

## Parallel sensitivity analysis for DAEs with many parameters

WENJIE ZHU<sup>1</sup> AND LINDA PETZOLD\*<sup>2</sup>

<sup>1</sup>*Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455, USA*

<sup>2</sup>*Department of Mechanical and Environmental Engineering, University of California, Santa Barbara, CA 93106-5070, USA*

---

### SUMMARY

In this paper, we discuss the parallel computation of the sensitivity analysis of systems of differential-algebraic equations (DAEs) with a moderate number of state variables and a large number of sensitivity parameters. Several parallel implementations based on DASSLSO are explored and their performance when using the Message Passing Interface (MPI) on an SGI Origin 2000 is compared. Copyright © 1999 John Wiley & Sons, Ltd.

### 1. INTRODUCTION

Sensitivity analysis of large-scale differential-algebraic systems is important in many engineering and scientific applications, such as chemical, mechanical and electrical engineering and economics. Sensitivity analysis generates essential information for parameter estimation, optimization, control, model simplification and experimental design, and for assessing the effects of uncertainty in the model and the computation. Consequently, algorithms which perform such an analysis in an efficient and rapid manner are invaluable to researchers in many fields.

For sensitivity analysis of DAE systems with relatively few parameters, the issues of parallelization are essentially the same as for the solution of ODEs or DAEs. A survey of parallel numerical methods for initial value problems (IVP) for ODEs is given by [1]. The methods for achieving parallelism in initial value problem solution can be classified into two main categories:

1. parallelism across the method
2. parallelism across the problem.

The first class of methods exploits the parallelism in the numerical methods for ODEs/DAEs, such as parallel block Runge–Kutta methods [2–4] which make use of several concurrent function evaluations within each step. Despite its appealing generality and problem-independence, there are rather severe limitations to the amount of parallelism which is achievable by this approach. The second class of methods makes use of the parallelism in the ODEs themselves. *Dynamic iteration* or *waveform relaxation* [5–8] methods involve a partitioning of the ODE/DAE system onto the parallel processors, and offer a potential of using many processors to speed up the solution of large-scale problems.

\*Correspondence to: Linda Petzold, Department of Mechanical and Environmental Engineering, University of California, Santa Barbara, CA 93106-5070, USA.

Contract/grant sponsor: DOE, USA; Contract/grant number: DE-F603-98ER25354

Contract/grant sponsor: National Science Foundation; Contract/grant number: CCR-98-9896198

Another possibility for large-scale ODE/DAE systems is to parallelize at the level of the linear algebra. In [9], a parallel version with good scalability of the large-scale DAE solver DASPK is developed for very large scale DAE problems, such as those generated from semi-discretization of PDEs in two or three dimensions.

Here we are concerned with a different class of problems: sensitivity analysis for DAEs with many parameters. For example, in some optimization and optimal control problems, the number of optimization parameters, i.e. sensitivity parameters, can be much larger than the size of the original DAE system.<sup>†‡</sup> Although one integration of the DAE system may not be costly, in a design optimization the original and sensitivity systems will be integrated many times. Hence, efficiency is a very important issue. A potential source of parallelism for these problems is across the parameters. In this paper we will investigate several possible strategies for parallelization across the parameters, with the aim of good performance on the distributed memory systems which have in recent years become economically so attractive. Our numerical experiments are actually done with MPI[10] on a shared memory SGI O2000 system. In the MPI standard, the processors communicate only by passing messages. However, since the SGI O2000 is a shared memory machine, the MPI implementation on this machine is optimized to use the underlying shared memory, and the communication cost is reduced. Thus, the communication cost on it is lower than on real distributed memory systems. This issue will be discussed in more detail in Section 4. We will see that because of the costs of communication and synchronization, some of the more obvious approaches for parallelization of this class of problems may not be effective, and that the strategy with the least centralized control performs best.

To facilitate our discussion of parallel methods for sensitivity analysis, we first introduce three well-established methods for sensitivity analysis in some detail, and briefly compare them for sequential computation in Section 2. Several possible strategies for parallel implementation are described in Section 3. Computational results are presented and analyzed in Section 4.

## 2. SENSITIVITY ANALYSIS OF DAEs

### 2.1. Problem definition

To illustrate the basic approach for sensitivity analysis, consider the general DAE system with parameters

$$F(t, y, y', p) = 0, \quad y(0) = y_0 \quad (1)$$

where  $y \in R^{n_y}$ ,  $p \in R^{n_p}$ . Here,  $n_y$  and  $n_p$  are the dimension and the number of parameters in the original DAE system, respectively. Sensitivity analysis entails finding the derivative of the above system with respect to each parameter. This produces an additional  $n_s = n_p \cdot n_y$  sensitivity equations which, together with the original system, yields

$$F(t, y, y', p) = 0 \quad (2)$$

$$\frac{\partial F}{\partial y} s_i + \frac{\partial F}{\partial y'} s'_i + \frac{\partial F}{\partial p_i} = 0, \quad i = 1, \dots, n_p \quad (3)$$

<sup>†</sup>L. R. Petzold and W. Zhu, 'Model reduction for chemical kinetics: An optimization approach', *AICLe Journal*, April 1999, pp. 869–886.

<sup>‡</sup>L. R. Petzold and W. Zhu, 'Sensitivity analysis of differential-algebraic equations: A comparison of methods on a special problem', *Applied Numerical Mathematics*, to appear.

where  $s_i = \partial y / \partial p_i$ . There are three well-established direct methods to solve the sensitivity system (3). Here we describe the three methods and discuss briefly their performance. A detailed comparison of computational complexity and efficiency on sequential computers of the methods applied to the class of problems described here is given elsewhere.<sup>‡</sup>

In the following discussion, we assume that the basic algorithm for the DAE problem (1) is DASSL[11]. For each time step, a predictor polynomial that interpolates the solution at previous time steps is used to obtain an initial guess  $y_{n+1}^{(0)}$  and  $y'_{n+1}{}^{(0)}$ . Then a modified Newton iteration is used to solve the corrector formula for  $y_{n+1}$ . DASSL uses the fixed leading coefficient form of the  $k$ th-order BDF formula for the corrector[11]. For all three sensitivity methods, the predictor step for the sensitivity equations is the same but the corrector formula is solved in different ways, which we will describe in detail later.

### 2.2. Simultaneous corrector method

We discuss the simultaneous corrector method in the context of DASSLSO, an extension of DASSL for sensitivity analysis[12]. The most recent version<sup>§</sup> has an option for using ADIFOR-generated sensitivity equations and Jacobian matrix. ADIFOR is an automatic differentiation package for Fortran code. Supplied with a user defined code to compute a function  $y = f(x)$ , where  $x$  is the independent variable and  $y$  the dependent variable, ADIFOR generates a new code for computing the exact value of  $\frac{\partial y}{\partial x}$  together with  $y$ . ADIFOR generated code is found to be more reliable and efficient than the finite difference method in many cases,<sup>§</sup> especially for highly non-linear functions. The sensitivity equations are formed by the seed-matrix option of ADIFOR2.0. The seed-matrix option of ADIFOR2.0 allows the direct computation of the product of the Jacobian and another matrix, which is called the seed-matrix. If the seed-matrix is of much lower rank than the Jacobian, this product can be computed at a cost which is much lower than that of the matrix vector product.

Defining  $Y = [y, s_1, \dots, s_{n_p}]^T$  and  $F = [F(t, y, y', p), \frac{\partial F}{\partial y} s_1 + \frac{\partial F}{\partial y'} s'_1 + \frac{\partial F}{\partial p_1}, \dots, \frac{\partial F}{\partial y} s_{n_p} + \frac{\partial F}{\partial y'} s'_{n_p} + \frac{\partial F}{\partial p_{n_p}}]^T$ , the combined system can be rewritten

$$F(t, Y, Y', p) = 0, \quad Y(0) = \begin{pmatrix} y_0 \\ \frac{\partial y_0}{\partial p_1} \\ \vdots \\ \frac{\partial y_0}{\partial p_{n_p}} \end{pmatrix} \tag{4}$$

The simultaneous corrector method solves (4) as one non-linear dynamic system without making use of the linearity of the sensitivity equations and the fact that the linear sensitivity equations can be obtained exactly after solving for the state variables.

Approximating the solution to the combined system by a numerical method, for example the  $k$ th-order BDF formula with step size  $h_{n+1}$ , yields the non-linear system of the corrector iteration

$$G(Y_{n+1}) = F \left( t_{n+1}, Y_{n+1}, Y'_{n+1}{}^{(0)} - \frac{\alpha_s}{h_{n+1}} (Y_{n+1} - Y_{n+1}{}^{(0)}), p \right) = 0 \tag{5}$$

<sup>§</sup>D. Clancey, 'Automatic differentiation in the numerical solution and sensitivity analysis of differential algebraic equations', Dept. of Computer Science and Engineering, University of Minnesota, <http://www-users.cs.umn.edu/~clancey>.

System (5) is solved for  $Y_{n+1}$ , where  $Y_{n+1}^{(0)}$  and  $Y'_{n+1}^{(0)}$  are predicted values of  $Y_{n+1}$  and  $Y'_{n+1}$ , which are obtained via polynomial extrapolation of past values [11]. Also,  $\alpha_s$  is the fixed leading coefficient which is defined in [11] and is not important to our discussion here.

Newton's method for the non-linear system produces the iteration

$$Y_{n+1}^{(k+1)} = Y_{n+1}^{(k)} - J^{-1} G(Y_{n+1}^{(k)}) \quad (6)$$

where

$$J = \begin{bmatrix} J & & & & \\ J_1 & J & & & \\ J_2 & 0 & J & & \\ \vdots & \vdots & \vdots & \ddots & \\ J_{n_p} & 0 & \dots & 0 & J \end{bmatrix} \quad (7)$$

$$J = \alpha \frac{\partial F}{\partial y} + \frac{\partial F}{\partial y}, J_i = \frac{\partial J}{\partial y} s_i + \frac{\partial J}{\partial p_i} \text{ and } \alpha = -\alpha_s / h_{n+1}.$$

DASSLSO uses a direct method to solve the linear system arising from the Newton iteration for solving the non-linear system at each time step. In DASSLSO, the full Jacobian matrix  $J$  is not actually computed. Instead, it is approximated by its block diagonal in the Newton iteration. It has been shown in [12] that the resulting iteration is two-step quadratically convergent for full Newton and convergent for modified Newton iteration.

### 2.3. Staggered direct method

Noting that the differential equations for the state variables are independent of the sensitivity variables, the staggered direct method first solves the differential equations for the state variables at each time step.

After the Newton iteration for the state variables has converged, the coefficient matrices of the sensitivity equations are obtained at the current step. This requires updating the Jacobian matrix on every time step. In this way the sensitivity equations are obtained exactly at each step. The linear sensitivity equations are discretized with the same numerical scheme as the state variables. For example, using the  $k$ th-order BDF method with step-size  $h_{n+1}$  as before, we solve, for the  $i$ th parameter,

$$\frac{\partial F}{\partial y'_{n+1}} \left( s'_{i(n+1)} - \frac{\alpha_s}{h_{n+1}} (s_{i(n+1)} - s_{i(n+1)}^{(0)}) \right) + \frac{\partial F}{\partial y_{n+1}} s_{i(n+1)} + \frac{\partial F}{\partial p_i} = 0 \quad (8)$$

for  $s_{i(n+1)}$ .

Because (8) is linear, it can be solved for  $s_{i(n+1)}$  directly without using the Newton iteration

$$J s_{i(n+1)} = \left( -\frac{\partial F}{\partial y'_{n+1}} \beta_i - \frac{\partial F}{\partial p_i} \right) \quad (9)$$

where  $\beta_i = s'_{i(n+1)} - \alpha_s s_{i(n+1)}^{(0)}$ . However, to solve the linear system in this way requires computation and factorization of the Jacobian matrix at each step and also extra storage for

the matrix  $\partial F/\partial y'_{n+1}$ .<sup>¶</sup>

We note that in practice the Jacobian matrix is evaluated using the predicted values of the state variables  $y$  and  $y'$  and then factored. This new Jacobian is then used in the Newton iterations for the state variables and in forming the sensitivity equations. Since the order of the predictor is the same as the corrector, the Jacobian matrix evaluated at the predicted values is an accurate approximation to the Jacobian evaluated after the corrector step. In this way the sensitivity equations are obtained accurately and also the Jacobian is always current for the Newton iterations for the state variables. Thus there are fewer Newton iterations for the state variables for problems with a rapidly changing Jacobian matrix. This scheme is also used in the staggered direct method with iterative improvement.

#### 2.4. Staggered corrector method

The staggered corrector method is similar to the staggered direct method. However, instead of solving the linear system (8) directly, a Newton iteration is used,

$$s_{i(n+1)}^{(k+1)} = s_{i(n+1)}^{(k)} - \hat{J}^{-1} \left( J s_{i(n+1)}^{(k)} + \frac{\partial F}{\partial y'_{n+1}} \beta_i + \frac{\partial F}{\partial p_i} \right) \quad (10)$$

where  $\hat{J}$  may be a factored Jacobian matrix which is saved from a past step and used in the Newton iteration for the state variables, while  $J$  is the current unfactored Jacobian. The updated Jacobian will be factored if the current version cannot converge the Newton iteration. Using this method, an extra copy of the unfactored updated Jacobian  $J$  should be stored for computing the residuals in later iterations as in [13]. Alternatively, as in the simultaneous corrector method, we can use the seed-matrix option of ADIFOR2.0 to avoid evaluating the Jacobian and matrix-vector product; thus the Jacobian matrix is evaluated and factored only when necessary. This can substantially improve the efficiency of the staggered corrector method.

#### 2.5. Staggered direct method with iterative improvement

For some problems with a rapidly changing and/or ill-conditioned Jacobian matrix, the Jacobian needs to be evaluated and factored very frequently to enable the Newton iteration to converge. In fact, the Newton iteration can 'converge' to inaccurate solutions when using old Jacobian matrices in this case, as pointed out elsewhere.<sup>||</sup> On the other hand, the staggered direct method can fail on some problems with an ill-conditioned Jacobian. An iterative improvement method [14] can be used to improve the accuracy of the solution of the linear sensitivity system. This iterative improvement process is actually equivalent to the staggered corrector method with an updated factorization of the Jacobian matrix, i.e.  $\hat{J} = J$  in (10). This of course is the most expensive method. However, it can be the most efficient and robust when ill-conditioned problems are solved.

<sup>¶</sup>Alternatively, the seed-matrix option of ADIFOR2.0 can be used to obtain the matrix vector product  $\frac{\partial F}{\partial y'_{n+1}} \beta_i$  to improve the efficiency and avoid storage of  $\partial F/\partial y'_{n+1}$ .

<sup>||</sup>C. T. Kelley, C. T. Miller and M. D. Tocci, 'Termination of Newton/Chord iterations and the method of lines', *SIAM J. Sci. Comput.*, **19**(1), pp. 280-290.

## 2.6. Brief comparison

Elsewhere,<sup>‡</sup> a detailed comparison of the complexity and efficiency of the three sensitivity analysis methods on a special class of problems is given for sequential computation. It was found that the simultaneous corrector method can be made nearly as efficient as the staggered corrector method via the seed-matrix option of ADIFOR2.0, provided that the diagonal approximation of the full Jacobian converges the Newton iteration well, which happens for most problems. The staggered direct method can be efficient for problems with a well-conditioned Jacobian matrix, but for an ill-conditioned Jacobian matrix this method can fail and iterative improvement is needed. For problems with a rapidly changing Jacobian matrix, the staggered direct method with iterative improvement can be more efficient than the staggered corrector method and simultaneous corrector method.

Balancing the computational requirements of large problems with few parameters against those for moderate-size problems with many parameters, and noting that the staggered direct method may not always yield acceptable results, it appears that the staggered corrector method, implemented via the seed-matrix option of ADIFOR, may be the best overall choice on sequential computers.

For the reasons mentioned above, in the rest of this paper we will just test the staggered direct method with iterative improvement, the simultaneous corrector method and the staggered corrector method, all with seed-matrix only.

## 3. PARALLEL COMPUTING

For sensitivity analysis problems with a large number of parameters, parallel computing is an important issue. Since the sensitivity variables are independent of each other, this can be a source of parallelism. However, it is difficult to give a precise analytic evaluation of the performance of the parallelized versions of the different sensitivity analysis methods. This is because we need to consider the cost of communication between processes (or processors). Different implementations of the parallelization have different communication costs. Our objective is to minimize the cost of communication and maximize the balance of work load between the processors.

We have tried three different ways to parallelize DASSLSO, using MPI[10] and the master-slave paradigm. The master processor is in charge of distributing work and collecting results and in some implementations it is also the centre of control over branching of the computation. Because DASSLSO has a very complex controller to determine the convergence of the non-linear iteration, satisfaction of the error test and choice of step size and order for the next integration step, we decided that the parallel implementation should not change these features. This means that the slave processors should have either maximum control over all the choices mentioned before, leaving the master processor with minimum control, or else have minimum control, leaving the master processor with maximum control. An implementation in between these two implementations will be very complex and will also have more communication and synchronization costs, as we will see later.

### 3.1. Distributed parameters only

The simplest way to achieve parallelism does not require any modification to the DASSLSO code. The sensitivity parameters are divided into different sets and distributed

---

to different processors, with each processor running a copy of DASSLSO and computing a subset of the sensitivity variables. In this way, there is no communication between the processors and there is no need to modify the original DASSLSO code. Also it is obvious that this method is independent of the three different sensitivity analysis methods. The work load can be balanced between processors by allocating the parameters randomly to the processors so that they will be more likely to take a similar number of steps for the whole integration interval and finish their work in a similar amount of time.

The only weak point of this method is that every processor computes the state variables locally and also the Jacobian matrix is computed and factorized locally when needed, which means that the computation of the Jacobian matrix is not parallelized and in some sense is repeated. So the cost of the parallelization is the repeated computation of the Jacobian matrix and state variables for each processor. For many problems the computation time for the Jacobian matrix and the state variables is very small compared to the time for the system with a large number of sensitivity variables, and the former can be ignored. We have seen problems for which the computing of the Jacobian matrix and state variables costs less than 5% of the total computational time. However, there are other problems for which the difference is not that large and a different approach should be considered.

This method can be viewed as the one in which the slaves have maximum control over the computation. Actually the master processor is not really needed for this method.

### 3.2. Distributed parameters with centralized Jacobian computation

Computational experiments show that for many problems, evaluation and factorization of the Jacobian matrix is the most significant source of computation. This is often the case when solving systems containing only state variables, even when the system is only of moderate size. One possible way to improve the method mentioned in the previous Section is to have the Jacobian matrix evaluated and factored centrally, and leave the state variables solved by processors locally.

To do this, we introduce a master processor and two sets of slave processors. One set is the DASSLSO slaves which are used to compute the sensitivity variables together with the state variables. Another set of slaves are Jacobian slaves that do nothing else but compute the Jacobian matrix. Whenever a DASSLSO slave needs to update its Jacobian factorization, it will send a Jacobian updating request to the master processor which will check if there is a newly computed Jacobian matrix available. If there is a factorization of the Jacobian which is obtained close to the time at which the new Jacobian is required, then the master will just send this Jacobian to the DASSLSO slave instead of asking the Jacobian slaves to compute a new one. If there is no such Jacobian available then a new one will be computed by the Jacobian slaves and sent to the DASSLSO slave. In this way, we try to avoid repeated computation of the Jacobian matrix.

An improvement to this method is that several newly computed Jacobian matrices can be kept at the master processor with different time stamps indicating at which time they are evaluated and factored. When a DASSLSO slave sends a Jacobian request to the master, the time stamps will be checked and the Jacobian with the time stamp closest to the time at which a new Jacobian is required will be sent, if the time stamp is within an acceptable time interval. This is done to reduce the recomputing of the Jacobian as much as possible because different DASSLSO slaves may need the Jacobian at very different time points.

The method described above requires communication between processors to obtain the

Jacobian matrix. However, the time spent on the communication should be insignificant since for each processor, the Jacobian is not updated frequently and such communication costs are inevitable if the goal is to share the Jacobian matrix. However, there is another cost which may make these methods impractical. Since DASSLSO is a variable step size, variable order method, there is no guarantee that the DASSLSO slaves will require the Jacobian matrix at even similar time points. So there is a chance that when there are  $n$  DASSLSO slaves, they all require the Jacobian matrices at very different time points so that no work on the Jacobian matrix is actually saved. If the computation time for the Jacobian is small compared to the Newton iteration on the large system of sensitivity variables, then it is possible that the Jacobian slaves will not be busy. But when we have more and more DASSLSO slaves, the number of sensitivity variables per processor decreases. Thus the time spent on computation of the Jacobian matrix may be comparable to the time spent on the Newton iteration of the sensitivity variables. In this case, the Jacobian slaves will receive Jacobian updating requests more frequently from the DASSLSO slaves and the requests will have to be queued, which makes the Jacobian slaves the bottleneck. Though storing multiple copies of newly updated Jacobian matrices can relieve this problem, it cannot guarantee to eliminate it. So this method may be a perfect solution for fixed-step-size methods but may not be good for variable-step-size methods like DASSLSO.

Another modification to this method is to control the step-sizes and orders of the DASSLSO slaves so that they will take the same step-size and order and then the Jacobian matrix can be reused more. This can be done by letting all the DASSLSO slaves negotiate a common step-size and order after each step. After each step, the DASSLSO slaves will communicate to each other to see if every one has had a successful step. If one slave has failed, it is considered that the step is failed for all other slaves and the step-size and order is reduced by each slave using its local information independently and at last a common step-size and order is negotiated. If every slave has a successful step, then similarly each slave will decide its step-size and order for the next step and finally negotiate a common step size and order. The step-size will always be the smallest step-size among the slaves and also the order is taken to be the lowest. In this manner, the Jacobian matrix is reused much more as numerical experiments show. However, synchronization among DASSLSO slaves can be a very significant cost since they have to wait for the slowest one to finish at every step to negotiate a common step-size and order. Thus every step now will take the time of the slowest processor and the whole integration can be slowed down significantly. The gain from the reuse of the Jacobian matrix may not be able to offset this cost.

Yet another possibility might be to control error and step-size based on the state variables only. Then all the DASSLSO slaves will take the same order and step-size and thus they would not need to synchronize and negotiate a common order and step-size after each step. But there is a danger that some of the sensitivities might not be computed accurately.

These methods give most of the control to the DASSLSO slaves and there is very little centralized control by the master processor, which is used to reduce the cost of Jacobian computations.

### 3.3. Fine grain parallelization

The methods mentioned above can be considered as coarse grained parallelization since the parallelization is done at a high level and slave processors have maximum control over the computation. We also tried the fine grained method which gives the master processor

the maximum control and lets the slave do only pure computation distributed by the master processor.

In this method, the master makes all the decisions in the computation: it decides whether the linear and non-linear iterations have converged, the error tests are satisfied and what order and step size to take for the next time step. Whenever the master needs to do expensive computation such as the computation of the Jacobian and the residual, the work is distributed to the slaves. This is done by the master sending the necessary information to the slaves and then collecting the results from the slaves. For example, when the master needs the residual information, it sends the state and sensitivity variable values to the slaves. Then the residuals are computed in parallel by the slaves, with each slave computing part of the residual. Finally, the results are sent back to the master. Similarly, when the master needs to update the Jacobian matrix, it sends state variable values to the slaves and gathers the different parts of the Jacobian from the slaves. When the master needs to solve the linear systems inside the Newton iteration, the Jacobian matrix is sent to slaves, together with different right hand side vectors to different slaves, and then the solutions for different linear systems are gathered from the slaves. These different kinds of results are collected by the master, where they are used to make decisions to control the computation. The slaves are passive computational nodes in that they just receive information and compute results, having no control over the computing process.

It is obvious that this is the method which requires the most communication and synchronization, though the balancing of the work load is done well, which may reduce the cost of synchronization. However, the communication cost can be large, which makes this method impractical on the systems considered here. Though improvements can be made to reduce the cost of communication, the results are still not satisfactory. This method is the one with the most centralized control by the master processor.

#### 4. NUMERICAL EXPERIMENTS

In this Section we compare the performance of the different implementations using a chemical model reduction problem<sup>†</sup> as an example. In this computation, an optimization method is used to reduce the number of reactions in an original chemical mechanism to obtain a reduced mechanism which can approximate the important features of the original one. This aids in understanding of the original mechanism; also the reduced mechanism can be used in place of the original one in later computations to save computational work. A norm, which is chosen according to the future use of the reduced mechanism, is to be minimized by the optimizer. Thus the gradient information about the norm is needed, which is obtained by using sensitivity analysis of the differential equations defining the norm.

The species concentrations and temperature can be described by a system of ordinary differential equations,

$$y' = \sum_{r=1}^N S_r F_r(y), \quad y(0) = y_0 \quad (11)$$

where  $y$  is the vector of species concentrations and temperature, and  $S_r \in R^n$  is the stoichiometric vector for reaction  $r$ . Here  $n$  is the dimension of  $y$ ,  $N$  is the number of reactions in the original mechanism and  $F_r(y)$  is the reaction rate of reaction  $r$ .

Table 1. Sensitivity analysis and parallelization methods

SIC	Simultaneous corrector method
STC	Staggered corrector method
SDI	Staggered direct method with iterative improvement
DPO	Distributed parameters only method
DPCJD	Distributed parameters, centralized Jacobian, different step-size
DPCJC	Distributed parameters, centralized Jacobian, common step-size
FGP	Fine grain parallelization

Thus the optimization problem is to find a set of parameters  $d_i = 0$  or  $1$ ,  $i = 1, \dots, N$ , which defines the approximate system

$$z' = \sum_{r=1}^N S_r d_r F_r(z), \quad z(0) = y_0 \quad (12)$$

and makes the error norm  $\|y - z\|$  as small as possible. A reaction is kept in the reduced mechanism if its corresponding parameter is 1 and it is deleted from the original mechanism if the value of the parameter is 0. To achieve a reduction of the number of reactions, we also require  $\sum_{r=1}^N d_r = k \ll N$ .

We can see from the definition of the optimization problem that the number of optimization parameters is much larger than the number of ODEs. This is due to the fact that the number of reactions in a chemical mechanism is usually much larger than the number of species.

The numerical experiments repeated here were done on the Exxon model[15], which contains 116 species and 447 reversible reactions. Details about the dynamic system and parameter values are omitted. In the example, there are about 250 equations in the original DAE system and about 450 sensitivity parameters. The time interval of the integration is taken as  $[0, 0.129]$ .

We use the notation given in Table 1 for convenience of presentation of the numerical results.

The computer used in these tests is crunch/mrl.ucsb.edu, which has 2 GB of main memory, 32 R10000 CPUs at 200 MHz, and 35 GB of disk space available in a 2 rack SGI Origin 2000. Though each processor has its own local memory, the entire physical memory is accessible to all the processors in the system. As mentioned before, we use the MPI standard as the communication tool between processors. The MPI standard is targeted for distributed memory systems so all the communication between processors is done through message passing. However, on a shared memory machine like the SGI O2000, the implementation of MPI can be done in two different ways. One of them is the *network* implementation in which the communication is actually done by passing messages via the network and sockets; the other is the *shared memory* implementation which uses the shared memory architecture to reduce the communication cost. The MPI implementation on the SGI O2000 is a shared memory implementation and is optimized to reduce the communication cost. The shared memory implementation of MPI usually has better performance than the network implementation on a shared memory machine. It is more likely that a parallel implementation which requires extensive communication

Table 2. CPU times for one process at light load

Method	447 parameters	1 parameter	0 parameter	Speedup bound
SIC	1023	41	18	25
STC	1284	54	18	24
SDI	904	180	18	N/A

Table 3. CPU times for one process at heavy load

Method	447 parameters	1 parameter	0 parameter	Speedup bound
SIC	1144	49	22	23
STC	1512	56	22	27
SDI	1089	214	22	N/A

may perform worse on real distributed memory systems which have only the network implementation of MPI than on a shared memory system.

Tables 2 and 3 list the CPU seconds for sensitivity analysis with different numbers of sensitivity parameters. Only one processor is used for these three methods. We also list the approximate upper bounds of speedup for sensitivity analysis when no sharing of Jacobian information is done. Since in this paper we are parallelizing the sensitivity analysis only, these upper bounds are given by the ratio of the CPU times for one parameter and 447 parameters.

Since we find that the timing results can be very different for all the methods except those of DPO type, we present the results obtained at both heavy and light loads of the computer system. For some methods, the results at heavy system load are either very bad or difficult to obtain due to a lack of system resources.

The large difference in CPU times between the case of one sensitivity parameter and no sensitivity analysis is mainly caused by the difference between the number of time steps used in these two cases. When sensitivity analysis is required for one parameter for this problem, the number of time steps required nearly doubles and so does the number of Jacobian matrix updates. The SDI method is certainly not good for the case where only one sensitivity parameter is considered since it updates the Jacobian on every step but the sensitivity system is of very small size.

We can see from Tables 2 and 3 that for the 447 sensitivity parameters case, the SDI method has the best performance and the second is the SIC method, with the slowest method being the STC method. This result is consistent with those given elsewhere.<sup>‡</sup>

The following Tables in this Section list the longest CPU times in seconds among all the processes used by a method, which is now a combination of one of the sensitivity analysis methods and one of the parallelization methods. All these CPU time results are obtained when sensitivity analysis is done on 447 parameters. For all of these methods, we use one master process and several DASSLSO slave processes. For methods of DPCJD and DPCJC types, a proper number of Jacobian slave processes are used too. We use the longest CPU times among all the processes instead of the wall-clock time of the entire job to reduce the effect of memory swapping on timing results. The speedup results are given in parenthesis.

Table 4. CPU times (speedups) for four DASSLSO (four Jacobian) slaves at light load

Method	DPO	DPCJD	DPCJC	FGP
SIC	316(3.2)	263(3.9)	425(2.5)	339(3.0)
STC	368(3.5)	386(3.3)	535(2.4)	442(2.9)
SDI	330(2.7)	241(3.8)	253(3.6)	311(2.0)

Table 5. CPU times (speedups) for four DASSLSO (four Jacobian) slaves at heavy load

Method	DPO	DPCJD	DPCJC	FGP
SIC	342(3.3)	358(3.2)	473(2.4)	465(2.5)
STC	411(3.7)	454(3.3)	556(2.7)	550(2.8)
SDI	377(2.9)	288(3.8)	346(3.2)	367(3.0)

We use different numbers of processes to test the scalability of the methods.

The following tables list the CPU times when four DASSLSO slaves are used. For methods of DPCJD and DPCJC types, another four Jacobian slaves are used.

From Tables 4 and 5, we can see that the DPO and DPCJD types have generally better performance than the others. Under both heavy and light loads of the system, the SDI method with DPCJD is the fastest here. This is because the number of DASSLSO slaves is not large and the Jacobian slaves reduce the computation time for the Jacobian matrix well. But we will see later that its performance degrades as the number of DASSLSO slaves increases.

For all methods except those of DPO type, the speedups are generally higher when the system is at light load than when the system is at heavy load. These methods all require synchronization which costs more when the system is at heavy load. Synchronization is not required by DPO type methods so the gain from a light loaded system is not very obvious for methods of this type and their speedups actually decreased.

We can see from Tables 4 and 5 that, for the DPO type, the SIC method has the best performance rather than the SDI method. This is because after the time spent on the non-linear iterations for sensitivity variables is decreased by using four processes, evaluation and factorization of the Jacobian can be a significant cost as its computation is not parallelized. This cost is even more obvious for the SDI method, which needs to compute the Jacobian at every step.

The DPCJC results do not reveal good performance except for the SDI method. When the number of DASSLSO slaves increases, the performance of this method is even worse. We suspect that the reason for its bad performance is due to excessive synchronization needed by this method, especially when the system is at heavy load. Since we do not run our jobs in exclusive mode, we have to share system resources with other users. Sometimes the system may not have enough memory and free processors. At such a time, it is possible that some processes of our jobs are swapped out of the CPUs and memory and the other processes, even in the CPUs, can do nothing but wait. The performance of this method is better when the system is at light load but still the results are not satisfactory. Another disadvantage of this method is that it may increase the number of steps needed and thus

Table 6. CPU times (speedups) for eight DASSLSO (four Jacobian) slaves at light load

Method	DPO	DPCJD	DPCJC	FGP
SIC	172(5.9)	145(7.1)	465(2.2)	251(4.1)
STC	226(5.7)	196(6.6)	535(2.4)	295(4.6)
SDI	245(3.7)	241(3.8)	253(3.6)	254(3.6)

Table 7. CPU times (speedups) for eight DASSLSO (four Jacobian) slaves at heavy load

Method	DPO	DPCJD	DPCJC	FGP
SIC	166(6.9)	196(5.8)	Bad(N/A)	318(3.6)
STC	236(6.4)	212(7.1)	Bad(N/A)	353(4.3)
SDI	250(4.4)	276(3.9)	Bad(N/A)	295(3.7)

Table 8. CPU times (speedups) for sixteen DASSLSO (eight Jacobian) slaves at light load

Method	DPO	DPCJD	DPCJC	FGP
SIC	90(12.7)	97(11.8)	Bad(N/A)	186(6.2)
STC	108(14.0)	102(14.8)	Bad(N/A)	237(6.4)
SDI	202(5.4)	226(4.8)	Bad(N/A)	177(6.2)

compromise the advantage of the better Jacobian reuse across DASSLSO slaves.

Tables 6 and 7 list the results of using eight DASSLSO slaves (and four Jacobian slaves for DPCJD only).

We can see from the above Tables that the DPO and DPCJD methods still have the best performance, especially with the SIC method. The performance of the SDI method is comparatively worse for the reasons we mentioned before. The FGP method does not exhibit good scalability because of excessive communication even with the optimized implementation of MPI on our test machine.

Table 8 lists the results using 16 DASSLSO slaves (and eight Jacobian slaves for DPCJD only) when the system is at light load. We were unable to run the same test when the system is at heavy load, due to lack of system resources. The results are generally similar to those when eight processors are used. Note that SDI with FGP is better than SDI with DPO and DPCJD. This is because when the number of sensitivity variables that each DASSLSO slave has is reduced, SDI with DPO spends too much time on the Jacobian matrix computation and SDI with DPCJD does not reuse the Jacobian matrix well across the DASSLSO slaves.

We can see from Tables 6–8 that the STC method with DPCJD is generally better than the STC method with DPO. It is not clear where this somewhat better performance comes from. We find the performance of this method on Jacobian matrix reuse is almost the same as the SIC with DPCJD, so it does not come from a better Jacobian matrix reuse. However the improvement is not satisfactory since we use four or eight more processors.

From the available results, for the SGI implementation of MPI on the O2000, the

best performance is achieved by the SIC and STC methods with DPO. Since this implementation of MPI already used the shared memory architecture of the O2000 to reduce communication time, it is likely that the methods of FGP type will have worse performance on real distributed memory systems.

#### 4.1. Comparison of the methodologies

For every parallel implementation, we are trying to do three things at the same time: parallelize everything that can be parallelized, reduce the communication needed and balance the work load of different processors to minimize the synchronization cost. For the first method where only the parameters are distributed, the communication is zero and the work load can be balanced well, but the Jacobian is not computed in parallel and it is computed repeatedly at different processors. For the distributed parameters with centralized control on Jacobian computation method, the communication is not significant and everything is computed in parallel but either the work loads of the Jacobian slaves are much greater than those of other processors or the work load between DASSLSO slaves is not balanced well at each time step. For the fine grained method, the work load is balanced well and everything is computed in parallel, but the communication is very costly. As we can see, the more centralized the implementation, the better the work load balancing the parallelization, but also the higher the communication cost. For our problem, the one with the least centralized control seems to be best. Based on these results, we conclude that for the class of problems under consideration, where the number of parameters is large relative to the size of the original system, the least centralized control (DPO) yields the best performance. We can see from Tables 6–8 that the staggered corrector and the simultaneous corrector methods of DPO type can achieve speedups close to the number of processes used: when eight DASSLSO slaves are used, the speedups are around 6, and when 16 DASSLSO slaves are used, the speedups are around 13. We suspect that using more than 16 DASSLSO slaves will further improve the speedups. Since we are mainly parallelizing the computation of sensitivity variables, use of more than 32 slave processes will not help much, as the speedup bounds for these methods are around 25.

#### ACKNOWLEDGEMENTS

This work was partially supported by DOE contract number DE-F603-98ER25354 and by National Science Foundation grant CCR-98-9896198.

#### REFERENCES

1. K. R. Jackson, 'A survey of parallel numerical methods for initial value problems for ordinary differential equations', *IEEE Trans. Magn.*, **27**(5), 3792–3797 (1991).
2. L. G. Birta and O. Abou-Rabia, 'Parallel block predictor-corrector methods for ODE's', *IEEE Trans. Comput.*, **C-36**, 299–311 (1987).
3. M. T. Chu and H. Hamilton, 'Parallel solution of ODEs by multi-block methods', *SIAM J. Sci. Stat. Comput.*, **8**, 342–353 (1987).
4. P. J. van der Houwen and B. P. Sommeijer, 'Parallel iteration of high-order Runge–Kutta methods with stepsize control', *J. Comput. Appl. Math.*, **29**, 111–127 (1990).
5. E. Lelarasmee, A. Ruehli and A. L. Sangiovanni-Vincentelli, 'The waveform relaxation method for time-domain analysis of large scale integrated circuits', *IEEE Trans. Comput.-Aided Des., CAD-1*, 131–145 (1982).

6. U. Miekkala and O. Nevanlinna, 'Convergence of dynamic iteration methods for initial value problems', *SIAM J. Sci. Stat. Comput.*, **8**, 459–482 (1987).
7. C. W. Gear, 'Parallel methods for ordinary differential equations', Tech. Rep. UIUCDCS-R-87-1369, Comp. Sci. Dept., Univ. of Illinois, Urbana, IL, 1987.
8. R. D. Skeel, 'Waveform iteration and the shifted Picard splitting', *SIAM J. Sci. Stat. Comput.*, **10**, 756–776 (1989).
9. R. S. Maier, L. R. Petzold and W. Rath, 'Parallel solution of large-scale differential-algebraic systems', *Concurrency: Pract. Exp.*, **7**(8), 795–822 (1995).
10. Message Passing Interface Forum, 'MPI: a message-passing interface standard', *Int. J. Supercomput. Appl.*, **8**(3/4), (1994).
11. K. E. Brenan, S. L. Campbell and L. R. Petzold, *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, 2 Edn, SIAM, 1995.
12. T. Maly and L. R. Petzold, 'Numerical methods and software for sensitivity analysis of differential-algebraic systems', *Appl. Numer. Math.*, **20**, 57–79 (1997).
13. W. F. Feehery, J. E. Tolsma and P. I. Barton, 'Efficient sensitivity analysis of large-scale differential-algebraic systems', *Appl. Numer. Math.*, **25**, 41–54 (1997).
14. G. H. Golub and C. F. Van Loan, *Matrix Computations*, 2 Edn, Johns Hopkins, 1989.
15. C. A. Mims, R. Mauti, A. M. Dean and K. D. Rose, 'Radical chemistry in methane oxidative coupling: tracing of ethylene secondary reactions with computer models and isotopes', *J. Phys. Chem.*, **98**, 13357–13372. (1994).
16. C. Bischof, A. Carle, G. Corliss, A. Griewank and P. Hovland, 'ADIFOR – generating derivative codes from Fortran programs', *Sci. Program.*, **1**, 11–29 (1992).
17. M. Caracotsios and W. E. Stewart, 'Sensitivity analysis of initial value problems with mixed ODEs and algebraic equations', *Comput. Chem. Eng.*, **9**(4), 359–365 (1985).
18. J. E. Dennis and R. B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice Hall, Englewood Cliffs, NJ, 1983.
19. C. W. Gear, 'The potential for parallelism in ordinary differential equations', Tech. Rep. UIUCDCS-R-86-1246, Comp. Sci. Dept., Univ. of Illinois, Urbana, IL, 1986.
20. P. E. Gill, W. Murray and M. H. Wright, *Practical Optimization*, Academic Press, New York, 1981.
21. E. J. Haug, R. Wehage and N. C. Barman, 'Design sensitivity analysis of planar mechanism and machine dynamics', *Trans. ASME*, **103**, (1981).
22. M. A. Kramer and J. R. Leis, 'The simultaneous solution and sensitivity analysis of systems described by ordinary differential equations', *ACM Trans. Math. Softw.* **14**, 45–60 (1988).
23. L. R. Petzold, J. B. Rosen, P. E. Gill, L. O. Jay and K. Park, 'Numerical optimal control of parabolic PDEs using DASOPT', *Large Scale Optimization with Applications. Part II Optimal Design and Control*, Eds. L. Biegler, T. Coleman, A. Conn and F. Santosa, *IMA Volumes in Mathematics and its Applications*, Vol. 93 (1997), 271–300.
24. H. Rabitz, M. Kramer and D. Dacol, 'Sensitivity analysis in chemical kinetics', *Ann. Rev. Phys. Chem.*, **34**, 419–461 (1983).