PARALLELISING MATLAB CODE TO IMPROVE SPEED

Parallelising splits the operations contained within a piece of code and sends it to multiple processors in your computer. This should speed up computation time as multiple processors execute the code simultaneously. Writing parallelisable code is especially helpful when working with computationally intensive problems.

In this post, I will focus on transforming a for loop to a parfor loop in MATLAB. The parfor loop will execute all the statements contained in the loop body in parallel, unlike in the for loop where each iteration must wait its turn. Parfor is contained in the Parallel Computing Toolbox. As an example, I will show how parallelising code can be useful if we wish to solve a cost minimisation for several firms and aim to improve the speed of computation. See Fernández-Villaverde and Valencia (2018) for other applications of parallel computing to economics, such as value function iteration.

Before getting into the specifics, it is helpful to understand what is happening behind the scenes when a parfor loop is executed. This process can be split into three steps.

1. First, the MATLAB client (i.e. the MATLAB interface where the code is written up) issues the parfor command and oversees the MATLAB workers in the parallel pool. MATLAB workers run in the background and follow the commands issued by the MATLAB client, executing the loops in parallel.
2. Second, the data needed for the workers is sent and the computations are executed.
3. Third, these results are then sent back to the MATLAB client and assembled.

Parfor is best used in computationally demanding problems as the communication between the client and workers, plus the reassembly of the data all takes time. Therefore, you must be sure this parallel overhead is time saving for your problem, which it may not always be if the for loop is only needed for a computationally simple task.

Each cycle through a parfor loop is termed an iteration. It's important to note that MATLAB workers do not evaluate iterations in a particular order. Therefore, each iteration in a parfor loop must be independent and self-contained.

How to set up parfor loop

This worked example will feature a cost minimisation problem including ten firms, indexed by j in the loop, each of which will employ ten intermediate inputs and labour to be used in its production. Each firm will solve its cost minimisation problem, minimising its cost function subject to its production function.

For more background on setting up a constrained optimisation problem in MATLAB and solving it using fmincon, please see my previous Coders' Corner post How to Solve Constrained Optimisation Problems in MATLAB.

Before running the parfor loop, I create a matrix and vector in which the results will be saved. X_quant is the quantity of each intermediate input and the quantity of labour that will be employed by the firm to minimise its cost. Fval_quant will be the associated minimum cost. Each firm's results will be displayed in the corresponding row.

```
x_quant = nan(n,n+1);
fval_quant = nan(n,1);
```

At the start of the parfor loop I create a temporary matrix/vector. The dimensions of the matrix must be consistent with the dimensions of the results for a single firm, not all the firms through which we'll be looping. By construction, a firm will minimise its costs of employing ten intermediate inputs and labour subject to its production function. Temp_x_quant will be a row vector of length 11 (the ten intermediate inputs plus labour), and temp_fval_quant, a scalar of the associated cost of production. Both must be pre-allocated in order for parfor to run without error.

These temporary vectors are necessary for us to transfer these results of interest to a storage matrix that will exist in the MATLAB workspace after the parfor loop is executed. You'll notice when we run the code, temp_x_quant and temp_fval_quant, will not appear in the MATLAB workspace.

```
parfor j = 1:n

        temp_x_quant = nan(1,n+1)
        temp_fval_quant = nan(1,1);
```

The remaining body of the parfor loop then executes the firms' cost minimisation problem. Please review the aforementioned pdf file for a detailed explanation of the code written for cost minimisation. Costfun is a function that assembles the firm's linear cost function, and prodfun is the constraint.

The results from fmincon are stored in the temporary matrices I created.

```
        objective = @(par) costfun(par,p,w,n);
        constraint = @(par) prodfun(par,a_vec,alpha_m,q,j,n);

        %solve
        lbtmp = zeros(1,1); % min intermediate input quantities
        lmin = 0.0001; % min labour quantity, essential input
        lb =  [lbtmp, lmin];

        ubtmp = repmat(1000,1,1)'; % max int input quantities
        lmax = 1000; % max labour quantity
        ub = [ubtmp, lmax];
        % no linear constraints so set those arguments to []
        A = [];
        b = [];
        Aeq = [];
        beq = [];

        % choose initial starting point satisfying constraints
        % optimal input choice order of matrix: [x1, x2, x3, ..., l]
        x0 = repmat(0.001,n+1,1)';

        options = optimset('display','iter', 'PlotFcn', {@optimplotfval}, 'MaxFunEvals', 10000);

        [temp_x_quant(1,1:n+1), temp_fval_quant(1,1)] = fmincon(objective,x0,A,b,Aeq,beq,lb,ub,constraint,options);
```

We must transfer the results stored in the temporary vectors to the storage matrix created before the parfor loop so that we can access the results in the workspace. Each firm's results will enter as a different row in the x_quant and fval_quant matrices.

```
        x_quant(j,:) = temp_x_quant;
        fval_quant(j,1) = temp_fval_quant;
end
```

For this problem, using a for loop takes MATLAB 56 seconds to execute the code on my computer, while the parfor loop takes 50 seconds. Saving 6 seconds can be very useful, especially if this parfor loop is contained within another for loop that may have many iterations itself. The time savings can be much more substantial for more computationally demanding problems, especially if the problem in the loop is more time-consuming.

Note that you cannot nest a parfor loop in another parfor loop. Therefore, you should parallelise your most computationally intensive forloop to see the most gains.

While tinkering with parallelising your code, you cannot break out of a parfor loop early, e.g. using commands such as keyboard to check your code. It's best to write the code consistent with parallelisation, and check a few lines with a for loop before running it. Once you want to check whether it is fully operational as parallelised code, replace the for loop with a parfor loop and iterate until there is no error.

References

Fernández-Villaverde, J. and Valencia, D.Z., 2018. A practical guide to parallelization in economics (No. w24561). National Bureau of Economic Research.

Diana Beltekian, PhD Candidate in Economics, University of Nottingham

23 November 2021