maxThreads>8</maxThreads>
   <stochModelFactory
      className="org.openda.models.lorenz.LorenzStochModelFactory">
      <workingDirectory>model</workingDirectory>
      <configFile>LorenzStochModel.xml</configFile>
   </stochModelFactory>
</threadConfigstoch>
```

NOTE1: The compute method of your model MUST be thread safe. Java models will often be thread safe. Native models are in general not thread safe.

NOTE2: The working directory for the back-end model is NOT relative to the location of the `ThreadStochModelFactory` configuration file but to the "main" OpenDA configuration file.

2.2.2 advanced features

The basic usage of thread stochastic model only includes specifying the max number of threads and the back-end model factory. There are some additional options that can improve the performance of your parallel application. These advanced options are described in this section. The impact of these options depend heavily on the used model and architecture.

- `cashState` (boolean) if set "true", a parallel nonblocking getState method is invoked after a parallel compute is completed. This parallel getState method is invoked on a new thread and not limited by the maxThread option. In this way, the data assimilation algorithm already receives a copy of the state while other model instances are still computing. This can be useful when the time of the getState method is dominated by IO (file or network). The state is stored internally until the getState method is invoked.

- `nonBlockingAxpy` (boolean) if set "true", a invocation of the axpyState method will be non-blocking. This option might improve performance when the time of the axpyState method is dominated by IO (file or network).

The XML configuration using these two options looks like:

```
<threadConfig>
   <maxThreads>2</maxThreads>
   <cashState>true</cashState>
   <nonBlockingAxpy>true</nonBlockingAxpy>
   <stochModelFactory
    className="org.openda.models.rmiModel.RmiClientStochModelFactory">
      <workingDirectory>./stochModel</workingDirectory>
      <configFile>RmiStochModel2.xml</configFile>
```

```
      </stochModelFactory>
</threadConfig>
```

3 Remote Method Invocation

3.1 Introduction

RMI (Remote Method Invocation) is a way in java to make a connection between objects in various virtual machines (executables). In OpenDA we provide a front-end model that enables us to create and use stochastic models on various executables and computers. Note that RMI allows us to use multiple processes in our application but that computations are not automatically in parallel.

3.2 RMI Stochastic Model

The RMI provides the (parallel) computation of model simulation steps in a different virtual machine as the data assimilation method or the model calibration method. This serves two goals:

- multiple model time steps can be computed in parallel (in combination with the Thread model front-end)
- the model can runs on a dedicated machine (server) where the data assimilation method runs locally.

The OpenDA application running the data assimilation algorithm or model calibration algorithm is called the client. This executable will make use of one or more servers, possibly running on remote machines.

The RMI front-end model factory is

`org.openda.models.rmiModel.RmiClientStochModelFactory`

The configuration file of the `RmiClientStochModelFactory` specifies

- serverAddress; The names of the computers running the servers. There are three ways of configuration possible:
  - empty; all servers run on the local host
  - single machine name; all servers run on this machine

  - names of all computers, comma separated. Note that the number of computer names must be the same as the number of specified factoryIDs. The same computer name can be used multiple times when multiple servers run on that computer.

- factoryID; The unique IDs of all the servers, comma separated, implementing the remote models and model factories.

- stochModelFactory; the configuration of the the back-end stochastic model factory. **Important note**: The `workingDirectory` and `configFile` are communicated to the server processes. The use of relative paths is only possible when the servers are started from an appropriate location. Full paths are therefore often a better choice.

A typical example of a `RmiClientStochModelFactory` configuration file is

```
<rmiConfig>
   <serverAddress></serverAddress>
   <factoryID>IRmiIStochModel_1,IRmiIStochModel_2</factoryID>
   <stochModelFactory
      className="org.openda.models.lorenz.LorenzStochModelFactory">
      <workingDirectory>./model</workingDirectory>
      <configFile>LorenzStochModel.xml</configFile>
   </stochModelFactory>
</rmiConfig>
```

3.3 RMI Stochastic Model Server

In the previous section we have explained how to use the RMI front-end model from the client side. The other side are the server processes. The java class that implements the server is `org.openda.models.rmiModel.Server`.

Before the servers can be started a special daemon process must be started `rmiregistry` on each host computer. This daemon process allows RMI clients and servers to connect. The following example shows a shell script that starts and initialises two servers on local host.

```
#!/bin/sh

# append all jars in opendabindir to java classpath
for file in $OPENDADIR/*.jar ; do
```

```
   if [ -f "$file" ] ; then
       export CLASSPATH=$CLASSPATH:$file
   fi
done

rmiregistry &
echo wait 5 seconds for rmiregistry to start and initialize
sleep 5

# start the server with a non-default factory ID
echo starting server
java  -Djava.rmi.server.codebase=file:///$OPENDADIR/ \
      org.openda.models.rmiModel.Server IRmiIStochModel_1 0 2&
java  -Djava.rmi.server.codebase=file:///$OPENDADIR/ \
      org.openda.models.rmiModel.Server IRmiIStochModel_2 1 2&
```

Note the `-Djava.rmi.server.codebase=file:$OPENDADIR` option. This specifies the starting point to the classes that are loaded by the server. The two arguments have the following meaning:

1. (string) ID of this server
2. (int) the sequence number of this server (0,...,number of servers -1)
3. (int) total number of servers

The last two numbers are used to configure the DistributedCounter as will be explained in the following section.

3.4 InfiniBand and multiple network cards

Some nodes have multiple network connections. On supercomputers you will typically see that nodes both have an ethernet and an InfiniBand connection. In this case the computer will have two ip addresses and probably two names. The RMI server will use the network connection that corresponds to the $HOSTNAME of the node. This is typically the ethernet connection and not the fast InfiniBand we would like to use. The java option

`-Djava.rmi.server.hostname="InfiniBandHostname"`

can be passed to Java when starting the servers. In this way, the servers will use the InfiniBand connection for communication to the algorithm process.

3.5 Random numbers and DistributedCounter

When running in parallel you sometimes need to generate "unique" numbers in the model instances. E.g. when each model instance generates (or reads) its own input and output files and when you want to use some numbering to distinguish them. In order to globally define unique counters we have introduced the DistributedCounter class in OpenDa. When used in a sequential run, a DistributedCounter instance is just a integer counter 0,1,2,3,etc. In parallel runs, the DistributedCounter instance will generate a sequence $i_p + jn_p$ for j=$0, 1, ...$ where i_p denotes the process number and n_p the total number of processes.

A similar issue is the generation of pseudo random numbers. We must be very careful when we generate random numbers in a parallel environment. If the random generators in various threads of processes are all initialised with the same initial seed, we will get wrong results since the same pseudo random numbers will be drawn on the various processes. The DistributedCounter can be used to initialise pseudo random generators. When the user uses one of the noise models from OpenDA this should work correct in parallel since these random generators make use of the DistributedCounter.

3.6 Serialization

All arguments (classes) that are passed between the remote model instances need to "extends Serializable". If this is not the case Java cannot pack the content and send the object to an other processes. Many of the relevant classes in OpenDA are already Serializable but user provided classes are often not. When a class is not Serializable you will get a run-time error when you try to run in parallel. Fortunately, this is often easy to fix by adding "extends Serializable" to these classes (and subclasses).

3.7 Limitations

The RMI front-end model is work in progress. There are therefore some limitations including

- Only java implementations of (Tree-)Vectors can be used because the serialization of native objects is not yet implemented

- We have not done any work/testing on client/server security issues but we did not encounter any problems on the HPC computer systems we have used. There might be some more work to be done for using these tools in more secured environments.

4 Parallel computing with multiple executables

The Thread Stochastic model allows parallel computation of model time steps. This parallelism is limited to a single executable and maximized by the amount of cores of a single computer. The RMI Stochastic model allows multiple executables to be used for the model computations. But The RMI model does not by itself support parallelism. However when both front-end models are combined we are able to compute model time steps in parallel using multiple processes and multiple computers. The Thread model is then used as a frond-end to the RMI model which is the front-end of the physical model.

Models that are not thread safe can be parallelized by combining both front end models because at the server side no time steps are computed in parallel.