PARALLEL STRATEGIES FOR LOCAL SENSITIVITY ANALYSIS OF LARGE CHEMICAL MECHANISMS*

A. MARK PAYNE†

Abstract. Performing sensitivity analysis is vitally important to the construction of large chemical mechanisms. Sensitivity analysis allows the user to determine which subset of the total model parameters should be refined by higher level methods. There are two primary methods for conducting sensitivity analysis for mechanisms of this size: forward sensitivity analysis, and adjoint sensitivity analysis, with the latter suspected of being more favorable for large models. Both of these methods are inherently parallelizable, though this is rarely done. In this work we implemented methods for performing these analyses in parallel and comparing their performance. In both cases parallelization resulted in tremendous speed-up. In fact, one technique, known as batching, is so important for forward sensitivity analysis that it is recommended that it be used even when running in serial. With these improvements, the forward sensitivity method IFPFSA described in this work was able to beat out the adjoint method by a factor of 5 even when the model contained 3345 parameters. Despite this, adjoint sensitivity analysis could be the preferred method if only a subset of the analysis needed to be performed, which is common when working with chemical mechanisms.

Key words. RMG, sensitivity analysis, adjoint sensitivity

1. Introduction.

1.1. Motivation.

Generating high fidelity models for chemical systems is a vital part of understanding many problems in science in engineering, from reducing soot formation, to designing new biofuel engines, to understanding how pharmaceutical APIs break down over time. These models can involves many components, including computational fluid dynamics (CFD) simulation; however, at the heart of any of these modeling approaches is a chemical mechanism. These mechanisms consist of elementary reactions that describe how fast one chemical species transforms into another. Once constructed, this mechanism is used to define a system of ordinary differential equations (ODEs) to describe how the concentration of all of the species in the model (the solution variables of the ODE) change with time.

There are many different methods for constructing chemical mechanisms. One common approach is to assume that the reactions between species follow first-order kinetics, and then fitting the correspond rate constants to experimental data obtained for the system. Another common method is generating a mechanism based off of elementary reactions, where the reactions present were chosen by an expert chemist "by hand". In the past decade, though, one method that has become increasingly utilized is constructing mechanisms using automatic mechanism generation software, such as the Reaction Mechanism Generator (RMG)[3]. Automatic mechanism generation software like RMG construct chemical mechanisms by enumerating all possible reactions that can occur between the initial products (and then their byproducts), and determining which of these reactions are relevant enough to include in the final model based on the conditions of the model. By doing all of this in automated fashion, it has now become possible to generate very large chemical mechanisms, allowing very large and complex chemical systems to be studied. For example, it is possible for RMG to generate mechanisms that contain O(1,000) species and O(100,000) reactions.

Perhaps the biggest challenge for automatically constructing chemical mechanisms

*Submitted to the class instructors December 18th, 2020.

†Department of Chemical Engineering, Massachusetts Institute of Technology, Cambridge, MA (ampayne@mit.edu).

is obtaining values for all of the parameters in the mechanism itself. For a mechanism based on elementary reactions, every reaction must have an associated forward rate constant, k_f, and every species in the model must have known thermochemistry, making the total number of parameters $n_{species} + n_{rxns}$. For large chemical mechanisms this means that the number of parameters can easily extend into the hundreds of thousands. Unfortunately, the vast majority of the reactions and species in a model have never been studied before, either computationally or experimentally, and thus there is not value that one can use from the literature for these rate constants and species thermochemistry.

Thankfully, automatic mechanism generation software such as RMG have methods for estimating rate constants and species thermochemistry. The quality of these estimates varies depending on the exact method used and how close the reaction/species are to previously studied reactions/species (which are used to train the estimation methods). While this guarantees that automatic mechanism generation software can include a value for all necessary parameters, this leaves open the very likely possibility that some of the parameters used are off by several orders of magnitude, especially when generating mechanisms for new types of chemical systems previously never studied before. With this, the predictive capability of the generated mechanism could be compromised due to these poor parameter estimates.

Given this challenge, the reason that automatic mechanism generation software has become moderately successful over the past decade is that advances in quantum chemistry software over the past few decades has allowed for the accurate calculations of reaction rates and species thermochemistry, allowing researchers to get better parameter estimates for chemical mechanisms. While advances are continually made to allow for faster calculations on increasingly larger species, it is simply not possible to calculate all O(100,000) parameters accurately using quantum chemistry software.

The solution to this problem is to only refine model parameters that would significantly impact the key observables (often solution variables) of the model if they were off from the true value by even a small amount. Serendipitously, it is often the case that of the O(100,000) parameters in a large chemical mechanism, only a small fraction of them fall into this category, which is usually a small enough number to refine exhaustively by quantum chemistry calculations.

Determining how much model parameters affect the solution variables in a model can be done by performing sensitivity analysis on the model. Given the importance of finding the key subset of parameters to refine and the scale of the models generated by automatic mechanism generation software, it is crucial to be able to perform sensitivity analysis on models containing many parameters in a quick and efficient manner. This work explores strategies for performing sensitivity analysis on large models. While this work draws heavily on already existing software packages for simulating chemical methods and performing various forms of sensitivity analysis, this work builds on these packages by exploiting the nature of the problem to enable speed-up via parallelization.

1.2. Parallel Nature of Sensitivity Analysis. Sensitivity analysis aims to determine of small perturbations in a parameter affect the resulting solution variables of the model. There are two different types of sensitivity analysis: local sensitivity analysis and global sensitivity analysis. Local sensitivity analysis calculates the derivative $s_{ij} = \frac{\partial y_i}{\partial p_j}$ for all solution variables y_i and parameters p_j. Since local sensitivity analysis only calculates these first-order derivatives, the sensitivities are only valid for small perturbations around the parameters. Global sensitivity analysis on the

other hand tries to capture information about the higher order derivatives (without directly calculating them) so that it is valid for larger perturbations in the model parameters. The are a variety of methods for performing global sensitivity analysis, including Monte Carlo Simulation and polynomial chaos expansions (PCEs), but these methods are computationally expensive. For large chemical methods these methods prove to be too costly, and thus were not explored in this work, despite the fact that large errors in parameter estimation in chemical mechanisms can necessitate global sensitivities.

Performing local sensitivity analysis involves calculating the full sensitivity matrix, usually at specific time points, shown in Equation (1.1). Note that the matrix is structured much like that parameter Jacobian matrix for the ODE function.

$$S(t) = \begin{bmatrix} \frac{\partial y_1}{\partial p_1} & \frac{\partial y_1}{\partial p_2} & \cdots & \frac{\partial y_1}{\partial p_k} \\ \frac{\partial y_2}{\partial p_1} & \frac{\partial y_2}{\partial p_2} & \cdots & \frac{\partial y_2}{\partial p_k} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial p_1} & \frac{\partial y_n}{\partial p_2} & \cdots & \frac{\partial y_n}{\partial p_k} \end{bmatrix} \tag{1.1}$$

We point out the structure of the sensitivity matrix to make an analogy towards auto differentiation methods that can be used to calculate Jacobian matrices. Although calculating the sensitivity matrix is a little more involved, we will show that there are methods that fundamentally calculate columns of this matrix, and then there are methods that fundamentally calculate rows of this matrix. This key insight is really at the heart of strategies aimed at parallelizing this procedure.

The first method for performing local sensitivity analysis is forward sensitivity analysis. Similar to forward-mode automatic differentiation (forward AD), the elements of the sensitivity matrix are calculated in the same forward pass through the ODE solve for the solution variables. In fact forward-mode AD is often used to calculate the parameter Jacobian needed for this process. Also like forward-mode AD, this forward sensitivity analysis also fundamentally calculates columns of the sensitivity matrix, as we will show.

Forward sensitivity analysis involves solving the following systems of ODEs in Equation (1.2) in addition to the ODEs for the solution variables:

$$\frac{d}{dt}\left(\mathbf{S}[:,k]\right) = \frac{\partial \mathbf{f}}{\partial \mathbf{u}}\mathbf{S}[:,k] + \frac{\partial \mathbf{f}}{\partial p_k} \tag{1.2}$$

Where $\mathbf{S}[:,k]$ is the kth column of the sensitivity matrix (Nx1), $\frac{\partial \mathbf{f}}{\partial \mathbf{u}}$ is the NxN Jacobian matrix of the original ODE function $\mathbf{f}$ with respect to the solution variables, and $\frac{\partial \mathbf{f}}{\partial p_k}$ is the kth column of the parameter Jacobian matrix (Nx1). Both of the Jacobian matrices are functions of the solution variables, time, and the model parameters, but are not a function of the elements of the sensitivity matrix.

The key insight here is that as seen in Equation (1.2), calculating columns of the sensitivity matrix, $\mathbf{S}[:,k]$, only depends on $\mathbf{S}[:,k]$, the solution variables, the model parameters, and time. Crucially, it does not depend on other columns of the sensitivity matrix. This makes the ODEs for calculating a single column of the sensitivity matrix completely decoupled from the ODEs for calculating any other column of the sensitivity matrix. In fact, it is possible to calculate a single column of this matrix using forward sensitivity analysis without ever calculating any of the remaining columns.

We can exploit this property of forward sensitivity analysis to parallelize the whole calculation by calculating a single column or even batches of columns at a time in a single thread. Note however that the forward sensitivity equations do depend on the solution variables as a function of time. Therefore, the solution variables must be calculated either simultaneously with the sensitivity equations, or solved first to supply an interpolated solution for the subsequent forward sensitivity solves.

As we alluded to earlier, though, forward sensitivity analysis is not the only method for performing local sensitivity analysis. Adjoint sensitivity analysis, which utilizes reverse-mode AD, also calculates local sensitivities. In much the same way that reverse-mode AD calculates rows of a Jacobian, adjoint sensitivity analysis calculates rows of the sensitivity matrix at a specific time point. By solving the adjoint problem, we get as an output the sensitivity of a single solution variable with respect to all of the parameters, which is just a row of the sensitivity matrix. The solving the adjoint problem also requires the forward solution, but unlike forward sensitivity analysis the forward solution cannot be solved simultaneously with the adjoint problem. Therefore, the forward solution is first calculated, and then it is used to solve the adjoint problem in the reverse pass. Solution to various rows returned by adjoint sensitivity analysis can be calculated independently of one another, and as such can be performed on different threads to calculate the whole matrix.

1.3. Performance Implications of Forward versus Adjoint Sensitivity Analysis. Much like forward-mode versus reverse-mode AD, there are performance considerations to each when calculating the full sensitivity matrix. In general, reverse-mode AD tends to have a higher overhead cost compared to forward-mode AD. Since forward sensitivity analysis utilizes forward-mode AD (and vice-versa for adjoint sensitivity analysis), adjoint sensitivity analysis can be slower when used to calculate the whole sensitivity matrix. Work in this area by Rackauckas et al.[4] showed that forward sensitivity analysis performed better than adjoint sensitivity analysis when the number of parameters is small (less than 100). However, they also showed that adjoint sensitivity analysis can be more efficient for much larger models, like the ones that might be constructed using software like RMG.

In related problems of calculating Jacobians, one technique for maximizing efficiency is to use mixed-mode AD (forward-mode AD for columns, reverse-mode AD for rows) to calculate rows/columns of the Jacobian based off of the sparsity pattern of the Jacobian. Unfortunately, mixed-mode sensitivity analysis would offer little benefit, as the sensitivity matrix is dense.

Despite this, there are still some optimizations to be made in the specific context of refining large chemical mechanisms. Large chemical mechanisms, there are many more intermediate species than there are species that are considered key observables. This means that to refine these models one only has to calculate a small subset of the rows of the sensitivity matrix, which makes adjoint sensitivity analysis quite an appealing option. A saving feature though of forward sensitivity analysis is that it can calculate the sensitivity matrix at multiple time points in a single pass, making it possible to get an interpolated continuous solution for the sensitivity matrix. Given this, the most efficient calculation path might be a balancing act based on the number of key observable species, the size of the model, and the number of time points that need investigating

2. Methods.

2.1. Simulation Software. While there are many software packages specifically designed for simulating and analyzing chemical methods across many different programming languages, we used the Reactions Mechanism Simulator (RMS) software[2], which is an open-source software package written in Julia. As it is a sister project under the Reaction Mechanism Generator organization, it is specifically designed to work well RMG generated mechanisms. The software serves to load in a chemical mechanism and setup the corresponding system of ODEs for simulation. RMS relies heavily on the DifferentialEquations.jl[1] software suite for simulating mechanisms, and performing both forward sensitivity analysis and adjoint sensitivity analysis using the DifferentialEqautions.jl API. Instructions for installing RMS and its necessary environment are available on the RMS wiki page on GitHub.

Prior to this work, the authors and developers of RMS (which at the time the author of this work was not a part of) had already implemented serial adjoint sensitivity analysis and serial forward sensitivity analysis. Included in this already existing implementation were function calls for the analytic Jacobian matrices (with respect to the solution variables and separately the parameters), and function calls for calculating these matrices using ForwardDiff.jl. In this work we exclusively used the constant temperature and pressure ideal gas batch reactors already implemented in RMS. The authors of RMS have spent and continue to spend a lot of effort making sure that RMS is a reasonably efficient code, and as such little effort was spent on this work in optimizing the RMS serial code.

2.2. Benchmarking. Two different chemical mechanisms were used in this study, called "small" and "medium" based on their size relative to commonly generated RMG mechanisms. The small model consists of a mechanism for ethane pyrolysis at 1350 K and 1 bar. The model only contains 27 solution variables and 96 parameters. Simulating the model to 1 ms starting from pure ethane yielded good enough conversion for benchmarking. The medium model consists of a n-octane pyrolysis mechanism at 1000 K and 20 bar. The model consists of 120 solution variables and 3345 parameters. Simulating the model to 0.2 s starting from pure n-octane yielded good enough conversion for benchmarking.

Benchmarking was performed on two different systems. The first was a Dell Latitude E7270 laptop with a dual-core Intel Core i7-6600U CPU at 2.60 GHz and 16 GB of RAM. The second system, where the majority of the benchmarking took place was part of the MIT Supercloud cluster using a single node, 40-core Intel Xeon Gold 6248 CPU at 2.50GHz equipped with 377 GB of RAM. When running on supercloud, exclusive access to the node was requested to avoid errors in timings due to competing processes.

The BenchmarkTools.jl macro @btime was used to perform the benchmarking to gather timings and memory usage due to allocations. The medium model was too big to profile for performing the full sensitivity analysis. Instead, a subset of the rows/columns were calculated, and the exact number used is noted in this work where needed. For this reason the medium model was only ever benchmarked on Supercloud. As a baseline method, we tested running forward sensitivity analysis on the small mechanism, both using the Jacobian functions available in RMS and without, which then relied on ODEForwardSensitivityProblem from DifferentialEquations.jl to use ForwardDiff.jl for calculating the Jacobians. Both methods had very similar timings, but the method without analytical Jacobians was used as the baseline.

2.3. Parallelization of Simultaneous Forward Sensitivity Analysis (SFSA). In serial operation of forward sensitivity without analytical Jacobians, RMS defines

a function, ‘dydt!(dy, y, p, t)‘ that defines the ODE system for the solution variables y. Without being given the analytical Jacobians for the solution variables and parameters to ‘dydt¡, ODEForwardSensitivityProblem calculates the Jacobian using ForwardDiff. To do this, it figures out the number of parameters from the length of the parameter vector supplied to ‘dydt¡.

Parallelizing this procedure relies on calculating the sensitivity to only a subset of the parameters, with the number of parameters in a single calculation being the batch size. While there are likely other places to exploit parallelization, such as within the ODE solver itself, a general rule of thumb is to parallelize the outer most loop, which is what we calculating a series of problems with a batch of parameters accomplishes. To do this, we need to supply the ODE solver with the batch of parameters that we want to calculate, while holding the other parameters fixed. The easiest way to accomplish this is to enclose the fixed parameters (using a closure) to trick the ODE solver into thinking that only the parameters include only those in the batch. With this, a list of several ODEForwardSensitivityProblem objects can be constructed to cover all parameters in batches, which can be calculated by looping over the list with the @threads macro in Julia. This was chosen over Threads.@spawn with the assumption that any iteration should take roughly the same time. An outline of this procedure is given below in Algorithm 2.1

Algorithm 2.1 SFSA

```
Define probList = []
while remaining parameters do
  Define batchDydt!(dy, y, p, t) = dydt!(dy, y, [batchPs; fixedPs])
  Append ODEForwardSensitivityProblem(batchDydt) → probList
end while
for prob in probList using @threads do
  solve(prob)
end for
```

Note that while this algorithm is simplistic, it has a potential downside in that every single problem being solved by the threads is re-solving the forward solution.

2.4. Parallelization of Interpolated Forward Pass Forward Sensitivity Analysis (IFPFSA). Recalculating the forward solution every batch can be avoided by calculating the forward solution up-front. By default, the DifferentialEquations.jl package returns interpolated forward solution objects, which are function that can be passed to setup separate systems of ODEs that only contain the forward sensitivity equations.

In this method, the analytical Jacobians were used to help create the forward sensitivity equations, which were coded by hand. To keep things simple, the batch size fixed at 1 (i.e. every problem a thread tackles computes one column of the sensitivity matrix at a time). An outline of this methodology is given by Algorithm 2.2

2.5. Parallelization of Adjoint Sensitivity Analysis (ASA). Adjoint sensitivity analysis was already implemented in RMS prior to this work. For adjoint methods the forward pass is needed each time, but can be re-used from solve to solve. Therefore, parallelizing this procedure required practically no effort. While the solver tolerances for forward sensitivity analysis matched that of the forward solution of atol=1e-20 and rtol=1e-12, tolerances of atol=1e-8 and rtol=1e-6 were used for the adjoint sensitivity analysis, with atol-1e-6 being used for the medium model. The

Algorithm 2.2 IFPFSA

```
Define probList = []
Solve fsol = solve(dydt)
while remaining parameters do
    Define dsdt = d/dt s_k = J_y(fsol)*s_k + J_p(fsol)[k]
    Append ODEProblem(dsdt) → probList
end while
for prob in probList using @threads do
    solve(prob)
end for
```

exact value used for the tolerances seemed to have a significant effect on the runtime. It was observed that if the tolerances are set too high, then some of the rows can take significantly longer to calculate the others. In this case, Threads.@spawn should be used since its dynamic scheduler can balance the load.

2.6. Code and Materials. All of the code used in this work is available on GitHub at . Included are the RMS input files to load the mechanisms in, log files with profiler timings, and a file 'project.jmd' that includes all of the relevant code for this work. Several other Julia files are present that contain the same code but packaged for profiling the code on Supercloud.

3. Results. Overall, it was discovered that each of the parallelization schemes were significantly faster than the serial scheme, even for just 2 threads. For example, the baseline simulation of calculating forward sensitivity analysis on the small model in serial took 70 seconds on the laptop, whereas the slowest of the parallel methods even for just 2 threads was around 10 seconds, with similar timings on Supercloud.

For the first parallel method of Simultaneous Forward Sensitivity Analysis (SFSA), where the forward solution is solved with the sensitivities for batches of parameters, the total calculation method does not seem to be a strong function of the number of threads used, as seen in Figure 1.

Regardless of the number of threads used, the total time is about an order of magnitude reduction from the serial method. Increasing the number of threads helps a little bit at first, which suggests that the time limiting step is not solving the ODE systems. A better picture of what is going on here can be gleaned from performing SFSA at different batch sizes for the small model as shown in Figure 2.

It turns out that the batch size plays a crucial role in total time required, with smaller batches being significantly faster than larger batches, despite requiring more problems to solve to calculate all columns of the sensitivity matrix. A key here might be the fact that the total memory taken up by allocations increases as the batch size increases, and the total timings seems to be tracking this perfectly. It is worth noting that in any case the memory usage is less than what was used for the serial baseline calculation (which uses only previously existing RMS code) of 70 GB, so it is unlikely the memory usage is a result of this implementation of SFSA.

We see similar behavior from studying the medium model versus the batch size, as shown in Figure 3.

Once again, increasing the batch size increases the memory of allocations, which tracks with increasing total time to calculate the first 256 columns of the sensitivity matrix.

While for the small model total time did not depend strongly on the number of

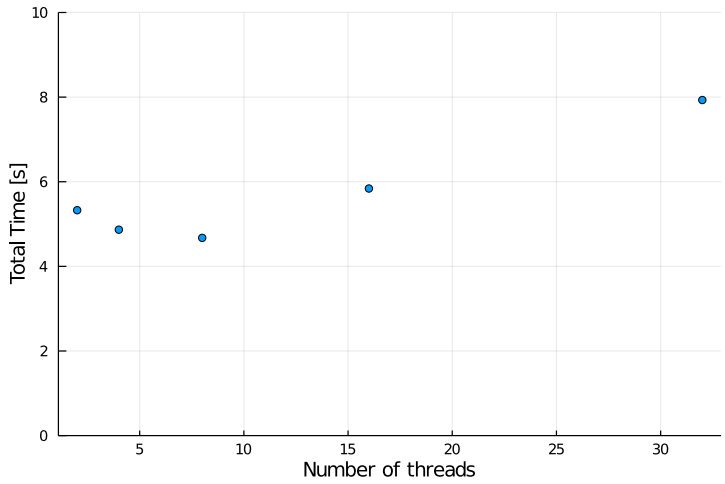

FIG. 1. *Total time to calculate the full sensitivity matrix of the small model using the SFSA parallel methodology with a batch size of 2. The increase in total time for the larger number of threads is likely a consequence of the fact that only 96 columns need to be calculated*

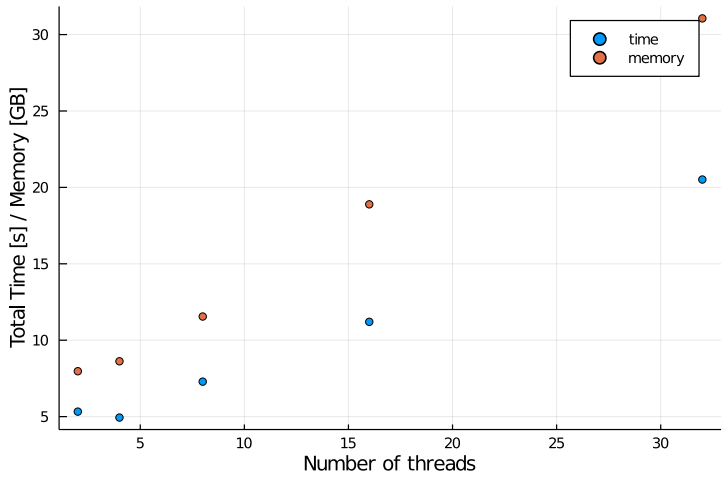

FIG. 2. *Total time to calculate the full sensitivity matrix of the small model using the SFSA parallel methodology with 2 threads versus batch size. The memory usage of allocations goes up with the batch size, as does the total time required*

threads used, we expect in general for this to be the case, especially when the time limiting step is solving the ODE problems as opposed to setting them up. That is exactly what we see for the SFSA method when we study the medium model as shown in Figure 4.

As expected, increasing the number of threads decreases the total time, with decent but not superb parallelization efficiency. Going from 2 threads to 32 threads decreases the total calculation time by a factor of 6 instead of the theoretical maximum of 16.

There are several possible sources for this inefficiency. Some of the time is spent

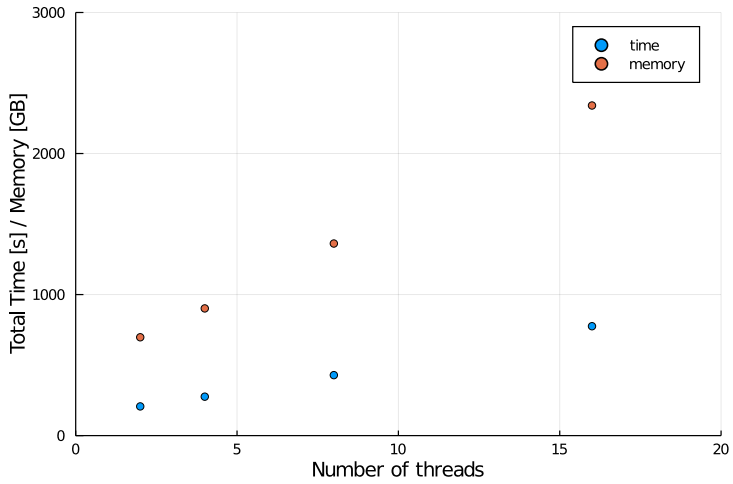

FIG. 3. *Total time to calculate the full sensitivity matrix of the medium model using the SFSA parallel methodology with 2 threads versus batch size. The memory usage of allocations goes up with the batch size, as does the total time required*

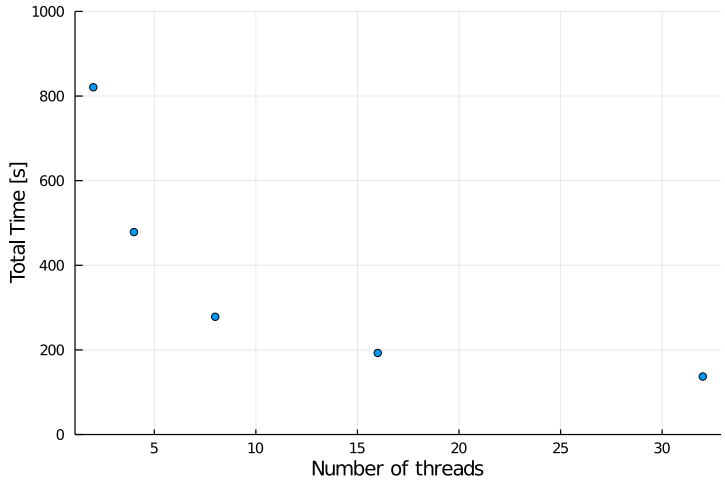

FIG. 4. *Time to calculate the 256 columns of the sensitivity matrix of the medium model using the SFSA parallel methodology with a batch size of 2 versus number of threads. As expected, increasing the number of threads reduces the total time of the calculation*

in setting up the ODEProblems, which is not parallelized. The methodology also has other inefficiencies that do not depend on the number of threads used. Every time another iteration is done to calculate another ODEProblem in SFSA, the forward solution is being resolved for. If the batch size is small (which otherwise reduces the total time) then these solution variables are roughly the same size as the sensitivity variables being solved for, which results in a lot of wasted time. This fact was the motivation behind implementing the Interpolated Forward Pass Forward Sensitivity Analysis (IFPFSA), which solves for the solution variables only once. The SFSA and IFPFSA methods are compared against each other for the small model in Figure 5.

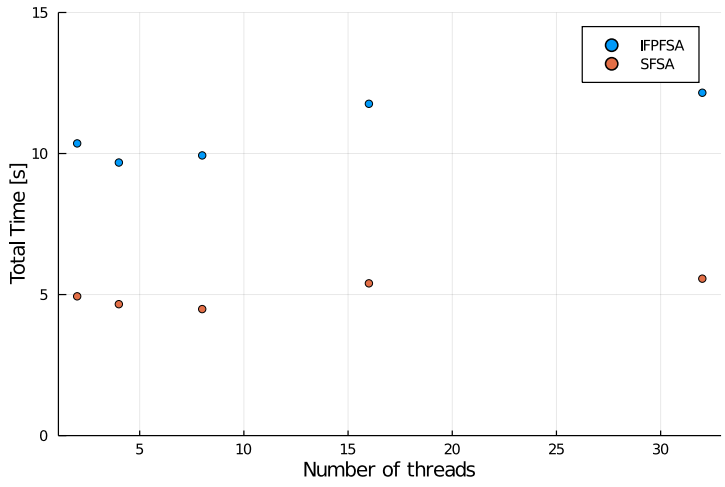

FIG. 5. *Comparison of IFPFSA to SFSA with a batch size of 4 versus the number of threads for the small model. Neither method depends strongly on the number of threads, but the SFSA method consistently beats out the IFPFSA method at this scale, also using less memory*

For the small model, neither methodology depends strongly on the number of threads at this scale. Interestingly though, SFSA consistently beats out IFPFSA at this scale. This is likely due to the specific implementation of IFPFSA. The method was handed analytical Jacobians, but due to how this functionality is implemented in RMS this involves evaluating the whole parameter Jacobian each time when needed as opposed to just evaluating the columns needed for the parameters being calculated at that instant. This adds some overhead to the method, but despite this, we expect this method to perform better for larger systems, as this overhead should be no match for the reduction in variables being solved for in the ODE in IFPFSA. This is exactly what we see in Figure 6.

For the medium model, both methods see a reduction in calculation time with increasing number of threads as expected. However, this time IFPFSA is consistently faster than SFSA, by a factor of 6.5 for 2 threads, and still a factor of 4.5 for 32 threads. This means that the parallelization efficiency of IFPFSA is less than that of SFSA, likely the result of the fact that the time to setup the problem is closer to the time needed for all of the solves, mostly because the solves in IFPFSA are faster. In this way, it appears that IFPFSA is the better algorithm for larger scales.

Finally, it is worth comparing the results of parallelizing Adjoint Sensitivity Analysis (ASA). Despite this, it is actually a little bit complicated to compare the two methods. For one, ASA only calculates the sensitivity matrix at a single time point, unlike SFSA or IFPFSA which return interpolated continuous solution up to the fial time point of the simulation. Secondly, the tolerances supplied to the different methodologies are not the same, and ASA seems highly sensitive to tolerances. What can take 3 seconds to do at atol=1e-8 for ASA can take at least a factor of 10 longer when atol=1e-10. A fair comparison would account for this by determining the minimum tolerance needed to achieve a threshold accuracy. Even with these complications, it is worth comparing the performance of ASA to SFSA (which beat out IFPFSA for the small model) on the small model as shown in Figure 7.

Once again, neither ASA or SFSA depend much on the number of threads used.

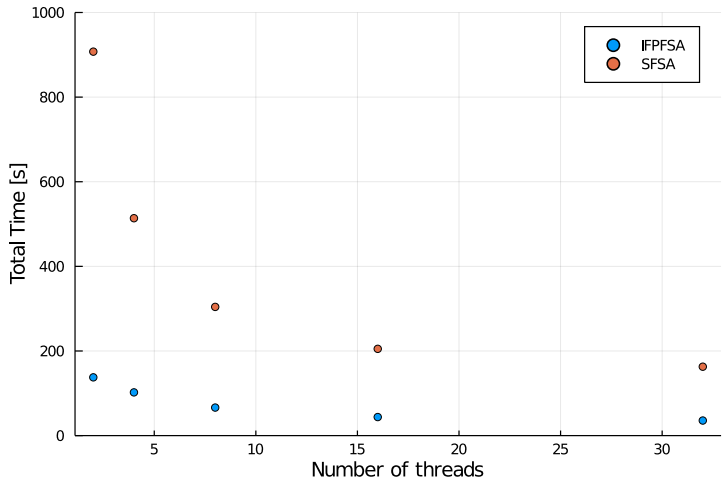

FIG. 6. *Comparison of IFPFSA to SFSA with a batch size of 1 versus the number of threads for the medium model. Both methods see improvements with additional threads, but the IFPFSA methods is consistently better than the SFSA method at this scale*

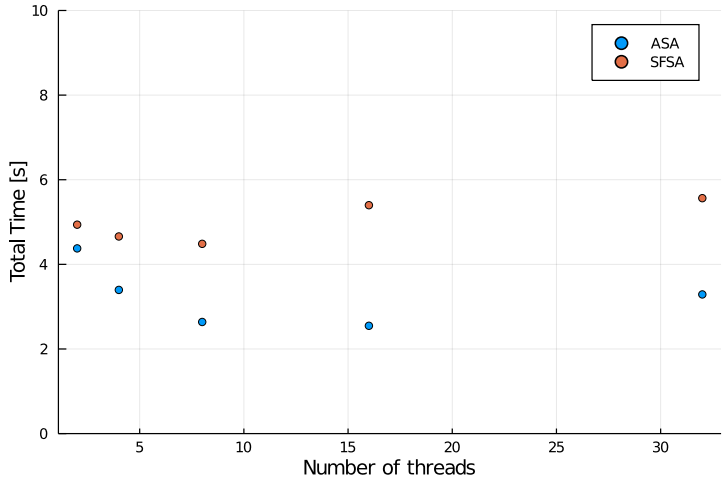

FIG. 7. *Comparison of ASA to SFSA with a batch size of 4 versus the number of threads for the small model. Neither method depends strongly on the number of threads. It is hard to directly compare timings between these methods, but ASA was able to have consistently smaller timings*

.

However, ASA was able to be out SFSA just barely. For the reasons stated above, this should be taken with a slight grain of salt, but it at least seems like ASA takes the same order of magnitude as SFSA.

Finally, we can see that ASA does in fact benefit from being parallelized, as shown in Figure 8.

In this test ASA was used to only calculate the first 16 rows of the sensitivity matrix, which results in some odd behavior when the number of threads equals the number of rows.

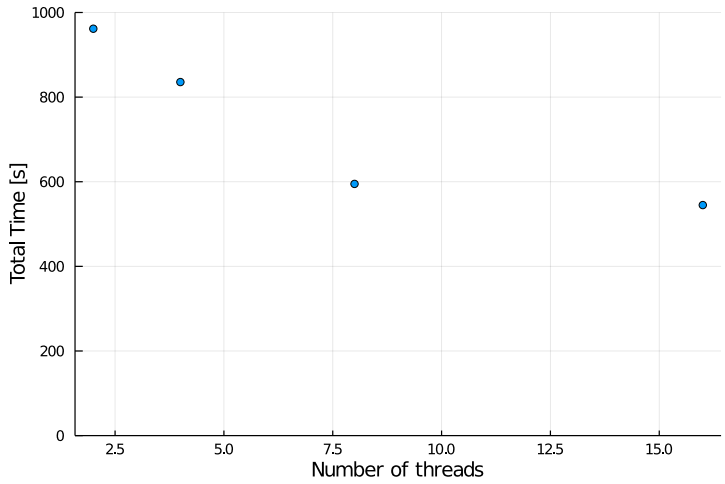

FIG. 8. *Time to calculate 16 rows of the sensitivity matrix of the medium model using the ASA parallel methodology. The increase in total time for the larger number of threads is likely a consequence of the fact that 16 rows were being calculated by 16 threads. Threads.@spawn should be used in this scenario*

4. Analysis.

4.1. Decreases in memory tracks with decreases in timings. The biggest surprise from this work was that parallelization strategies for sensitivity analysis tend to be fastest when the batch size is small. Our prior expectation before this work was that the optimal batch size would be the one that allocates an even load across all available threads such that each thread only solves one problem in total.

One possible explanation for this is that reducing the batch size seems to correlate strongly with reducing the memory usage of allocations. Intuitively this makes sense, as using a smaller batch size leads to smaller-sized ODEs for each iteration. This mean that any vectors, and more crucially and matrices that must be allocated during the solve are also much smaller. Reducing the size of vectors by a factor of F does not matter in this case, because this reduction increases the number of ODE systems by a factor of F. However, matrices would be decreased by a factor of F^2, which still nets a decrease in total memory by a factor of F despite needing F times as many ODE systems to solve. This linear reduction in memory with decreasing batch sizes seems to be consistent with this fact, and if the runtime of the algorithm is being dominated by allocating memory (seems likely as even the small model needed 70 GB in allocations for serial) then this explains the decrease in runtime with small batch sizes perfectly.

4.2. Theoretically Small Batch Sizes are Best. On top of this, there is compelling reasoning why keeping the batch size small is beneficial from a theoretical standpoint. If the time limiting step for sensitivity analysis is solving systems of ODEs, then it makes sense to talk about the time complexity of ODE solvers. The time complexity of an ODE solve is usually $O(N^3)$, where N is the number of variables being solved for at once. Serial forward sensitivity analysis solves not only for the number of solution variables Y but also all elements of the sensitivity matrix for P parameters, which is $Y * P$ elements in total. Therefore, $N = Y + Y * P$. For the parallel implementation, the batch size determines the number of iterations M, as

$M = \frac{P}{batchsize}$. As the batch size decreases, M increases.

In parallel, we now have to solve M ODE systems, but the size of these systems has gone down. For SFSA the size of each ODE problem is now $Y + \frac{Y*P}{M}$ (for IFPFSA it is $\frac{Y*P}{M}$). Since for our models $P * Y >> Y$, it follows that $Y + \frac{Y*P}{M} \approx \frac{N}{M}$, leading to a time complexity of $O((\frac{N}{M})^3) = \frac{1}{M^3} * O(N^3)$. Since we have to solve M of these systems, the overall time complexity is $\frac{1}{M^2} * O(N^3)$, a reduction by a factor of $\frac{1}{M^2}$.

Although this reasoning appears sounds, decreasing the batch size does not results in a speed-up nearly as much as this suggests. It is possible that this is because the time complexity depends mostly on the memory allocations, and this effect would be noticeable if the number of allocations were low.

4.3. Adjoint versus Forward Sensitivity Analysis. Finally, although it is hard to compare the adjoint method directly with forward sensitivity analysis, it is still worth exploring whether forward sensitivity analysis is by far inferior to adjoint methods for larger models. To determine this, we can extrapolate to calculate the time it would have taken IFPFSA and ASA to calculate the full sensitivity matrix. IFPFSA was able to calculate 256 columns out of 3345 of the sensitivity matrix using 4 threads in 102 seconds. Extrapolating this out, this would take 22 minutes to calculate the full sensitivity matrix. On the other hand, calculating 16 rows of the sensitivity matrix using ASA with 4 threads took 835 seconds. Extrapolating this out to all 120 rows, it would take ASA 104 minutes to calculate the full matrix. Surprisingly, even though the medium model had many parameters at 3345, forward sensitivity analysis was able to beat out adjoint sensitivity analysis by a factor of almost 5.

It is possible that this is an unknown implementation issue with adjoint sensitivity analysis in RMS, but at the very least forward sensitivity analysis is very competitive with adjoint sensitivity analysis even for large systems.

Of course, this is assuming that the full sensitivity matrix is desired. For working with large chemical mechanisms, only a subset of the rows are need, potentially a small enough fraction even in this case for adjoint sensitivity analysis to be the faster option. Balancing this with tolerances and obtaining values at enough time points, it is likely that determining the optimal algorithm is an open question to be solved on a case by case basis.

5. Implementation Performance Considerations. When implementing these methods, there were a number of performance considerations that have yet to be mentioned, especially with regards to the implementation of IFPFSA. As hinted earlier, the implementation as it stands in fine but not optimal. For one, this method involves computing the full parameter Jacobian for each problem instead of just the necessary columns that match the non-fixed parameters. To get around this issue, an implementation of IFPFSA was coded up (see alternative IFPFSA in project.jmd) that relied on using ForwardDiff.jacobian to calculate the Jacobian for only the parameters of interest. Unfortunately, this implementation proved slower, so it was not studied further.

Another area of improvement for the IFPFSA method is the fact that Jacbian matrices have to be allocated each time before calling the functions for calculating the Jacobians. One discarded implementation tried to improve this by pre-allocating one Jacobian matrix per thread up front, but this also proved to be slower.

The biggest improvement that could be made (beside general improvements to RMS to reduce allocations) would be to create a function able to return specific columns of the analytical Jacobians.

Finally, various adjoint sensitivity algorithms available in DifferentialEquations.jl

were tested out. However, InterpolatingAdjoint appeared to work the best.

6. Conclusions. In this work, we were able to implement and analyze methods for performing local sensitivity analysis capable of handling large chemical mechanisms. The parallel methods for performing forward sensitivity analysis were significantly superior to the previous serial implementations, in large part due to the speed-up from calculating the sensitivities in batches. In fact, even in serial, forward sensitivity analysis should be performed in batches using small batch sizes. There are theoretical reasons for why this should be the case, but it is possible that this is simply a result of reducing memory allocation and that RMS needs to be further optimized. To rule this out further studies should be performed using a fully optimized code for building models to see if memory allocation still appear to be rate limiting or if the theoretical speed-up of $\frac{1}{M^2}$ is achievable.

Adjoint sensitivity analysis in parallel also performed well, although it was not able to be out IFPFSA for a model with a large number of parameters at roughly 3345. This could also be an implementation detail inside of RMS, and should be explored further by testing on optimized code.

Still, adjoint sensitivity analysis might be preferred over forward sensitivity analysis is only a small subset of the rows in the sensitivity matrix are needed. This should be determined on a case-by-case basis.

Acknowledgments. The author thanks Chris Rackauckas for many helpful discussions and guidance on this work. The author also thanks Matt Johnson and Hao-Wei Pang for their help with RMS and for implementing serial code for forward and adjoint sensitivity analysis. The author also thanks Ranjan Anantharaman and Sungwoo Jeong for their help and guidance. Finally, the author thanks Yen-Ting Wang for supplying the mechanism files for the medium model.

REFERENCES

[1] *Differentialequations.jl*, https://github.com/SciML/DifferentialEquations.jl.
[2] *Reactionmechanismsimulator.jl*, https://github.com/ReactionMechanismGenerator/ReactionMechanismSimulator.jl.
[3] C. W. Gao, J. W. Allen, W. H. Green, and R. H. West, *Reaction Mechanism Generator: Automatic construction of chemical kinetic mechanisms*, Computer Physics Communications, 203 (2016), pp. 212–225, https://doi.org/10.1016/j.cpc.2016.02.013, http://dx.doi.org/10.1016/j.cpc.2016.02.013.
[4] C. Rackauckas, Y. Ma, V. Dixit, X. Guo, M. Innes, J. Revels, J. Nyberg, and V. Ivaturi, *A Comparison of Automatic Differentiation and Continuous Sensitivity Analysis for Derivatives of Differential Equation Solutions*, arXiv, (2018), pp. 1–14, https://arxiv.org/abs/1812.01892.