An Adjoint-Based Scalable Algorithm for Time-Parallel Integration

Vishwas Rao*a, Adrian Sandu*a

*aComputational Science Laboratory
Department of Computer Science
Virginia Polytechnic Institute and State University, Blacksburg, VA 24061
E-mail: {visrao, sandu}@vt.edu

Abstract

As parallel architectures evolve the number of available cores continues to increase. Applications need to display a high degree of concurrency in order to effectively utilize the available resources. Large scale partial differential equations mainly rely on a spatial domain decomposition approach, where the number of parallel tasks is limited by the size of the spatial domain. Time parallelism offers a promising approach to increase the degree of concurrency. ‘Parareal’ is an iterative parallel in time algorithm that uses both low and high accuracy numerical solvers. Though the high accuracy solvers are computed in parallel, the low accuracy ones are serial, and this considerably hinders the scalability of the entire scheme.

This paper presents a new parallel in time discretization algorithm based on a nonlinear optimization approach. Like in the traditional ‘Parareal’ method, the time interval is partitioned into subintervals, and local time integrations are carried out in parallel. The objective cost function quantifies the mismatch of local solutions between adjacent subintervals. The optimization problem is solved iteratively using gradient-based methods. All the computational steps - forward solutions, gradients, and Hessian-vector products - involve only ideally parallel computations and therefore are highly scalable.

The feasibility of the proposed algorithm is studied on three different model problems, namely, heat equation, Arenstorf’s orbit, and the Lorenz model.

Keywords: Parareal, Adjoint, Sensitivity analysis

1. Introduction

Computational sciences use numerical models to simulate time-evolving systems, where the governing physical laws are formulated as partial differential equations (PDEs). The computations associated with initial value problems are typically sequential, and advance the solution one time step after another. This mimics the physical behavior of the system where the current state determines its future evolution.

Considerable effort is spent in developing large-scale numerical PDE models that can run efficiently on high-performance computing architectures. The ubiquitous strategy is to parallelize the workload in space by mapping different parts of the simulation grid onto individual compute nodes. In contrast, parallelizing in time is not common practice. In principle the simulation time interval can also be split into subintervals: this decomposes the initial value problem into a sequence of initial value subproblems. The challenge for solving these subproblems in parallel is that each of them requires a different initial condition.

The ‘Parareal’ algorithm proposed by Lions, Maday and Turnici in [1] computes the intermediate initial conditions for each subproblem by a scalar, coarse numerical integration. The coarse (low cost, low accuracy) and fine (high cost,
Similarly, the set of consecutive numerical steps that evolve the solution from \( t_i \) to \( t_{i+1} \), will be denoted by:

\[
y_i = c_{i+j} (y_{i+j-1}) , \quad j = 1, \ldots, N.
\]

Similarly, the set of consecutive numerical steps that evolve the solution from \( t_k \) to \( t_{\ell} \), \( \ell > k \), will be denoted by:

\[
y_{k+\ell} = M_{k,\ell} (y_k) , \quad \forall \ell > k.
\]

For example,

\[
y_{k+3} = M_{k,k+3} (y_k) = M_{k+2,k+3} \left( M_{k+1,k+2} (M_{k,k+1} (y_k)) \right).
\]

The serial solution procedure consists in applying \( N \) consecutive steps of the method, starting from the initial condition, to obtain the solution at the final time:

\[
y(t_{\text{final}}) \approx y_N = M_{0,N} (y_0).
\]

In the context of the ‘Parareal’ algorithm, we regard \( M \) to be a “fine” (high compute time and high accuracy) solver. Similarly, we consider a “coarse” (low accuracy and high efficiency) numerical process, and denote it by:

\[
y_{i} = G_{i-1,i} (y_{i-1}) , \quad i = 1, \ldots, N.
\]
2.2. Partitioning the simulation time interval

For a parallel-in-time solution we partition the simulation time interval \([t_{\text{initial}}, t_{\text{final}}]\) into \(M\) subintervals, with boundaries as shown below:

\[ t_{\text{initial}} = T_0 < T_1 < \cdots < T_{M-1} < T_M = t_{\text{final}}. \tag{4} \]

We choose the subinterval boundaries such that they correspond to integer steps of the “fine” numerical method,

\[ T_i = t_{\ell_i}, \quad i = 0, \ldots, M, \]

where \(\ell_i\) are integers. This implies that \(\ell_0 = 0\) and \(\ell_M = N\). The first subinterval consists of the first \(\ell_1\) steps of the numerical method

\[ [T_0, T_1] = [t_0, t_{\ell_1}]. \]

The second subinterval consists of the next \(\ell_2 - \ell_1\) steps of the numerical method

\[ [T_1, T_2] = [t_{\ell_1}, t_{\ell_2}] = [t_{\ell_1}, t_{\ell_1+1}] \cup [t_{\ell_1+1}, t_{\ell_1+2}] \cup \cdots \cup [t_{\ell_2-1}, t_{\ell_2}], \]

and so on.

Consider the following approximations of the solution at the interval boundaries:

\[ u_0 = y_0; \quad u_i \approx y(t_i^+), \quad i = 1, \ldots, M - 1. \]

Then a numerical solution can be computed on each interval \([t_{i-1}, t_i]\) starting from the initial condition \(u_{i-1}\):

\[ \tilde{u}_i = \mathcal{M}_{t_i-t_{i-1}}(u_{i-1}) \quad \text{in parallel for } i = 1, \ldots, M. \tag{5} \]

Since each interval has its own known initial condition, the integrations (5) proceed independently and can be computed in parallel.

2.3. The traditional ‘Parareal’ iterations

Gander and Vanderwile [7] define the ‘Parareal’ algorithm using both the fine (2) and the coarse (3) numerical solution operators. Let \(u_i^{(0)} = y(T_i)\) denote the initial conditions for the interval \([T_i, T_{i+1}]\) at the \(k^{th}\) iteration. The following iterative scheme is used to obtain improved numerical solutions

\[ u_i^{(k+1)} = \mathcal{G}_{i+1} \left( u_i^{(k+1)} \right) + \mathcal{M}_{i+1} \left( u_i^{(k)} \right) - \mathcal{G}_{i+1} \left( u_i^{(k)} \right). \tag{6} \]

Note that the coarse integration is carried out serially, while the fine integration is carried out in parallel (5). The serial coarse solver represents a major bottleneck that prevents the scalability of traditional ‘Parareal’ (6) on a very large number of threads.

3. An Optimization Approach to Time Parallel Discretizations

In this section we propose new parallel in time solution procedure derived in a nonlinear optimization framework.

3.1. Define a cost function to penalize jumps

The set of ODEs (5) lead to a solution that is piecewise continuous within each interval \([T_{i-1}, T_i]\). At each interval boundary \(T_i\) there is a jump between \(\tilde{u}_i \sim y(T_i^-)\), the solution computed on \([T_{i-1}, T_i]\), and \(u_i \sim y(T_i^+)\), the initial condition for the next interval \([T_i, T_{i+1}]\). To obtain the solution of the original ODE we need to choose the initial conditions \(u_i\) such that the result of the integration on the current interval matches the initial conditions of the next interval,

\[ u_i = \tilde{u}_i, \quad i = 1, \ldots, M. \]

We define a cost function that penalizes the jumps across interval boundaries:

\[ J(u_1, \ldots, u_{M-1}) = \frac{1}{2} \sum_{i=1}^{N-1} (u_i - \mathcal{M}_{i-1, i}(u_{i-1}))^T \mathcal{R}^{-1} (u_i - \mathcal{M}_{i-1, i}(u_{i-1})). \tag{7} \]
Here $R_i$'s are symmetric weight matrices and can be chosen as follows:

$$R_i = \text{diag}_{1 \leq j \leq n} \left( \text{Atol}_j + Rtol_j \|u_i\|_j \right)^2,$$

where $\text{Atol}$ and $\text{Rtol}$ are the absolute and relative tolerances used in the ODE integration, respectively. Note that the cost function is greater than or equal to zero. The initial conditions for each subinterval determine the entire solution.

In order to enforce continuity across boundaries, they are obtained by minimizing this cost function

$$\left(u_1^{\text{opt}}, \ldots, u_{M-1}^{\text{opt}}\right) = \arg \min_{(u_1, \ldots, u_{M-1})} J(u_1, \ldots, u_{M-1}).$$

We know that the optimal value of the cost function is $J = 0$, and the optimal values of the initial conditions are the numerical solutions obtained with the serial algorithm:

$$u_i^{\text{opt}} = y_i, \quad i = 1, \ldots, M - 1.$$

In practice it suffices to decrease the jumps until each difference $(u_i - \widetilde{u}_i)$ is of the order of the local truncation error of the underlying numerical method [12].

### 3.2. Gradient of the cost function

The minimization problem (8) is solved using gradient based methods. We use sensitivity analysis [13, 14, 15] to compute the gradient and the (approximate) Hessian of the cost function (7). The discrete tangent linear model associated with the numerical solution (2) is denoted by

$$\delta y_{j+1} = M'_{j,j+1} (y_j) \cdot \delta y_j = M_{j+1} \delta y_j, \quad j = 0, \ldots, N - 1,$$

where $M'_{j,j+1} = M_{j,j+1}$ is the linearized numerical method; the linearization point is. The discrete adjoint (of the above tangent linear) model is denoted by

$$A_j = (M'_{j,j+1})^* A_{j+1} = M^*_{j+1,j} A_{j+1}, \quad j = N - 1, \ldots, 0.$$

where $M^*_{j+1,j} = M^T_{j,j+1}$ is the discrete adjoint of the (linearized) numerical method [16]. The cost function (7) has a gradient of the form

$$\nabla_{u_i} J = R_i^{-1} (u_i - M_{i-1} (u_{i-1})) - M^*_{i+1,i} R_i^{-1} (u_{i+1} - M_{i+1} (u_i)).$$

Further, the gradient can be computed in parallel as follows:

**Algorithm 1 Parallel Gradient**

```plaintext
1: procedure Parallel_Gradient
2:  for all $1 \leq i \leq M$ do in parallel
3:     $\widetilde{u}_i \leftarrow M_{i,i-1} (u_{i-1}), \quad T_{i-1} \leq t \leq T_i.$ \> Integrate the forward ODE
4:     $v_l \leftarrow u_l - \widetilde{u}_i.$
5:     $\overline{v}_i \leftarrow M^*_{i,1} R_i^{-1} v_{i+1}, \quad T_{i} \leq t \leq T_{i+1}.$ \> Integrate the adjoint ODE
6:  end for all
7:  for all $1 \leq i \leq M - 1$ do in parallel
8:     $\nabla_{u_i} J \leftarrow R_i^{-1} v_l - \overline{v}_i.$ \> Final gradient
9: end for all
10: end procedure
```
3.3. Hessian of the cost function

The Hessian \( (\nabla^2 J) \) of the cost function (7) has a block tridiagonal structure. The analytical expression is as given below for each block row \( i = 1, \ldots, N \):

\[
\nabla^2_{u_i, u_i} J = - R_i^{-1} M_{i, i} L_i, \quad (12a)
\]

\[
\nabla^2_{\tilde{u}_i, u_i} J = R_i^{-1} + M^*_{i, i} L_i R_i^{-1} M_{i, i} - \frac{d}{du_i} \left( M^*_{i, i} (u_i) \right) R_i^{-1} (u_{i+1} - M_{i, i} (u_i)), \quad (12b)
\]

\[
\nabla^2_{u_i, \tilde{u}_i} J = - M^*_{i, i} L_i R_i^{-1}. \quad (12c)
\]

In the equation (12b), we employ the Gauss-Newton approximation and neglect the second term involving derivatives of the adjoint operator. This term is expensive to compute, as it requires a second order adjoint model. More over, the residual \( u_{i+1} - \tilde{u}_{i+1} \) is small when one is close to the continuous ODE solution, and the term does not contribute significantly to the Hessian. Hence the approximate Hessian \( H \approx \nabla^2_{u_i, u_i} J \) is given by

\[
H = \begin{pmatrix}
R_i^{-1} + M^*_{i, i} L_i R_i^{-1} M_{i, i} & \ldots & 0 \\
\ldots & \ldots & \ldots \\
0 & \ldots & -R_{M-1}^{-1} M_{M-1, M-1} + M^*_{M-1, M-1} R_{M-1}^{-1}
\end{pmatrix}.
\]

Furthermore, the Gauss-Newton Hessian (13) admits a Cholesky decomposition \( H = L^T L \) with

\[
L = \begin{pmatrix}
R_1^{-1} & 0 & \ldots & 0 \\
-R_2^{-1} M_{1, 1} & R_2^{-1} & \ldots & 0 \\
\ldots & \ldots & \ldots & \ldots \\
0 & \ldots & R_{M-1}^{-1} M_{M-1, M-1} & R_{M-1}^{-1}
\end{pmatrix}.
\]

The inverse can be computed as follows

\[
L^{-1} = \begin{pmatrix}
I & 0 & 0 & \ldots & 0 \\
M_{1, 1} & I & 0 & \ldots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
M_{M-1, M-1} & M_{M-1, M-1} & \ldots & M_{M-1, M-1} & I
\end{pmatrix}.
\]

3.4. Hessian-vector products

Computing the full Hessian is very expensive, as it requires \( \sim 4M \) adjoint and tangent linear model runs. Hence the need for Hessian-vector products. Hessian-vector products play a very important role in the optimization of cost functions. They are necessary for solving the linear systems involving the Hessian, if necessary. Hence having a parallel algorithm to compute the Hessian-vector is very important. Assume that Hessian is a \( M \) by \( M \) block matrix. Further, assume that the vector \( V \) is \( M \) by 1 block vector. Let denote \( i^th \) block of the vector by \( V_i \). Also denote the resulting Hessian-vector product by \( HV \). The algorithm 2 give the steps to compute the Hessian vector product in parallel. Since the Hessian is block tridiagonal, both \( \tilde{W}_M \) and \( \tilde{U}_1 \) are 0. Here steps 3, 4 and 5 need to be performed in order. Steps 8 and 11 can proceed independently. Hence further parallelization can be achieved by performing steps 3, 4 and 5 on one set of processors, step 8 on a second set of processors and step 11 on the third set of processors. This makes the Hessian-vector product computation a very scalable operation.
Algorithm 2 Parallel Hessian-vector product

1: procedure Parallel_Hessian-vector_product
2:   For all 1 ≤ i ≤ M do in parallel
3:     \( \tilde{V}_i \leftarrow M_{t_i,t_{i+1}} V_i, \quad T_i \leq t \leq T_{i+1} \) \text{.}  \hspace{1cm} \triangleright \text{Integrate the tangent linear ODE}
4:     \( \tilde{V}_i \leftarrow M_{t_i,t_{i+1}}^{-1} R_{t_i} \tilde{V}_i, \quad T_{i+1} \leq t \leq T_i \) \text{.}  \hspace{1cm} \triangleright \text{Integrate the adjoint ODE}
5:     \( \tilde{V}_i \leftarrow R_{t_i}^{-1} \tilde{V}_i + \tilde{V}_i \).  \hspace{1cm} \triangleright \text{Final Hessian-vector product}
6:   end For all
7:   For all 2 ≤ i ≤ M do in parallel
8:     \( \tilde{U}_i \leftarrow R_{t_i}^{-1} M_{t_i,t_{i-1}} V_{i-1}, \quad T_{i-1} \leq t \leq T_i \).  \hspace{1cm} \triangleright \text{Integrate the tangent linear ODE}
9:   end For all
10:  For all 1 ≤ i ≤ M − 1 do in parallel
11:   \( \tilde{W}_i \leftarrow M_{t_i,t_{i+1}}^{-1} R_{t_{i+1}} \tilde{V}_{i+1}, \quad T_{i+1} \leq t \leq T_i \).  \hspace{1cm} \triangleright \text{Integrate the adjoint ODE}
12:  end For all
13: end procedure

3.5. Optimization algorithms

We have tried four gradient based optimization methods to minimize the cost function: Gauss-Newton, L-BFGS, truncated Newton and nonlinear conjugate gradient. In this sub-section we represent the solution vector as \( U = (u_1, \ldots, u_{M-1}) \). In the following sub-sections we briefly discuss the optimization strategies employed.

3.5.1. Gauss-Newton

Gauss-Newton solves non-linear least squares problems [17] via the following iterations:

\[
\begin{align*}
H(U^{(k)}) \cdot \Delta U & = -\nabla_U J(U^{(k)}), \\
U^{(k+1)} & = U^{(k)} + \Delta U.
\end{align*}
\]

(16a)

(16b)

The Gauss-Newton approach approximates the Hessian by ignoring second-order derivative information. The structure of the gradient (right hand side) and the methodology to compute it has already been described in section 3.2. The structure of the approximate Hessian and its decomposition in two block bidiagonal matrices is described in section 3.3. The linear system (16a) can be solved using iterative methods applied directly to the block tridiagonal matrix (13). This requires the computation of Hessian-vector products, which has been described in Section 3.4. Alternatively, the system(16a) can be solved via forward and backward substitutions. Further, the lower and upper block bidiagonal system solutions (15) can be obtained using only matrix-vector products, calculated by running the adjoint or the tangent linear model.

3.5.2. L-BFGS

L-BFGS belongs to the class of quasi-Newton methods and is considered the gold standard for large-scale unconstrained nonlinear optimization [18]. Quasi-Newton methods build an approximation of the Hessian inverse using the solution and gradient information from past iterations. This leads to only one forward model run and one first-order adjoint model run for each intermediate solution. In our numerical experiments we used the L-BFGS-B package [19].

3.5.3. Nonlinear Conjugate Gradients

The nonlinear conjugate gradients (NLCG) method finds the local minimum of a nonlinear function using only gradient \( \nabla U J(U) \) information. A detailed description of the method can be found in [20, 21]. In our experiments we use the Fletcher-Reeves version of NLCG. In our numerical experiments we use the Matlab package Poblano [22].
3.5.4. Truncated Newton

Truncated Newton methods use the same principle as the Newton’s method (16). The linear system (16a) is solved inexactly by an iterative method, such as preconditioned (linear) conjugate gradients. The iterations are “truncated” (terminated before getting an accurate solution) since it suffices to only find a descent direction for the nonlinear minimization. In our numerical experiments we use the Matlab package Poblano [22] and the Fortran codes based on [23] and [19].

4. Numerical Experiments

4.1. Experimental setup

We apply the proposed algorithm to solve three problems of varying sizes, namely, Arenstorf’s orbit, the Lorenz-96 model, and the heat equation. For each case the time integration was performed using Singly Diagonally Implicit Runge Kutta (SDIRK) method described in [24]. The experiments were performed on a 48 core cluster. Number of threads used to perform the computations are of the same order as the number of intervals into which the time domain is divided. Adaptive step size control is used to perform the numerical integration in each of the scenario. Traditional Parareal is performed by setting relative tolerance as $10^{-3}$ for coarse solver and $10^{-10}$ for fine grained solver. Adjoint Parareal algorithm is carried out by setting the relative tolerance as $10^{-10}$ for each subinterval. The following subsections briefly describe the equations governing the models, the space and time domains and the discretizations schemes used to perform the computations.

4.1.1. Arenstorf’s orbit

The Arenstorf orbit is a special case of a three body problem. The analytical solution is unknown because of the complexity of the interactions between the three bodies. In this model, two bodies of mass $1-\mu$ and $\mu$ rotating in a circular plane are considered with a third body of negligible mass moving around in the same plane. The following equations govern the motion [24]:

$$
\begin{align*}
\ddot{y}_1 &= y_1 + 2\dot{y}_2 - (1-\mu)\frac{y_1 + \mu}{D_1} - \mu \frac{y_1 - (1-\mu)}{D_2}, \\
\ddot{y}_2 &= y_2 - 2\dot{y}_1 - y_2 \left(\frac{1-\mu}{D_1} + \frac{\mu}{D_2}\right),
\end{align*}
$$

where

\begin{align*}
\mu &= \frac{m_1}{m_1 + m_2}, \\
D_1 &= \left((y_1 + \mu)^2 + y_2^2\right)^{\frac{3}{2}}, \\
D_2 &= \left((y_1 - (1-\mu)^2 + y_2^2\right)^{\frac{3}{2}}.
\end{align*}

The solution to the above set of equations are periodic. The second order Arenstorf equations are converted to a first order system by adding the variables $y_3$ and $y_4$, and the extra equations $y_3 = \dot{y}_1$ and $y_4 = \dot{y}_2$. The simulation time interval is $[0, 1.706]$ s. The number of steps taken by the solver to perform the integration over a subinterval depends on the number of intervals. When the integration interval is divided into four parts, the SDIRK integrator requires 52 steps to perform the integration and the maximum step size is 0.0106.

4.1.2. Heat equation

We consider the following one dimensional heat equation, which governs the evolution of the temperature profile $w$ of a solid

$$
\frac{w_t}{w_{xx}} + x^4(1-x) + t^2.
$$

(17)
The PDE (17) is discretized in space using a central difference scheme for \( w_{xx} \). This leads to the ODE

\[
\dot{W} = \frac{1}{h^2} A W + f(t),
\]

where \( W \) is the vector of approximate temperature values at the grid points, and \( f \) is the discretized source term.

The simulation time interval is \([0, 1] \) s. The number of steps taken by the solver to perform the integration over a subinterval depends on the number of intervals. When the integration interval is divided into four parts, the SDIRK integrator requires 141 steps to perform the integration over one subinterval and the maximum step size is \( 2.97E - 3 \).

4.1.3. Lorenz model

This chaotic Lorenz-96 model [25] has \( n = 40 \) states and is described by the following equations:

\[
\frac{dy_j}{dt} = -y_{j-1} \left( y_{j-2} - y_{j+1} \right) - y_j + F, \quad j = 1, \ldots, n,
\]

\[
y_{j-1} = y_{n-1}, \quad y_0 = y_n, \quad y_{n+1} = y_1.
\]

The forcing term is \( F = 8.0 \). The simulation time interval is \([0, 1] \) s. The number of steps taken by the solver to perform the integration over a subinterval depends on the number of intervals. When the integration interval is divided into four parts, the SDIRK integrator requires 54 steps to perform the integration over one subinterval and the maximum step size is \( 0.0598 \).

4.2. Relative error

The parallel in time solution obtained from optimization is compared with a reference solution given by a high accuracy serial solver (Matlab’s ode45, ode15s routines using very tight relative tolerances). We denote by \( U \) the optimization solution, i.e., the vector of initial conditions for each subinterval (see Section 3.5). Further, we denote the reference solution values at the beginning of each subintervals \((4)\) by \( U_{\text{ref}} \). We report the following \( L^2 \) norm of the relative error:

\[
RE = \sqrt{\sum_i \left( \frac{U_i - U_{i,\text{ref}}}{U_{i,\text{ref}}} \right)^2}.
\]

4.3. Convergence of the optimization process and the number of subintervals

The exact solution to the optimization problem (8) leads to a cost function value (7) equal to zero. Figure 1 shows the behavior of cost function for different values of \( M \) (the number of subintervals). The cost function decreases as the Gauss-Newton iterations progress. The cost function reduction is relatively insensitive to the number of subintervals for the Arenstorf and heat equation problems. For the Lorenz problem, however, more iterations are required to decrease the cost function for a larger number of subintervals.

4.4. Relative performance of different optimization methods

The convergence of different optimization algorithms is shown in Figure 2 for Arenstorf’s orbit, in Figures 3 for the heat equation, and in Figure 4 for the Lorenz system. For all problems we consider two cases, with the simulation interval decided into \( M = 20 \) and \( M = 50 \) subintervals, respectively. Gauss-Newton converges to a solution of good accuracy in the smallest number of iterations. Other optimization techniques (L-BFGS, Truncated Newton, NLCG) are not as effective, although they give good results in case of heat equation.

All methods require gradient computations. The truncated Newton approach also requires Hessian-vector products. Each Gauss-Newton iteration requires a gradient computation and an approximate solution of a linear system \((16a)\). One way to solve the system is to compute the approximate Hessian and solve a block tridiagonal system. Another way is to decompose the Hessian into a block lower bidiagonal and block upper bidiagonal matrices as described in Section 3.5.1 and solve the linear system by forward and backward substitution. In both cases parallel solutions of the block tridiagonal and bidiagonal systems are possible, and required. One can employ parallel direct solvers \([26, 27]\) or parallel iterative methods. In the results presented in this section we solve the linear system via forward and backward substitutions.
Figure 1: Decrease of cost function for several test problems using the Gauss-Newton method. The problem is solved with different numbers of subintervals $M$. 
Figure 2: Comparison of various optimization techniques for Arenstorf’s orbit for different numbers of subintervals (different values of $M$).

Table 1: Timings for the Arenstorf’s orbit solved with Gauss-Newton approach (all reported CPU times are in seconds).
Figure 3: Comparison of various optimization techniques for Heat equation for different numbers of subintervals (different values of $M$).

Table 2: Timings for the heat equation solved with Gauss-Newton approach (all reported CPU times are in seconds).
Figure 4: Comparison of various optimization techniques for Lorenz model for different numbers of subintervals (different values of $M$).

Table 3: Timings for the Lorenz model solved with Gauss-Newton approach (all reported CPU times are in seconds).
Figure 5: Strong scalability of gradient and Hessian-vector product computations for different model problems. The number of threads is essentially equal to the number of subintervals.

4.5. Scalability

In this sub-section we discuss the scalability of the gradient and Hessian-vector product computations which are required in the implementation of numerical optimization routines.

4.5.1. Gradient

Each of the optimization algorithms discussed here requires one gradient calculation per iteration, therefore the scalability of gradient computation plays an important role in the scalability of the entire solution process. The parallel gradient calculation is described in detail in Section 3.2. Figure 5 shows the time taken by the gradient calculations for the same problem divided into different numbers of subintervals. The number of threads is essentially the number of subintervals. The gradient calculations show an almost ideal strong scalability with an increasing number of subintervals/threads.
4.5.2. Hessian-vector product

Each Hessian-vector product calculation requires one adjoint model run and one tangent linear model run. The parallel algorithm is described in detail in the Section 3.4. Figure 5 presents the compute times for Hessian-vector products. The number of threads is essentially the number of subintervals. Like with gradients, Hessian-vector calculations show an almost ideal strong scalability curve.

4.6. Timings

We consider the CPU times required by the Gauss-Newton solutions to assess the practical applicability of the method. Tables 1, 2 and 3 display the timings for Arenstorf’s orbit, heat equation and the Lorenz model, respectively. The solution of the linear system requires a lot of time and is a bottleneck for the entire algorithm. This can be overcome by employing parallel algorithms to solve the block tridiagonal systems. A survey of available algorithms to solve block bidiagonal system is presented in [26]. A highly scalable algorithm is described in [27]. An efficient algorithm to solve block tridiagonal systems on multi-core CPUs and GPUs is described in [28]. Significant speedups for solving the linear system alone can, in principle, be obtained by using the aforementioned algorithm. Consider the heat equation for example. If we denote the serial time by $T_s$, the data in Table 2 reveals that the time needed to solve the linear system for 4 threads is about $T_s^6$. In table 4 we report the timings for the Lorenz model solution in a practical setting that uses fewer Gauss-Newton iterations. The parallel timings are smaller than the serial time.

We also compare the CPU times of Gauss-Newton-adjoint ‘Parareal’ and the traditional ‘Parareal’ for solving the heat equation in Figure 6. It can be seen that the new adjoint ‘Parareal’ algorithm performs better than traditional ‘Parareal’ in terms of the timings and accuracy. This result can be further improved with more efficient parallel strategies for solving the Gauss-Newton linear system.

5. Conclusions and Future Work

Efficient use of emerging parallel architectures will require applications to effectively employ large numbers of compute cores. In simulations of evolutionary PDEs the level of parallelism obtained solely by spatial domain decomposition is inherently restricted. Time parallelism is necessary to increase the degree of concurrency. The traditional ‘Parareal’ approach has limited scalability because a coarse solution is serially computed in each iteration. In this paper we propose an algorithm for time parallelization that is potentially highly scalable. The method is formulated in the framework of nonlinear optimization, where the initial conditions on each subinterval are updated such as to minimize the solution jumps. Derivatives required to solve the optimization problem are computed using the adjoint model. The forward solution, the gradients, and the Hessian-vector products involve only ideally parallel computations and therefore have an excellent potential for scalability on large numbers of cores.

Numerical experiments are conducted with three different problems of varying sizes. Among the nonlinear optimization solvers tested, Gauss-Newton performs the best in terms of accuracy and the total number of iterations. Based on the nature of the problem, other techniques like truncated Newton and L-BFGS can work reasonably well. The solution of the Gauss-Newton linear system (16a) can become a bottleneck for achieving good scalability and speedups, and we discuss ways to overcome it.

Further research is needed to make the adjoint-based ‘Parareal’ a practical method of choice. A good preconditioner for the linear system (16a) would be desirable to speed up the iterative solvers. Multigrid strategies and limited memory preconditioners can be used to accelerate the convergence of nonlinear optimizers. We are working on implementing an optimized parallel block tridiagonal linear system solver for (16a). The overall approach can potentially benefit from an error control mechanism that allows to stop iterations when the optimization errors are of the order

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0.147</td>
<td>0.023</td>
<td>0.170</td>
<td>0.240</td>
<td>7.60e-13</td>
<td>6.47e-08</td>
</tr>
<tr>
<td>8</td>
<td>0.095</td>
<td>0.053</td>
<td>0.148</td>
<td>0.240</td>
<td>7.81e-06</td>
<td>1.30e-04</td>
</tr>
<tr>
<td>16</td>
<td>0.060</td>
<td>0.098</td>
<td>0.158</td>
<td>0.240</td>
<td>4.65e-04</td>
<td>1.15e-03</td>
</tr>
<tr>
<td>32</td>
<td>0.035</td>
<td>0.200</td>
<td>0.235</td>
<td>0.240</td>
<td>1.45e-03</td>
<td>2.56e-03</td>
</tr>
</tbody>
</table>

Table 4: Timings for Lorenz model with fewer Gauss-Newton iterations (all reported CPU times are in seconds).

Vishwas Rao and Adrian Sandu / Journal of Computational Science 00 (2013) 1–17
Figure 6: Comparison of Gauss-Newton-adjoint ‘Parareal’ and traditional ‘Parareal’ for solving the heat equation with different numbers of threads.
of truncation errors, and to dynamically switch between different versions of `Parareal`, including the traditional and adjoint-based ones.

Acknowledgements

This work has been supported in part by NSF through awards NSF OCI–8670904397, NSF CCF–0916493, NSF DMS–0915047, NSF CMMI–1130667, NSF CCF–1218454, AFOSR FA9550–12–1–0293–DEF, and by the Computational Science Laboratory at Virginia Tech.

References

