A PETSc Parallel BVP Code

Jason Boisvert

Abstract

Solutions to ordinary differential equations with boundary conditions (BVPs) are often approximated by a numerical method. Typically, this method involves setting up a large system of nonlinear equations for which the unknowns are discrete numerical approximations of the solution to the BVP. This nonlinear system has the potential to be large. Therefore, routines that parallelize both the generation and solution of the nonlinear system could reduce the overall computational time. This report describes a BVP code called PETSc_BVP that contains parallel implementations of both of these routines. The parallel scientific toolbox PETSc is used to create the code. The report also describes the results of several numerical experiments used to show that for certain problems, a reduction of overall computational time is achieved when multiple processors are used to solve the problem.

1 Introduction

In this report, we consider software to numerically solve boundary value problems (BVPs) that consist of a system of first order ODEs

$$\mathbf{u}' = \mathbf{f}(x, \mathbf{u}(x)), \quad a \leq x \leq b, \tag{1a}$$

and a system of two-point boundary conditions

$$\mathbf{0} = \mathbf{g}(\mathbf{u}(a), \mathbf{u}(b)), \tag{1b}$$

where $\mathbf{u}$, $\mathbf{f}$, and $\mathbf{g}$ are vectors of length d.

Typically, exact solutions to BVPs are unknown and therefore a numerical approximation of the solution is often found instead. A common approach to finding the numerical solution involves generating a large system of nonlinear equations. The unknowns to the system of nonlinear equations are discrete numerical approximations of the solution to the BVP. Popular BVP codes that use this approach are `bvp4c` [12], `bvp5c` [13], `BVP_SOLVER` [16] and `COLNEW` [6].

The size of the system of nonlinear equations depends on both the number of differential solution components d and the number of descrite numerical approximations to the BVP. For a BVP with a large d or a problem that requires many discrete points, the system of nonlinear equations become large and therefore computational timely to both generate and solve. These particular problems would benefit from a BVP code that parallelize the routines used to accomplish these tasks. None of the above mention codes do this.

This report describes a BVP code that uses parallel implementations of the routines that generate and solve the system of nonlinear equations. The toolkit `PETSc` [7] is used to implement the BVP code and therefore the code is called `PETSc_BVP`. The remainder of the report can be divided into the following sections. Section 2 describes the method used by `PETSc_BVP` to solve BVPs of the form (1). Section 3 describes the parallel implementation of the method. Section 4 describes several numerical experiments used to show a performance increase when multiple processors are used. Section 5 describes some conclusions and future work.

2 A MIRK method to solve BVP

In this section, we present a global approach to the numerical solution of BVPs that makes use of mono-implicit Runge–Kutta (MIRK) formulas.

A global approach involves generating a system of generally nonlinear equations for which the unknowns $\mathbf{u}_i$ are numerical approximations of the solution values $\mathbf{u}(x_i)$ at a mesh point x_i. A mesh consists of discrete points that partition the solution domain

$$a = x_0 \leq x_1 \leq \cdots \leq x_{N-1} \leq x_N = b.$$

Using MIRK formulas, the nonlinear equations for each subinterval $[x_i, x_{i+1}], \quad i = 0, \ldots, N-1$, where $h_{i+1} = x_{i+1} - x_i$, have the form

$$\boldsymbol{\phi}_{i+1}(\mathbf{u}_i, \mathbf{u}_{i+1}) = \mathbf{u}_{i+1} - \mathbf{u}_i - h_{i+1} \sum_{j=1}^{s} b_j \mathbf{f}(x_i + c_j h_{i+1}, \mathbf{u}_{i,j}) = \mathbf{0}, \tag{2a}$$

where $\boldsymbol{\phi_{i+1}}$ is a vector of length d and

$$\mathbf{u}_{i,j} = (1 - v_j)\mathbf{u}_i + v_j \mathbf{u}_{i+1} + h_{i+1} \sum_{k=1}^{j-1} a_{j,k} \mathbf{f}(x_i + c_k h_{i+1}, \mathbf{u}_{i,k}) \tag{2b}$$

In (2), $j = 1, 2, \ldots, s$ are the stages of the MIRK method. The coefficients, $a_{j,k}$, b_j, v_j, c_j are defined by the MIRK method begin used. The coefficient $c_j = v_j + \sum_{k=1}^{j-1} a_{j,k}$ for $k = 1, 2, \ldots, j-1$.

For each subinterval, equation (2a) represents d nonlinear equations. The solution point $\mathbf{u}_{i,j}$ is determined explicitly and therefore is not part of the overall system of equations. In other words, the MIRK method is a semi-explicit method and therefore can be said to use parameter condensation to eliminate the local unknowns for each subinterval from the overall system of equations. Including the d boundary conditions, the entire system consists of $(N+1)d$ non-linear equations and has the form

$$\boldsymbol{\Phi}(\mathbf{U}) = \begin{pmatrix} \boldsymbol{\phi}_1(\mathbf{u}_0, \mathbf{u}_1) \\ \boldsymbol{\phi}_2(\mathbf{u}_1, \mathbf{u}_2) \\ \vdots \\ \boldsymbol{\phi}_N(\mathbf{u}_{N-1}, \mathbf{u}_N) \\ \mathbf{g}(\mathbf{u}(a), \mathbf{u}(b)) \end{pmatrix} = \mathbf{0}, \tag{3}$$

where $\mathbf{U} = [\mathbf{u}_o, \mathbf{u}_1, \ldots, \mathbf{u}_{N-1}, \mathbf{u}_N]^T$ is the vector of unknowns and $\boldsymbol{\Phi}(\mathbf{U})$ is a residual function.

System (3) is typically solved using Newton's method. This method begins with an initial guess $\mathbf{U}_0$ of the solution to (3). After, the solution to the nonlinear equations can be iteratively determined by evaluating

$$\mathbf{U}_v = \mathbf{U}_{v-1} - \boldsymbol{\Phi}'(\mathbf{U}_{v-1})^{-1} \boldsymbol{\Phi}(\mathbf{U}_{v-1}),$$

where $\mathbf{U}_v$ is the solution for the vth iteration of the Newton's method and $\boldsymbol{\Phi}'(\mathbf{U}_{v-1})$ is the Jacobian matrix

$$\boldsymbol{\Phi}'(\mathbf{U}_{v-1}) = \left. \frac{\partial \boldsymbol{\Phi}}{\partial \mathbf{U}} \right|_{\mathbf{U}=\mathbf{U}_{v-1}}.$$

Iterations continue until some termination criteria is met. For example,

$$\|\mathbf{\Phi}(\mathbf{U}_v)\|_\infty < \tau,$$

where $\tau << 1$.

In practice, the inverse of the Jacobian matrix is computationally expensive to evaluate. A Newton direction $\boldsymbol{\delta}$ is therefore determined for each iteration by solving the linear system

$$\mathbf{\Phi}'(\mathbf{U}_{v-1})\delta = -\mathbf{\Phi}(\mathbf{U}_{v-1}). \tag{4}$$

The solution for the current iteration can then be determined by

$$\mathbf{U}_v = \mathbf{U}_{v-1} + \boldsymbol{\delta}.$$

In the case of system (3), the linear equations form an almost-block diagonal (ABD) system when solving (4) [5]. Therefore, a specialized ABD linear solver can be used to solve the system and greatly reduce the overall computational time [2].

The Newton's method presented above fails to converge for poor initial guesses [11]. In practice, a modified Newton's method is used that only applies a fraction λ, where $0 < \lambda \leq 1$, of the Newton direction to the previous solution $\mathbf{U}_{v-1}$, i.e.

$$\mathbf{U}_v = \mathbf{U}_{v-1} + \lambda\boldsymbol{\delta}.$$

The method used to determine λ distinguishes the Newton's method and is also referred to as a global convergence strategy. Typically, a damped Newton's method is used for BVP software [4]. However, this is only one of the many variations of Newton's method that could be applied to the nonlinear system (3).

Once a numerical solution is determined, the global error of the solution can be estimated through a variety of ways. Typically, a method based on Richardson Extrapolation is used. For a numerical solution $\mathbf{u}^\pi$ determined on mesh π, a new solution $\mathbf{u}^{2\pi}$ is found by having having the mesh. After, a GE estimate is given by

$$\epsilon_g = \left(\frac{2^p}{2^p - 1}\right) \max_{i,j} \frac{|u^\pi_{i,j} - u^{2\pi}_{i,j}|}{1 + |u^\pi_{i,j}|},$$

where $u^\pi_{i,j}$ is the numerical approximation of the jth component of $\mathbf{u}$ for the mesh point x_i and p is the order of the MIRK method. The order of the method dictates the rate of the

convergence of the method to the exact solution as the mesh size increases. A few other methods of global-error estimation have been applied to BVP codes and have been shown to have performance improvements over Richardson extrapolation [9].

3 A parallel BVP code

This section describes a parallel BVP code created to solve problems of the form (1). The toolkit `PETSc` version 3.2 [7] is used to implement the BVP code. This toolkit provides a variety of data structures and routines that are typically used to solve partial-differential equations in parallel. The parallel routines found in `PETSc` are implemented with the `pthreads` toolkit for shared memory computing and a message passing interface toolkit (MPI) for both shared and distributed memory computing. The parallel BVP toolkit only uses the MPI implementation of the routines found in `PETSc`. By doing so, the code can be run on both shared and distributed memory computers.

The implementation of `PETSc_BVP` focuses on the parallelization of two of the steps performed during the numerical solution method described in Section 2. These steps include the formation of the system of nonlinear equations (3) and the solution of the system of nonlinear equations. It should be noted that `PETSc_BVP` is still in the early stages of development and therefore is missing many of the important features found in other BVP codes. For example, `PETSc_BVP` does not have the ability to estimate the error of the numerical solution, e.g., using the method described at the end of Section 2. Instead, the user must verify the solution by either using an exact solution or a reference solution. At this point, the purpose of the current state of the implementation is simply to show the benefits of both generating and solving the system of nonlinear equations in parallel.

3.1 Parallelization of the generation of the nonlinear equations

This section describes how the system of nonlinear equations (3) is evaluated in parallel. In order to generate the system, it is assumed that the user provides a function that describes the odes

```
PetscErrorCode odes(PetscReal x, PetscReal *u, PetscReal *f);
```

where the parameters correspond to (1a). The user can make use of `PETSc` error codes to prevent further computation if an issue is detected. Also, it is not necessary for the user to parallelize the function `odes` and therefore the method of parallelization used in `PETSc_BVP` can be abstracted from the user. The user-supplied function `odes` can then be passed to `PETSc_BVP` with the following function

```
PetscErrorCode setOdeFunction (BvdaeCtx *sol,
                    PetscErrorCode (*f)(PetscReal, PetscReal *, PetscReal *));
```

where `sol` is a data structure created for `PETSc_BVP` and is initialized with the function

```
PetscErrorCode initBVDAE(BvdaeCtx *sol, PetscReal a,PetscReal b, PetscInt d);
```

where `a` and `b` are the boundary points and `d` is the number of differential solution components. Similarly, the user must provide a function that describes the boundary conditions

```
PetscErrorCode bc(PetscReal *ua, PetscReal *ub, PetscReal *g);
```

where `ua` is an array that holds the values for $\mathbf{u}(a)$ and `ub` is an array that holds the values of $\mathbf{u}(b)$. The user-supplied function can then be passed to `PETSc_BVP` by the following function

```
PetscErrorCode setBCFunction (BvdaeCtx *sol,
                    PetscErrorCode (*bc)(PetscReal *, PetscReal *, PetscReal *));
```

Because a variation of Newton's method must be used to solve the nonlinear equations, an initial guess for the solution components can be provided by the user as a function

```
PetscErrorCode guess(PetscReal x, PetscReal *u);
```

The user-supplied initial guess is passed to `PETSc_BVP` by using the function

```
extern PetscErrorCode setInitialGuess(BvdaeCtx *sol,
                    PetscErrorCode (*guess)(PetscReal, PetscReal *));
```

Once a specification of the BVP is provided to PETSc_BVP, the pattern of distribution among P processors can be determined. This pattern determines how the mesh points, nonlinear equations, and solution components are distributed among the different processors. This pattern is determined by the size of the mesh provided to PETSc_BVP by either the user or through the following function

```
extern PetscErrorCode uniformMesh(BvdaeCtx *sol, PetscInt meshSize);
```

where meshSize is the size of the mesh, i.e., $N+1$ points. By using the above function, a uniform mesh is obtained. After a mesh is provided, we choose a distribution pattern that distributes the mesh evenly among P processors. For example, the first processor is assigned the first $\frac{N+1}{P}$ mesh points, the second processor is then assigned the next $\frac{N+1}{P}$ points. This continues until all points are distributed among processors. If P does not divide the total number of points evenly, the remainder of the mesh points are assigned to the last processor. Once the pattern is determined, it is used to generate PETSc distributed arrays for the solution components and the results of the residual function. These distributed arrays are used by PETSc to provide a local array of size dM_i, where M_i is the number of mesh points assigned to processor i, for each processor. These local arrays are used to setup the system of equations in parallel.

When setting up the nonlinear system, each processor is responsible for the generation of M_i components of (3). In a similar fashion to the mesh points, the first processor is responsible for the first M_i components of (3), the next processor is responsible for the next M_i components and so on. The last processor is responsible for the evaluation of the boundary conditions as well. Except for the boundary conditions, each component of (3) requires d evaluations of the MIRK formula (2), i.e., one for each differential solution component. For the evaluation of the MIRK formulas, users are able to choose between a 2nd, 4th, and 6th order MIRK method found in [14]. Users choose the order of the MIRK method with the following

```
extern PetscErrorCode setMirkOrder (BvdaeCtx *sol, PetscInt p);
```

A larger MIRK order implies more stage computations and therefore the benefit of parallelization may be greater.

Some communication between processors must occur when the system of equations is generated. For example, a solution component must have access to $\mathtt{u}_{i+1}$ in order to evaluate $\boldsymbol{\phi}_i(\mathbf{u}_i, \mathbf{u}_{i+1})$. Therefore the local array of dM_i solution components for a processor does not contain all the solution components required to generate M_i components of (3). Therefore, `PETSc` allows the local array of each processor to contain a buffer of solution components that have been assigned to other processors. In this case, we allow for a buffer of $2d$ solution components. Unfortunately, in order to use the buffered local arrays provided by `PETSc`, the total mesh size must be evenly divisible by $2*d+1$. As a consequence, additional mesh points are added when required. The buffered solution components are shared between processors before (3) is generated. Also, the first processor shares the current solution of $\mathbf{u}(a)$ with the last processor in order for the boundary conditions to be evaluated.

The system of nonlinear equations are generated every time an evaluation of the residual function is required for the Newton's method.

3.2 A Parallel Newton's method

The parallel toolkit `PETSc` provides access to parallel implementations of different Newton methods. By default, `PETSc_BVP` determines a Newton direction through LU decomposition and the parallel sparse direct solver `MUMPS` [1]. A cubic line search method is used as the global convergence strategy [11]. `PETSc` also provides several parallel implementations of inexact Newton-Krylov methods [7]. These methods use iterative methods to solve the linear system (4). Users of `PETSc_BVP` are free to apply the inexact methods to (3), however users may have to try a variety of methods in order to determine a successful one for a particular BVP.

Despite which Newton's method is used, a Jacobian matrix must be evaluated. By default, `PETSc_BVP` uses a finite difference scheme that exploits matrix sparsity through the use of matrix colouring [7]. Due to the sparse nature of the Jacobian matrix generated from (3), only a small portion of the Jacobian matrix must be evaluated. It has been noticed that this method is considerably faster than finite-difference scheme for a dense Jacobian matrix. Other BVP codes that allow a Jacobian to be evaluated for (3) by finite differences, `pythODE` for example [8], should employ a similar finite-difference scheme.

4 Numerical Experiments

Numerical experiments were performed in order to validate the parallel implementation of the numerical method and to show some speedup as the number of processors increase. For the latter, speedup is shown for both large mesh sizes and large problem sizes.

Experiments were performed on a single machine using shared memory. The machine consists of two 2.26 GHz Quad-Core Intel Xeon processors and 16 GB of DDR3 RAM. The operating system is Mac OS X 10.7.2. As stated in the previous section, `PETSc` 3.2 is used to build `PETSc_BVP` along with the `PETSc` distribution of `MPICH2` 1.4.1p1 and `gcc` 4.2. The software can be run on distributed machines. However, `PETSc` 3.2 must be used to build the code due to changes in the naming conventions made from the previous version.

4.1 Method validation

The BVP code `PETSc_BVP` comes bundled with several test problems that can be used to validate the parallel implementation of the MIRK method described in Section 2. These problems are especially useful for ensuring that `PETSc_BVP` is properly built for a given system. One such test problem is Bratu's equation

$$u''(x) - \lambda \exp^{u(x)} = 0, \quad 0 \leq x \leq 1,$$

with the boundary conditions

$$u(0) = u(1) = 0.$$

We solve the problem using `PETSc_BVP` for $\lambda = 1$ and an initial guess of

$$u(x) = x - x^2,$$

$$u'(x) = 1 - 2x.$$

A uniform mesh of 10 points and a 4th order MIRK method is used to determine a solution.

Figure 1 shows the result of using `PETSc_BVP` when solving the problem with one processor. A solution is also found using the `MATLAB` BVP code `bvp4c`. Excellent agreement between the two solutions is shown.

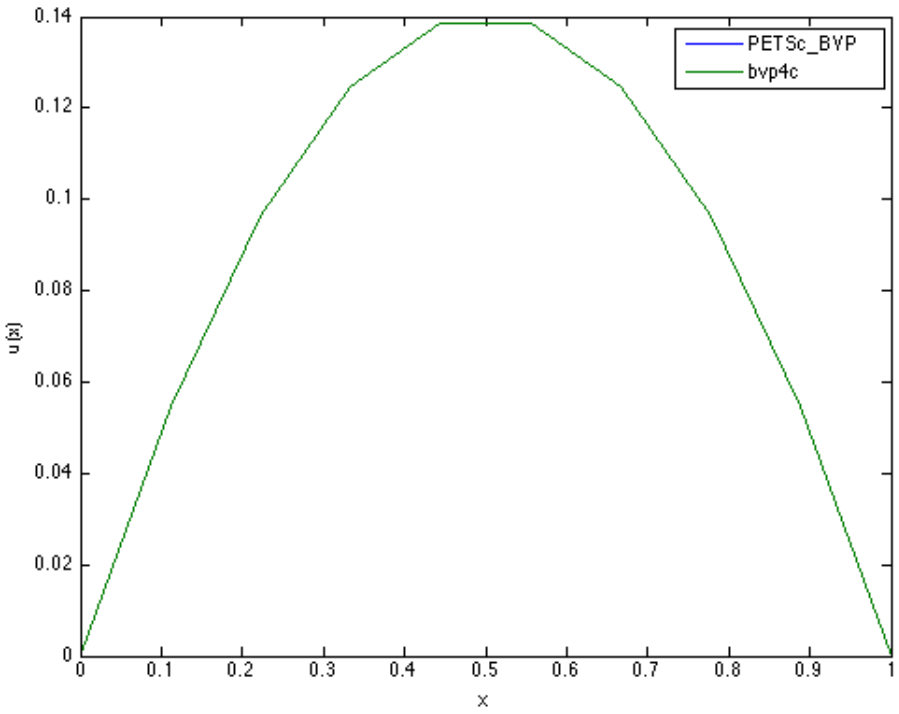

Figure 1: Solution to Bratu's equation using `PETSc_BVP` with one processor and `bvp4c`.

It is also important to ensure that the same solution is obtained despite the number of processors used. Figure 2 shows the result of using `PETSc_BVP` to solve Bratu's equation as the number of processors are varied. Excellent agreement between the solutions are shown.

4.2 Solving large mesh problems

For certain problems, BVP codes require a large mesh in order to determine a solution that meets a user-supplied tolerance placed on the error of the numerical solution. In many cases, these problems require a fine mesh to ensure the stability of the numerical method. These problems are called *stiff* problems [3]. Examples of stiff problems can be found in [9]. The authors use these problems to compare different global error estimation methods for BVP codes.

For this experiment, we demonstrate the performance of `PETSc_BVP` when solving a BVP

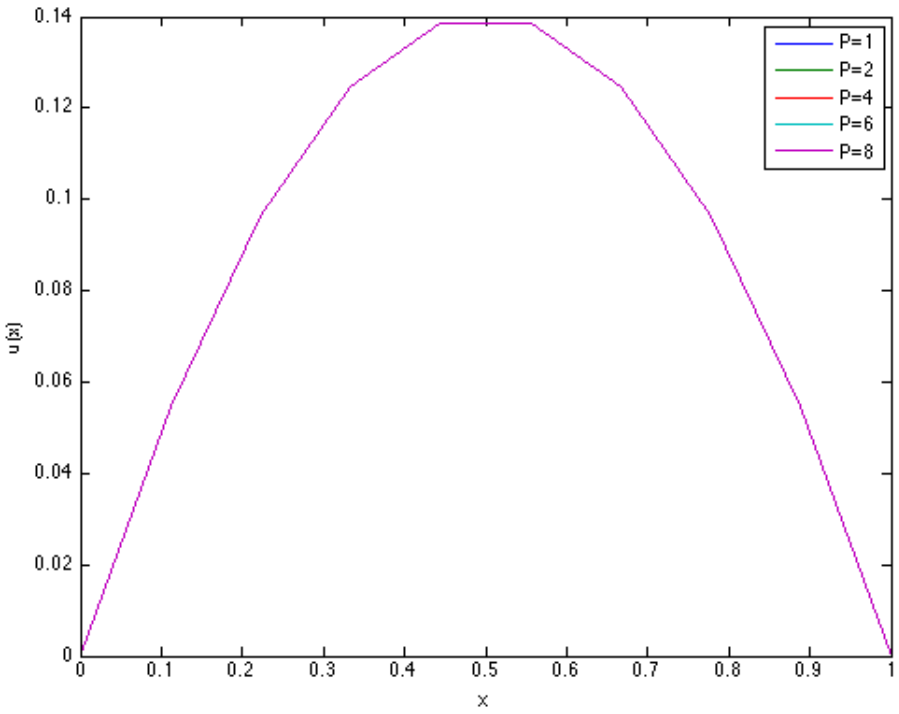

Figure 2: Solution to Bratu's equation using `PETSc_BVP` with different number of processors.

using a large mesh. A linear BVP

$$u''(x) = u(x), \quad 0 \leq x \leq 1, \tag{6a}$$

with the boundary conditions

$$u(0) = 1, \quad u(1) = 0, \tag{6b}$$

is used for this experiment. An initial guess of

$$u(x) = 1 - x,$$

$$u'(x) = -1,$$

is used. The problem is solved using 4th order MIRK formulas and a uniform mesh of size 5000 and 10000 points. Results are obtained for different number of processors.

Despite varying the number of mesh points, similar results are obtained. Figure 3 shows the time in seconds as additional processors are used to solve the problem. The time is

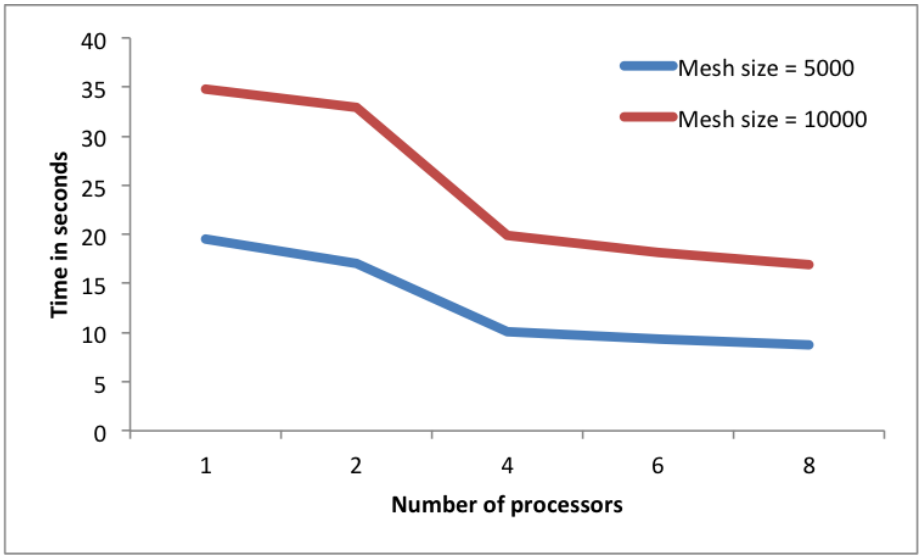

Figure 3: Time in seconds required to solve (6) using `PETSc_BVP` with different number of processors.

obtained using `MPI_WCLOCK`. The time required to determine a solution is reduced as the number of processors increase. However, the amount that the time is reduced also decreases. Figure 4 further illustrates this result by showing the speedup as the number of processors increase. The speedup is defined as

$$s_i = \frac{t_1}{t_i},$$

where s_i is the speedup for i processors and t_i is the time required to determine a solution for i processors. Figure 5 shows the change in efficiency as the number of processors increase. The efficiency can be defined as

$$e_i = \frac{s_i}{P}.$$

For i processors, e_i is the efficiency for each processor. The efficiency is reduced as the number of processors is increased. This is probably due to the increase in communication time as the number of processors also increase.

For mesh sizes less than a thousand points, additional processors may result in a larger time required to determine a solution. A sufficiently large mesh is required to offset the communication cost created by more processors.

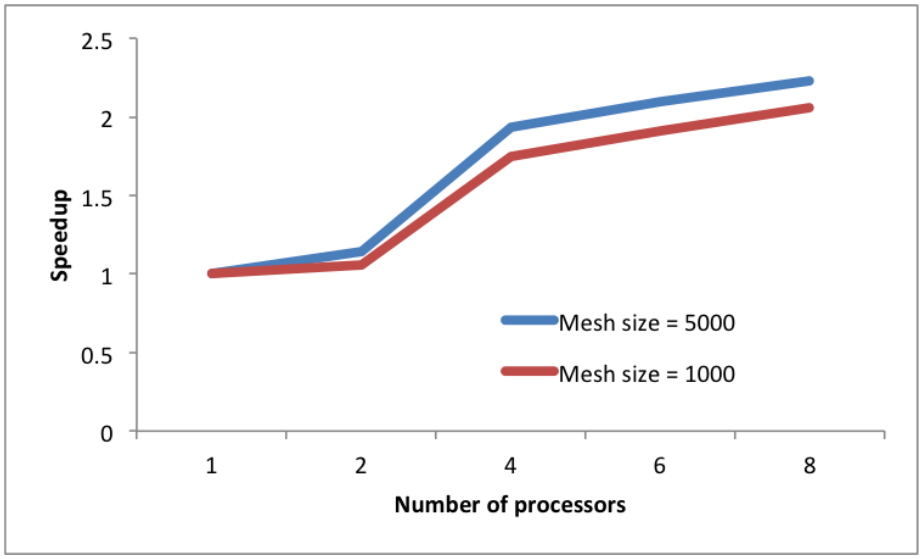

Figure 4: Speedup obtained when solving (6) using `PETSc_BVP` with different number of processors.

4.3 Solving large problems

BVPs that consist of a large system of ODEs and boundary conditions have considerably more solution components than smaller problems and therefore require more computational time to generate and solve the system of nonlinear equations (3). Large problems can benefit greatly from parallelization. In this section, we show the benefit of using multiple processors to solve one such problem.

A Raman fiber laser consists of a double-clade laser that launches a high-powered pump light into a fiber tube. The steady state of a nth order Raman fiber laser can be simulated through a series $2n+2$ ODEs and boundary conditions [10]. We start with the forward and backward-propagating pump fields. Let P_0^+ be the forward-propagating pump field and P_0^- be the backward propagating pump field. These can be described by two first-order ODEs

$$\frac{dP_0^+(z)}{dz} = P_0^+(z)\left[-a_0 - \frac{w_0}{w_1}\gamma_1[P_1^+(z) + P_0^-(z)]\right], \quad 0 \le z \le L$$

$$\frac{dP_0^-(z)}{dz} = -P_0^-(z)\left[-a_0 - \frac{w_0}{w_1}\gamma_1[P_1^+(z) + P_0^-(z)]\right],$$

where a_i is the intrinsic loss of the host glass at various filed wavelengths, w_0 is the frequency at the pumps, and w_i is the frequency of the ith Stokes field. The parameter $\gamma_i = 4.89 \times$

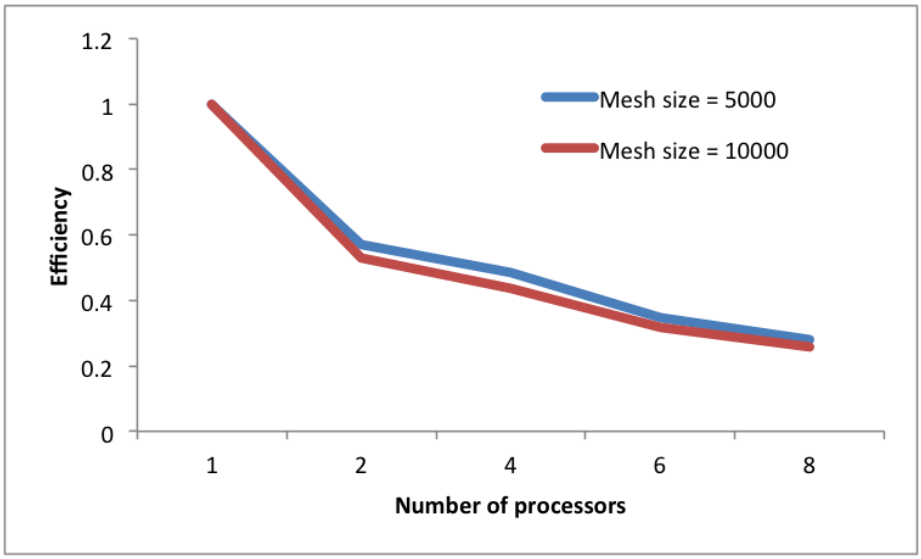

Figure 5: Efficiency when solving (6) using `PETSc_BVP` with different number of processors.

$10^{-14}/(\lambda_i/A_{\text{eff}})$ is a Raman-gain coefficient, where λ_i is the wave length of the ith Stokes field and A_{eff} is the effective core area. The parameter L is the fiber length. The forward-propagating Stokes fields P_i^+ and backward-propagating Stokes field P_i^- are given by the ODEs

$$\frac{dP_i^+(z)}{dz} = P_i^+(z)\left[-a_i + \gamma_i[P_{i-1}^+(z) + P_{i-1}^-(z)] - \gamma_{i+1}\frac{w_i}{w_{i+1}}[P_{i+1}^+(z) + P_{i+1}^-(z)]\right],$$

$$\frac{dP_i^-(z)}{dz} = -P_i^-(z)\left[-a_i + \gamma_i[P_{i-1}^+(z) + P_{i-1}^-(z)] - \gamma_{i+1}\frac{w_i}{w_{i+1}}[P_{i+1}^+(z) + P_{i+1}^-(z)]\right],$$

for $i = 1, 2, \ldots, n$ and $0 \le z \le L$. The nth forward-propagating and backward-propagating stokes field is given by

$$\frac{dP_n^+(z)}{dz} = P_n^+(z)\left[-a_n + \gamma_n[P_{n-1}^+(z) + P_{n-1}^-(z)]\right],$$

$$\frac{dP_n^-(z)}{dz} = -P_n^-(z)\left[-a_n + \gamma_n[P_{n-1}^+(z) + P_{n-1}^-(z)]\right],$$

for $0 \le z \le L$. The boundary conditions are given by

$$P_0^+(0) = P_{\text{launch}} + R_0 P_0^-(0),$$

$$P_i^+(0) = P_i^-(0), \quad i = 1, 2, \ldots n,$$

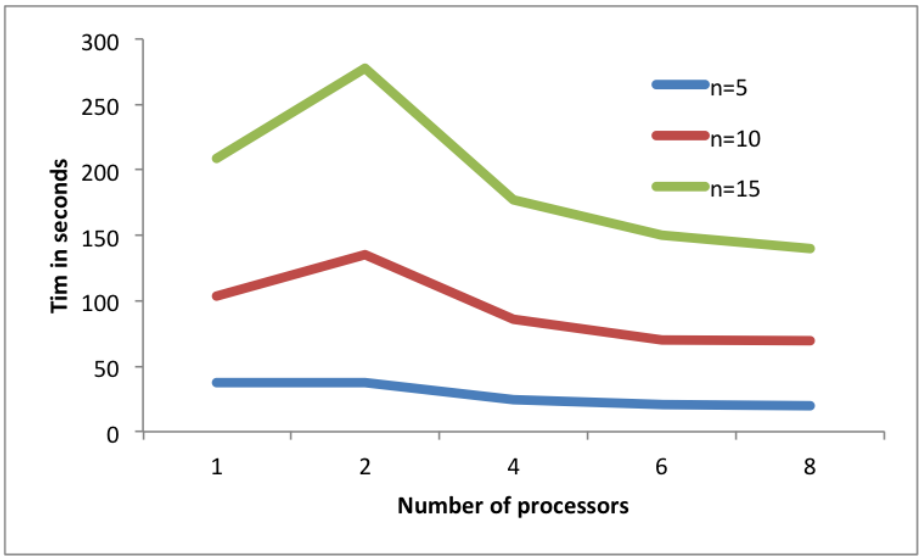

Figure 6: Time in seconds required to solve a nth order Raman fiber laser model using `PETSc_BVP` with different number of processors.

$$P_i^-(L) = P_i^+(L), \quad i = 0, 2, \ldots n-1,$$

$$P_n^-(L) = R_N P_n^+(L),$$

where R_0 is the reflectivity of the pump light at input, R_n is the reflectivity of the nth Stokes mode at output, and P_{launch} is the launched pump power.

The model is solved with `PETSc_BVP` using the parameters $L = 1\text{km}$, $a_0 = 1.7\text{dB/km}$, $a_i = 1.0\text{dB/km}$ for $i = 1, 2, \ldots, n$, and $\gamma_i = 5.20 \times 10^{-3}(\text{m} \cdot \text{W})^{-1}$ for $i = 0, 1, \ldots, n$. We let $w_0 = w_1 = \cdots = w_n$. For the boundary conditions, we use $P_{\text{launch}} = 3.3\text{W}$, $R_0 = 0.95$, and $R_n = 0.10$. These parameters were used for some of the simulations performed in [10]. A constant initial guess of

$$P_i^+(z) = \frac{1}{2},$$

$$P_i^-(z) = \frac{1}{2},$$

is used. The problem is solved for $n = 5$ with 100 mesh points, $n = 10$ with 135 mesh points, and $n = 15$ with 130 mesh points. They mesh points vary due to the requirement that $2 * d + 1$ must divide the size of the mesh evenly; see Section 3.1. A 4th order MIRK method is used for all cases.

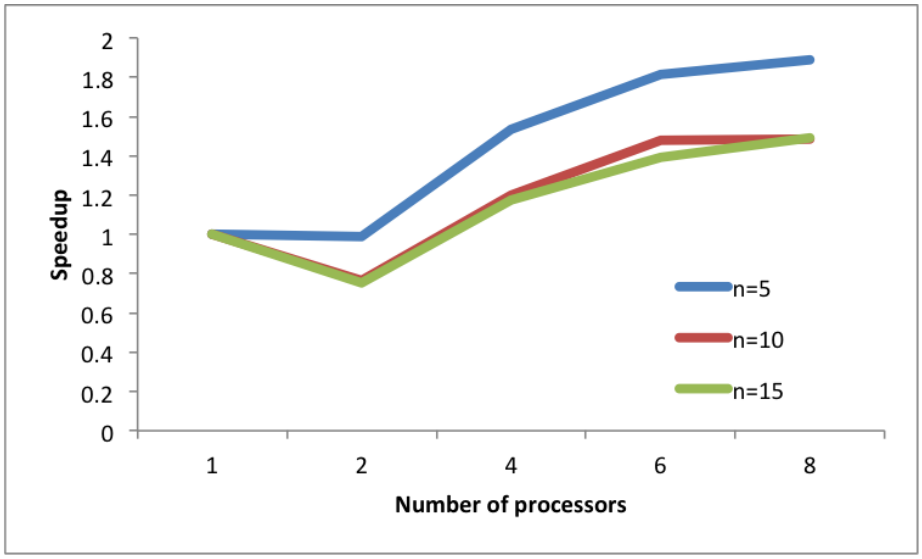

Figure 7: Speedup obtained when solving a nth order Raman fiber laser model using `PETSc_BVP` with different number of processors.

Figure 6 shows the time in seconds required to solve the Raman fiber laser model as the number of processors increase. For all three values of n, more time is required to solve the problem using two processor as opposed to one. This is likely due to the benefit of two processors being unable to offset the initialization cost of `PETSc` for multiple processors, e.g., initializing MPI and creating local distributive arrays. However, additional experiments must be performed to find the exact reason. For processors greater than two, however, a reduction of time is seen when compared to one processor. Figure 7 further illustrates this point by displaying the speedup. Figure 8 displays the change in efficiency as the number of processors grow. For all values of n, the efficiency is reduced as the number of processors is increased. Again, this is probably due to the increase in communication time as the number of processors increase.

5 Conclusions and future work

The numerical experiments show that a reduction of time can be achieved by parallelizing both the generation and solution of the system on nonlinear equations (3). However, the

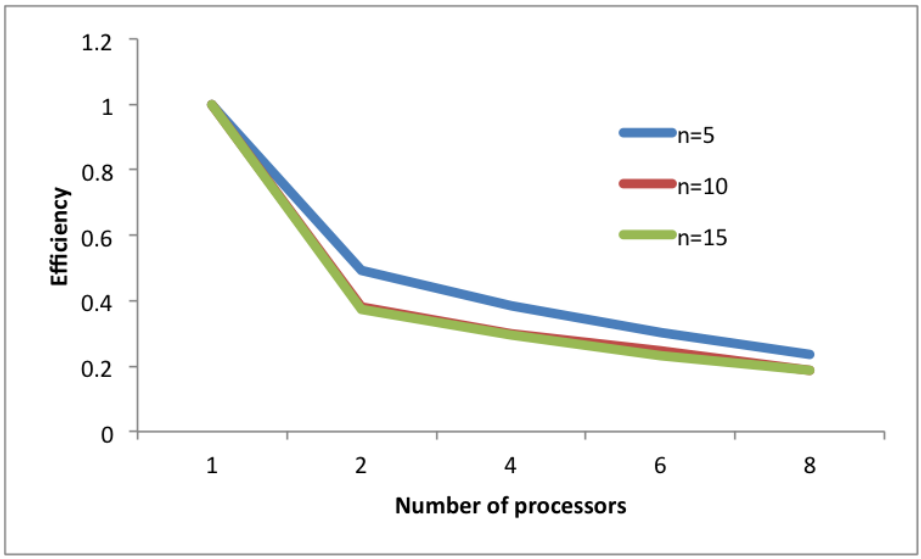

Figure 8: Efficiency when solving a nth order Raman fiber laser model using `PETSc_BVP` with different number of processors.

problem must either require enough mesh points or the problem must be large enough to offset the communication cost between processors.

This report is concluded with several suggestion for future work:

- The BVP code `PETSc_BVP` is missing many components of a fully featured BVP code such as `BVP_SOLVER` and `bvp4c`. In particular, a global error estimation method and an efficient mesh selection method should be added to the code. An efficient mesh selection method attempts to distribute mesh points in way that ensures the error on each subinterval is less than a user-supplied tolerance, e.g., mesh equidistribution [4]. `PETSc` should also be used to parallelize these routines.

- The numerical experiments described in this report should be performed on a distributed memory cluster. By doing so, the cost of inter-node communication can be compared to the cost of the processor communication on a shared memory machine.

- By default, `PETSc` uses the parallel sparse direct solver `MUMPS` to determine the Newton direction. However, this solver does not take advantage of the known ABD sparsity structure of the Jacobian matrix of (3). Instead, additional improvements of time may

be achieved by using a parallel direct solver that is optimized for the solution of ABD matrices. `RSCALE` is an example of one such solver [15].

References

[1] Miumps, April 2012. `http://graal.ens-lyon.fr/MUMPS/`.

[2] P. Amodio, J. R. Cash, G. Roussos, R. W. Wright, G. Fairweather, I. Gladwell, G. L. Kraut, and M. Paprzycki. Almost block diagonal linear systems: sequential and parallel solution techniques, and applications. *Numerical Linear Algebra with Applications*, 7(5):275–317, 2000.

[3] U. Ascher. On numerical differential algebraic problems with application to semiconductor device simulation. *SIAM J. Numer. Anal.*, 26(3):517–538, 1989.

[4] U. M. Ascher, R. M. M. Mattheij, and R. D. Russell. *Numerical solution of boundary value problems for ordinary differential equations*, volume 13 of *Classics in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1995. Corrected reprint of the 1988 original.

[5] U. M. Ascher and L. R. Petzold. *Computer methods for ordinary differential equations and differential-algebraic equations*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1998.

[6] G. Bader and U. Ascher. A new basis implementation for a mixed order boundary value ODE solver. *SIAM J. Sci. Statist. Comput.*, 8(4):483–500, 1987.

[7] S. Balay, J. Brown, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.2, Argonne National Laboratory, 2011.

[8] J. J. Boisvert. A problem-solving environment for the numerical solution of boundary value problems, 2010.

[9] J. J. Boisvert, P. H. Muir, and R. J. Spiteri. A numerical study of global error estimation schemes for defect control bvode codes. *Saint Mary's University, Dept. of Math. and Comp. Sci. Technical Report Series,cs.smu.ca/tech_reports/*, 2009.

[10] S. D. Jackson and P. H. Muir. Theory and numerical simulation of nth-order cascaded raman fiber lasers. *J. Opt. Soc. Am. B*, 18(9):1297–1306, Sep 2001.

[11] C. T. Kelley. *Iterative methods for linear and nonlinear equations*, volume 16 of *Frontiers in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1995. With separately available software.

[12] J. Kierzenka and L. F. Shampine. A BVP solver based on residual control and the MATLAB PSE. *ACM Trans. Math. Software*, 27(3):299–316, 2001.

[13] J. Kierzenka and L. F. Shampine. A BVP solver that controls residual and error. *JNAIAM J. Numer. Anal. Ind. Appl. Math.*, 3(1-2):27–41, 2008.

[14] P. H. Muir. Optimal discrete and continuous mono-implicit Runge–Kutta schemes for BVODEs. *Adv. Comput. Math.*, 10(2):135–167, 1999.

[15] P. H. Muir, R. N. Pancer, and K. R. Jackson. PMIRKDC: a parallel mono-implicit Runge-Kutta code with defect control for boundary value ODEs. *Parallel Comput.*, 29(6):711–741, 2003.

[16] L. F. Shampine, P. H. Muir, and H. Xu. A user-friendly Fortran BVP solver. *JNAIAM J. Numer. Anal. Ind. Appl. Math.*, 1(2):201–217, 2006.