CRC Report No. A-110

SCOPING STUDY FOR REWRITE OF MOVES FOR EFFICIENCY

Final Report

August 2017

COORDINATING RESEARCH COUNCIL, INC.
5755 NORTH POINT PARKWAY • SUITE 265 • ALPHARETTA, GA 30022

The Coordinating Research Council, Inc. (CRC) is a non-profit corporation supported by the petroleum and automotive equipment industries. CRC operates through the committees made up of technical experts from industry and government who voluntarily participate. The four main areas of research within CRC are: air pollution (atmospheric and engineering studies); aviation fuels, lubricants, and equipment performance, heavy-duty vehicle fuels, lubricants, and equipment performance (e.g., diesel trucks); and light-duty vehicle fuels, lubricants, and equipment performance (e.g., passenger cars). CRC's function is to provide the mechanism for joint research conducted by the two industries that will help in determining the optimum combination of petroleum products and automotive equipment. CRC's work is limited to research that is mutually beneficial to the two industries involved. The final results of the research conducted by, or under the auspices of, CRC are available to the public.

Eastern Research Group, Inc.

Scoping Study for Rewrite of MOVES for Efficiency

FINAL REPORT (Version 2)

Prepared for:

Coordinating Research Council

Prepared by:

Eastern Research Group, Inc.

August 31, 2017

ERG Project No.: 4082.00.001.001

Scoping Study for Rewrite of MOVES for Efficiency

FINAL REPORT (Version 2)

Prepared for:

Coordinating Research Council

Prepared by:

Doug Jackson
Wes Faler (Fluid & Reason LLC)
John Koupal
Scott Fincher

Eastern Research Group, Inc.
3508 Far West Blvd., Suite 210
Austin, TX 78731

August 31, 2017

Table of Contents

Tables

Figures

Acronyms and terms

AMI	Amazon Machine Image providing an operating system and programs for a cloud computer
AWS	Amazon Web Services cloud platform
bucket	a storage location in S3, similar to a hard drive for an AWS cloud computer
CDB	MOVES county database
CMIT	MOVES Core Model Input Table
CPU	central processing unit
disk I/O	input/output: transfer of data to and from a hard drive
EC2	Amazon Elastic Cloud Compute providing virtual computers in the AWS cloud
Go	a programming language created by Google
GUI	Graphical User Interface: an application window with buttons, menus, etc. to interface with a computer program
instance	a single virtual computer running in the cloud
JAR	Java ARchive file used to store Java code in a single, sharable file
multithreading	parallel/simultaneous execution of multiple calculations, possible on modern computers with multiple processors/cores
MySQL	An open source database management system developed by Oracle Corporation
parallelization	running computer processes simultaneously, as opposed to in series, to reduce execution time (clock time)
RAM	Random Access Memory: high speed, short-term memory used to store a computer's active programs and data
runspec	a MOVES configuration file containing all of the settings for a single MOVES run
S3	Simple Store Service providing storage space for AWS cloud computers
SDK	Software Development Kit: a set of software tools to create programs for specific applications, e.g., to interface with AWS
SIP	State Implementation Plan
SQL	Structured Query Language used for programming database operations and calculations
SQS	Simple Queue Service used to send messages to and from an instance in the AWS cloud
VMT	Vehicle Miles Traveled
VSP	Vehicle Specific Power

1.0 Introduction

This report documents the CRC-sponsored *Scoping Study for Rewrite of MOVES for Efficiency* performed by ERG in collaboration Wes Faler, president and founder of Fluid & Reason LLC. We describe our evaluation and plan for implementing three separate approaches to improving the efficiency of the U.S. EPA's MOtor Vehicle Emission Simulator (MOVES): 1) recoding the model; 2) taking advantage of cloud resources to a larger degree while making user interaction with the cloud simple and direct; and 3) automating application of advanced performance features that already exist in MOVES.

Our approach for this project is to consider how to improve the speed of MOVES at the national or county scale without sacrificing functionality (runtime is less of an issue for project scale). MOVES is designed with flexibility to allow users to customize inputs based on local data. While MOVES has default information to enable modeling for the entire U.S., including meteorology, fuel properties, I/M program parameters, and vehicle activity, EPA modeling guidance for state implementation plans (SIP) and transportation conformity recommends use of local data where available. To maintain this feature of MOVES, performance improvement cannot come at the expense of model functionality. For example, approaches which pre-generate MOVES emissions rates into lookup tables and couple these with VMT estimates can produce mass emission estimates very quickly, but only for the specific set of inputs used to generate the rates. Any change in fuel properties, I/M program parameters, etc. could not be accommodated without first regenerating emission rates, thus negating the performance improvements of a rate lookup table approach. Our approach seeks similar reductions in runtime from a rate lookup approach, while retaining MOVES' flexibility to accept alternate user inputs.

This project consisted of four tasks, structured around the three approaches listed above plus development of a scoping plan. Under Task 1, Mr. Faler led an evaluation of the potential to recode the model. Under Task 2, ERG and Mr. Faler describe a Runspec Execution Utility that could be used to execute any arbitrary MOVES runspec(s) in the cloud and a Runspec Creation Utility that would facilitate the batch creation of runspecs. Task 2 also includes a Worker Cloud Gateway that, with some changes to the MOVES architecture, would enable the deployment of large numbers of workers in the cloud to process bundles generated by a local master. Under Task 3, we evaluate the potential for automated application of Advanced Performance Features to eliminate redundant processing through storage and re-use of common intermediate and "core model input" tables (CMITs). Finally, Task 4 is a scoping document which includes the results of Tasks 1-3 along with a scoping plan and projected budget for each element.

By design, each of the elements developed in Tasks 1-3 can be pursued independently of the others. For short term improvements, Tasks (2) and (3) could be pursued without changes to the current MOVES 2014a model, with the exception of the Worker Cloud Gateway (meant to optimize cloud application of a MOVES version that adopted the recoding recommendations of Task 1). This approach provides more flexibility for EPA and the user community to begin implementing performance improvements that do not immediately require the EPA to modify the code. Ultimately, our vision is that all three approaches could be implemented in parallel to maximize the speed of the model.

The improvements identified in Task 1, including converting key generators and calculators to the Go language and speeding up distributed processing by 50%, are estimated to provide an

87% reduction in MOVES runtime; assuming the theoretical limit of completely eliminating distributed processing would result in a 92% reduction. The cloud interface utilities described in Task 2 would reduce clock time in proportion to the number of Amazon AWS instances available to the user – on the order of hundreds of virtual machines – with some overhead required to transfer data to and from the cloud. Cloud-based utilities would also entail some cost for renting AWS computing resources, though these costs would be relatively small in the context of a typical MOVES project. Bypassing key generators for a typical MOVES 2014a run, as described in Task 3, would reduce total runtime by up to 45%. These are not completely compounding – for example, Task 1 recoding would obviate most if not all of the Task 3 reduction, which is based on inefficiencies in the current code. However, if taken together we think it is plausible to reduce current MOVES runtime by over 95 percent.

Details on the three approaches are provided in Section 2.0 (recoding), Section 3.0 (cloud) and Section 4.0 (eliminating redundancy). Section 5.0 provides a broad scope for the work required to implement these options.

2.0 Task 1: Evaluate Recoding for Maximum Processing Speed

Disk operations to transfer data are several thousand times slower than in-memory computations. MOVES is slow primarily because it has too much disk input/output (I/O). While an appropriate design choice at the inception of MOVES, modern computing demands a different approach.

It is important to realize how MOVES uses its Java and MySQL components. The Java components do almost no computation. There is a persistent and common misconception that MOVES uses Java to calculate results. Instead, Java is used to build and coordinate the execution of MySQL statements, meaning that MOVES uses MySQL to perform its computations, not just to store inputs and results. Most smart watches have more computing power than is needed to execute the Java elements of MOVES.

Thinking of an equation as a series of steps, MOVES' use of MySQL to perform computation means all pollutant calculations must go through each step before any can go through a subsequent step. The problem is that the large number of pollution records cannot fit into memory, so they must be streamed from disk, to memory, perform a single calculation step, then written back to disk. This is repeated for each step, with tremendous disk I/O overhead. Further, as disk I/O is so slow relative to central processing unit (CPU) speed, increasing the number CPU cores available to perform a step does not improve performance because the disk is the bottleneck.

Contrast the disk approach to a CPU-centric one. A CPU-centric approach creates a pollution record in memory and then performs most, if not all, computation steps on the record before writing it to disk for storage. As simple as it seems, the difficulty comes with the need to know emission factors, fuel adjustments, temperature coefficients, and myriad other equation terms that must be available to complete the calculations. Pulling this data from disk on demand would defeat the CPU benefits. Rather, this data must be kept in memory in an efficient data structure. Recent widespread availability of gigabyte-class computers enables in-memory storage of these factors, finally allowing the MOVES architecture to be changed from the initial design structure that dates back to the early 2000s. A CPU-centric approach performs no more CPU instructions

on a pollution record than the disk approach while performing an order of magnitude fewer disk operations, making the CPU-centric approach inherently faster.

EPA has already begun converting slow disk-centric portions of code into CPU-centric modules. These have included both calculators, run on workers, and generators, which are run on the master nodes. EPA selected the Go language for these modules. A trade study was performed, recreating a bottleneck module in Go, C++, Java, and several versions of Python. ***All used multithreading to make maximum use of CPU resources.*** C++ had the best performance at 20 times faster than MySQL, required the most effort to create, and had the highest expected cost of ownership. Java and Python offered similar performance, roughly 5 times faster than the MySQL computations, with Python having low creation effort and the lowest cost of ownership. The Go language module, popularized by Google and used extensively in Silicon Valley, had performance within a few percent of C++, the same low effort as Python, and midrange cost of ownership.

The rapid pace of technology adoption means a closer look at the economics of the Go programming language is in order. From a study of computing language popularity on the Internet, especially among open source projects, we find Go declared "Programming Language of the Year":

Jan 2017	Jan 2016	Change	Programming Language	Ratings	Change
1	1		Java	17.278%	-4.19%
2	2		C	9.349%	-6.69%
3	3		C++	6.301%	-0.61%
4	4		C#	4.039%	-0.67%
5	5		Python	3.465%	-0.39%
6	7	^	Visual Basic .NET	2.960%	+0.38%
7	8	^	JavaScript	2.850%	+0.29%
8	11	^	Perl	2.750%	+0.91%
9	9		Assembly language	2.701%	+0.61%
10	6	vv	PHP	2.564%	-0.14%
11	12	^	Delphi/Object Pascal	2.561%	+0.78%
12	10	v	Ruby	2.546%	+0.50%
13	54	^^	Go	2.325%	+2.16%
14	14		Swift	1.932%	+0.57%
15	13	v	Visual Basic	1.912%	+0.23%
16	19	^	R	1.787%	+0.73%
17	26	^^	Dart	1.720%	+0.95%

Source: http://www.zdnet.com/article/googles-go-beats-java-c-python-to-programming-language-of-the-year-crown/

MOVES' Java language continues to be the most popular language, 3 times more popular than C++. From EPA's actual testing, multithreaded Java can perform calculations 5 times faster than the current SQL-based algorithms. Python remains quite popular. EPA's actual measurements of multithreaded Python showed it performing on par with multithreaded Java, about 5 times faster than SQL.

The Go language is now more popular than Swift, which is used for iPhone apps, and Visual Basic, which is incredibly common for corporate software developers. The rising popularity reduces concerns about technology lock-in. It is common to perceive Java and C++ staff availability as equal, despite C++ being only 1/3 as popular as Java. With this new data, it is now more true to say that C++ and Go are closer in popularity than C++ and Java.

The only language more popular than Go that is faster than Java is C++. EPA's actual measurements showed multithreaded C++ running 20 times faster than SQL, about 4 times faster than Java or Python. This is certainly within the speedup range desired for any rewrite of MOVES.

Multithreaded Go's measured performance is within a few percent of C++'s, 20 times faster than SQL and in the desired speedup range. Level of effort, including time to debug multithreaded systems, differentiates Go from C++. Level of effort for code creation depends greatly upon program length. C++ and Java required about the same number of lines of code in the EPA study. Python and Go required only 1/3 this number.

With a Go developer salary average of $97,500/year (https://remoteok.io/remote-golang-jobs) and a C++ average salary of $115,575/year (https://www.indeed.com/salaries/C++-Developer-Salaries), we can expect a C++ MOVES module to cost 3.6 times more than a Go module (3x lines of code * C++ salary/Go salary). Go has equivalent measured performance (for MOVES' purposes), lower cost of ownership, and rapidly rising popularity in the labor pool.

As part of this pilot study, many chained calculators were rewritten in Go, moving the bottleneck from workers to the master node. Generators on the critical path are being rewritten as well. A nondisclosure agreement prevents release of further details. EPA may release a version of MOVES with most, if not all, calculators and critical-path generators rewritten in Go.

Commercial database systems, such as Microsoft's SQL Server, use multiple threads to service a single SQL command. MySQL, however, is single threaded. A single SQL command in MySQL will fully occupy exactly one thread, and a full CPU core when not disk I/O limited. The traditional MOVES framework views generators and calculators as long sequences of SQL statements, with generators having some Java code used only for decision making and no actual computations. As such, the traditional framework confines each generator and calculator to a single thread.

The models within MOVES's generators cause dependencies that nearly eliminate the potential to execute multiple generators concurrently. Thus, the MOVES Master is essentially single threaded as all of the SQL commands it issues are done sequentially and MySQL uses but a single CPU core to service each.

A MOVES Worker performs all the SQL statements for multiple calculators, but all the SQL statements for the calculators are done in a flat sequential list without concurrency. To get

concurrency, multiple worker instances can be run on the same computer. This practice will consume more of the available CPU resources only until the machine's disk I/O bandwidth is reached. At that point, adding additional worker instances no longer increases CPU usage. Instead, CPU usage plummets as I/O latency climbs in the oversubscribed disk system. The tipping point is often at 4 workers, making the total combination of a master and 4 workers consume roughly half the CPU resources of a single quad core processor (i.e. a typical i5 or i7 CPU found on modern desktop computers).

Designed specifically to get past this limit, generators and calculators written in Go are aggressively multithreaded and routinely consume 100% of CPU clock cycles even for multi-processor, multi-core machines. The Go-based calculators and generators have been tested on 32 core, quad processor computers (each computer with 4 processors, each processor with 4 hyperthreaded cores (i.e. 2 logical cores per physical core), totaling 32 CPU cores on a single computer). The Go code has a pipeline design that completely eliminates disk I/O between any step in the calculation of a pollutant or generation of activity data. Both calculator and generator Go frameworks have pipeline designs. The design allowed a single program instance to consume the full resources of all CPU cores. The implementation has sufficient in-memory buffering to eliminate the disk write bottleneck by performing disk writes in optimally sized segments. A separate study was performed to find the optimal amount of data to write to disk in each disk I/O event.

However, the release could still be improved upon. Improvement will come from integrating multiple generators and even calculators into a single CPU-centric program, completely eliminating disk storage of intermediate results. The new Go-based generators, while multithreaded and pipelined to avoid disk I/O between generators, still must store their results to disk for later access by pollution calculators, themselves multithreaded and pipelined. To get maximum performance, the activity information created by generator algorithms must be accessible by pollution calculations without intermediate disk operations. This must occur regardless of the use of local or distributed processing. Where there are now two separate Go-based programs, one for generators and one for calculators, a single program doing both in a single instance would eliminate the disk operations bottleneck between the two.

In the extreme, the nature of MOVES' bundles will be changed, moving all generators to the workers along with the calculators. In this circumstance, the notion of "bundle" would degenerate to a short high-level description of place and time along with a list of modules to be evaluated. Such a short notion of bundle lends itself very well to entirely local processing. It could also lend itself well to distributed computing (either via a local network or via the cloud). Integration of generators and calculators is made tractable because of the modular method in which the EPA Go code is written, itself a mandate of the MOVES design team. Calculation steps are linked together in memory using Go's "channels", a unique language feature that drastically simplifies multithreading, while driving modularity and decreasing expensive common multithreading code defects. The released modules can be connected in a new channel pipeline that maximizes computation and minimizes disk I/O. This makes the rewrite much more of an administrative coding task, subject to much less risk and testing costs than a wholesale model rewrite but with similar benefits. Changes to MOVES' MasterLoop Java class will be required to iterate using the new bundling scheme. Again, this is more administrative than scientific and lends itself well to unit tests to verify functionality.

To summarize:

- MOVES is slow because it does too much disk I/O. Measurements (see below) will dictate how much disk I/O is due to master/worker distributed computing and how much is due to SQL-based computations.
- Running multiple instances of traditional MOVES masters and workers on a single computer rapidly hits the computer's disk I/O bandwidth limit, underutilizing the computer's CPU and memory resources.
- No amount of reduction in scope of runspecs or other schemes to eliminate master/worker distributed work or eliminate network file shares can get around this limit. It is simply a consequence of the MySQL-based nature of computations that prevents CPU-based streaming of calculations.
- MOVES' generators and calculators should be multithreaded and should perform computations using CPU-centric streaming/pipelining logic outside of SQL systems. The evidence of why they "should" is simply that they already have been converted and the results have been dramatic improvements in speed.
- The improved calculators and generators fully consume the CPU resources of large systems (32 core, 4 processor systems). Thus, multiple instances of the new software should not be run on a typical single computer (pending direct evidence to the contrary since hardware system tuning permits many optimizations).
- MOVES' remaining generators and calculators not already converted to CPU-centric algorithms should be converted and likely will be converted by EPA independent of CRC funding.
- The next major improvement will come by integrating generators and calculators into a single multithreaded CPU-centric program. This will also simplify the use of MOVES on large clusters as it could eliminate the master/worker paradigm.

In the following, we describe the results of a bottleneck analysis used to identify further opportunities to improve the performance of MOVES. Based on this analysis, we recommend a variety of design changes.

2.1 Task 1 measurements

A representative MOVES scenario was simulated. Its performance was categorized to gain insight into the potential for improvement. The code and database version used are the latest EPA internal version and not yet publicly released. This version includes several recent improvements implemented by EPA and already results in several hours of time savings over the publicly available version.

The scope of the scenario is shown in the table below. This run is a typical daily run similar to what might be used for SIP modeling, and is comparable to the test run used in Task 3, so was considered a good example run. The performance improvements may vary depending on the scope of the run.

RunSpec	CRC IO Test
Output/Run	crciotest RunID=1
	National
Geography	Washtenaw County, Michigan
Year	1
Month	1
Daytype	2
Hours	24
Source/Fuel Types	All (30)
Road Types	3 (Off network, 2 Urban types)
Pollutants	27
Processes	7 (Running, Starts, Evap x3, Ext Idle, Aux Power)

Table 2-1. Representative MOVES scenario run specifications

The scenario needed 5,968 seconds to complete. Data from the scenario's BundleTracking output table follows. 5,734 seconds are accounted for in this table. "M" or "W" refer to whether the operating is occurring on Master or Worker.

loopableClassName	hostType	seconds	category	loopableType
TOGSpeciationCalculator	W	1,395	Core	calc
RatesOperatingModeDistributionGenerator	M	1,132	Core	gen
EvaporativePermeationCalculator	W	833	Core	calc
AirToxicsCalculator	W	714	Core	calc
OutputAggregation	W	380	Dist	agg
TankVaporVentingCalculator	W	217	Core	calc
GeneralAggregation	W	194	IO	agg
TankTemperatureGenerator	M	168	Core	gen
BaseRateGenerator	M	120	Core	gen
StartOperatingModeDistributionGenerator	M	95	Core	gen
ExternalCalcRun	W	63	Core	calc
LiquidLeakingCalculator	W	55	Core	calc
BundleResults	W	50	Dist	agg
HCSpeciationCalculator	W	48	Core	calc
ExternalCalcReadResults	W	43	Dist	calc
Other	W	33	Core	agg
BaseRateCalculator	W	31	Core	calc
SourceBinDistributionGenerator	M	31	Core	gen
SulfatePMCalculator	W	29	Core	calc
FuelEffectsGenerator	M	22	Core	gen
BaseRateCalculator	M	19	Dist	calc
ActivityAggregation	W	19	Core	agg
TotalActivityGenerator	M	12	Core	gen
TankVaporVentingCalculator	M	9	Dist	calc
EvaporativeEmissionsOperatingMode DistributionGenerator	M	5	Core	gen
DistanceCalculator	W	4	Core	calc
LiquidLeakingCalculator	M	3	Dist	calc
ExternalCalcWriteInput	W	3	Dist	calc
EvaporativePermeationCalculator	M	2	Dist	calc
TankFuelGenerator	M	2	Core	gen
MeteorologyGenerator	M	1	Core	gen
DistanceCalculator	M	1	Dist	calc
NonroadAggregation	W	0	Core	agg

Table 2-2. BundleTracking output from representative MOVES scenario

The distribution of time spent on the Master and Worker is:

Location	Total Seconds	% of total time
Master	1,623	27%
Worker	4,111	69%
Other	234	4%
Total	5,968	100%

Table 2-3. Distribution of time spent on the Master and Worker in representative MOVES scenario

Bundle Tracking entries are assigned a "Category," one of:

- Core – computations and data movement essential to calculation of pollution and activity.
- Dist – operations related to MOVES' distributed computing architecture. Examples include calculator time that occurs on the master computer as well as aggregation of intermediate results for each bundle.
- IO – disk I/O or SQL-related overhead likely better served with code-centric in-memory operations. From bundle tracking, only the General Aggregation entry is sufficiently narrow that it can be said to be the IO category.

The time breakdown by Category is:

Category	Total Seconds	% of total time
Core	5,030	84%
Dist	510	9%
IO	194	3%
Other	234	4%
Total	5,968	100%

Table 2-4. Time breakdown by category in representative MOVES scenario

Bundle Tracking entries were also assigned a "Loopable Type" designation, one of:

- Agg – consolidation of data, reducing record count, equivalent to the SQL GROUP BY clause.
- Calc – operations related to MOVES' calculator modules.
- Gen – operations related to MOVES' generator modules.

The time breakdown by Loopable Type is:

Loopable Type	Total Seconds	% of total time
Agg	676	11%
Calc	3,470	58%
Gen	1,588	27%
Other	234	4%
Total	5,968	100%

Table 2-5. Time breakdown by Loopable Type in representative MOVES scenario

During the run, MySQL's slow query feature was used. This feature records all SQL statements that require more than 1 second to complete. MOVES' slow query analyzer code was used to consolidate the MySQL output. Representative SQL statements are shown below. The complete list is omitted for brevity.

Total Seconds	Calls	Total Records	Category	SQL
453.3	12	41,419,530	Core	insert into togTemp (MOVESRunID,iterationID,yearID,monthID,dayID,hourID,stateID,countyID,zoneID,linkID,pollutantID, processID,sourceTypeID,regClassID,fuelTypeID,modelYearID,roadTypeID,SCC,emissionQuant,emissionRate,engTechID,sectorID,hpID) select a.MOVESRunID,a.iterationID,a.yearID,a.monthID,a.dayID,a.hourID,a.stateID,a.countyID,a.zoneID,a.linkID, b.outPollutantID as pollutantID, a.processID,a.sourceTypeID,a.regClassID,a.fuelTypeID,a.modelYearID,a.roadTypeID,a.SCC, emissionQuant*factor as emissionQuant, emissionRate*factor as emissionRate, a.engTechID,a.sectorID,a.hpID from TOGWorkerOutputIntegrated a inner join togSpeciationCountyYear b on (a.mechanismID = b.mechanismID and a.integratedSpeciesSetID = b.integratedSpeciesSetID and a.countyID = b.countyID and a.monthID = b.monthID and a.yearID = b.yearID and a.processID = b.inProcessID and a.pollutantID = b.inPollutantID and a.fuelTypeID = b.fuelTypeID and a.modelYearID >= b.minModelYearID and a.modelYearID <= b.maxModelYearID and (a.regClassID = b.regClassID or b.regClassID=0));
337.8	12	42,941,679	Dist	INSERT INTO WorkerOutputTemp (MOVESRunID, iterationID, yearID, monthID, dayID, hourID, pollutantID, stateID, countyID, zoneID, linkID, roadTypeID, processID,

				fuelTypeID, fuelSubTypeID, modelYearID, sourceTypeID, regClassID, SCC, engTechID, sectorID, hpID, emissionQuant, emissionRate) SELECT MOVESRunID, iterationID, yearID, monthID, dayID, hourID, pollutantID, stateID, countyID, null as zoneID, null as linkID, roadTypeID, processID, fuelTypeID, null as fuelSubTypeID, modelYearID, sourceTypeID, null as regClassID, null as SCC, null as engTechID, null as sectorID, null as hpID, SUM(emissionQuant*(case dayID when 2 then 0.5 when 5 then 0.2 else 1 end)) AS emissionQuant, SUM(emissionRate*(case dayID when 2 then 0.5 when 5 then 0.2 else 1 end)) AS emissionRate FROM MOVESWorkerOutput GROUP BY MOVESRunID, iterationID, yearID, monthID, dayID, hourID, pollutantID, stateID, countyID, roadTypeID, processID, fuelTypeID, modelYearID, sourceTypeID;
15.4	1	0	Dist	LOAD DATA INFILE 'C:\\EPA\\MOVES\\MOVESGHGSource\\MOVESTemporary\\DONEProcessing\\BaseRateOutput.tbl' INTO TABLE tempBaseRateOutput;

621.2	38	1,178	IO	INSERT INTO TemperatureAdjustByOpMode (zoneID, monthID, hourDayID, tankTemperatureGroupID, opModeID, polProcessID, fuelTypeID, temperatureAdjustByOpMode, modelYearID) SELECT zoneID, monthID, hourDayID, tankTemperatureGroupID, opModeID, polProcessID, fuelTypeID, tempAdjustTermA*EXP(tempAdjustTermB*averageTankTemperature) AS temperatureAdjustByOpMode, modelYearID FROM AverageTankTemperature INNER JOIN TemperatureAdjustment INNER JOIN ModelYearmy on (modelYearID between minModelYearID and maxModelYearID);

Table 2-6. MySQL slow query output for representative SQL statements

From the slow SQL queries, the time spent in each category is:

Category	Total Seconds	% of total time
Core	1,799	30%
Dist	822	14%
IO	1,561	26%
Other	1,785	30%
Total	5,968	100%

Table 2-7. MySQL slow query output: Time spent by category

A full 30% of MOVES' execution time consists of fast SQL statements and/or non-SQL operations.

Calculators converted to Go typically are 95% faster, executing in only 5% of the time required of the SQL-based version. This has been consistent with the comparisons done when comparing Go, C++, and other languages. From a categorization standpoint, we can view unconverted calculators as 5% "Core" and 95% "I/O". The I/O category is easiest to spot in slow SQL statements, though they only account for 30% of the runtime.

2.2 Task 1 discussion

Division of time by Master and Worker sheds little light on redesign opportunities.

Examining Bundle Tracking data shows that the combination of TOGSpeciationCalculator, RatesOperatingModeDistributionGenerator, and EvaporativePermeationCalculator accounts for 56% of MOVES' runtime.

TOGSpeciationCalculator is presently still using SQL scripts. Based upon experience converting the BaseRateCalculator (the previous slowest calculator) to Go, TOG speciation would likely be much faster when converted to Go. Being the slowest calculator in MOVES, this is likely to be converted by EPA.

RatesOperatingModeDistributionGenerator has already been partially converted to Go in the internal MOVES version that was tested. Its performance is strongly affected by rulemaking-related tables and is the subject of ongoing optimizations.

EvaporativePermeationCalculator uses SQL scripts, accounting for 14% of runtime. It too should be significantly faster after conversion to Go.

For design purposes, we can presume that calculators and generators can and will be converted to Go, achieving 95% speedup of each. This will have the same speedup on the I/O category. When the categorized time is examined under this presumption, we see that runtime is reduced 83% and that distributed processing overhead dominates:

Category	Total Seconds	Speedup	New Seconds	%
Core	5,030	95%	251	25%
Dist	510	0%	510	51%
IO	194	95%	10	1%
Other	234	0%	234	23%
Total	5,968	83%	1,005	

Table 2-8. Speedup by category

Distributed processing arose in initial MOVES design as a way to reduce time-on-the-wall required (clock time) for a scenario. Distributed processing overhead is eliminated in a single computer, single program solution. This solution has no data sharing between multiple programs. Naturally, it extends the time required to arrive at a solution. However, it is likely that there exists a cross over point at which distributed processing provides benefit.

Distributed processing overhead is mostly disk I/O and only improves by performing less of it, ideally only invoking the overhead when doing so has a net benefit. In the improved scenario above, distributed processing is roughly 2:1 the Core time. So, distributing ½ of the bundles to a separate worker computer would break even. Revising the speedup chart to include a 50% speedup (via reduction in usage) of distributed processing gives these timings:

Category	Total Seconds	Speedup	New Seconds	%
Core	5,030	95%	251	25%
Dist	510	50%	255	25%
IO	194	95%	10	1%
Other	234	0%	234	23%
Total	5,968	87%	750	

Table 2-9. Speedup by category with 50% speedup of distributed processing

The total runtime improves a little to 87% reduction. Taking the distributed overhead to zero (i.e. a 100% reduction) would give a total time of 495 seconds, a 92% reduction.

2.3 Task 1 recommended design changes

The bulk of improvements happen when converting calculators and generators from SQL and Java to Go. To maximize this benefit, calculators and generators need to execute in the same multithreaded program without storing data to disk between the two. ***This is a major departure from MOVES' design roots.*** MOVES has always presumed that computation results are slower to obtain than looking them up from a disk file. Recent tests show that modern CPUs perform far faster than their disks. It may be desirable to accept redundant computations rather than disk access.

With recoding to a completely CPU-centric approach, it is possible to completely eliminate the Master/Worker paradigm and eliminate distributed processing from MOVES. This should not be done lightly, however, as the computation burden may still be large enough to benefit from multiple computers, not just multiple processors and cores. Rather, we propose that MOVES should retain the option of Master/Worker processing, and that users determine whether the performance benefit of a Master/Worker paradigm outweighs the cost of data transfer. A long term goal would be to automate the measurement of Master/Worker overhead, maximizing the benefit to the majority of MOVES' users that have little IT background.

One function of Master/Worker approach is to compile and store output data from multiple processing threads in a database program, for ready manipulation. Though a fully CPU-centric approach provides the most performance benefit, it sacrifices this centralized data compilation function, and it is unclear if MOVES users would embrace the chore of manually assembling output from many new-MOVES instances. Most users lack the computer science scripting skills needed for this task, instead focusing on environmental concerns and data quality. These users seem well served by continuing the MOVES "master" paradigm in that, from the user's perspective, they interact with a single instance of a computer program and that program maximizes the utilization of their computing resources. However, sophisticated users should have the option of configuring their computing system as they see fit and should be able to manually choose the use of distributed computing or single-instance runs.

The design changes required are:

- Unify the Go-based external generator and external calculator code into a single program, called "gencalc" in the rest of this document. Continue to make use of Go's "channels" so as to achieve good modularity in the code despite the external appearance of combination.
- Update gencalc to perform the RatesOperatingModeDistributionGenerator, BaseRateGenerator, BaseRateCalculator, and any chained calculators in a single invocation.
- Update gencalc to read and write from the database directly. This mode is needed for local execution. This is a network operation and should not involve disk I/O. The generator code already operates this way and the calculator code needs to be updated.

- Update gencalc to also accept data files instead of the database. This mode is needed for distributed execution. The calculator code already operates this way and the generator code needs to be updated.

- Update the Java-based MasterLoop to use lazy invocation, invoking any calculator predecessor generators only on demand when a calculator is needed. Knowledge of the predecessors can come from gencalc itself or a database table. Generators that are inherent to gencalc can be merely marked as complete and not invoked directly.

- Compute the utility of making a distributed bundle.

- If bundling is called for, create bundles but do not load bundled table files into a database nor retrieve gencalc results from a database. Doing so adds unwanted disk I/O. Rather, gencalc will read the files directly and write its results into a file suitable for use by the master.

- When a bundle is not called for, do not create a bundle file or otherwise do unnecessary disk I/O. Invoking gencalc locally does not require disk I/O since it operates directly from the execution database.

3.0 Task 2: Evaluate Optimizing Multiprocessing: Smart Cloud Interface

As discussed under Task 1, the current MOVES design provides the option for distributed processing to allow calculation of many individual bundles in parallel. Cloud computing is a way to significantly increase the number of processers that can be brought to bear on parallel operations. We have considered how cloud computing could be used for the current configuration, and for updates proposed under Task 1, which would change how multiple parallel processers could best be used to reduce runtime. This provides options for using the cloud to speed up runtime significantly for users of MOVES2014a, and then using cloud capability to further enhance the performance improvements if recoding recommendations made under Task 1 are pursued.

Amazon Web Services (AWS) is a cloud computing platform which provides on-demand, simultaneous access to hundreds or thousands of virtual computers. AWS opens up a tremendous opportunity for speeding up resource-intensive simulation tasks through massive parallelization without the costs associated with setting up and maintaining in-house hardware.

MOVES is well-suited to being deployed on AWS, since many MOVES projects are embarrassingly[1] parallel (separable into individual tasks with little effort) and thus can be readily broken up into individual runs that can be executed independently on separate computers. The potential benefit of this approach has already been demonstrated in practice: for the U.S. EPA, ERG ran the Mexico version of MOVES in AWS to produce annual emission inventories for

[1] Editor note: I'm assured by the author this is a real term

approximately 2,500 *municipios* (akin to counties) in the clock time required for a single county run.

Despite the potential power of AWS for speeding up MOVES runs, this approach has been underutilized to date because of the difficulty involved in setting up, launching, and harvesting results from MOVES runs in AWS. A substantial amount of groundwork has already been done to enable AWS deployment of MOVES, but configuring and launching AWS runs currently requires numerous Perl scripts and batch files that must be custom configured and run from the command line, a complex and daunting process that is accessible only to those with fairly advanced programming and computer skills – far beyond the skills required to run MOVES itself.

The tremendous potential of AWS for MOVES modeling, coupled with the prohibitive complexity currently involved in exploiting this resource, presents a compelling case for the development of user-friendly utilities to bridge the gap between MOVES and AWS.

In the following, we provide a high-level overview of how MOVES runs are currently deployed to AWS. Then, we outline three proposed utilities.

3.1 Overview of AWS

For the purposes of MOVES modeling, AWS can be envisioned as three primary components that together make up the cloud computing platform: Elastic Compute Cloud (EC2), Simple Storage Service (S3), and Simple Queue Service (SQS).

The EC2 component can be thought of as providing a virtual computer for the operating system (e.g., Windows) and MOVES code to run on, including the central processing units (CPUs) and the memory required to store and execute the operating system and MOVES.

When a virtual computer is created by EC2, it is a blank slate, with no associated operating system or programs. Therefore, the user must provide what is called an Amazon Machine Image (AMI), which essentially contains a copy of the operating system and any programs that the user has installed and configured. This AMI can be used as a template to create and launch any number of identical virtual computers, each of which is called an instance.

The S3 component is like a hard drive, and is used to store input and output data that an instance may need or generate during execution. For example, the MOVES code itself, county databases (CDBs), and MOVES outputs can be stored in S3. A storage location in S3 is called a bucket.

SQS is used to transfer messages for communication with the instances. Under typical operation, the user does not interact directly with an instance using a mouse, keyboard, and screen; instead, each instance automatically boots itself, launches whatever programs are scheduled to launch automatically on boot-up, and then polls the SQS queue for tasks to execute.

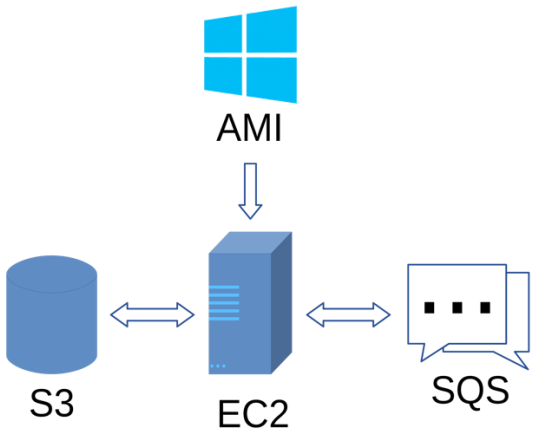

Figure 3-1. High-level overview of AWS components

3.2 Current process for deploying MOVES on AWS

Before MOVES can be deployed on AWS, a number of preparatory steps must be completed, such as obtaining an AWS account, configuring various AWS account settings, and obtaining access to an AMI configured to run MOVES. These are one-time tasks, and therefore are not the focus of this project, as automating them would not provide a substantial usability benefit. For the following description, we assume that these preparatory tasks have already been completed.

Currently, deploying MOVES on AWS requires a series of steps, some of which are carried out manually through the AWS web interface, and some of which are accomplished by writing/modifying and launching scripts.

1. Manually store MOVES default database in an S3 bucket using the AWS web interface
2. Manually store MOVES code in an S3 bucket using the AWS web interface
3. Manually create an S3 bucket to store MOVES jobs and results using the AWS web interface
4. Write/modify and launch scripts to generate MOVES runspecs for each required MOVES run, including all of the appropriate references to the various input databases required for each run
5. Write/modify and launch scripts to bundle and compress files required for MOVES jobs (runspecs, input databases, etc.) in preparation for uploading to AWS
6. Write/modify and launch scripts to upload jobs to S3 bucket
7. Write/modify and launch scripts to add jobs to the SQS queue
8. Manually configure and launch multiple EC2 instances using the AWS web interface
9. Monitor the instances for completion using the AWS web interface
10. When the instances have successfully completed all of the scheduled jobs, write/modify and launch a script to download the MOVES output databases from the S3 bucket to the local computer

11. Write/modify and launch a script to uncompress and rename all of the downloaded results and import them into local MySQL databases

As can be seen from this list, there are a number of manual steps required, each one of which encapsulates a large amount of complexity, and each one of which can fail due to any number of subtle details.

3.3 Proposed utilities

Looking closely at the steps listed in Section 3.2, they can be broken into two categories: step 4, which is the MOVES-specific step of generating the runspecs and their associated CDBs and/or user input databases; and all the other steps, which are AWS-specific tasks required to load the jobs into S3, schedule the jobs using SQS, prepare and launch the EC2 instances, and download the results to the local computer.

Much of the user-facing complexity involved in deploying MOVES to AWS is due to the latter category – the AWS-specific tasks. Fortunately, these are also the tasks that are most amenable to complete automation, as there are existing software development kits (SDKs) that enable software to interface directly with AWS. For example, the Python Boto 3 SDK, which ERG already uses for automating other AWS-based software tools, enables the programmatic creation and control of EC2 instances, S3 buckets, and SQS queues. These tasks are also conceptually simpler because the MOVES runspecs, once they are created, contain almost all of the information required to completely specify the components of a MOVES run. Determining the input databases to upload, the name of the output database, etc., is simply a matter of parsing the runspec, a relatively trivial programming task.

The MOVES-specific task of creating the runspecs is potentially more challenging, as it involves translating the user's project goals into concrete run specifications. For example, if a user wishes to perform county-scale runs for every county in the U.S., applying one age distribution to states bordering Mexico and a different age distribution to non-border states, two different types of runspecs would have to be generated: one set referencing the input database with the border state age distribution, and another set referencing the input database with the non-border state age distribution. While this scenario may not seem terribly complicated, complexity can rapidly increase as more combinations and variations are required, as is typical in most real-world projects. Currently, all of these combinations must be specified using custom variations of the Perl scripts used to construct the runspecs, a requirement that can add substantially to the time and difficulty of setting up and executing a project.

Considering the above, there is an opportunity for two different levels of automation. The first level would involve a utility that automates all of the tasks required to execute a MOVES runspec using AWS (Figure 3-2 and Figure 3-4). Provided one or more runspecs and all of the input databases referenced by the runspecs, the utility would automatically handle the bundling of the runspecs and databases; transferring them to an S3 bucket; creating, launching, and monitoring the EC2 instances; and downloading the output databases onto the local machine and importing them into MySQL.

The second level of automation would assist with the creation of large numbers of runspecs (Figure 3-3 and Figure 3-5).

Further detail related to how these utilities might work is provided in Sections 3.4 and 3.5.

In addition to these utilities, which automate running both the masters and workers in the cloud, under some worker-bound scenarios it may be beneficial to use the cloud simply to provide additional workers upon demand. This Worker Cloud Gateway utility is described in Section 3.7

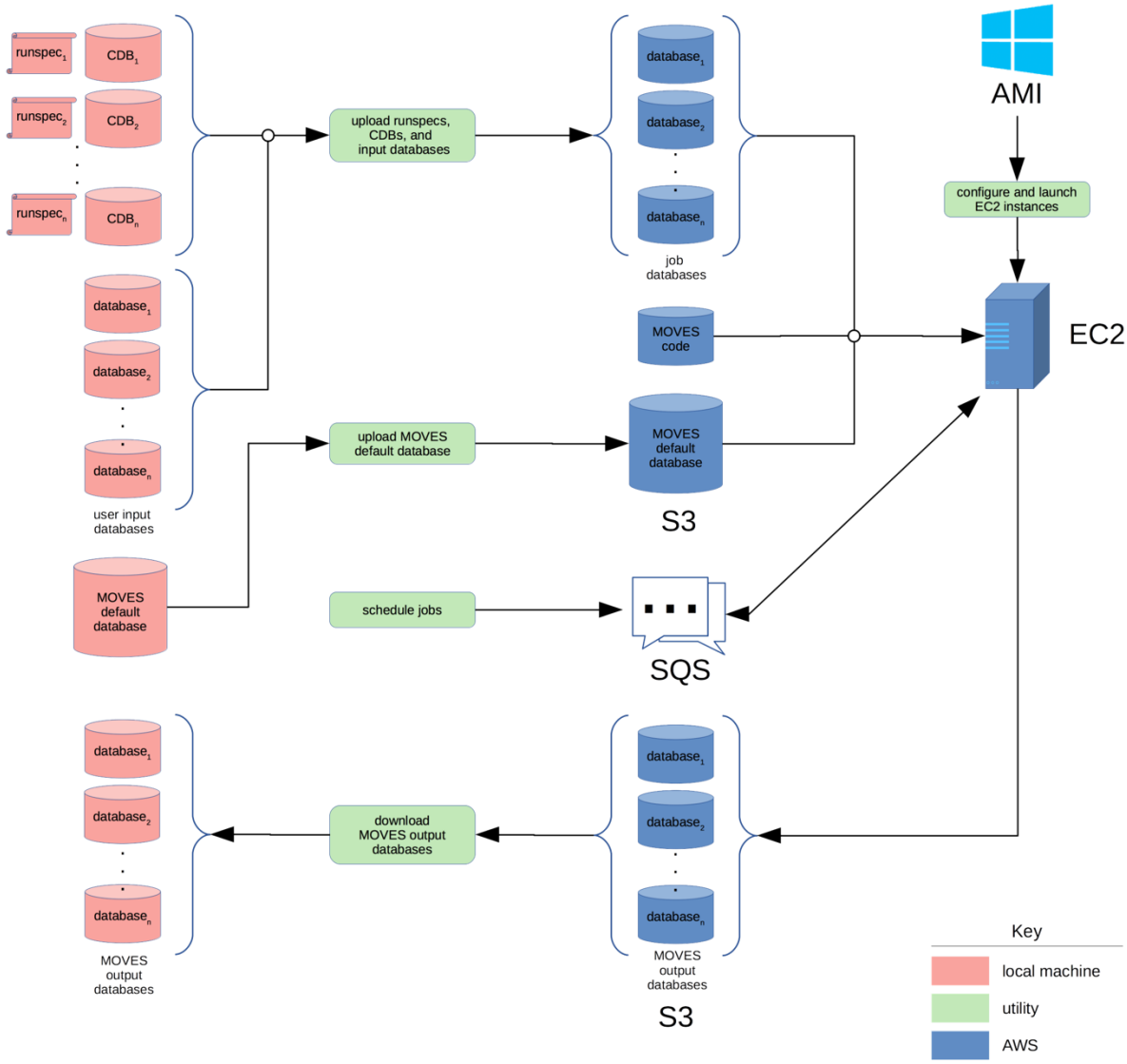

Figure 3-2. Conceptual diagram of AWS Runspec Execution Utility

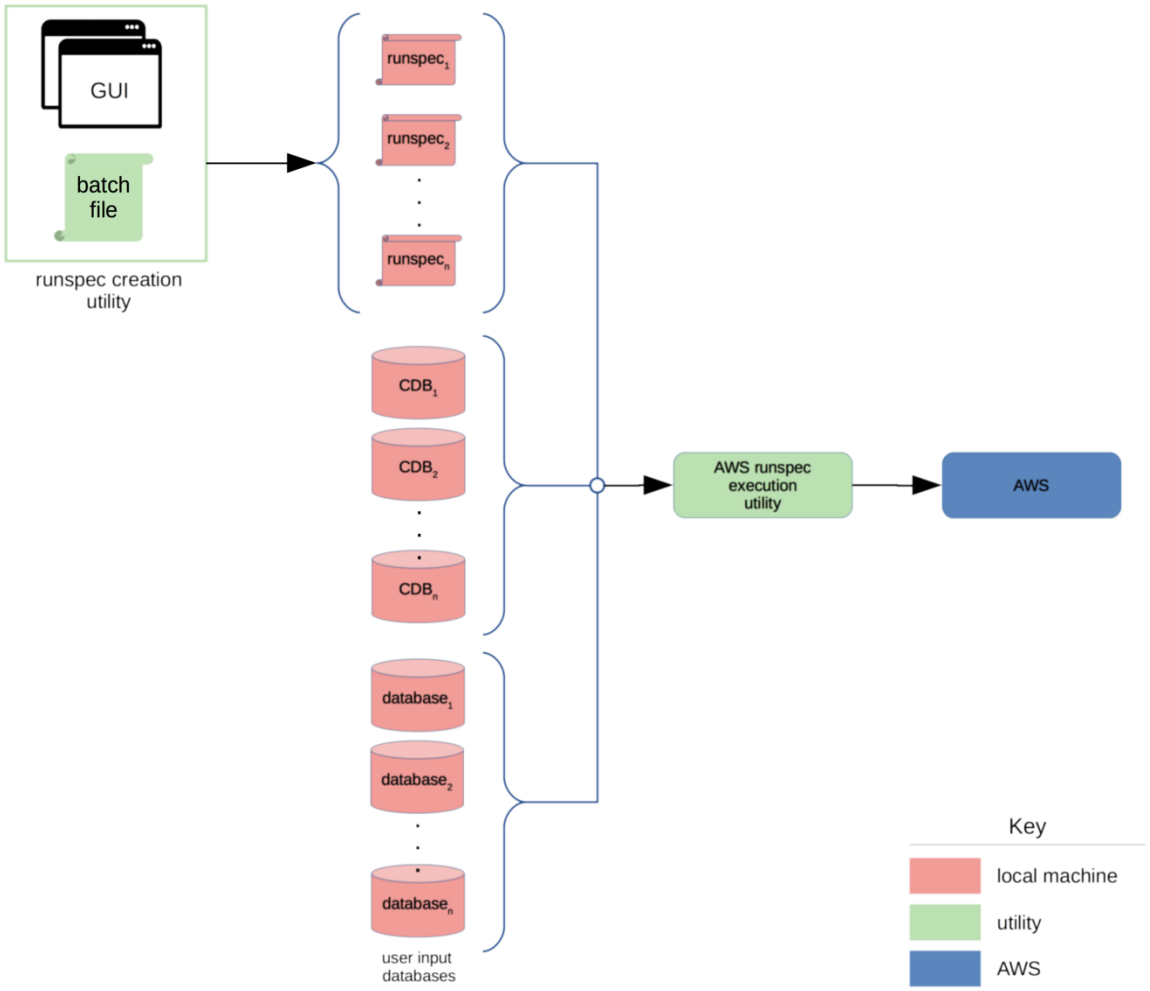

Figure 3-3. Conceptual diagram of Runspec Creation Utility

3.4 Runspec Execution Utility

The Runspec Execution Utility would automate the execution of MOVES runspecs in the cloud, allowing users to set up, deploy, and download results using a simple GUI interface such as the mock-up shown in Figure 3-4.

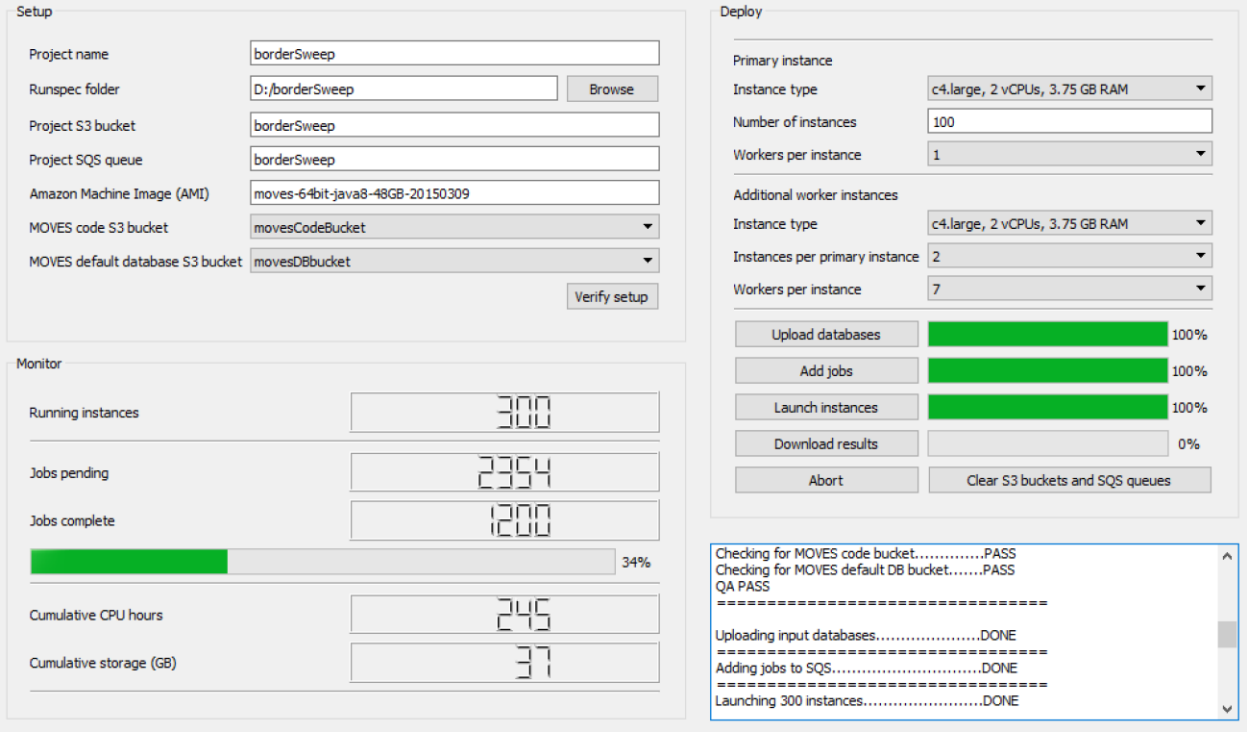

Figure 3-4. Mock-up of GUI for Runspec Execution Utility

As mentioned previously, use of this utility would require a one-time setup of an AWS account, including credentials files that would be stored on the user's local machine and automatically used by the utility to access the appropriate AWS account. In addition, an AWS machine image (AMI) pre-configured to run MOVES on AWS would have to be available, perhaps developed by the EPA for use by the community of MOVES users. The MOVES code and default database would also need to be uploaded to S3 buckets; these would also likely be obtained from a central provider, e.g., the EPA.

With this one-time setup complete, users would then create the runspecs and input databases for their MOVES runs, either using the MOVES GUI to create individual runspecs or using the Runspec Creation Utility to create a batch of runspecs. These runspecs would be saved in a single folder so they could be deployed using the Runspec Execution Utility.

After creating the runspecs, the user would use the Setup pane in the GUI to specify a project name, which could be used to keep track of the total AWS resources used for a particular set of MOVES runs, identify log files created by the utility, etc. Next, they would specify the folder containing the MOVES runspecs. Names for the project's S3 bucket (used to store the input and output databases) and SQS queue would be entered. The name of the pre-configured AMI would then be entered, and the S3 buckets containing the MOVES code and default database would be

chosen from a drop-down menu that the utility would have populated with all available buckets associated with the user's AWS account.

After the setup is complete, the user could click the "Verify setup" button to run a series of checks to ensure that the AWS account is accessible and configured correctly; that the AMI exists; and that the MOVES code and default database buckets exist.

The Deploy pane would be used to actually schedule, launch, and harvest the results of the MOVES runs. First, the user would select the type of instance that they want to use from the available options. A variety of AWS instances are available with different numbers of virtual CPUs (vCPUs), different processor speeds, and varying amounts of RAM. The number of instances to run concurrently would be selected next. The maximum number of instances for a given user is determined by Amazon. The number of MOVES workers to run on the primary instance would then be chosen from a dropdown menu.

In some cases, MOVES runs are worker bound, meaning that the total runtime is largely determined by the number of MOVES workers. If so, it may be beneficial to launch additional instances to run more MOVES workers than could be executed on a single machine. This would be similar to the sort of master machine/worker machines setup that can be used with a shared folder on a local network except the machines and the shared folder would all exist in the cloud. Behind the scenes, the utility would handle setting up a shared folder in an S3 bucket for the master instance and the worker instances to use for posting tasks and processed jobs. The user would specify the number of additional worker instances to launch for every primary (master) instance as well as the number of workers to run on each of these additional instances.

For publicly released versions of MOVES, Master/Worker is not enabled in the cloud. There is no file sharing between Amazon computers in current configurations. Nothing in MOVES prevents this, but setting up file shares via cloud scripts would require additional time and expense that would not pay off. This is because multithreading performance in the cloud has the same issues highlighted in Task 1. The current public versions of MOVES are limited by I/O bandwidth, not by the number of CPU cores on a single machine; adding workers via the cloud does not solve this problem After the changes from Task 1, MOVES will be CPU limited and will likely not benefit much from the master/worker paradigm. This updated version of MOVES would have a single master/worker instance on each cloud machine and no dedicated worker cloud machines. In this case, adding CPU instances in the cloud would directly benefit MOVES runtime.

After configuring the instances, the user would click the "Upload databases" button to transfer all of the input databases, including any CDBs, to the project's S3 bucket. The progress indicator would show the percentage of the data uploaded to AWS. After all of the input data are uploaded, the user could then proceed to add the jobs to the SQS queue and launch the instances.

The Monitor pane would be used to monitor the state of the AWS resources, including the number of running instances, the number of jobs pending, the number of jobs completed, and the overall progress. The cumulative CPU hours and storage use are important, as they determine the total cost for the AWS runs, so these would also be displayed in the GUI.

At any point during execution, the user could click the "Download results" button to download results from any completed MOVES runs and load them into the local MySQL server.

Buttons for aborting all runs and clearing the S3 buckets and SQS queues would also be provided.

3.5 Runspec Creation Utility

For simplicity, the Runspec Creation Utility could rely on the assumption that all runspecs within a batch share the same pollutants, processes, vehicles, and other run settings, i.e., the only differences between individual runspecs would be their geographic locations, time periods, and input databases. With this restriction, the utility could use an existing MOVES runspec as a template. The user would then specify the geographic locations and time periods, either via a GUI or a batch file. Then, for the locations specified, the user would select one or more input databases that would be incorporated into the runspecs. Again, this could be done either via a GUI or a batch file.

The use of a runspec template would allow the Runspec Creation Utility to be much simpler than it would be otherwise. It could rely on the MOVES GUI to specify most of the settings in the runspec template, highlighted in blue in the following example runspec. The utility would only need to alter the sections highlighted in orange in order to generate runspecs for the individual runs.

```
<runspec version='MOVES2014a-20151201'>
    <description><![CDATA[EPA MOVES Mexico Runspec]]></description>
    <models>
        <model value='ONROAD'/>
    </models>
    <modelscale value='Inv'/>
    <modeldomain value='NATIONAL'/>
    <geographicselections>
            <geographicselection type='COUNTY' key='1001' description='Autauga County'/>
    <geographicselections>
    <timespan>
        <year key='2028'/>
        <month id='1'/>
        <month id='7'/>
        <day id='2'/>
        <day id='5'/>
        <beginhour id='1'/>
        <endhour id='24'/>
        <aggregateBy key='24-Hour Day'/>
    </timespan>
    <onroadvehicleselections>
        <onroadvehicleselection fueltypeid='1' fueltypedesc='Gasoline' sourcetypeid='61'
               sourcetypename='Combination Short-haul Truck'/>
        <onroadvehicleselection fueltypeid='1' fueltypedesc='Gasoline' sourcetypeid='32'
               sourcetypename='Light Commercial Truck'/>
        <onroadvehicleselection fueltypeid='1' fueltypedesc='Gasoline' sourcetypeid='11'
               sourcetypename='Motorcycle'/>
        <onroadvehicleselection fueltypeid='1' fueltypedesc='Gasoline' sourcetypeid='21'
               sourcetypename='Passenger Car'/>
        <onroadvehicleselection fueltypeid='1' fueltypedesc='Gasoline' sourcetypeid='31'
               sourcetypename='Passenger Truck'/>
        <onroadvehicleselection fueltypeid='1' fueltypedesc='Gasoline' sourcetypeid='53'
               sourcetypename='Single Unit Long-haul Truck'/>
        <onroadvehicleselection fueltypeid='1' fueltypedesc='Gasoline' sourcetypeid='52'
               sourcetypename='Single Unit Short-haul Truck'/>
        <onroadvehicleselection fueltypeid='1' fueltypedesc='Gasoline' sourcetypeid='42'
                sourcetypename='Transit Bus'/>
    </onroadvehicleselections>
```

```
    <offroadvehicleselections>
    </offroadvehicleselections>
    <offroadvehiclesccs>
    </offroadvehiclesccs>
    <roadtypes separateramps='false'>
        <roadtype roadtypeid='1' roadtypename='Off-Network' modelCombination='M1'/>
        <roadtype roadtypeid='2' roadtypename='Rural Restricted Access' modelCombination='M1'/>
        <roadtype roadtypeid='3' roadtypename='Rural Unrestricted Access' modelCombination='M1'/>
        <roadtype roadtypeid='4' roadtypename='Urban Restricted Access' modelCombination='M1'/>
        <roadtype roadtypeid='5' roadtypename='Urban Unrestricted Access' modelCombination='M1'/>
    </roadtypes>
    <pollutantprocessassociations>
        <pollutantprocessassociation pollutantkey='2' pollutantname='Carbon Monoxide (CO)'
               processkey='1' processname='Running Exhaust'/>
        <pollutantprocessassociation pollutantkey='2' pollutantname='Carbon Monoxide (CO)'
               processkey='15' processname='Crankcase Running Exhaust'/>
        <pollutantprocessassociation pollutantkey='3' pollutantname='Oxides of Nitrogen (NOx)'
               processkey='1' processname='Running Exhaust'/>
        <pollutantprocessassociation pollutantkey='3' pollutantname='Oxides of Nitrogen (NOx)'
               processkey='15' processname='Crankcase Running Exhaust'/>
    </pollutantprocessassociations>
    <databaseselections>
        <databaseselection servername='' databasename='borderAgeDistribution' description=''/>
        <databaseselection servername='' databasename='borderFuels' description=''/>
    </databaseselections>
    <internalcontrolstrategies>
        <internalcontrolstrategy
               classname='gov.epa.otaq.moves.master.implementation.ghg.internalcontrolstrategies.
               rateofprogress.RateOfProgressStrategy'><![CDATA[
        useParameters  No

        ]]></internalcontrolstrategy>
    </internalcontrolstrategies>
    <inputdatabase servername='' databasename='' description=''/>
    <uncertaintyparameters uncertaintymodeenabled='false' numberofrunspersimulation='0'
               numberofsimulations='0'/>
    <geographicoutputdetail description='COUNTY'/>
    <outputemissionsbreakdownselection>
        <modelyear selected='false'/>
        <fueltype selected='true'/>
        <fuelsubtype selected='false'/>
        <emissionprocess selected='true'/>
        <onroadoffroad selected='true'/>
        <roadtype selected='true'/>
        <sourceusetype selected='true'/>
        <movesvehicletype selected='false'/>
        <onroadscc selected='true'/>
        <estimateuncertainty selected='false' numberOfIterations='2' keepSampledData='false'
               keepIterations='false'/>
        <sector selected='false'/>
        <engtechid selected='false'/>
        <hpclass selected='false'/>
        <regclassid selected='false'/>
    </outputemissionsbreakdownselection>
    <outputdatabase servername="" databasename="borderSweep_12jun17" description=""/>
    <outputtimestep value='24-Hour Day'/>
    <outputvmtdata value='true'/>
    <outputsho value='true'/>
    <outputsh value='true'/>
    <outputshp value='true'/>
    <outputshidling value='true'/>
    <outputstarts value='true'/>
    <outputpopulation value='true'/>
    <scaleinputdatabase servername='' databasename='' description=''/>
    <pmsize value='0'/>
    <outputfactors>
       <timefactors selected='true' units='Days'/>
        <distancefactors selected='true' units='Miles'/>
        <massfactors selected='true' units='Grams' energyunits='Joules'/>
    </outputfactors>
    <savedata>
```

```
    </savedata>

    <donotexecute>

    </donotexecute>

    <generatordatabase shouldsave='false' servername='' databasename='' description=''/>
        <donotperformfinalaggregation selected='false'/>
    <lookuptableflags scenarioid='' truncateoutput='true' truncateactivity='true'
truncatebaserates='true'/>
</runspec>
```

A mock-up of the Runspec Creation Utility GUI is shown in Figure 3-5. Using this GUI, the user would specify a project name, with a project being a group of batches. Each batch would be a group of runspecs that share a common configuration, including the time period (years, months, days, and hours), the input databases, and the geographic bounds. All of the batches would use the same runspec template, and all of the generated runspecs would be saved to a common output folder so they could be launched together using the Runspec Execution Utility. County-specific data could be provided by county databases (CDBs) adhering to the naming convention specified in the GUI. The MOVES output databases generated by the runs would adhere to a naming convention specified in the project settings, e.g., borderSweep_*countyID*_12jun17 in the GUI mock-up, with *countyID* indicating a placeholder that would be replaced by each MOVES run's actual countyID.

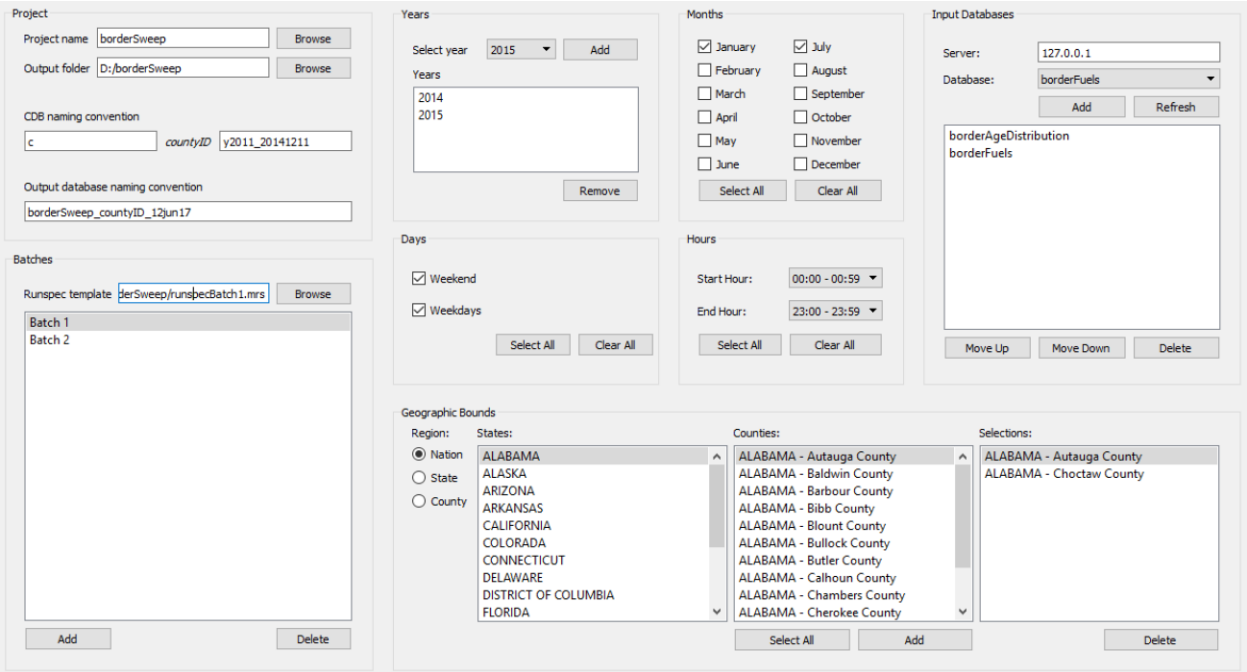

Figure 3-5. Mock-up of GUI for Runspec Creation Utility

At a minimum, the utility could generate individual runspecs for each unique year/county combination. However, it may be possible to build in some logic to adaptively split and/or combine runspecs. For example, if very few pollutants are chosen, it may be better to combine multiple years into a single runspec, as this would reduce the number of AWS instances that

would have to be launched – potentially saving costs, as Amazon charges for CPU time in one hour increments, so each additional instance costs at least one hour of CPU time.

3.6 Example scenario using Runspec Creation and Execution Utilities

In the following, we describe an example scenario to illustrate the potential time savings achievable using the cloud. To be as realistic as possible, this example describes a setup similar to one already implemented by ERG, allowing for relatively precise estimates of runtime.

The example scenario is as follows: generate annual emissions inventories for all 3,228 counties in the U.S.; two calendar years; all months, day types, and hours; all gasoline and diesel source types; and all road types. All pollutants should be included except petroleum energy consumption; fossil fuel energy consumption; CO_2 equivalent; metals; and dioxins and furans. Both evaporative and non-evaporative processes should be modeled.

For efficiency, the evaporative and non-evaporative processes can be modeled using separate runs, as that allows the non-evaporative processes to use a daily time aggregation level instead of the hourly time aggregation required for evaporative processes. With this approach, the evaporative runs only need to include the gasoline source types, while the non-evaporative runs include both gasoline and diesel source types.

The computational resources to run this scenario on a single computer would be prohibitive. On a typical desktop computer with a single worker, executing the MOVES runs for a single county and year would require approximately 5 hours (approximately 270 minutes for the evaporative run and 30 minutes for the non-evaporative run). Scaling up to all 3,228 counties and both years would require 3,228×2×5 = 32,280 hours, or 3.7 years.

Deploying these runs in the cloud would make this impractical scenario feasible. Assuming that 400 instances are available to the AWS account (the default limit is lower, but Amazon will increase this limit upon request, as they have for ERG), and assuming that each instance executes a single evaporative run (5 hours of CPU time) or non-evaporative run (1 hour of CPU time) for a single county and year, the clock time required to execute the runs will be [(3,228×2×5) + (3,228×2×1)]÷400 = 97 hours, or 4 days.

In addition to the execution time, the cloud runs would entail a certain amount of overhead to transfer the input databases to the cloud and download the results. This would depend on the local network speed, but even if these were to add an additional day to the clock time (probably a conservative estimate), the entire inventory could be completed in a week's time instead of taking years – or, more likely, not being possible at all.

Although the scenario as outlined above demonstrates a dramatic reduction in the clock time required to generate the hypothetical inventory, it probably underestimates the potential increase in efficiency, as it assumes that no additional MOVES workers are used. It is probable that these runs would be worker bound to at least some extent, so launching additional workers, either on the primary (master) instance or using additional worker instances, could provide another substantial improvement in runtime.

The cost to execute these runs in the AWS cloud would depend on the details, e.g., the total size of the input databases and the exact runtime. However, it can be approximated using the hours

estimated above and a total rate of $0.13/CPU hour, which is roughly what ERG has historically paid for CPU time and typical storage and transfer fees. With these assumptions, the total cost would be approximately [(3,228×2×5) + (3,228×2×1)]×$0.13, or $5,036.

The procedure for executing this scenario using the proposed runspec creation and runspec execution utilities would be as follows: First, the user would use the standard MOVES GUI to generate one runspec template for the evaporative processes and one runspec template for the non-evaporative processes. Then, using the Runspec Creation Utility, the user would create two batches – one for the evaporative runs, and one for the non-evaporative runs. The user would use the GUI to specify the two scenario years, as well as the time ranges and geographic bounds. They would also specify any input databases, including any CDBs containing county-specific input data. The utility would use these inputs to create 3,228×2 = 6,456 runspecs for each batch, for a total of 12,912 runspecs. These would be saved to a single output folder for use by the Runspec Execution Utility.

Launching these runspecs with the Runspec Execution Utility would be a simple matter of using the GUI to specify the location of the runspecs and then executing the steps required to specify the AWS instances; load the input databases to AWS; launch and monitor the instances; and then download the results to the local MySQL database.

3.7 Worker Cloud Gateway

Another potential way to leverage AWS to speed up worker-bound MOVES runs would be to deploy just MOVES workers in the cloud, keeping the MOVES master on a local machine. Deploying just the workers would be simpler, as the MOVES default database and input databases would not need to be transferred to the cloud, obviating the need to create S3 buckets to store these data. This approach would be facilitated by changes to the MOVES architecture, as described below. Changing the MOVES architecture to move some part of the generators to the workers, as discussed in Section 2.0, would also increase the value of deploying a large number of workers for a single master; while running workers in the cloud could provide some benefit for MOVES 2014a, its greatest benefit would be realized after the Task 1 changes are implemented.

This streamlined approach would enable the creation of a Worker Cloud Gateway utility (Figure 3-6). This utility would allow the cloud to be treated as simply another collection of worker machines, with the number of machines easily scalable as required for the particular MOVES run. The user would simply launch the Worker Cloud Gateway and specify the number of workers to request. In the background, the utility would handle the tasks required to launch instance types appropriate for an individual worker or small number of workers (selection of an optimal instance type would be determined during development of the utility).

The utility would poll the MOVES SharedWork folder for available bundles to process, sending them to an S3 bucket shared by the workers and placing messages on SQS notifying the workers that TODO bundles are available to process. The SQS message would contain the name of the file to be fetched from S3, the master ID, and the master code version. The master's code itself would be stored in a Java ARchive (JAR) file in the SharedWork folder; this file would serve as both the master's heartbeat file and as a way to ensure that any worker processing bundles

associated with that master's ID would also have access to the correct code required to process the bundles.

After a worker finishes processing a bundle, it would place the DONE bundle in S3 and post an SQS message informing the Worker Cloud Gateway utility that a bundle is ready to download. The DONE bundle would also contain the master's ID number so the Worker Cloud Gateway would know which local folder to place the downloaded bundle in. Once the final master bundle is detected, the worker instances could be automatically triggered to shut down.

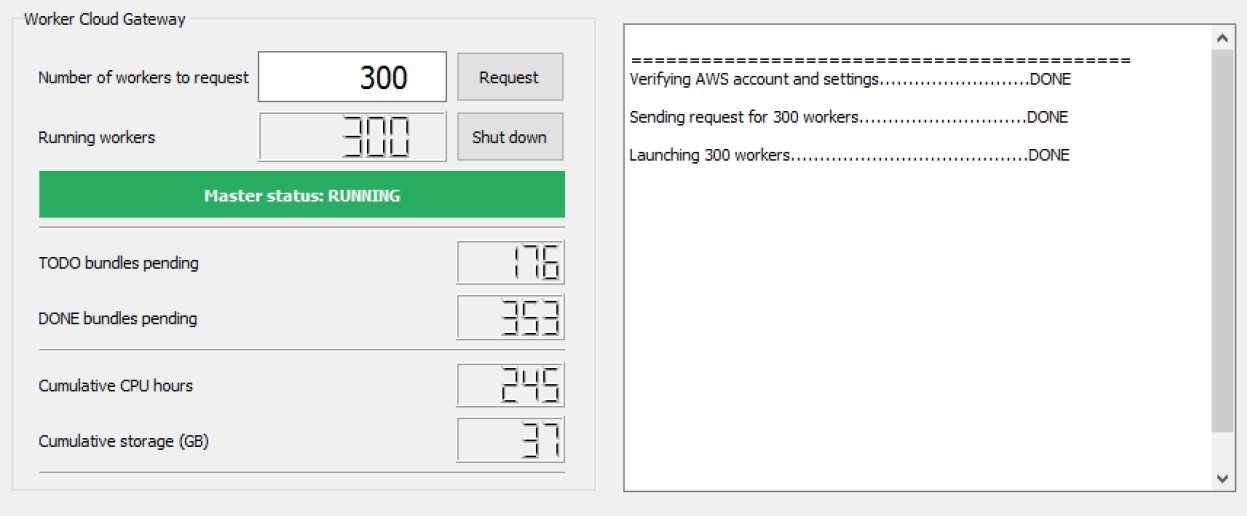

Figure 3-6. Mock-up of Worker Cloud Gateway utility

4.0 Task 3: Eliminating Redundancies

Task 3 focuses on what users could do to improve performance of the current version of MOVES. As noted under Task 1, the recoding would eliminate the bottlenecks and redundancies highlighted in this section.

The original design concept of MOVES was to create core calculation functionality around activity and emissions rates defined by unique modes of vehicle operation. For example, running emissions, the core MOVES "calculator" design uses load-based (vehicle specific power, VSP) operating modes, with emission rates expressed per time spent in each operating mode. Because activity data available to the user takes different forms (e.g. VMT, average speed), the MOVES design added "generators" to convert readily-available data to the core data needed by calculators. Generators were designed separately from calculators to provide flexibility for users to alter key inputs and have the model adapt "on-the-fly"; a prime example is the Operating Mode Distribution Generator, which was developed to allow users to enter alternative drive cycles into MOVES, and have the model adapt emission rates based on the inherent distribution of VSP-based operating modes.

Redundant calculations are most likely to occur, and easiest to isolate, in "generators" which convert intermediate data to the core data needed by the model to directly calculation emissions rates and inventory. If users do not change inputs to generators, the generators will produce the same core model inputs every time the model is run. This creates unnecessary redundancy; in

these cases, core model inputs could be provided directly to the model MOVES calculators, without the need for generators to run. This would reduce runtime to the degree generators are contributing to overall execution time.

To demonstrate the contribution of generators to a typical run, and to isolate which generators are the biggest bottlenecks, a test run with following specifications was run on the current publicly available version of MOVES, MOVES2014a. This run is a typical daily run similar to what might be used for SIP modeling, so was considered a good example run. The performance improvements may vary depending on the scope of the run.

RunSpec	ACELA_ZMVM
Output/Run	ACELA_Newblends RunID=15
	National
Geography	State Aggregation / 2 States
Year	1
Month	1
Daytype	1
Hours	24
Source/Fuel Types	8
Road Types	All
Pollutants	12
Processes	All

Table 4-1. MOVES test run specifications

In the MOVES output database, the table BundleTracker logs the run duration for all generators, calculations and aggregation steps. Figure 4-1 shows the general breakdown between these three groups:

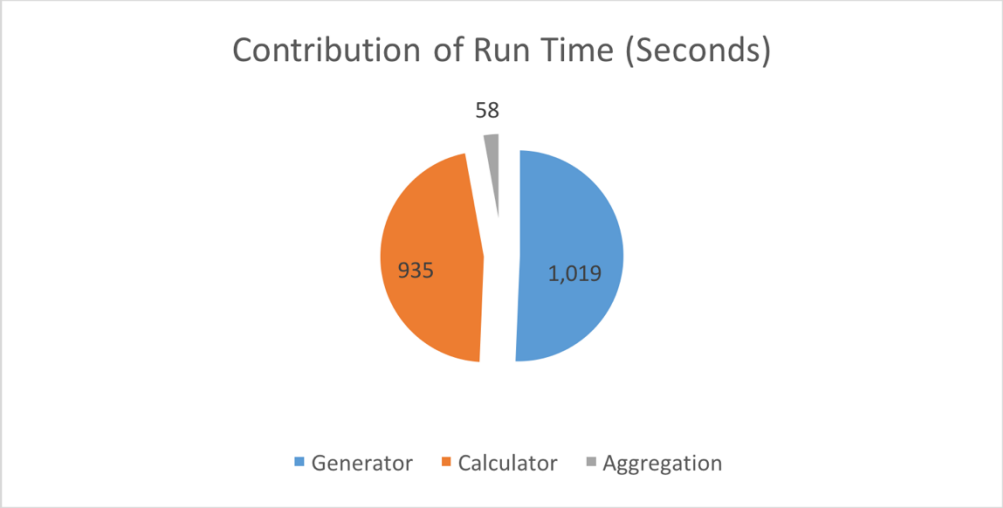

Figure 4-1. Contribution to runtime, example run

As shown, generators account for about ½ of total run time. A further breakdown of individual run time generators is shown in Figure 4-2.

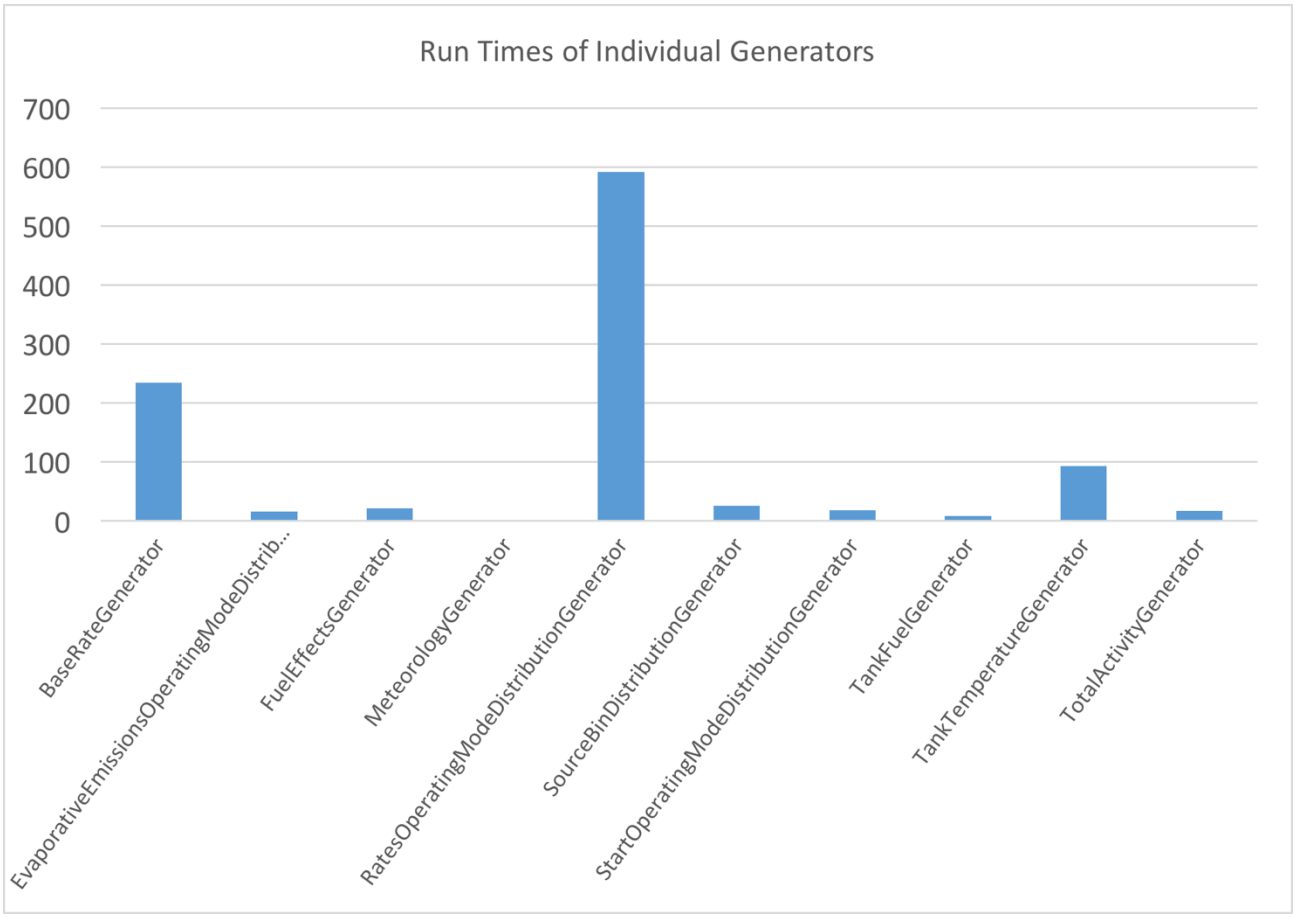

Figure 4-2. Runtime by generator

As shown, the RatesOperatingModeDistribution and BaseRate generators accounts for the majority of total generator run time – 884 seconds, which is 87 percent of total generator runtime, and 44 percent of overall runtime. Bypassing these two generators would therefore reduce runtime considerably for this run. This example is illustrative only; the amount of benefit would vary with changes in the scope of the runspec, for example the number of source types, road types, months, days or hours.

Generators can be considered redundant if users are not likely to change data used as input to the generator. Table 4-2 shows the input tables used by each generate to produce the core data needed by MOVES calculators.

Generator	Input Tables	User Likely to Change?
RatesOperatingModeDistributionGenerator	AvgSpeedBin	No
	DriveSchedule	No
	DriveScheduleAssoc	No
	DriveScheduleSecond	No
	hotellingActivityDistribution	No
	OMDGPolProcessRepresented	No
	OperatingMode	No
	OpModePolProcAssoc	No
	PollutantProcessAssoc	No
	RoadOpmodeDistribution	No
	RoadType	No
	SourceTypePolProcess	No
BaseRateGenerator	dioxinemissionrate	No
	EmissionRate	No
	EmissionRateByAge	No
	FullACAdjustment	No
	hotellingActivityDistribution	No
	metalemissionrate	No
	ModelYearGroup	No
	PollutantProcessAssoc	No
	PollutantProcessModelYear	No
	SourceBin	No
	SourceBinDistribution	No
	SourceTypeModelYear	No

Table 4-2. Input tables for two bottleneck generators

As shown in Table 4-2, it is very unlikely users would change the input tables used by these generators. Our assessment is based on broad experience running MOVES, including support for clients setting up MOVES runs at the local, state, regional and national levels. None of the tables listed above are addressed in EPA's technical guidance for running MOVES for SIP and Conformity purposes, and none are included within the primary set accessed via the County Data Manager. These generators are prime candidates to bypass, but would require providing a pre-generated core model input table (CMIT) to MOVES. Only BaseRateGenerator produces a CMIT, but based on output from the RatesOpModeDistribution generator. This table, RatesOpModeDistribution, contained about 2.6 million records for this run. This is the combination of each source type (8), road type (5), pollutant (5), process (2), average speed bin (16) and hour/day (48) used in the run; the size of this table will vary based on the selections of these parameters. The Core Model Input Table produced by the BaseRateGenerator, is BaseRateByAge. This is a combination of source type, regulatory class, fuel type, road type, pollutant, process, average speed bin, resulting in about 0.5 million records. As mentioned, BaseRateByAge encompasses RateOpModeDistribution, so both generators could be bypassed if only the BaseRateByAge table is pregenerated. For the test run, bypassing the generators would reduce total runtime by up to 45 percent in MOVES2014a.

A pre-generated BaseRateByAge table could be provided by EPA based on national defaults. In general, EPA may consider providing default CMITs for all generators, so a "calculator only" version of the model could be run if no inputs are changed.

MOVES2014a has functionality in Advanced Performance Features GUI pane to turn off generators and calculators. Unfortunately, this has not been extended to RatesOperatingModeDistribution or BaseRate generators. Code update would therefore be required to allow bypass of these generators when a pre-generated CMIT is available, in order to realize the performance improvements. As part of this update, the model could be programmed to detect if a) relevant inputs have not been changed and b) pre-generated CMITs are populated, and automatically bypass the generators if these conditions are met. The hour and cost estimates detailed in Section 5 include this feature.

5.0 Task 4: Effort Required

This section lists the individual subtasks required to enact the recommendations listed under Task 1-3, to serve as a qualitative estimate of effort for work to accomplish Tasks 1-3. Specific hours for each subtask are not shown, as these would only be provided as CBI for a specific bid. To meet the requirements of the RFP, a supplemental cost estimate has been provided to CRC.

5.1 Task 1 subtasks

Subtasks needed for the recommended architecture changes are shown below:

Task 1: Recode
Create gencalc program from generator and calculator programs
Modify gencalc generators to have file-based mode
Modify gencalc calculators to have database mode
Update gencalc to chain generators and calculators
Compute benefit from a distributed bundle
Invoke a local bundle
Invoke a distributed bundle
Updates to MasterLoop
Testing and validation
Architecture documentation updates
Contingency

Table 5-1. Task 1 subtasks

5.2 Task 2 subtasks

Subtasks for Task 2 are shows for each runspec utility, and for the Worker Cloud Gateway

Task 2: Runspec Execution Utility
Develop scripts to create S3 buckets; create SQS queues; transfer data to and from S3 buckets; and transfer messages to instances via SQS
Develop scripts to configure, launch, monitor, and manage instances
Develop scripts to download and process MOVES outputs and load them to the local MySQL server
Design the GUI and develop the underlying code to manage the backend scripts
Testing and validation
Documentation

Table 5-2. Task 2 Runspec Execution Utility subtasks

Task 2: Runspec Creation Utility
Develop scripts to parse runspec templates and develop runspecs for batches
Design the GUI and develop the underlying code to manage the backend scripts
Testing and validation
Documentation

Table 5-3. Task 2 Runspec Creation Utility subtasks

Task 2: Worker Cloud Gateway
MOVES architectures changes: bundling MOVES code in a JAR file and using it as the heartbeat file; modifying workers to use bundled MOVES code
Configure AMI, EC2 instances, and worker code to implement workers on AWS
Develop scripts to create S3 buckets; create SQS queues; transfer data to and from S3 buckets; and transfer messages to instances via SQS
Develop scripts to configure, launch, monitor, and manage instances
Develop scripts to manage bundles on AWS and local machine
Design the GUI and develop the underlying code to manage the backend scripts
Testing and validation
Documentation

Table 5-4. Task 2 Worker Cloud Gateway subtasks

5.3 Task 3 effort estimates

Task
Pre-generate default BaseRatebyAge CMIT
Allow bypass of RatesOperatingModeDistribution and BaseRate generators , inc. automatic bypass if relevant input tables are not changed, and CMITs populated
Provide guidance on re-generating CMIT in case any inputs are changed (e.g. driving schedules)

Table 5-5. Task 3 subtasks