Partitioning the time interval

March 2, 2017

1 The problem

Decomposing the time interval $I = [0, T]$ into N equally sized subintervals $I_i = [T_i, T_{i+1}]$, and solving the state and adjoint equations separately on each subinterval, allows our algorithm to be run in parallel. Decomposing I is simple in the continuous case, however in practice we are solving these equations numerically, and in the discrete case, partitioning I is more involved. To explain how we decompose I in the discrete case, lets look at how to do it for the state equation. Define a differential equation F:

$$\begin{cases} F(y(t), v(t)) = 0 \ \textit{For} \ t \in [0, T] \\ y(0) = y_0 \end{cases}$$

We then decompose I, and assume that we have $N-1$ intermediate initial conditions $\{\lambda_i\}_{i=1}^{N-1}$, such that we get a solvable equation on each subinterval:

$$\begin{cases} F^i(y_i(t), v(t)) = 0 \ \textit{For} \ t \in [T_i, T_{i+1}] \\ y(T_i) = \lambda_i \end{cases}$$

Now lets look at what happens when we discretize I. Lets divide I into n parts of length $\Delta t = \frac{T}{n}$, and set $t_k = k\Delta t$. This gives us a sequence $I_{\Delta t} = \{t_k\}_{k=0}^{n}$ as a discrete representation of the interval I. Using some finite difference scheme, we can transform the differential equation F into a difference equation $F_{\Delta t}$:

$$\begin{cases} F_{\Delta t}(y^k, v(t_k)) = 0 \ \textit{For} \ k = 1, ..., n \\ y^0 = y_0 \end{cases}$$

The next step is to decompose the discrete interval $I_{\Delta t}$ into N discrete subintervals. This is simply done by extracting a subsequence $\{t_{k_i}\}_{i=0}^{N} \subset I_{\Delta t}$ where $t_{k_0} = t_0$ and $t_{k_N} = t_n$. This results in N sequences on the form $I_{\Delta t}^i = \{t_{k_i}, t_{k_i+1}, ..., t_{k_{i+1}}\}$, and if we assume, as we did in the continuous case, that we have some intermediate initial conditions $\{\lambda_i\}_{i=1}^{N-1}$, we can solve $F_{\Delta t}$ separately on each $I_{\Delta t}^i$:

$$\begin{cases} F_{\Delta t}^i(y_i(t_k), v(t_k)) = 0 \ \textit{For} \ k \in \{k_i, k_i + 1, ..., k_{i+1}\} \\ y(t_{k_i}) = \lambda_i \end{cases}$$

There is one minor issue with decomposing $I_{\Delta t}$, which I did not mention above, and that has to do with the choice of the subsequence $\{t_{k_i}\}_{i=0}^{N}$. In theory, one could of course freely chose any subsequence of $I_{\Delta t}$, but we generally want the difference $t_{k_i} - t_{k_{i+1}}$ to be constant for all i. This is however not always possible, since there is no guarantee that n is divisible by N.

2 Partitioning

The general rule for partitioning a task between N processes, is to distribute the work of the task as evenly as possible. The task in the above setting is solving $F_{\Delta t}$, and the work to be distributed, is the computations required to move the solution from one time step to the next for all time steps. Since there are n time steps, we should be able to say that the main task of solving $F_{\Delta t}$ can be divided into n subtasks, and it is these n tasks that we want to distribute between the N processes. Now deciding how many subtasks each process should get is quite simple. Start with defining the numbers:

$$\begin{aligned} q &= \lfloor \frac{n}{N} \rfloor \\ r &= N \mod n \end{aligned}$$

Then we give each process q tasks, and then add one task to r processes. To which processes you give the extra task does not really matter, but the most straightforward way of doing it is just to give the first r processes the extra task. I however chose to look at the distributing problem slightly different, by instead of trying to distribute the n tasks, looking at how to evenly divide the $n+1$ points $\{t_k\}_{k=0}^{n}$ among the N processes. Again lets define q and r as:

$$\begin{aligned} q &= \lfloor \frac{n+1}{N} \rfloor \\ r &= N \mod n+1 \end{aligned}$$

Every process now gets q points, but due to overlap every process gets an extra point excluding the first process. Then the remaining r points are given to the first r processes. This allows me to define the sequence $\{k_i\}_{i=0}^{N}$ recursively as follows:

$$k_{i+1} = k_i + q + \delta_{i\neq 0} + \delta_{i<r}$$

Here the δs are conditional functions defined as:

$$\delta_S = \begin{cases} 1 \; S = \text{True} \\ 0 \; S = \text{False} \end{cases}$$

We now have a way of decomposing the discrete time interval, and therefore also a way of partitioning the finite difference solver of the state equation in temporal direction. However both when we want try to find the gradient in a penalized optimal control problem, or when we just want to parallelize solving a differential equation using the parareal scheme, some communication between the processes is required.

3 Communication without shared memory

If we assume that we are solving our optimal control problem in parallel on processes that do not share any memory, there will have to be communication between the processes. The parts of the algorithm that we are parallelizing using the penalty method, is the evaluation of the objective function and its gradient for a given control variable v. The function evaluation requires us to solve the state equation, while we to calculate the gradient need both the solution of the state and adjoint equation. the required communication between the processes are different for function and gradient evaluation, so I will describe them separately. However they both share the same starting point, which I explain below.

Lets assume that we have N processes, which we name $\{P_i\}_{i=0}^{N-1}$. Then assume that each process P_i only knows the parts of the control that are required for the process to be able to solve the state equation and to locally evaluate the objective function. This also includes the the control variables $\{\lambda_i\}_{i=1}^{N-1}$ that originates from the penalty terms in the functional. After each P_i then have solved their part of the state equation, they all have the following data stored locally:

Control variable: v_i

Penalty control variable: λ_i

Solution to local state equation: $y^i = \{y^i_j\}_{j=k_i}^{k_{i+1}}$

Using this data we should be able to evaluate the penalized objective function, or to calculate its gradient.

3.0.1 Communication in functional evaluation

The penalized objective function consists of two parts:

$$\hat{J}_\mu(v,\lambda) = \hat{J}(y(v),v) + \frac{\mu}{2}\sum_{j=1}^{N-1}(y^{j-1}(T_j)-\lambda_j)^2$$

Lets begin with the penalty term. Since each process P_i only have λ_i and $y^i(T_i+1)$ stored locally. This means that to calculate all penalty terms the processes will have to send either λ_i or $y^i(T_i + 1)$ to one of it neighbours. For example P_i could send λ_i to P_{i-1} for $i = 1, ..., N-1$:

$$P_0 \xleftarrow{\lambda_1} P_1 \xleftarrow{\lambda_2} P_2 \xleftarrow{\lambda_3} \cdots \xleftarrow{\lambda_{N-2}} P_{N-2} \xleftarrow{\lambda_{N-1}} P_{N-1}$$

For the evaluation of $\hat{J}(y(v), v)$. lets assume that there exists functions $\hat{J}^i(y^i(v_i), v_i)$, such that:

$$\hat{J}(y(v), v) = \sum_{j=0}^{N-1} \hat{J}^j(y^j(v_j), v_j)$$

If this is the case we can evaluate each part of the objective function locally, and then get the global $\hat{J}_\mu$ by doing one summation reduction. The penalized objective function evaluation algorithm will then look like the following:

1. P_i *calculates* $y^i \; \forall i$
2. $P_{i-1} \xleftarrow{\lambda_i} P_i$ *for* $i = 1, .., N-1$
3. $\hat{J}^i_\mu(y^i(v_i), v_i) = \hat{J}^i(y^i(v_i), v_i) + \frac{\mu}{2}(y^i(T_{i+1}) - \lambda_{i+1})^2$ *for* $i = 0, .., N-2$
   $\hat{J}^{N-1}_\mu(y^{N-1}(v_{N-1}), v_{N-1}) = \hat{J}^{N-1}(y^{N-1}(v_{N-1}), v_{N-1})$
4. $\hat{J}_\mu(y(u), u) = \mathbf{Reduce}(\hat{J}^i_\mu, +)$

3.0.2 Communication in gradient computation

The gradient of the penalized optimal control problem looks like the following:

$$\nabla \hat{J}_\mu(v, \lambda) = (J_v(y(v), v) - B^* p, \{p_i(T_i) - p_{i-1}(T_i)\}_{i=1}^{N-1})$$

p is here the solution to the adjoint equation, which has to be calculated before we can evaluate the gradient. For processes P_i, $i < N-1$, the initial condition of the adjoint equation is $p^i(T_{i+1}) = \mu(y^i(T_{i+1} - \lambda_{i+1})$. This means that the first step after solving the state equations for gradient evaluation, is the same as for function evaluation, i.e. we have to send λ_i from $P - i$ to P_{i-1}:

$$P_0 \xleftarrow{\lambda_1} P_1 \xleftarrow{\lambda_2} P_2 \xleftarrow{\lambda_3} \cdots \xleftarrow{\lambda_{N-2}} P_{N-2} \xleftarrow{\lambda_{N-1}} P_{N-1}$$

Each process can now solve its adjoint equation locally, and we can start to actually evaluate the gradient. The first step, would be to send $p_i(T_i)$ from P_i to P_{i-1} so that we can find the penalty part of the gradient. Each process should also be able to calculate their own part of the gradient as $\nabla \hat{J}^i = (J_v(y^i(v^i), v^i) - B_i^* p^i)$. The final step is now to gather all the local parts of the gradient to the form the actual gradient. In summation we get the following algorithm for gradient evaluation:

1. P_i *calculates* y^i $\forall i$
2. $P_{i-1} \xleftarrow{\lambda_i} P_i$ *for* $i = 1, .., N-1$
3. P_i *calculates* p^i $\forall i$
4. $P_{i-1} \xleftarrow{p^i(T_i)} P_i$ *for* $i = 1, .., N-1$
5. *Penalty part of gradient* $p_i(T_i) - p_{i-1}(T_i)$ *is stored for* $i \neq N-1$
   Local gradient if calcualated: $\nabla \hat{J}^i = (J_v(y^i(v^i), v^i) - B_i^* p^i)$
6. $\nabla \hat{J}_\mu = \mathbf{Gather}(\nabla \hat{J}^i, p_i(T_i) - p_{i-1}(T_i))$

4 Analysing parallel performance

Now that we what type of communication is involved in objective function evaluation and gradient computation, we can try to model the expected performance of the two algorithms. One way to measure performance of algorithms is to at their execution times. Therefore lets define T_s as execution time of sequential algorithm, and T_p as parallel algorithm execution time. Let us also define the speedup $S = \frac{T_s}{T_p}$. Since we for now only are modelling performance we do not actually calculate the execution times, but we do know that the run time of the algorithms are related to the size of the problem, namely the number of time-steps n. The last thing we need before we start our performance analysis, is a way to model communication between two processes. The communication time T_c for sending a message of size m between to processes, is usually modelled in the following way:

$$T_c = T_l + mT_w$$

Here T_l is a constant representing latency or startup time, while T_w is a constant representing the per message-unit transfer time. With these tools, we can now start analysing the performance of our algorithms.

4.1 Objective function evaluation speedup

The function evaluation consists of solving the state equation, which requires $O(n)$ operations, and then applying the functional on the control and the state on all $n+1$ time points, which probably also requires $O(n)$ operations, we can assume that the sequential Objective function evaluation execution time is

$$T_s = O(n)$$

For our parallel algorithm we also solve the state equation and apply the functional, but since we divide the time steps equally to each process, solving the state equation and applying the functional only requires $O(\frac{n}{N})$ operations. Since we also have penalty terms in our functional we also get an extra $O(N)$ operations. Now for the communication. We are doing two communication steps one is sharing the λs between process neighbours, and the other is reducing the local function values into one global function value. The send and receive time is given by $T_c = T_l + dim(\lambda_i)T_w$, which requires $O(1)$ operations, while the reduction time T_{red} can be modelled as:

$$\begin{aligned} T_{red} &= \log N(T_l + T_w) \\ &= O(\log N) \end{aligned}$$

Here we assume that the parallel architecture is made in a certain way, and that the local functional value is a float. This results in parallel function evaluation execution time:

$$\begin{aligned} T_p &= O(\frac{n}{N}) + O(N) + O(\log N) + O(1) \\ &= O(\frac{n}{N}) + O(N) \end{aligned}$$

The speedup is then:

$$\begin{aligned} S = \frac{T_s}{T_p} &= \frac{O(n)}{O(\frac{n}{N}) + O(N)} \\ &= O(N) \end{aligned}$$

This is an optimal speedup.

4.2 Gradient speedup

When we calculate the objective function gradient sequentially, we solve both the state and adjoint equations. Still the required operations is still in the order of number of time-steps, i.e:

$$T_s = O(n)$$

For the parallel algorithm the operations required to solve the local state and adjoint equations are $O(\frac{n}{N})$. We then have two $O(1)$ send receive communications similar to the send and receive for function evaluation. Lastly we need to model the gathering of the gradient. First define L to be the length of the gradient. The run time of the gather T_{gather}, can then be modelled as:

$$\begin{aligned} T_{gather} &= T_l \log N + \frac{L}{N} T_w(N-1) \\ &= O(\log N) + O(L) \end{aligned}$$

The execution time of the parallel algorithm is therefore:

$$T_p = O(\frac{n}{N}) + O(\log N) + O(L)$$

Again we find speedup by dividing T_s by T_p:

$$\begin{aligned} S = \frac{T_s}{T_p} &= \frac{O(n)}{O(\frac{n}{N}) + O(\log N) + O(L)} \\ &= \frac{1}{\frac{1}{N} + \frac{\log N}{n} + \frac{L}{n}} = \frac{1}{\frac{1}{N} + \frac{L}{n}} \end{aligned}$$

If L is independent of n, the speedup for gradient evaluation is $O(N)$, like it is for function evaluation, however if L is dependent on n this not the case, and we would instead get speedup $S = O(\frac{n}{L(n)})$. In a case where the control for example is the source term in the state equation, we would actually get $S = O(1)$, which is really bad, and we would not expect any improvement when using parallel, at least for large n values.