

# Matlab's Parallel Computation Toolbox

## And the Parallel Interpolation of Commodity Futures Curves

William Smith, March 2010

Verson 1.0 (check [www.CommodityModels.com](http://www.CommodityModels.com) for any updates)

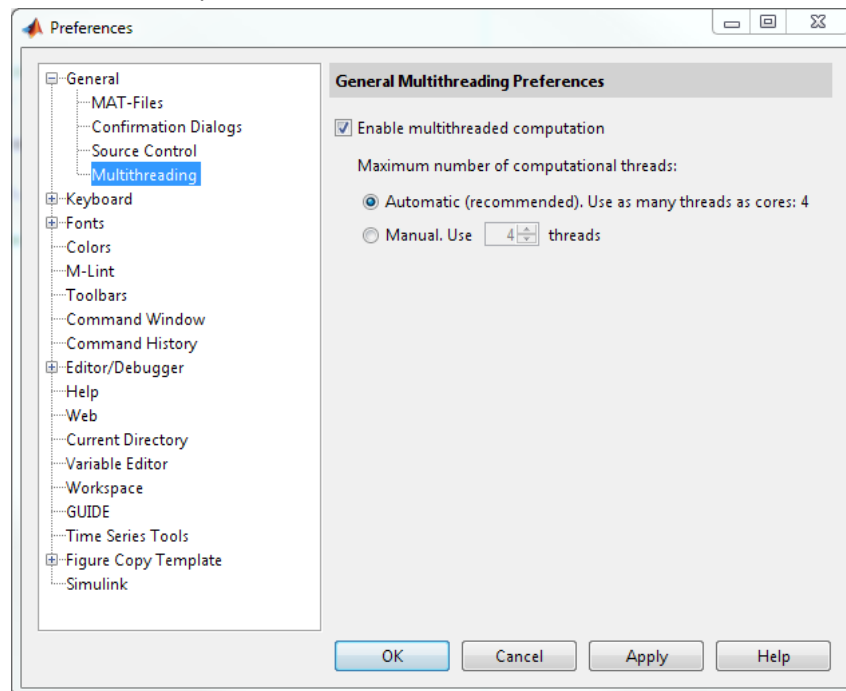
### Introduction

Matlab 2010a onwards finally enables the 'Parallel Computation Toolbox' for student use. It's not part of the core Matlab student package, but it is now available as an add-on toolbox. Of course, it's been available for commercial users for some time. I decided to learn some of its basic features and benchmark it against serial computation. I also aim for backwards compability and to implement parallel software that never run slower than its serial counterpart. As a sample problem which could be parallelised, I use my existing algorithm for interpolating maturity 'gaps' in commodity futures curves.

### Parallelism In Matlab

On a multi-core computer, Matlab can take advantage of the multiple processing units (which I'll call 'cores') to various extents:

- Many of the inbuilt functions<sup>1</sup> in Matlab are multi-core/ multi-thread aware. If the problem size is sufficiently large, they may run in parallel on your machine with no effort. Just make sure you set the relevant option in the Preferences:



<sup>1</sup> For a list, see <http://www.mathworks.com/support/solutions/en/data/1-4PG4AN/?solution=1-4PG4AN>

In Matlab 2009a onwards, this option in the preferences panel doesn't exist<sup>2</sup> and multithreading is always turned on unless you turn it off via the command-line at start-up.

- The 'parallel computing toolbox'<sup>3</sup> creates multiple workers (called 'labs' in Matlab jargon) on your local machine or even more on multiple machines in a cluster (you'll need the 'distributed computing server'<sup>4</sup> option for cluster work, not available for users of Matlab-student, and not cheap). You can then explicitly write your software to use the multiple labs, where possible.
- Other toolboxes are also aware of the parallel computing toolbox, if installed, and will take advantage of parallel computing to some extent<sup>5</sup>.

I cover below, my initial experimentation with the parallel computing toolbox, as part of Matlab-student-2010a, on my quad-core Windows-based machine.

---

<sup>2</sup> Thread option in control panel, see <http://www.mathworks.com/access/helpdesk/help/techdoc/rn/bry1ecg-1.html#br1ask3-1>

<sup>3</sup> Matlab parallel computing toolbox, see <http://www.mathworks.com/products/parallel-computing/>

<sup>4</sup> Matlab distributed computing server, see <http://www.mathworks.com/products/distriben/>

<sup>5</sup> Toolboxes aware and using parallel computing toolbox, see <http://www.mathworks.com/products/parallel-computing/builtin-parallel-support.html>

## Backwards Compatibility with Matlab without the Parallel Computing Toolbox.

For mixed environments when we can't be sure whether we have the parallel computing toolbox installed, it will be helpful, where possible, to write code that handles both cases.

We'll use the

```
parfor
    ...
end
```

syntax instead of the usual

```
for
    ...
end
```

syntax. We also need to start up the various labs. To make this backwards compatible, we create a utility function `isfunction`, which is sadly not implemented in Matlab:

```
function [ result ] = isfunction( function_string )
% Returns whether function_string is a known function.

result = 0; %#ok<NASGU>

% If it's a keyword, I'd say 'yes'. Although some might dispute this.
if iskeyword(function_string)
    result = 1;
else
    fh = str2func(function_string);
    f = functions(fh);
    % If it's a function, functions() will return the file it's in.
    if (~isempty(f.file))
        result = 1;
    else
        result = 0;
    end
end
end
```

we then can write code that will start the parallel labs only if we detect we have the parallel computing toolbox available, i.e. if the `'matlabpool'` function is present. Without this test, we get an `'Undefined function matlabpool'` error if we don't have the Parallel Computing Toolbox installed.

```
if (isfunction('matlabpool'))
    % Start labs if necessary.
    sz = matlabpool('size');
    if (sz == 0)
        matlabpool('open');
    end
    % Check we got some now.
    sz = matlabpool('size');
    if (sz == 0)
        error('Failed to open parallel workers');
    else
        fprintf('Running on %d workers\n', sz);
    end
end
```

## Simple Speed Test, Parallel vs. Serial

Let's write a simple function which takes an appreciable amount of time, and can be run in serial or parallel mode, with a variable number of iterations, and returns the time taken. We can then benchmark parallel and serial running for different problems sizes. For example:

```
function [ duration ] = TestMeParfor( count, mode )
%Just run a simple arithmetic calculation in a ParFor. Pass in 'serial' or
% 'parallel'.

A = zeros(count,1);
tic
if (strcmp(mode, 'serial') )
    fprintf('serial : %s\n', mode);
    for i=1:1:count
        for j=1:1:1000
            A(i) = A(i) + sin(i*2*pi/count)+j/1000000;
        end
    end
else
    fprintf('parallel : %s\n', mode);
    parfor i=1:count
        for j=1:1:1000
            A(i) = A(i) + sin(i*2*pi/count)+j/1000000;
        end
    end
end
duration = toc;

end
```

Now we call the function repeatedly for increasing problem size, and plot the results:

```
function [ ] = main( )
%Run a Simple Function in Parallel or Serial Model, compare timings and
%speedups.
%This function is compatible with Matlab, even if it doesn't have the
%parallel toolbox loaded.

fprintf('Testing Parfor Performance\n');

if (isfunction('matlabpool'))
    % Start labs if necessary.
    sz = matlabpool('size');
    if (sz ==0)
        matlabpool('open');
    end
    % Check we got some now.
    sz = matlabpool('size');
    if (sz ==0)
        error('Failed to open parallel workers');
    else
        fprintf('Running on %d workers\n', sz);
    end
end

iterations = [ ] ;
i = 1;
while (i < 10000)
    iterations = [ iterations floor(i) ];
end
```

```

    i = i * 1.5;
    i = i + 1;
end

partime = zeros(size(iterations));
sertime = zeros(size(iterations));
speedup = zeros(size(iterations));
for i=1:length(iterations)

    fprintf('Testing with %d iterations: ', iterations(i));
    partime(i) = TestMeParfor(iterations(i), 'parallel');
    sertime(i) = TestMeParfor(iterations(i), 'serial');
    speedup(i) = sertime(i) / partime(i);
    fprintf('%.2fx speedup (%.3f => %.3f seconds)\n', speedup(i), sertime(i), partime(i));
end

subplot(1,2,1);
plot(iterations, partime, ...
     iterations, sertime);
title('Serial vs Parallel Timings');

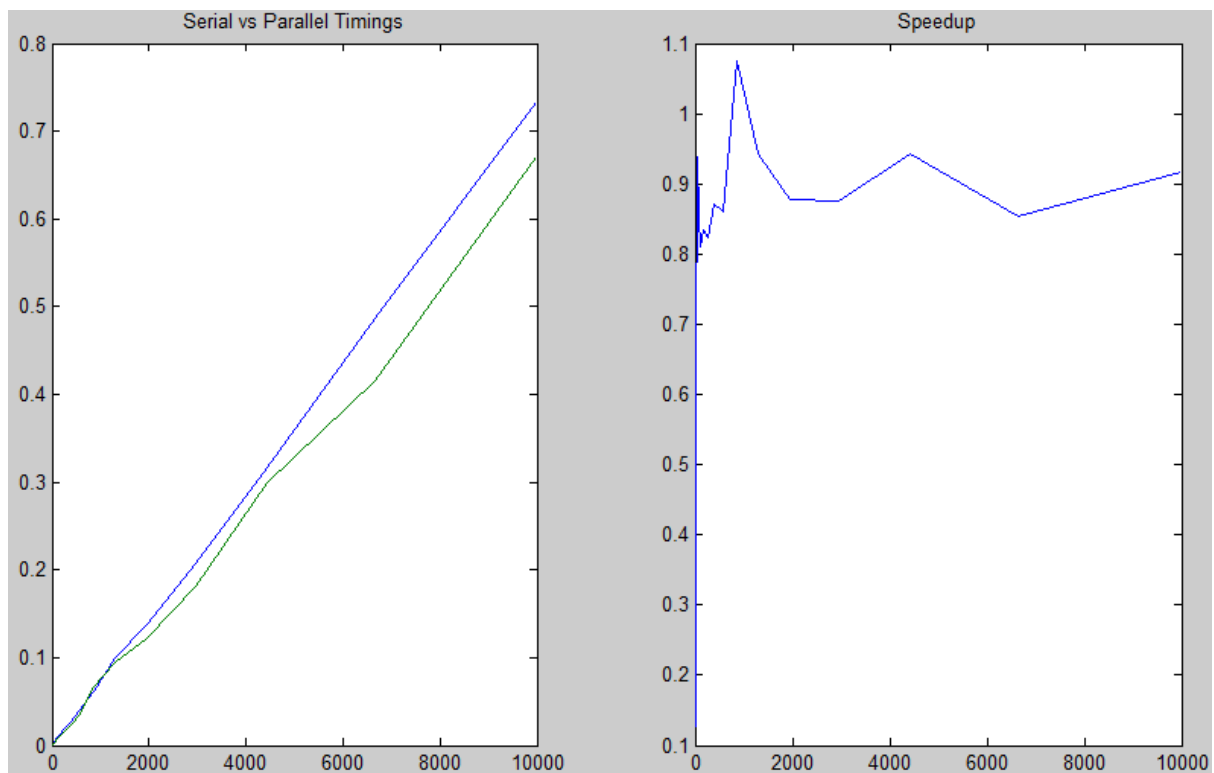
subplot(1,2,2);
plot(iterations, speedup);
title('Speedup');

end

```

## Results on Serial-Matlab

Running on Matlab with no parallel toolbox (for example, my old install of 2008b), we see the following:



Clearly there's no speedup, as expected. But we also see that the 'parfor' syntax works fine even without the parallel computing toolbox, and does not provide too much of an overhead. Exact results are:

```

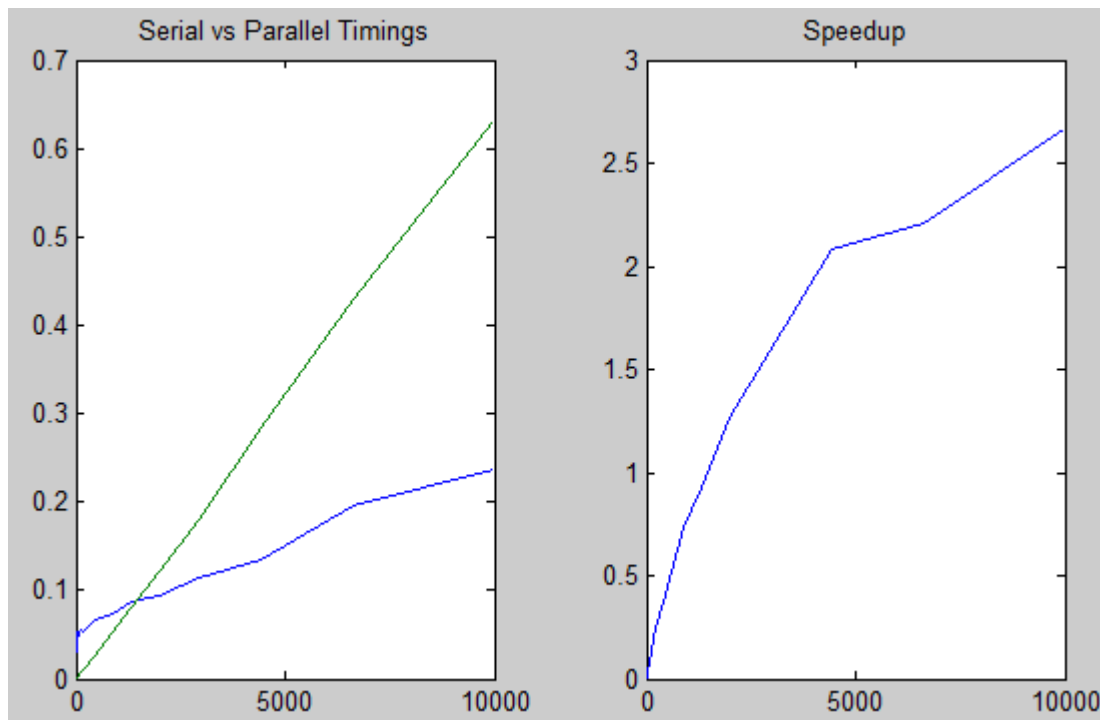
EDU>> main
Testing Parfor Performance
Testing with 1 iterations: 0.13x speedup (0.000 => 0.000 seconds)
Testing with 2 iterations: 0.31x speedup (0.000 => 0.000 seconds)
Testing with 4 iterations: 0.48x speedup (0.000 => 0.000 seconds)
Testing with 8 iterations: 0.58x speedup (0.000 => 0.001 seconds)
Testing with 13 iterations: 0.71x speedup (0.001 => 0.001 seconds)
Testing with 20 iterations: 0.75x speedup (0.001 => 0.002 seconds)
Testing with 32 iterations: 0.82x speedup (0.002 => 0.002 seconds)
Testing with 49 iterations: 0.94x speedup (0.003 => 0.004 seconds)
Testing with 74 iterations: 0.85x speedup (0.005 => 0.005 seconds)
Testing with 113 iterations: 0.81x speedup (0.007 => 0.009 seconds)
Testing with 170 iterations: 0.83x speedup (0.011 => 0.013 seconds)
Testing with 257 iterations: 0.82x speedup (0.016 => 0.019 seconds)
Testing with 387 iterations: 0.87x speedup (0.024 => 0.028 seconds)
Testing with 581 iterations: 0.86x speedup (0.036 => 0.042 seconds)
Testing with 873 iterations: 1.07x speedup (0.066 => 0.062 seconds)
Testing with 1311 iterations: 0.94x speedup (0.094 => 0.100 seconds)
Testing with 1968 iterations: 0.88x speedup (0.122 => 0.139 seconds)
Testing with 2953 iterations: 0.88x speedup (0.182 => 0.208 seconds)
Testing with 4431 iterations: 0.94x speedup (0.299 => 0.317 seconds)
Testing with 6648 iterations: 0.85x speedup (0.416 => 0.487 seconds)
Testing with 9973 iterations: 0.92x speedup (0.671 => 0.732 seconds)

```

Clearly, for small loops counts, there's some overhead to try to run a 'parfor', and for Matlab to realise this is not possible, so we get a slowdown (speedup < 1.0). But on a machine without the parallel computing toolbox, the overhead is not that significant, and we get to speedups of around 0.8 for problems of only a few milliseconds.

### Results on Parallel-Matlab

Running now on my newly installed Matlab-2010a, with parallel computing toolbox and on my quad-core computer (i.e. maximum theoretical speedup = 4), I get the following results:



We see that for problems above about 0.1 seconds big and above, the parallel execution (blue line) is below the serial implementation (green line). The graph on the right shows that we never reach the full expected speedup of 4.0.

In detail:

EDU>> main

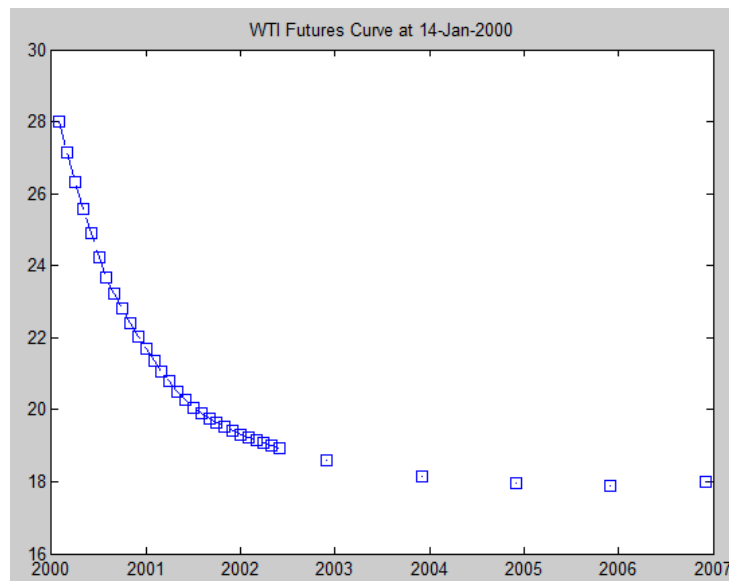
```
Testing Parfor Performance
Running on 4 workers
Testing with 1 iterations: 0.00x speedup (0.000 => 0.030 seconds)
Testing with 2 iterations: 0.00x speedup (0.000 => 0.030 seconds)
Testing with 4 iterations: 0.00x speedup (0.000 => 0.047 seconds)
Testing with 8 iterations: 0.01x speedup (0.000 => 0.048 seconds)
Testing with 13 iterations: 0.02x speedup (0.001 => 0.046 seconds)
Testing with 20 iterations: 0.03x speedup (0.001 => 0.050 seconds)
Testing with 32 iterations: 0.04x speedup (0.002 => 0.052 seconds)
Testing with 49 iterations: 0.06x speedup (0.003 => 0.048 seconds)
Testing with 74 iterations: 0.08x speedup (0.005 => 0.053 seconds)
Testing with 113 iterations: 0.13x speedup (0.007 => 0.055 seconds)
Testing with 170 iterations: 0.20x speedup (0.011 => 0.052 seconds)
Testing with 257 iterations: 0.28x speedup (0.016 => 0.057 seconds)
Testing with 387 iterations: 0.37x speedup (0.024 => 0.064 seconds)
Testing with 581 iterations: 0.52x speedup (0.036 => 0.068 seconds)
Testing with 873 iterations: 0.74x speedup (0.053 => 0.073 seconds)
Testing with 1311 iterations: 0.93x speedup (0.080 => 0.087 seconds)
Testing with 1968 iterations: 1.27x speedup (0.120 => 0.095 seconds)
Testing with 2953 iterations: 1.59x speedup (0.181 => 0.113 seconds)
Testing with 4431 iterations: 2.09x speedup (0.283 => 0.136 seconds)
Testing with 6648 iterations: 2.21x speedup (0.430 => 0.195 seconds)
Testing with 9973 iterations: 2.66x speedup (0.630 => 0.237 seconds)
```

We now see that running in parallel on small problem sizes is significantly lower. For small number of iterations, the parallel implementation is 50x or more slower, due to a significant 'constant' in the parallel timings, seen as the non-zero intercept of the blue line on the left-hand graph. Crossover (when parallel implementation begins to exceed serial) occurs at a problem size of around 0.1 seconds.

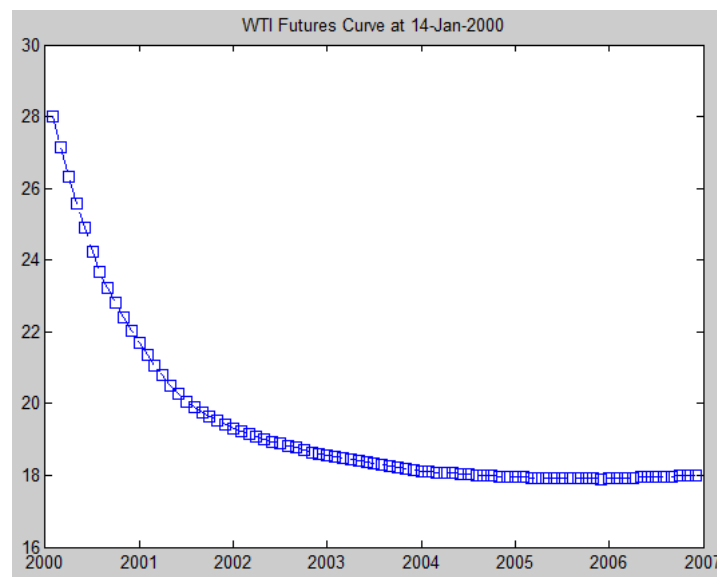
## Use in Futures Curve Interpolation

For many commodity futures curves, not all months are traded. Typically for the energy commodities, we can trade every single month up to maturities of about 2 years. After that, contracts are 'gappy'<sup>6</sup>. For agricultural commodities, the problem is worse, only some months are traded.<sup>7</sup>

There may be times when we wish to interpolate prices between known, actively traded months, to obtain a continuous futures curve. For example, the WTI crude oil futures curve at 14-January-2000 looked like



We can see gappy months beyond around 2.5 years. If we interpolate with a linear interpolation method along each curve, we get, for the 14-January-2000 above, the following:



<sup>6</sup> For example, see the WTI crude oil prices at <http://www.cmegroup.com/trading/energy/crude-oil/light-sweet-crude.html>

<sup>7</sup> For example, see wheat prices at <http://www.cmegroup.com/trading/commodities/grain-and-oilseed/wheat.html>

I implement below a simple Matlab algorithm to interpolate a square grid of futures prices, assuming time downwards and maturity acrosswards. I also assume a grid of dates (Matlab datenum format), being the maturity of each price in the corresponding position in the prices array.

For example, the first 4 trading days, and the first 6 maturities would be look like:

```
EDU>> display(prices(1:4,1:6))
ans =
    25.6         25         24.3         23.91         22.57         22
    25.55        24.84        24.13        23.43         22.79        22.23
    24.91        24.27        23.61        22.99         22.41        21.89
    24.78         24.2         23.54         22.92         22.34        21.82
```

The corresponding date array would look like

```
EDU>> display(dates(1:4,1:6))
ans =
    730517        730546        730577        730607        730638        730668
    730517        730546        730577        730607        730638        730668
    730517        730546        730577        730607        730638        730668
    730517        730546        730577        730607        730638        730668
```

The routine to perform interpolation on the grid of futures curves is:

```
function [ outprices, outdates ] = InterpolateFuturesCurves( inprices, indates )
%Interpolate but don't extrapolate an array of futures curves.
% Format should be m different futures curves x n periods
% The corresponding dates of the first-of-the-delivery month are assumed to
% be passed in 'indates', these will also be interpolated, assuming they
% have identical gaps to the prices.
curves = size(inprices,1);
periods = size(inprices,2);

% Preallocate result, same size as input. Unless we can interpolate
% further, we keep out=in.
outprices = inprices;
outdates = indates;
% We can do them all in parallel.
for i=1:curves
    % Only those rows with >= 2 points need to be (can be) interpolated.
    if (sum(~isnan(inprices(i,:))) > 1)
        % Only interpololote using the non-NaN points, or interpolate freaks out.
        % Where are the non-NaN's in this curve?
        [ dummy,c,dummy ] = find(~isnan(inprices(i,:)));

        % Pass the non-NaN positions and their values to the linear interpolator.
        % If we pass any NaN's into the interpolator, it complains.
        outprices(i,:) = interp1(c, inprices(i,c), 1:periods, 'linear');
        outdates(i,:) = interp1(c, indates (i,c), 1:periods, 'linear');
    end
end
end
```

In this case, we can also create a 2<sup>nd</sup> function `InterpolateFuturesCurvesParallel` where we trivially replace the 'for' loop with a 'parfor' loop.

We adjust the timing harness to be as follows:

```

function [ ] = main_interp( prices, dates )
%Run Yield Curve Interpolation
%This function is compatible with Matlab, even if it doesn't have the
%parallel toolbox loaded.

fprintf('Testing Parfor Yield Curve Interpolation Performance\n');

if (isfunction('matlabpool'))
    % Start labs if necessary.
    sz = matlabpool('size');
    if (sz ==0)
        matlabpool('open');
    end
    % Check we got some now.
    sz = matlabpool('size');
    if (sz ==0)
        error('Failed to open parallel workers');
    else
        fprintf('Running on %d workers\n', sz);
    end
end

curves = [ ] ;
i = 1;
while (i < size(prices,1))
    curves = [ curves floor(i) ];
    i = i * 1.5;
    i = i + 1;
end

partime = zeros(size(curves));
sertime = zeros(size(curves));
speedup = zeros(size(curves));
for i=1:length(curves)

    fprintf('Testing with %d curves: ', curves(i));

    % Construct part-arrays of only a few futures curves to interpolate.
    p = prices(1:curves(i),:);
    d = dates (1:curves(i),:);
    tic;
    [dummy, dummy] = InterpolateFuturesCurvesParallel(p, d);
    partime(i) = toc;

    tic;
    [dummy, dummy] = InterpolateFuturesCurves(p, d);
    sertime(i) = toc;

    speedup(i) = sertime(i) / partime(i);
    fprintf('%.2fx speedup (%.3f => %.3f seconds)\n', speedup(i), sertime(i), partime(i));
end

subplot(1,2,1);
plot(curves, partime, ...
     curves, sertime);
title('Serial vs Parallel Timings');

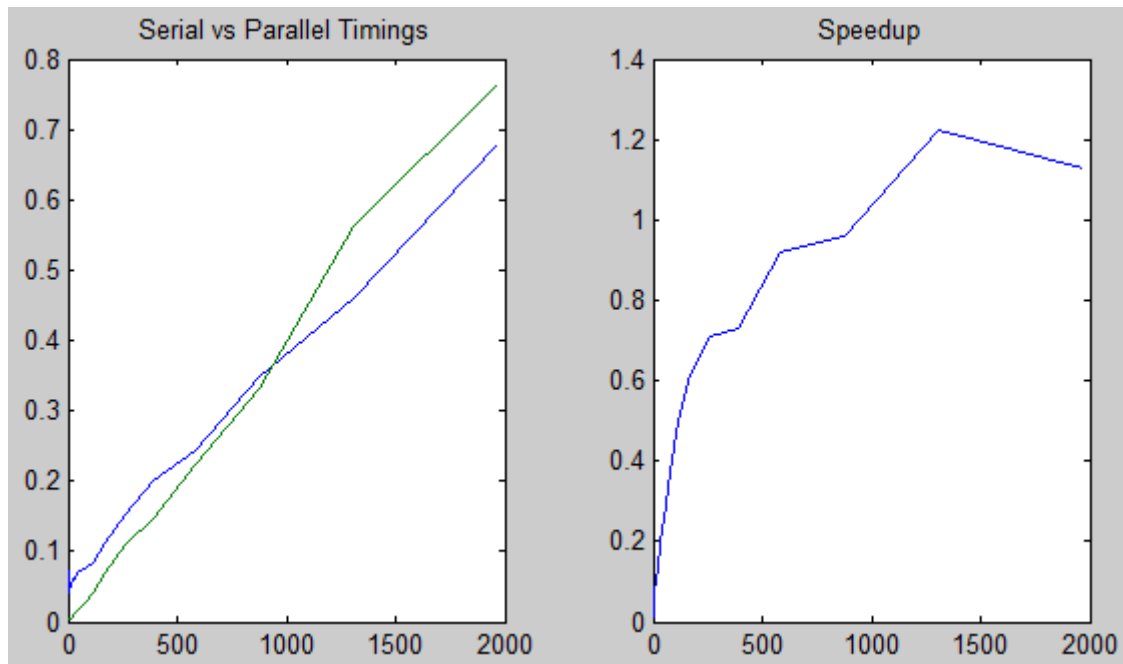
subplot(1,2,2);
plot(curves, speedup);
title('Speedup');

end

```

## Performance Results

We now get the following results, when we pass in the WTI futures curves from 2000-2009 (ten years), and call our interpretation routines on an increasing number of curves each time:



```

EDU>> main interp(dates, prices)
Testing Parfor Yield Curve Interpolation Performance
Running on 4 workers
Testing with 1 curves: 0.01x speedup (0.001 => 0.072 seconds)
Testing with 2 curves: 0.02x speedup (0.001 => 0.041 seconds)
Testing with 4 curves: 0.03x speedup (0.003 => 0.074 seconds)
Testing with 8 curves: 0.07x speedup (0.004 => 0.048 seconds)
Testing with 13 curves: 0.11x speedup (0.005 => 0.051 seconds)
Testing with 20 curves: 0.13x speedup (0.008 => 0.061 seconds)
Testing with 32 curves: 0.20x speedup (0.012 => 0.062 seconds)
Testing with 49 curves: 0.26x speedup (0.019 => 0.072 seconds)
Testing with 74 curves: 0.37x speedup (0.028 => 0.075 seconds)
Testing with 113 curves: 0.50x speedup (0.042 => 0.085 seconds)
Testing with 170 curves: 0.61x speedup (0.069 => 0.112 seconds)
Testing with 257 curves: 0.71x speedup (0.108 => 0.153 seconds)
Testing with 387 curves: 0.73x speedup (0.147 => 0.201 seconds)
Testing with 581 curves: 0.92x speedup (0.224 => 0.243 seconds)
Testing with 873 curves: 0.96x speedup (0.333 => 0.348 seconds)
Testing with 1311 curves: 1.22x speedup (0.563 => 0.460 seconds)
Testing with 1968 curves: 1.13x speedup (0.764 => 0.678 seconds)

```

In conclusion, we see little speedup, only about 1.2, and only above around 1000 curves (problem size of around 0.4 seconds) being interpolated at one time, and the risk of slowdown if we try to process amounts of curves in parallel.

## Performance Results after Recoding

Further reading of the Matlab parallel toolbox manual reveals a problem with the `inprices(i,c)` term in the interpolation algorithm. This is not a 'sliced' variable to Matlab, meaning `inprices()` cannot be efficiently split up, and only the relevant part sent to each worker. A recoding as below turns out to be more efficient for `InterpolateFuturesCurves`, with its brother process `InterpolateFuturesCurvesParallel` being identical, with 'parfor' instead of 'for'.

```

function [ outprices, outdates ] = InterpolateFuturesCurves( inprices, indates )
%Interpolate but don't extrapolate an array of futures curves.
% Format should be m different futures curves x n periods
% The corresponding dates of the first-of-the-delivery month are assumed to
% be passed in 'indates', these will also be interpolated, assuming they
% have identical gaps to the prices.
curves = size(inprices,1);

```

```

periods = size(inprices,2);

% Preallocate result, same size as input. Unless we can interpolate
% further, we keep out=in.
outprices = inprices;
outdates = indates;
% We can do them all in parallel.
for i=1:curves
    pricevec = inprices(i,:);
    datevec = indates(i,:);

    % Only those rows with >= 2 points need to be (can be) interpolated.
    if (sum(~isnan(pricevec)) > 1)
        % Only interpololate using the non-NaN points, or interpolate freaks out.
        % Where are the non-NaN's in this curve?
        [ dummy,c,dummy ] = find(~isnan(pricevec));

        % Pass the non-NaN positions and their values to the linear interpolator.
        % If we pass any NaN's into the interpolator, it complains.

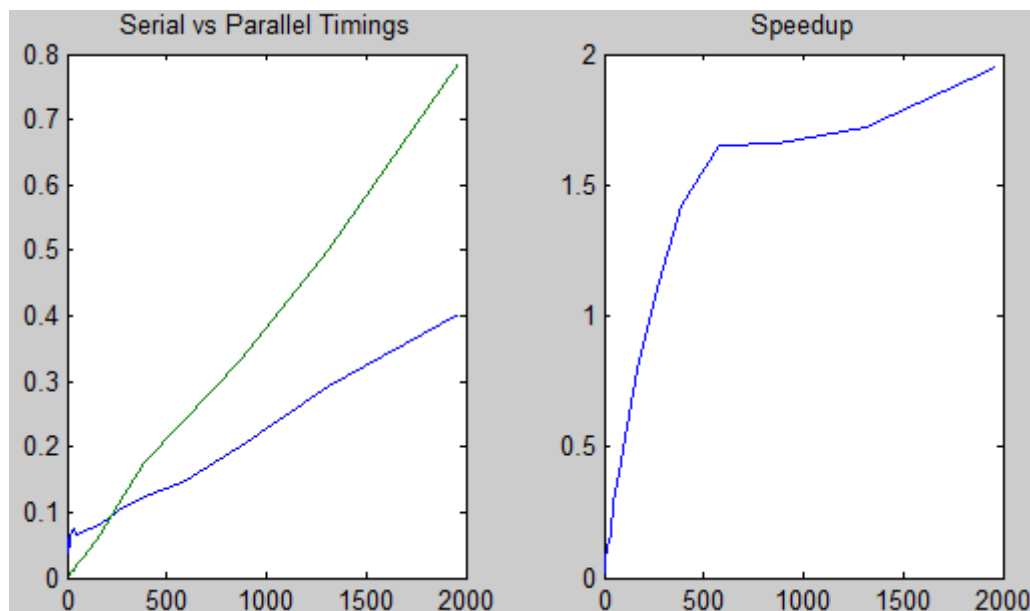
        outprices(i,:) = interp1(c, pricevec(c), 1:periods, 'linear');
        outdates(i,:) = interp1(c, datevec(c), 1:periods, 'linear');

    end
end
end

```

Note we now 'slice' up the input arrays at the start of the `for/parfor` loop, assign to temporary variables and perform the relevant processes on them.

Results after recoding are as follows:



```

EDU>> main interp(dates, prices)
Testing Parfor Yield Curve Interplation Performance
Running on 4 workers
Testing with 1 curves: 0.02x speedup (0.001 => 0.039 seconds)
Testing with 2 curves: 0.02x speedup (0.001 => 0.045 seconds)
Testing with 4 curves: 0.03x speedup (0.002 => 0.059 seconds)
Testing with 8 curves: 0.06x speedup (0.003 => 0.053 seconds)
Testing with 13 curves: 0.12x speedup (0.006 => 0.048 seconds)
Testing with 20 curves: 0.12x speedup (0.008 => 0.064 seconds)
Testing with 32 curves: 0.16x speedup (0.012 => 0.075 seconds)
Testing with 49 curves: 0.29x speedup (0.019 => 0.065 seconds)

```

```

Testing with 74 curves: 0.40x speedup (0.028 => 0.070 seconds)
Testing with 113 curves: 0.58x speedup (0.043 => 0.074 seconds)
Testing with 170 curves: 0.79x speedup (0.064 => 0.081 seconds)
Testing with 257 curves: 1.08x speedup (0.111 => 0.103 seconds)
Testing with 387 curves: 1.42x speedup (0.177 => 0.125 seconds)
Testing with 581 curves: 1.65x speedup (0.238 => 0.144 seconds)
Testing with 873 curves: 1.66x speedup (0.332 => 0.201 seconds)
Testing with 1311 curves: 1.72x speedup (0.502 => 0.293 seconds)
Testing with 1968 curves: 1.95x speedup (0.786 => 0.404 seconds)

```

We have now achieved a speedup of up to 2x if we need to process 10 years (2500 days) of futures curves at one time. The crossover is now at about 200 curves, or just under 1 years' worth of data. Notably, the crossover is again around the 0.1 seconds mark.

## Optimum, Intelligent Algorithm

A nice feature of the 'parfor' syntax lets us choose how many labs to run on. For example,

```

parfor (i=1:1:1000 , labcount)
    ...
end

```

lets us choose how many labs to use. If we specify 0 (*not 1*), the code runs serially on the local Matlab. If we specify more labs than are actually running, parfor will silently just use all that are available. We can thus set up intelligent libraries that switch between parallel and serial mode depending on the problem size. This syntax also works at least as far back at Matlab-2008b.

If we now recode `InterpolateFuturesCurvesOptimal` as follows, to only run in parallel (choosing 8 labs maximum in readiness for the time I get an 8-core desktop):

```

function [ outprices, outdates ] = InterpolateFuturesCurvesOptimal( inprices, indates )
%Interpolate but don't extrapolate an array of futures curves.
% Format should be m different futures curves x n periods
% The corresponding dates of the first-of-the-delivery month are assumed to
% be passed in 'indates', these will also be interpolated, assuming they
% have identical gaps to the prices.
curves = size(inprices,1);
periods = size(inprices,2);

% Preallocate result, same size as input. Unless we can interpolate
% further, we keep out=in.
outprices = inprices;
outdates = indates;

% Run in parallel only if curves > 200
optimumlabs = 0 + 8*(curves > 200);

parfor (i=1:curves , optimumlabs)

    inp = inprices(i,:);
    ind = indates(i,:);

    % Only those rows with >= 2 points need to be (can be) interpolated.
    if (sum(~isnan(inp)) > 1)
        % Only interpololote using the non-NaN points, or interpolate freaks out.
        % Where are the non-NaN's in this curve?
        [ dummy,c,dummy ] = find(~isnan(inp));

```

```

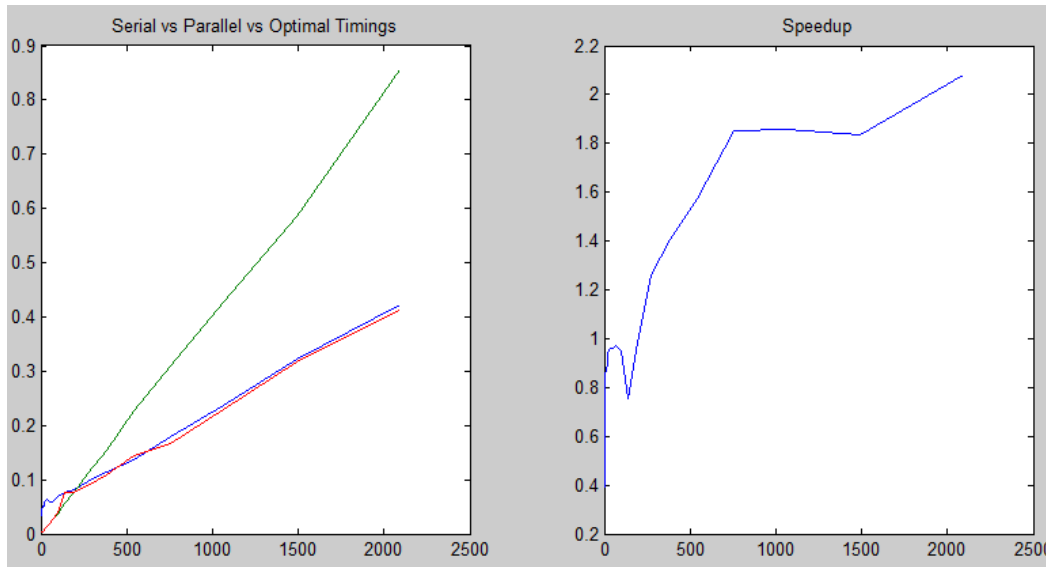
% Pass the non-NaN positions and their values to the linear interpolator.
% If we pass any NaN's into the interpolator, it complains.

outprices(i,:) = interp1(c, inp(c), 1:periods, 'linear');
outdates(i,:) = interp1(c, ind(c), 1:periods, 'linear');

end
end
end

```

we now get the following performance curve (note in particular the red line):



The red line above is the new execution time (Optimal algorithm), and the speedup compares Optimal with Serial. Note we now get the best of both worlds, the algorithm executes in serial time (green) for small problem sizes, and in parallel-time (blue) for large problem sizes. The 'kink' in the red line around the crossover point shows that the heuristic crossover point, coded as 200 curves, is a little too low.

## Conclusion, Discussion, Recommendations

- The Parallel Computing Toolbox in Matlab is no silver bullet for Matlab parallelization.
- For small problem sizes, naïve use of the 'parfor' parallelization technique with multiple workers adds significant overhead and can introduce *slowdowns* of 1, 2 or more orders of magnitude.
- Crossover, when speedup is experienced, is only when small problems reach the order of 100's or 1000's of iterations. A reasonable heuristic seems to be, simple tasks taking a total of above 0.1 seconds may experience speedup from running in parallel.

- Minimal recoding of software to 'help' Matlab parallelize the task efficiently can be beneficial.
- Although Matlab software written to make use of the parallel computing toolbox is not compatible with Matlab instances where the toolbox is unavailable, it is possible to engineer software to examine its environment and 'do the right thing'.
- Paradoxically, where parallel computing toolbox is *not* available, code written with the 'parfor' syntax runs at almost the full speed of the equivalent 'for' syntax.
- Alternatives to the 'parfor' parallelization technique exist, such as 'co-distributed arrays' or use of the 'batch' construct to 'fork' parallel tasks. However, these may require greater recoding of the problem, and reduce the likelihood of creating backwards compatible, redistributable code.
- Parallelization should be implemented at the higher levels in the program, not in very short-lived library routines.
- Library routines can examine the size of the problem they are given, and choose whether to run in serial (locally), or in parallel, on a pool of labs, via an extension to the 'parfor' syntax. With a well-configured crossover point (configured empirically for each algorithm), this gives the best of both worlds.