Using Cloud Computing to Do Large Numbers of MOVES Runs

Wesley Faler
Fluid & Reason, LLC, Dearborn, MI 48120
wes.faler@gmail.com

Harvey Michaels and William Aikman
U.S. Environmental Protection Agency, OTAQ, 2565 Plymouth Rd, Ann Arbor, MI 48105
michaels.harvey@epa.gov, aikman.william@epa.gov

ABSTRACT

EPA's Office of Transportation and Air Quality (OTAQ) turned to cloud computing because MOVES is computing-intensive, we have a lot of runs to do, and cloud computing provides cheap, abundant computing resources on demand. Mobile-source inventory generation is an ideal application for cloud computing, because the calculations for each geographic unit, time period, and vehicle class are independent and can therefore be run on separate computers in parallel. The challenges have been in managing large numbers of runs, tracking and recovering from errors, and integrating the results into useful output. In this paper, we describe in some detail how we use the cloud to create and post-process MOVES rate tables for eventual air quality modeling. Cloud computing vendors differ in their interfaces, so what we have done is not universal, but it provides a potentially useful picture of the processes, complexities, pitfalls, and rewards of such an effort.

INTRODUCTION

"The cloud" refers to computers that we access via the internet. We are able to access them on demand and pay only for the time our programs are running. We are able to specify the configuration we want: operating system, memory, storage, and installed software. We don't need to buy, install, maintain, repair, or upgrade the computers. We are not interested in using the cloud to run a single computer. We run hundreds of computers at a time, for limited periods.

EPA's Office of Transportation and Air Quality (OTAQ) turned to cloud computing because MOVES is computing-intensive, we have a lot of runs to do, and cloud computing provides cheap, abundant computing resources on demand. Mobile-source inventory generation is an ideal application for cloud computing, because the calculations for each geographic unit, time period, and vehicle class are independent and can therefore be run on separate computers in parallel.

The purpose of this paper is to introduce other MOVES users to our methods, since the problem of finding sufficient computing resources to do many MOVES runs is a common one. At OTAQ, the problem of processing many MOVES runs in a reasonable period of time originated with our need to generate the National Emissions Inventory (NEI) and to produce emission factor lookup tables for air quality modeling. In this paper, we will concentrate on the MOVES runs and post-processing needed to prepare emission factors for developing air quality modeling inputs, although we use similar methods for the National Emission Inventory (NEI) and for other MOVES inventory runs.

By way of background, to facilitate air quality modeling, EPA's Office of Air Quality Planning and Standards(OAQPS) and OTAQ integrated OAQPS's emission-pre-processor SMOKE with OTAQ's MOVES model[1]. This new integrated software is called SMOKE-MOVES.[2] It was built to use grid-cell-hour-specific temperatures and speeds to construct grid-cell-hour-specific inventories for air quality modeling. Early in the SMOKE-MOVES process, MOVES is run to produce lookup tables

of emission factors by temperature and speed. These MOVES tables are post-processed to generate SMOKE-ready emission factor tables, which SMOKE multiplies times activity (distance or population) to produce inventories at the grid-cell-hour level.

We limit the number of MOVES runs and the size of the lookup tables by creating county groups and fuel-month groups. The United States counties in the group must have similar winter and summer fuels, age distributions, and I/M programs. A single county from the group is chosen as the "representing county," typically the one with the most VMT. Thus, only two sets of lookup tables for all the counties in the group are required: one for the summer months and one for the winter months. SMOKE-MOVES is designed to use this scheme of county and fuel month groups.

The MOVES run specification (runspec) for each representing county-month is constructed by SMOKE-MOVES to generate emission factors over the full temperature range for all of the hours for all of the counties in the group for the winter months (October-April) and, separately, for the summer months (May-September). The modeled temperatures cover the range in specified increments (we have used ten degrees F, but any increment is allowed). SMOKE interpolates between the emission factors to match the actual grid-cell-hour temperatures. SMOKE then multiplies the appropriate emission factors from the lookup tables times the VMT (by SCC) or the vehicle population (by vehicle type) for each grid cell. As an example of the savings generated by using county and fuel month groups, to run all counties in the US for all months, we would have to develop individual runs for 3,109 counties x 12 months = 37,308 county-months. By using 146 representing counties and 2 months, we only have to do runs for 292 county-months, a more than 100 fold reduction in the number of batches. SMOKE-MOVES is designed to take advantage of this technique by applying emission factors for the representing county to the entire county group. The list of county groups and the list of fuel months are inputs to SMOKE-MOVES. Running the 292 county-months in parallel results in another two orders of magnitude reduction in required clock time.

CREATING AND ORGANIZING THE RUNSPECS AND DATABASES

OTAQ chooses the county groups, representing counties, and fuel months, as described above. In practice, we use a MySQL script and are continuing to refine our methodology. Once we have chosen the county groups and fuel months, the meteorology data needed by SMOKE-MOVES is generated by a SMOKE-MOVES Perl script called "met4moves", which generates an output file that contains all the temperatures and profiles that must be generated for each representing county. Met4moves takes three inputs:

- The detailed grid-cell-hour temperatures and humidity prepared for air quality modeling.
- The county groups, representing counties, and fuel months prepared by OTAQ.
- The size of the temperature increment. (For example, if the full temperature range is 10 deg F to 100 deg F and the increment is 10 deg, MOVES will calculate emission factors for 10, 20, 30, …100 degrees F.)

After representing counties are chosen and met4moves is run, the SMOKE-MOVES "runspec_generator" (a Perl script) is run. Its inputs are the list of representing counties and the output from met4moves. The runspec_generator produces:

- All the runspecs needed to generate all the lookup tables for all the representing counties. Each runspec has a unique name identifying the representing county, the type of rate that it will generate (rate per distance, rate per profile, or rate per vehicle) and the temperature or temperature range.

- One MOVES zonemonthhour table (temperature and humidity) in comma-separated variable (csv) format for each runspec. The association between runspec and table is made by using the same file name, with a different extension. An OTAQ Perl script converts the zonemonthhour csv files into MySQL zonemonthhour tables inside MySQL database folders having the same name as the csv files.

For the example we have been discussing, there were 146 representing counties and 2 fuel months, for a total of 292 county-months. Each county-month is a "batch." In order to cover the range of temperatures needed, each batch has between 42 and 167 runspecs associated with it for a total of about 27,000 runspecs and the same number of zonemonthhour tables, one for each runspec. The average number of runspecs per batch is about 90.

A Perl script organizes the runspecs and databases in a precise structure on a local file server drive. Our cloud software is expecting that structure. The county-month is the unit for which SMOKE-MOVES needs lookup tables. Therefore, we use county-month as our basic unit, which we refer to as a "batch." We run a batch on a single cloud-based virtual machine. A batch corresponds to a folder. A batch has one or more "jobs," organized as subfolders to the batch, and may also have databases associated with it. A job corresponds to a runspec and any databases associated with it (for example, as described above, a zonemonthhour database). Because SMOKE-MOVES requires many runspecs to create the necessary lookup tables, each batch has as many jobs as it needs. Each batch also is in its own directory, and each job (runspec and supporting databases) is a subdirectory to its parent batch.

OUR CLOUD APPROACH

In the Amazon Elastic Compute Cloud (EC2),[3] we start Linux computers (called "instances"), each running one MOVES master and one worker. We are not trying to get a single run, or even a single county-month, done quickly. Instead, we are executing hundreds of county-months in parallel.

We have an automated system to set up, run, and download hundreds of "batches" at once. Each batch holds all of the MOVES runspecs and user input databases for a single county-month. A batch might contain only one MOVES run. For SMOKE-MOVES processing, a batch typically comprises 100 or so MOVES runs, consisting of different temperatures and profiles. Each "instance" executes a single batch, performing each runspec within the batch sequentially.

Our design places all MOVES source code (written in Java), default databases (using MySQL), and batches into Amazon Simple Storage Service (S3) file storage.[4] We use Amazon's Simple Queue Service (SQS) queue to hold the list of batches that should be processed and another SQS queue to hold status messages output from the cloud instances.[5] Output databases and log files are placed into S3 storage as well. A set of Java-based commands moves files to and from our local file store and the cloud as well as handling queue interactions.

The scale of the entire process causes many tens of thousands of log files to be produced. We copy all log files onto our local file server directories ("logqueue" folders) where they can be scanned by local Perl scripts for error messages.

Amazon Terminology

An Amazon "S3 bucket" holds files in the Amazon cloud, similar to a network file server that holds files and makes them available for any authorized user.

An "Amazon Machine Image" ("AMI") is a total copy of the hard drive of a single computer in the Amazon EC2 cloud. This hard drive image can be cloned into any number of actual running

computers, called "Instances". Each running instance starts as a clone of an AMI and then begins to run independently of the original image. A running instance may be stopped and its drives converted into a new AMI from which new instances can be created.[6]

Amazon's "SQS queue" provides a first-in-first-out queue system for short messages. Messages, but not entire files, can be queued. When a message is retrieved, it is gone forever from the queue and is retrieved only once. Therefore, instances can check a queue to retrieve work items, certain that no two instances will work on the same item. There is a safety feature in the SQS system, in which a message will be restored to the queue if the retrieving computer fails to complete its work. However, as MOVES runs take longer than the maximum safety timeout, this feature is not used.

A "JAR" file is a ZIP file that adheres to Java standards for internal directory structure and metadata files.

We define a "Job" to be a single MOVES runspec and input databases that are specific only to that runspec.

We define a "Batch" as a collection of one or more Jobs along with input databases required by all, or at least many, of the Jobs.

A "Scenario" is defined as one or more Batches and input databases required by all, or at least many, of the Batches and their Jobs.

Lifecycle without Post Processing

The overall steps for processing MOVES jobs on Amazon are shown below. Most are accomplished using Java-based command-line tools we have created.

1) An Amazon SQS queue is created to hold job processing commands.
2) An Amazon SQS queue is created to hold status messages.
3) MOVES code is placed into a JAR file.
4) The MOVES code JAR is placed into an Amazon S3 bucket.
5) A MOVES default database is placed into a JAR file.
6) The default database JAR file is placed into an Amazon S3 bucket.
7) A JAR file is created for each job. This file contains the job's runspec (.mrs file) and input databases, if any.
8) A JAR file is created for the batch-level input databases, if any.
9) A JAR file is created for the scenario-level input databases, if any.
10) The JAR files are uploaded to a single Amazon S3 bucket.
11) After all JAR files for jobs in a batch have been uploaded, a command to process all jobs in a batch is placed into the command queue.
12) One or more Amazon EC2 instances are started to process the commands.
13) The status queue is polled for messages originating from the EC2 instances.
14) Job result JAR files are downloaded from an Amazon S3 bucket.
15) Result JAR files and job JAR files are deleted from the bucket.
16) The result JAR file's contents are extracted, including the output database and log files.
17) Operating system log files are duplicated and placed into the batch's local logqueue directory for automated scanning.
18) Amazon EC2 instances shutdown automatically after processing all jobs in a batch.

Lifecycle with Post Processing

Post processing of MOVES results introduces a few new steps and concepts. First, the output of each job, while still consisting of a database and a log file, is split such that the database portion is left within the cloud and the log file is downloaded and removed from the cloud. Second, post processing commands and code must be queued. Lastly, post processing results are not necessarily MySQL databases and are left within the cloud to facilitate sharing with other departments.

The overall steps for processing MOVES jobs on Amazon that use post processing are:

1) An Amazon SQS queue is created to hold job processing commands.
2) An Amazon SQS queue is created to hold status messages.
3) MOVES code is placed into a JAR file.
4) The MOVES code JAR is placed into an Amazon S3 bucket.
5) A MOVES default database is placed into a JAR file.
6) The default database JAR file is placed into an Amazon S3 bucket.
7) A JAR file is created for each job. This file contains the job's runspec (.mrs file) and input databases, if any.
8) A JAR file is created for the batch-level input databases, if any.
9) A JAR file is created for the scenario-level input databases, if any.
10) The JAR files are uploaded to a single Amazon S3 bucket.
11) After all JAR files for jobs in a batch have been uploaded, a command to process all jobs in a batch is placed into the command queue.
12) One or more Amazon EC2 instances are started to process the commands. These instances are given the "SEPARATERESULTS=1" flag in their instance data.
13) The status queue is polled for messages originating from the EC2 instances.
14) Job result JAR files are downloaded from an Amazon S3 bucket. These JAR files contain only the log files.
15) Result JAR files and job JAR files are deleted from the bucket. Database result JAR files remain in the bucket.
16) The result JAR file's contents are extracted, including only log files.
17) Operating system log files are duplicated and placed into the batch's local logqueue directory for automated scanning.
18) Amazon EC2 instances shutdown automatically after processing all jobs in a batch.
19) An Amazon SQS queue is created to hold post processing commands.
20) An Amazon SQS queue is created to hold post processing status messages.
21) Post processing code and required databases are placed into a JAR file.
22) The post processing code JAR is placed into an Amazon S3 bucket.
23) A command to post process all jobs in a batch is placed into the post processing command queue.
24) One or more Amazon EC2 instances are started to process the commands. These instances are given the "JOBCOMMAND=batchpostprocess" flag in their instance data.
25) The status queue is polled for messages originating from the EC2 instances.
26) Batch result JAR files are downloaded from an Amazon S3 bucket. These JAR files contain batch-level post processing results and log files.
27) No result JAR files are deleted from the bucket.
28) The result JAR file's contents are extracted.
29) Operating system log files are duplicated and placed into the batch's logqueue directory for automated scanning.
30) Amazon EC2 instances shutdown automatically after processing all jobs in a batch.

INTERACTING WITH THE AMAZON APPLICATION PROGRAMMING INTERFACE (API)

Amazon provides a set of Java classes to access their cloud's services. We created our own Java classes to interface with this API, providing the abstractions of Batches and Runs. This code is contained within its own JAR file separate from the main-branch of the MOVES code, allowing it to be used with multiple versions of MOVES. An Ant interface has been created to simplify command-line interaction with our Java tools. (Ant is a open source tool for automating software build processes.)[7]

Our tools are divided into two categories: S3 and Job. The S3 commands manipulate local files and Amazon S3 buckets. These commands have no concept of Batches or Runs. An example S3 command line is shown below. This command is used to transfer a JAR file containing a MOVES default database into an Amazon bucket.

```
set DBNAME=movesdb20110315
call ant -Dcmd="put bucket MYBUCKET nameinbucket %DBNAME%.jar file %DBNAME%.jar" s3
```

A large portion of our code is dedicated to error handling. The Amazon API sends and retrieves all data using Internet-standard web HTTPS secure sockets (SSL). However, not all web interactions are immediately successful and must be resubmitted. It is essential that code recover from these "soft" errors. For instance, the core lines of our Java code to upload a file to a bucket are shown below. The bolded line referencing "s3.putObject" uses the Amazon API to transfer a file. Most of the other code shown is for logging and recovering from error conditions.

```
while(true) {
        try {
          s3.putObject(new PutObjectRequest(bucketName, nameInBucket, f));
          break;
        } catch (AmazonClientException ace) {
          String aceMessage = ace.getMessage().toLowerCase();
          if(aceMessage.indexOf(" hash ") >= 0 || aceMessage.indexOf(" hash:") >= 0
                      || aceMessage.indexOf(" integrity ") >= 0
                      || aceMessage.indexOf("unable to upload part") >= 0) {
               String errorMessage = "Retrying upload of " + bucketName + "/"
                              + nameInBucket + " after error: " + ace.getMessage();
               S3.log(errorMessage);
               log(errorMessage);
               // Delete the file in S3
               try {
                      s3.deleteObject(bucketName, nameInBucket);
               } catch(Exception e) {
                      // Nothing to do here, the file may not even exist in S3
               }
               // Loop around to retry the upload
               waitRandomSeconds();
               continue;
          }
        } catch(Exception e) {
          // Delete the file in S3
          try {
            s3.deleteObject(bucketName, nameInBucket);
          } catch(Exception e2) {
            // Nothing to do here, the file may not even exist in S3
          }
          // Loop around to retry the upload
          waitRandomSeconds();
          continue;
        }
}
S3.log("Stored \"" + nameInBucket + "\"");
```

MOVES AMAZON S3 INTERFACE API

We have created “wrappers” around the Amazon S3 file storage API. These wrappers handle soft errors as well as handle workflow issues such as automatically creating Amazon Buckets and avoiding redundant file transfers. These S3 commands are generic and have no concept of the higher-level Batch/Job system.

All of our S3 commands are available via Ant calls. This is the same mechanism used by MOVES to simplify compilation and running of the MOVES system. An example DOS/Windows command line for one such command is:

```
call ant -Dcmd="put bucket MYBUCKET nameinbucket MYDBNAME.jar file MYDBNAME.jar" s3
```

Our S3-wrapper commands are:

- login accesskey <keyvalue> secretkey <keyvalue>
- logout
- put bucket <bucketname> nameinbucket <nameinbucket> file <filenameandpath>
  - Creates the bucket if it does not exist and does not transfer if <nameinbucket> already exists.
- putmany bucket <bucketname> prefix <prefix> directory <fulldirectorypath>
  - Do a "put" on all files with the matching prefix. * is used to indicate all files.
- get bucket <bucketname> nameinbucket <nameinbucket> file <filenameandpath>
  - Does not transfer data if <filenameandpath> already exists.
- delete bucket <bucketname> nameinbucket <nameinbucket>
  - Reports warnings if the bucket or the named object do not exist.
- list bucket <bucketname>

MOVES AMAZON JOB API

We created a set of commands to handle Batches and Jobs. Like the S3 wrappers, these are Java-based routines made available via Ant command-line operations. Internally, these commands use the S3 wrappers to accomplish file transfers. An example command-line that transfers all job runspecs and databases for a Batch is:

```
call ant -Dcmd="uploadjobs batchdir \"C:\MYBATCHNAME\" jobbucket MYJOBBUCKET" job
```

To track the status of jobs, our Job commands manipulate a number of temporary status files in each Batch and Job directory. These files are used to track the state of a Job, preventing duplicate uploads or duplicate queue entries. This mechanism drastically reduces file transfer overhead with the cloud and allows controlling scripts to be restarted in the case of an error (as can occur during long weekend runs when local networks undergo maintenance).

The Job commands available are:

- createqueue queue <queuename> timeoutminutes <defaulttimeoutminutes>
- deletequeue queue <queuename>
- flushqueue queue <queuename>
- listqueues
- addstatus queue <queuename> status <statusmessage>
- getstatus queue <queuename> file <statuslogfile>
- waitforjob queue <queuename> command <commandtorun>
- getjob queue <queuename> command <commandtorun>completejob queue <queuename> jobid <jobid>
- addjob jobqueue <jobqueuename> statusqueue <statusqueuename> databasebucket <dbbucket> database <dbname> codebucket <codebucket> code <codename> jobbucket <jobbucket> jobdir <jobdirectory>
  - Does nothing if the job folder contains status.queued file.
  - Does nothing if the job's folder already contains results*.jar, *.log, *.txt files, and/or an output/ folder. (like downloadjobresults)
  - Complains if the job folder is missing status.uploaded file.

    - Before enqueuing the job, deletes results_<batch>_<job>.jar if it exists from the bucket.
    - After enqueuing the job, creates status.queued file.
- addjobs jobqueue <jobqueuename> statusqueue <statusqueuename> databasebucket <dbbucket> database <dbname> codebucket <codebucket> code <codename> jobbucket <jobbucket> batchdir <batchdirectory>
    - Adds all jobs in a batch using the addjob command, but only for jobs not already queued.
- readdjobs jobqueue <jobqueuename> statusqueue <statusqueuename> databasebucket <dbbucket> database <dbname> codebucket <codebucket> code <codename> jobbucket <jobbucket> batchdir <batchdirectory>
    - Adds all jobs in a batch using the addjob command, clearing any prior status.queued file for jobs not done.
- jarjob jobdir <jobdirectory>
- uploadjob jobdir <jobdirectory> jobbucket <jobbucket>
    - creates status.uploaded file in the job folder.
- jarjobs batchdir <batchdirectory>
    - Jars each job in a batch.
    - Creates a jar holding the batch's databases/ folder, unless that jar already exists, called databases_<batch>.jar in the batch's databases/
    - Creates a jar holding the shared databases_<scenario>/ folder, unless the jar already exists, called databases_<scenario>.jar in the shared databases_<scenario>/ folder
- uploadjobs batchdir <batchdirectory> jobbucket <jobbucket>
    - Uploads each job's jar
    - Uploads the batch's database jar
    - Uploads the scenario's shared database jar
- jarjobresults jobdir <jobdirectory>
    - Creates results_<jobdirectory>.jar expecting <jobdirectory> to be <batch>_<job>
    - Includes *.txt, *.log files and output/ folder.
- uploadjobresults jobdir <jobdirectory> jobbucket <jobbucket>
- removejobjars jobdir <jobdirectory> jobbucket <jobbucket>
    - Removes <batch>_<job>.jar and results_<batch>_<job>.jar files if they exist.
    - Creates status.cleaned file in the job folder.
- downloadjobresults jobdir <jobdirectory> jobbucket <jobbucket>
    - Does nothing if the job's folder already contains results*.jar, *.log, *.txt files, and/or an output/ folder.
    - Checks <jobbucket> for results_<batch>_<job>.jar file, downloading it if available.
    - After the download, removes the job's jar files from the bucket using removejobjars.
    - Creates status.done file in the job folder.
- downloadresults batchdir <batchdirectory> jobbucket <jobbucket>
    - Does downloadjobresults on each job folder in a batch.
- downloadjobdbresults jobdir <jobdirectory> jobbucket <jobbucket>
    - Does nothing if the job's folder already contains db_results*.jar and/or an output/ folder.
    - Checks <jobbucket> for db_results_<batch>_<job>.jar file, downloading it if available.
    - After the download, does not delete files from the bucket.
- downloaddbresults batchdir <batchdirectory> jobbucket <jobbucket>
    - Does downloadjobdbresults on each job folder in a batch.
- batchstatus batchdir <batchdirectory>
    - Create [a new] <batch>_status.csv file in <batchdirectory>
    - Checks each job directory for status.* files [.uploaded, .queued, .done, .cleaned]
- terminate instanceid <instanceid>
    - Terminate an EC2 instance

- addpostprocess jobqueue <jobqueuename> statusqueue <statusqueuename> codebucket <codebucket> code <codename> jobbucket <jobbucket> batchdir <batchdirectory>
    - Post process all jobs in a batch.
- getpostprocess queue <queuename> command <commandtorun> todir <downloaddirectory>
    - Execute <commandtorun>.
- downloadpostresults batchdir <batchdirectory> jobbucket <jobbucket>
    - Downloads a batch's post processing result file.
- downloadallpostresults todir <downloaddirectory> jobbucket <jobbucket>
    - Download post processing results for all batches.
- splitresults fromdir <downloaddirectory> todir <processeddirectory>
    - Split a results*.jar file into one with only logs and db_results*.jar with the output databases.
- scanlogs logqueuedir <logqueuedirectory>
    - Read all log files for errors, renaming them as OK.* or Failed.*

CREATING AMAZON MACHINE IMAGES (AMIS)

Our computer instances on Amazon are Linux machines specially setup with:

- bootup scripts to retrieve a Batch command from a queue.
- scripts to download, execute, and upload the results of all Jobs in a Batch.
- MySQL.
- Perl.
- Java.

Note the lack of MOVES itself on the Amazon Linux machines. Our implementation treats the MOVES code and default databases as installable items that are specified with each queued batch. As such, we can use multiple versions of MOVES in the cloud. This feature eases the tasks of obtaining results from historical versions of MOVES and testing new MOVES versions.

Creating these instances requires special steps. The Amazon default instances come with hard drives that are too small for MOVES (8GB by default). MOVES has run well with 24GB of storage, though some rare batches with huge numbers of jobs have needed 200GB of storage.

To facilitate a variety of users making images, we have created a zip file with all of the required Linux shell scripts and installation packages and placed it into an Amazon bucket, which we name the "moves-ami" bucket. Creating a new AMI requires the following steps.

Instructions for creating a MOVES AMI

1) Download onto your local machine and unzip the machine image zip file from the "moves-ami" bucket. This zip file contains the required Linux shell scripts as well as installation packages for MySQL, Java, and Perl.
2) Install the Amazon Web Services (AWS) command line tools onto your local machine.[8] These are needed only for the next steps that create an instance with a non-standard size hard drive.
3) Install Putty and WinSCP, or equivalent software for Linux SSH shell access and file transfers, onto your local machine.
4) Using the Amazon GUI, locate the ID of the AWS Security Group your instances will be using. An example group ID is "sg-ba8419d3".
5) Using the Amazon GUI, locate the name of the AWS security key your instances will be using. An example security key name is "EpaCloud2".

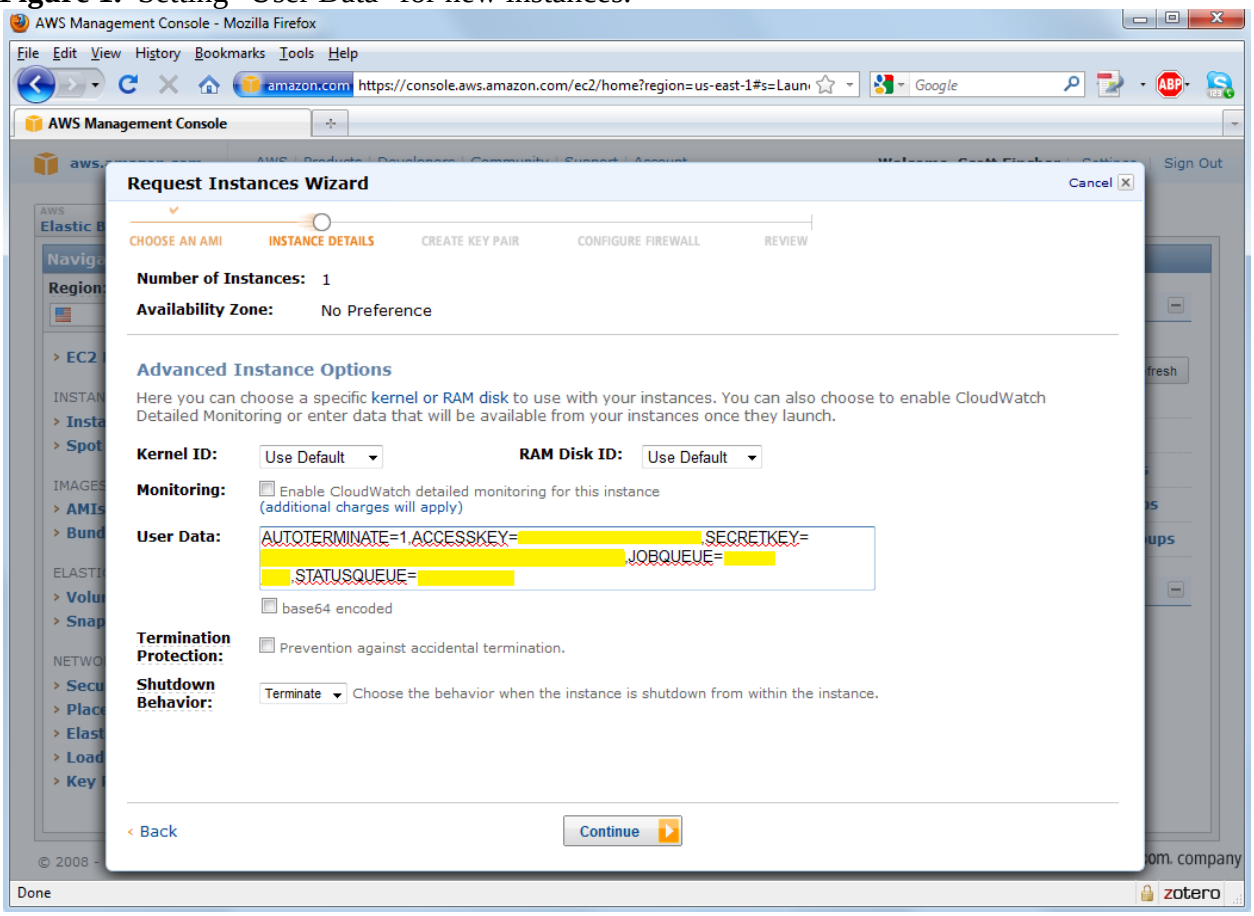
6) Using the Amazon GUI, locate the ID of an existing AWS AMI image that will be used. This should be a standard AMI offered by Amazon to start up a new image using its GUI tools. An example AMI image ID for the c1.medium machine type is "ami-31814f58". Note that Amazon sometimes changes its standard AMIs, changing their IDs. As such, scripts used today are not guaranteed to reference available AMIs in the future.
7) Decide upon the size of the virtual hard drive your instance will use. You are billed by the GB-hour while these virtual drives exist. Batches with many runs need larger hard drives, especially when used for post processing. Amazon limits most accounts to 2 TB of active drives, meaning you need smaller hard drives if you need more instances. MOVES has run well with 24GB of space but some runs have required 200GB.
8) Using the AWS command line tools on your local machine, start an instance with the custom hard drive size. An example command line using the example security group, key name, c1.medium machine type, a 24GB hard drive, and the "ami-31814f58" is:
```
ec2-run-instances --group sg-ba8419d3 --key EpaCloud2 --instance-type c1.medium --
instance-initiated-shutdown-behavior stop --block-device-mapping "/dev/sda1=:24" ami-
31814f58
```
9) Using the Amazon GUI, locate your running instance.
10) Open WinSCP and Putty connections to the new instance.
11) Login to the Putty terminal as "ec2-user".
12) Use the full size of the virtual hard drive by running the following commands at the instance's Linux command line:
```
sudo resize2fs /dev/sda1
df -h
```
13) Create the required MOVES directory structure by running the following commands at the instance's command line:
```
sudo mkdir /home/moves
sudo chmod a+rwx /home/moves
cd /home/moves
```
14) Use WinSCP to copy the image's files to the /home/moves directory. Among other things, this should yield this file:
    /home/moves/my.cnf
    and this directory:
    /home/moves/amazon/
15) Putty will have likely timed out, so create a new connection if it does. Run the following commands at the instance's command line:
```
cd /home/moves
sudo chmod ug+x setupinstance32.sh
sudo chmod ug+x setupinstance64.sh
```
16) If your chosen operating system is 32-bit, run this command:
```
/setupinstance32.sh
```
17) If your chosen operating system is 64-bit, run this command:
```
/setupinstance64.sh
```
During the installation process, you'll be prompted by the installers for MySQL and Perl. MySQL is installed first. When prompted for the MySQL password, enter:
```
root
```
18) then enter these commands at the MySQL command prompt (note the trailing semicolon required on MySQL commands):
```
grant all privileges on *.* to ''@'localhost' with grant option;
flush privileges;
exit;
```
19) Press ENTER when prompted.
20) ActivePerl is installed next. When prompted about reading the license file, enter:
```
yes
```
21) Agree to the license agreement:
```
yes
```
22) ENTER to accept the default directory.
23) Decline installation of HTML docs:
```
no
```

24) When prompted to proceed:
```
yes
```
25) Logout of Putty and WinSCP.
26) Return to the Amazon Management Console.
27) Stop (not "Terminate") the instance.
28) Use the Instance Action "Create Image (EBS AMI)".
29) Provide an AMI image name and description. It is a good practice to include the date, the bit size (32 or 64), and the Java version number.
30) Create the image and wait for it to become available. It is sometimes necessary to leave the AMI page and return to refresh the AMI status.
31) After the new AMI is available, Terminate your instance. This is done to stop Amazon billing on the instance. Do not terminate the instance until the AMI is available.

LAUNCHING AMAZON INSTANCES

Amazon provides both a GUI and command line tools to start instances from our AMIs. Either is sufficient to use MOVES in the cloud. The instances must be informed of the task queue to monitor as well their mode of operation. This information is provided in the "User Data" field that is passed by Amazon to each instance as it boots.

Figure 1. Setting "User Data" for new instances.



In the "User Data" section, paste in text of the form:

For inventory or rate runs without post processing:

AUTOTERMINATE=1,ACCESSKEY=*MYACCESSKEY*,SECRETKEY=*MYSECRETKEY*,JOBQUEUE=*JOBQUEUENAME*,STATUSQUEUE=*STATSQUEUENAME*,SEPARATERESULTS=0

For inventory or rate runs with post processing:

AUTOTERMINATE=1,ACCESSKEY=*MYACCESSKEY*,SECRETKEY=*MYSECRETKEY*,JOBQUEUE=*JOBQUEUENAME*,STATUSQUEUE=*STATSQUEUENAME*,SEPARATERESULTS=1

For post processing:

AUTOTERMINATE=1,ACCESSKEY=*MYACCESSKEY*,SECRETKEY=*MYSECRETKEY*,JOBQUEUE=*JOBQUEUENAME*,STATUSQUEUE=*STATSQUEUENAME*,JOBCOMMAND=batchpostprocess,SEPARATERESULTS=0

Enter as one line (though it will wrap around on screen) with no spaces anywhere and no trailing line ending (carriage return or line feed).

Without this user data, the instance will run (and be billed) until manually terminated without doing any work.

The "AUTOTERMINATE" flag tells the instance to shutdown and cease billable operation after it has processed all jobs from one batch. With AUTOTERMINATE set to 1, the instance will wait to find a batch command, automatically terminating after processing the batch or after 45 minutes of waiting for a batch to be queued.

Setting AUTOTERMINATE to 0 will leave the instance running indefinitely and will allow it to process multiple batches. Be cautious with this mode as there is no easy mechanism to determine if an instance is truly idle and can be safely manually terminated. It is recommended this mode be used with a queue of a single batch during debugging operations, perhaps with a separate debugging queue.

USER'S-EYE VIEW—SETTING UP AND EXECUTING A MOVES RUN ON THE CLOUD

1) Create a list of representing counties (rep counties) and fuel months.
2) The list of rep counties and fuel months are inputs for met4moves (one of several SMOKE-MOVES scripts). The output of met4moves is the meteorological input to the runspec_generator.
3) Run repcnties.plx to create the runspec input to the runspec_generator. (The "plx" extension indicates a Perl script.)
4) Run the runspec_generator—this is a SMOKE-MOVES Perl script from OAQPS, which creates runspecs and zonemonthhour tables in csv format.
5) Run loadZmh.plx to convert zonemonthhour tables from csv to MySQL.
6) Run CreateAndPopulateAmazonDirStructureSmokeMoves.plx to create a local set of folders, organized in a specific way, containing all the MOVES runspecs and databases needed for all the MOVES runs. This script also edits the runspecs in a variety of ways, specifying the county database as the scaleinputdatabase, the zonemonthhour and possibly other databases, such as fuels and VMT, as databaseselection databases. A script is needed because, for SMOKE-MOVES, there are typically 200 batches containing 100 jobs each, each job with its own zonemonthhour database.
7) Run CreateBatchFilesForAmazonRunsSmokeMoves.plx to create a set of "bat" files that are run locally to call our Amazon interface software, typically once for each county-month batch. This

Perl script inserts the correct parameters into the command for each batch. The parameters are listed in the "MOVES Amazon Job API" section above.

8) Log in to aws.amazon.com and create a "bucket," which is like a giant folder.
9) Run "createqueues.bat" to call our interface "createqueues," which creates the four queues required: one queue for the MOVES batches, one for the post-processing batches, one to retrieve diagnostic messages from the MOVES batches, and one to retrieve diagnostic messages from the post-processing batches. The CreateBatchFilesForAmazonRunsSmokeMoves.plx script inserts the correct parameters into this batch file.
10) Jar the MOVES code and transfer to the bucket.
11) Jar the MOVES default database and transfer to the bucket.
12) Jar the postprocess code and transfer to the bucket.
13) Uploadjobs.bat copies the JAR'd databases and jobs from the local directory to the bucket.
14) Addjobs.bat puts one message for each batch into the Amazon jobs queue.
15) Start instances to execute MOVES, one instance for each batch. Start instances by logging on to aws.amazon.com, selecting the image that you want, specifying the number of instances you want to start, and providing additional text information, such as the name of the queue. The required text string (see LAUNCHING AMAZON INSTANCES above) is produced by CreateBatchFilesForAmazonRunsSmokeMoves.plx in a text file which can be copied and pasted.
16) Downloadresults.bat downloads only the log files from the MOVES runs, which are analyzed locally for errors that might require one or more jobs to be rerun. The MOVES output databases are left in the bucket for post-processing. We do not usually run downloaddbresults, which will download the MOVES output databases, leaving copies in the cloud for post-processing.
17) Run addpostjobs.bat, if post-processing for SMOKE-MOVES. This bat file puts one message for each batch into the Amazon postjobs queue.
18) Start instances to execute postprocess, similar to Step 15 above.
19) Downloadpostresults. This command downloads SMOKE-ready emission factor files into the correct batch-specific locations on the local drive.
20) Notify OAQPS to downloadallpostresults. This command downloads all of the SMOKE-ready emission factor files into a directory of the user's choice. OAQPS runs the final SMOKE-MOVES script, which uses these emission factor files to generate grid-cell-hour inventories for air quality modeling.

RUN TIMING AND STATISTICS FOR SAMPLE RUNS WE HAVE DONE AT EPA

Time to run processes for napa2008, a run with 292 batches and 27,013 jobs, cdc2009, a run with 292 batches and 27,501 jobs (average 94 jobs/batch) are listed in Table 1. The ranges for local processes are clearly not related to differences in jobs, but to machines and network variables that can be unpredictable. A lot of processes have to go across network drives. And even on local machines, different levels of activity (e.g., patching, scans) can make a big difference. The batches run simultaneously in the cloud, so the time to complete the run is the time of the longest batch, which is 44 hours for the example shown below.

Table 1. Time to run processes for two example runs: napa2008 and cdc2009. Blanks indicate that timing data was missing.

Process	**Time (hours)**	
	napa2008	**cdc2009**
Loadzmh.plx	2.75	6.35
CreateAndPopulate*.plx	5.25	9.77
Uploadjobs	5.28	5.51
Addjobs	0.71	

Shortest Batch (42 jobs)		13.91
Longest Batch (167 jobs)		44.16
Downloadresults		13.14
Downloadpostresults		7.65
Batchstatus	0.15	0.18

Approximate cost for cloud processing: $0.17/hr * 32hr * 292 batches = $1,588, where 32 hours is the average run time per batch, including post-processing.

If we were to run all counties and months (i.e., not using representing counties and fuel months), we estimate the average time for a batch would be 14 hours, rather than 32, because the range of temperatures would be less for each batch. Using the 292 instances we used for the example runs above, it would take about 75 days to complete the run. If all the batches were run sequentially on a single machine, it would take about 60 years. Instead, the whole run takes about two days.

SUMMARY AND CONCLUSIONS

Cloud computing has been successfully employed to generate MOVES inventories and the lookup tables needed by SMOKE-MOVES. Runs that would take months or years if done without representing counties or on a single computer are completed within days due to parallel processing and the use of representing counties and fuel months. Mobile-source inventory generation is an ideal application for cloud computing, because the calculations for each geographic unit, time period, and vehicle class are independent and can therefore be run on separate computers that do not interact. We have been able to do large numbers of MOVES runs that would have been prohibitively time consuming if we had used our local distributed computing network. Software to upload, execute, post-process, and download MOVES jobs has been developed using Ant and the Java interface to the Amazon API. Perl scripts have been used to write the runspecs and create the local structure needed by the Java interface code. Note that cloud computing vendors differ in their interfaces, so what we have done is not universally directly applicable. But, for those interested in running MOVES in the cloud, our experience provides a potentially useful picture of the processes, complexities, pitfalls, and rewards of this effort.

DISCLAIMER

The content provided represents the work of the authors and does not reflect the policy, guidance, or procedures recommended by the U.S. EPA. This document is disseminated in the interest of information exchange and the U.S. Government assumes no liability for use of the information. This paper does not constitute a standard specification, regulation, or regulatory finding. The mention of specific commercial products and services does not constitute an endorsement by the U.S. EPA.

REFERENCES

[1] Detailed information about MOVES is available at http://www.epa.gov/otaq/models/moves/index.htm.

[2] For information on SMOKE and SMOKE-MOVES, see http://www.smoke-model.org/index.cfm.

[3] For information on the Amazon Elastic Compute Cloud (EC2), see http://aws.amazon.com/ec2/.

[4] For information on the Amazon Simple Storage Service (Amazon S3), see http://aws.amazon.com/s3/.

[5] For information on the Amazon Simple Queue Service (Amazon SQS), see http://aws.amazon.com/sqs/.

[6] For information on the Amazon Linux AMI, see http://aws.amazon.com/amazon-linux-ami/.

[7] For information on Ant, see http://ant.apache.org/.

[8] For Amazon Web Services (AWS) command line tools for EC2, see http://aws.amazon.com/developertools/351/

KEY WORDS

MOVES
SMOKE
SMOKE-MOVES
Cloud computing
Amazon
EC2
Onroad emission inventories
Mobile sources