Parallel MATLAB: Parallel For Loops

John Burkardt (FSU)
Gene Cliff (AOE/ICAM - ecliff@vt.edu)
Justin Krometis (ARC/ICAM - jkrometis@vt.edu)
3pm - 5pm, Tuesday, 3 April 2012
............. FDI .............

**FSU**: Florida State University
**AOE**: Department of Aerospace and Ocean Engineering
**ARC**: Advanced Research Computing
**ICAM**: Interdisciplinary Center for Applied Mathematics
Introduction
QUAD Example
Executing a PARFOR Program
Classification of variables
MD Example
PRIME Example
ODE_SWEEP Example
FMINCON Example
Conclusion
Parallel MATLAB is an extension of MATLAB that takes advantage of multicore desktop machines and clusters.

The *Parallel Computing Toolbox* or **PCT** runs on a desktop, and can take advantage of up to 8 cores there (a license limit). Parallel programs can be run interactively or in batch.

The **MATLAB Distributed Computing Server (MDCS)** controls parallel execution of MATLAB on a cluster with tens or hundreds of cores.

Virginia Tech’s Ithaca cluster allows supports a **MATLAB** queue that runs on 12 nodes. Each node has two, quad-core CPUs for a total 96 cores (a license limit).
INTRO: Local and Remote MATLAB Workers

- Desktop System
  - Parallel Computing Toolbox
    - Local Workers
  - Simulink, Blocksets, and Other Toolboxes
  - MATLAB

- Computer Cluster
  - MATLAB Distributed Computing Server
    - Workers
  - Scheduler

Diagram shows a connection between the Desktop System and the Computer Cluster, indicating how local and remote MATLAB workers communicate with the distributed computing server.
Three ways to write a parallel MATLAB program:

- suitable `for` loops can be made into `parfor` loops;
- the `spmd` statement can define cooperating synchronized processing;
- the `task` feature creates multiple independent programs.

The `parfor` approach is a limited but simple way to get started. `spmd` is powerful, but requires rethinking the program and data. The `task` approach is simple, but suitable only for computations that need almost no communication.
There are several ways to execute a parallel MATLAB program:
- interactive local (matlabpool), suitable for the desktop;
- indirect local, (batch or createJob/createTask);
- indirect remote, (batch or createJob/createTask), requires setup.

The Virginia Tech cluster Ithaca will accept parallel MATLAB jobs submitted from a user’s desktop, and will return the results when the job is completed.

Making this possible requires an Ithaca account and a one-time setup of the user’s machine (simplified since R2010b).
Today’s Lecture: PARFOR

The simplest path to parallelism is the `parfor` statement, which indicates that a given `for` loop can be executed in parallel.

When the “client” MATLAB reaches such a loop, the iterations of the loop are automatically divided up among the workers, and the results gathered back onto the client.

Using `parfor` requires that the iterations are completely independent; there are also some restrictions on data access.

OpenMP implements a directive for 'parallel for loops'
Lecture #2: SPMD

MATLAB can also work in a simplified kind of MPI model.
There is always a special “client” process.
Each worker process has its own memory and separate ID.
There is a single program, but it is divided into client and worker sections; the latter marked by special `spmd/end` statements.
Workers can “see” the client’s data; the client can access and change worker data.
The workers can also send messages to other workers.
OpenMP includes constructs similar to `spmd`.
SPMD programming includes distributed arrays.

A distributed array is logically one array, and a large set of MATLAB commands can treat it that way.

However, portions of the array are scattered across multiple processors. This means such an array can be really large.

The local part of a distributed array can be operated on by that processor very quickly.

A distributed array can be operated on by explicit commands to the SPMD workers that “own” pieces of the array, or implicitly by commands at the global or client level.
Parallel MATLAB jobs can be run directly, that is, interactively.

The `matlabpool` command is used to reserve a given number of workers on the local (or perhaps remote) machine.

Once these workers are available, the user can type commands, run scripts, or evaluate functions, which contain `parfor` statements. The workers will cooperate in producing results.

Interactive parallel execution is great for desktop debugging of short jobs.

It’s an inefficient way to work on a cluster, because no one else can use the workers until you release them!

So...don’t use the MATLAB queue on Ithaca, from your desktop machine or from an interactive session on an Ithaca login node! In our examples, we will indeed use Ithaca, but always through the indirect batch system.
Parallel PARFOR MATLAB jobs can be run indirectly.

The **batch** command is used to specify a MATLAB code to be executed, to indicate any files that will be needed, and how many workers are requested.

The **batch** command starts the computation in the background. The user can work on other things, and collect the results when the job is completed.

The **batch** command works on the desktop, and can be set up to access the Ithaca cluster.
Virginia Tech has installed the Ithaca cluster of 84 nodes. Each node has 2 quadcore processors and (at least) 24 GB of memory. This means each node can run 8 MATLAB processes.

12 nodes with 8 cores are dedicated to the parallel MATLAB cluster, so theoretically you can run a job with 96 workers.

You should not routinely ask for all 96 workers. If one job ties up all the workers, no one else can run. So we encourage the use of (no more than) 24 or 32 workers at a time instead.
MATLAB Parallel Computing

- Introduction
- QUAD Example
- Executing a PARFOR Program
- Classification of variables
- MD Example
- PRIME Example
- ODE_SWEEP Example
- FMINCON Example
- Conclusion
QUAD: Estimating an Integral

Approximating an integral

Graph showing the estimation of an integral.
function q = quad_fun( n, a, b )

    q=0.0;
    w=(b−a)/n;
    for i =1:n
        x=((n−i)*a+(i−1)*b)/(n−1);
        fx=bessely( 4.5 , x );
        q=q+w*fx ;
    end

    return
end
The function `quad_fun` estimates the integral of a particular function over the interval $[a, b]$.

It does this by evaluating the function at $n$ evenly spaced points, multiplying each value by the weight $(b - a)/n$.

These quantities can be regarded as the areas of little rectangles that lie under the curve, and their sum is an estimate for the total area under the curve from $a$ to $b$.

We could compute these subareas **in any order we want**.

We could even compute the subareas **at the same time**, assuming there is some method to save the partial results and add them together in an organized way.
function q = quad_fun( n, a, b )

    q = 0.0;
    w = (b - a) / n;
    parfor i = 1:n
        x = ((n - i) * a + (i - 1) * b) / (n - 1);
        fx = bessely(4.5, x);
        q = q + w * fx;
    end

    return
end
The parallel version of `quad_fun` does the same calculations.

The `parfor` statement changes how this program does the calculations. It asserts that all the iterations of the loop are independent, and can be done in any order, or in parallel.

Execution begins with a single processor, the client. When a `parfor` loop is encountered, the client is helped by a “pool” of workers.

Each worker is assigned some iterations of the loop. Once the loop is completed, the client resumes control of the execution.

MATLAB ensures that the results are the same whether the program is executed sequentially, or with the help of workers.

The user can wait until execution time to specify how many workers are actually available.
MATLAB Parallel Computing

- Introduction
- QUAD Example
- Executing a PARFOR Program
- Classification of variables
- MD Example
- PRIME Example
- ODE_SWEEP Example
- FMINCON Example
- Conclusion
EXECUTION: What Do You Need?

1. Your machine should have multiple processors or cores:
   - On a PC: Start :: Settings :: Control Panel :: System
   - On a Mac: Apple Menu :: About this Mac :: More Info...

2. Your MATLAB must be version 2009a or later:
   - Go to the HELP menu, and choose About Matlab.

3. You must have the Parallel Computing Toolbox:
   - At VT, the concurrent license MATLAB includes the PCT;
   - To list all your toolboxes, type the MATLAB command `ver`.
   - When using an MDCS (server) be sure to use the same version of MATLAB on your client machine.
Workers are gathered using the `matlabpool` command.

Interactively, we call `matlabpool` and then our function:

```matlab
matlabpool open local 4
(or)
matlabpool ('open', 'local', 4)
q = quad_fun ( n, a, b );
```

The `batch` command runs a script, with a `matlabpool` argument:

```matlab
job = batch ('quad_script', 'matlabpool', 4)
(or)
job = batch ('quad_script', 'matlabpool', 4, 'configuration', 'local')
```
To run *quad_fun.m* in parallel on your desktop, type:

```matlab
n = 10000; a = 0.5; b = 1;
matlabpool open local 4
q = quad_fun( n, a, b );
matlabpool close
```

The word **local** is choosing the local configuration, that is, the cores assigned to be workers will be on the local machine.

The value "4" is the number of workers you are asking for. It can be up to 8 on a local machine. It does not have to match the number of cores you have.
The batch command, for indirect execution, accepts scripts (and since R2010b functions). We can make a suitable script called `quad_script.m`:

```matlab
n = 10000; a = 0.5; b = 1;
q = quad_fun( n, a, b )
```

Now we assemble the job information needed to run the script and submit the job:

```matlab
job = batch( 'quad_script', 'matlabpool', 4, ... 'Configuration', 'local', ... 'FileDependencies', { 'quad_fun' } )
```

The following commands wait for the job to finish, gather the results, and clear out the job information:

```matlab
wait( job ); % no prompt until the job is finished
load( job ); % load data from the job’s Workspace
destroy( job ); % clean up
The batch command can send your job *anywhere*, and get the results back, as long as you have set up an account on the remote machine, and you have defined a *configuration* on your desktop that tells it how to access the remote machine.

At Virginia Tech, if your Ithaca account has been set up properly, your desktop can send a batch job there as easily as running locally:

```matlab
job = batch ( 'quad_script', 'matlabpool', 4, ...
    'Configuration', 'ithaca_2011b', ...
    'FileDependencies', { 'quad_fun' } )
```

The job is submitted. You may wait for it, load it and destroy it, all in the same way as for a local batch job.
MATLAB Parallel Computing

- Introduction
- QUAD Example
- Executing a PARFOR Program
- Classification of variables
- MD Example
- PRIME Example
- ODE_SWEEP Example
- FMINCON Example
- Conclusion
### Classification Description

<table>
<thead>
<tr>
<th>Classification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop</td>
<td>Serves as a loop index for arrays</td>
</tr>
<tr>
<td>Sliced</td>
<td>An array whose segments are operated on by different iterations of the loop</td>
</tr>
<tr>
<td>Broadcast</td>
<td>A variable defined before the loop whose value is used inside the loop, but never assigned inside the loop</td>
</tr>
<tr>
<td>Reduction</td>
<td>Accumulates a value across iterations of the loop, regardless of iteration order</td>
</tr>
<tr>
<td>Temporary</td>
<td>Variable created inside the loop, but unlike sliced or reduction variables, not available outside the loop</td>
</tr>
</tbody>
</table>
CLASSIFICATION: an example

```
a = 0;
c = pi;
z = 0;
r = rand(1,10);
parfor i = 1:10
    a = i;
z = z+i;
b(i) = r(i);
    if i <= c
        d = 2*a;
    end
end
```

NB: "The range of a parfor statement must be increasing consecutive integers"

Trick Ques.: What values to \( a \) and \( d \) have after exiting the loop?
Sliced variables:

Characteristics of a Sliced Variable. A variable in a `parfor`-loop is sliced if it has all of the following characteristics. A description of each characteristic follows the list:

- **Type of First-Level Indexing** — The first level of indexing is either parentheses, ( ), or braces, { }.
- **Fixed Index Listing** — Within the first-level parenthesis or braces, the list of indices is the same for all occurrences of a given variable.
- **Form of Indexing** — Within the list of indices for the variable, exactly one index involves the loop variable.
- **Shape of Array** — In assigning to a sliced variable, the right-hand side of the assignment is not [ ] or ' ' (these operators indicate deletion of elements).
• Introduction
• QUAD Example
• Executing a PARFOR Program
• Classification of variables
• MD Example
• PRIME Example
• ODE_SWEEP Example
• FMINCON Example
• Conclusion
Compute the positions and velocities of $N$ particles at a sequence of times. The particles exert a weak attractive force on each other.
The MD program runs a simple molecular dynamics simulation.

There are \( N \) molecules being simulated.

The program runs a long time; a parallel version would run faster.

There are many \texttt{for} loops in the program that we might replace by \texttt{parfor}, but it is a mistake to try to parallelize everything!

MATLAB has a \texttt{profile} command that can report where the CPU time was spent - which is where we should try to parallelize.
>> profile on
>> md
>> profile viewer

<table>
<thead>
<tr>
<th>Step</th>
<th>Potential Energy</th>
<th>Kinetic Energy</th>
<th>(P+K-E0)/E0 Energy Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>498108.113974</td>
<td>0.000000</td>
<td>0.000000e+00</td>
</tr>
<tr>
<td>2</td>
<td>498108.113974</td>
<td>0.000009</td>
<td>1.794265e-11</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>498108.111972</td>
<td>0.002011</td>
<td>1.794078e-11</td>
</tr>
<tr>
<td>10</td>
<td>498108.111400</td>
<td>0.002583</td>
<td>1.793996e-11</td>
</tr>
</tbody>
</table>

CPU time = 415.740000 seconds.
Wall time = 378.828021 seconds.
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Calls</th>
<th>Total Time</th>
<th>Self Time*</th>
<th>Total Time Plot</th>
</tr>
</thead>
<tbody>
<tr>
<td>md</td>
<td>1</td>
<td>415.847 s</td>
<td>0.096 s</td>
<td></td>
</tr>
<tr>
<td>compute</td>
<td>11</td>
<td>415.459 s</td>
<td>410.703 s</td>
<td></td>
</tr>
<tr>
<td>repmat</td>
<td>11000</td>
<td>4.755 s</td>
<td>4.755 s</td>
<td></td>
</tr>
<tr>
<td>timestamp</td>
<td>2</td>
<td>0.267 s</td>
<td>0.108 s</td>
<td></td>
</tr>
<tr>
<td>datestr</td>
<td>2</td>
<td>0.130 s</td>
<td>0.040 s</td>
<td></td>
</tr>
<tr>
<td>timefun/private/formatdate</td>
<td>2</td>
<td>0.084 s</td>
<td>0.084 s</td>
<td></td>
</tr>
<tr>
<td>update</td>
<td>10</td>
<td>0.019 s</td>
<td>0.019 s</td>
<td></td>
</tr>
<tr>
<td>datevec</td>
<td>2</td>
<td>0.017 s</td>
<td>0.017 s</td>
<td></td>
</tr>
<tr>
<td>now</td>
<td>2</td>
<td>0.013 s</td>
<td>0.001 s</td>
<td></td>
</tr>
<tr>
<td>datenum</td>
<td>4</td>
<td>0.012 s</td>
<td>0.012 s</td>
<td></td>
</tr>
<tr>
<td>datestr&gt;getdateform</td>
<td>2</td>
<td>0.005 s</td>
<td>0.005 s</td>
<td></td>
</tr>
<tr>
<td>initialize</td>
<td>1</td>
<td>0.005 s</td>
<td>0.005 s</td>
<td></td>
</tr>
<tr>
<td>etime</td>
<td>2</td>
<td>0.002 s</td>
<td>0.002 s</td>
<td></td>
</tr>
</tbody>
</table>

**Self time** is the time spent in a function excluding the time spent in its child functions. Self time also includes overhead resulting from the process of profiling.
function [ f, pot, kin ] = compute ( np, nd, pos, vel, mass )

f = zeros ( nd, np );
pot = 0.0;

for i = 1 : np
    for j = 1 : np
        if ( i ~= j )
            r_j = pos(1:nd,i) - pos(1:nd,j);
            d = sqrt ( sum ( r_j(1:nd).^2 ) );
            d2 = min ( d, pi / 2.0 );
            pot = pot + 0.5 * sin ( d2 ) * sin ( d2 );
            f(1:nd,i) = f(1:nd,i) - r_j(1:nd) * sin ( 2.0 * d2 ) / d;
        end
    end
end

kin = 0.5 * mass * sum ( vel(1:nd,1:np).^2 );

return
The **compute** function fills the force vector \( f(i) \) using a **for** loop.

Iteration \( i \) computes the force on particle \( i \), determining the distance to each particle \( j \), squaring, truncating, taking the sine.

The computation for each particle is “**independent**”; nothing computed in one iteration is needed by, nor affects, the computation in another iteration. We could compute each value on a separate worker, at the same time.

The MATLAB command **parfor** will distribute the iterations of this loop across the available workers.

Tricky question: Could we parallelize the \( j \) loop instead?

Tricky question: Could we parallelize **both** loops?
Replacing “for i” by “parfor i”, here is our speedup:
Parallel execution gives a huge improvement in this example.

There is some overhead in starting up the parallel process, and in transferring data to and from the workers each time a `parfor` loop is encountered. So we should not simply try to replace every `for` loop with `parfor`.

That’s why we first searched for the function that was using most of the execution time.

The `parfor` command is the simplest way to make a parallel program, but in other lectures we will see some alternatives.
We were only able to parallelize the loop because the iterations were independent, that is, the results did not depend on the order in which the iterations were carried out.

In fact, to use MATLAB’s `parfor` in this case requires some extra conditions, which are discussed in the PCT User’s Guide. Briefly, `parfor` is usable when vectors and arrays that are modified in the calculation can be divided up into distinct slices, so that each slice is only needed for one iteration.

This is a stronger requirement than independence of order!

Trick question: Why was the scalar value `POT` acceptable?
MATLAB Parallel Computing

- Introduction
- QUAD Example
- Executing a PARFOR Program
- Classification of variables
- MD Example
- PRIME Example
- ODE_SWEEP Example
- FMINCON Example
- Conclusion
For our next example, we want a simple computation involving a loop which we can set up to run for a long time.

We’ll choose a program that determines how many prime numbers there are between 1 and $N$.

If we want the program to run longer, we increase the variable $N$. Doubling $N$ multiplies the run time roughly by 4.
PRIME: The Sieve of Eratosthenes
function total = prime(n)

% PRIME returns the number of primes between 1 and N.

    total = 0;
    for i = 2 : n
        prime = 1;
        for j = 2 : i - 1
            if (mod(i,j) == 0)
                prime = 0;
            end
        end
        total = total + prime;
    end

    return
end
We can parallelize the loop whose index is $i$, replacing `for` by `parfor`. The computations for different values of $i$ are independent.

There is one variable that is not independent of the loops, namely `total`. This is simply computing a running sum (a reduction variable), and we only care about the final result. MATLAB is smart enough to be able to handle this summation in parallel.

To make the program parallel, we replace `for` by `parfor`. That’s all!
matlabpool ( 'open', 'local', 4) % function form

n=50;

while ( n <= 500000 )
    primes = prime_number_parfor ( n );
    fprintf ( 1, '__%8d__%8d
', n, primes );
    n = n * 10;
end

matlabpool ( 'close' )
**PRIME: Timing**

**PRIME_PARFOR_RUN**

Run PRIME_PARFOR with 0, 1, 2, and 4 labs.

<table>
<thead>
<tr>
<th>N</th>
<th>1+0</th>
<th>1+1</th>
<th>1+2</th>
<th>1+4</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>0.067</td>
<td>0.179</td>
<td>0.176</td>
<td>0.278</td>
</tr>
<tr>
<td>500</td>
<td>0.008</td>
<td>0.023</td>
<td>0.027</td>
<td>0.032</td>
</tr>
<tr>
<td>5000</td>
<td>0.100</td>
<td>0.142</td>
<td>0.097</td>
<td>0.061</td>
</tr>
<tr>
<td>50000</td>
<td>7.694</td>
<td>9.811</td>
<td>5.351</td>
<td>2.719</td>
</tr>
<tr>
<td>500000</td>
<td>609.764</td>
<td>826.534</td>
<td>432.233</td>
<td>222.284</td>
</tr>
</tbody>
</table>
There are many thoughts that come to mind from these results!

Why does 500 take less time than 50? (It doesn’t, really).

How can ”1+1” take longer than ”1+0”? (It does, but it’s probably not as bad as it looks!)

This data suggests two conclusions:

*Parallelism doesn’t pay until your problem is big enough;*

*AND*

*Parallelism doesn’t pay until you have a decent number of workers.*
MATLAB Parallel Computing

- Introduction
- QUAD Example
- Executing a PARFOR Program
- Classification of variables
- MD Example
- PRIME Example
- ODE_SWEEP Example
- FMINCON Example
- Classification of variables
- Conclusion
ODE: A Parameterized Problem

Consider a favorite ordinary differential equation, which describes the motion of a spring-mass system:

\[ m \frac{d^2x}{dt^2} + b \frac{dx}{dt} + kx = f(t) \]

![Diagram of a spring-mass system]
Solutions of this equation describe oscillatory behavior; $x(t)$ swings back and forth, in a pattern determined by the parameters $m, b, k, f$ and the initial conditions.

Each choice of parameters defines a solution, and let us suppose that the quantity of interest is the maximum deflection $x_{\text{max}}$ that occurs for each solution.

We may wish to investigate the influence of $b$ and $k$ on this quantity, leaving $m$ fixed and $f$ zero.

So our computation might involve creating a plot of $x_{\text{max}}(b, k)$. 
ODE: Each Solution has a Maximum Value

\( m = 5, \ b = 2, \ k = 2 \)

\( m = 5, \ b = 5, \ k = 5 \)
Evaluating the implicit function $x_{\text{max}}(b, k)$ requires selecting a pair of values for the parameters $b$ and $k$, solving the ODE over a fixed time range, and determining the maximum value of $x$ that is observed. Each point in our graph will cost us a significant amount of work.

On the other hand, it is clear that each evaluation is completely independent, and can be carried out in parallel. Moreover, if we use a few shortcuts in MATLAB, the whole operation becomes quite straightforward!
$m = 5.0;$
$bVals = 0.1 : 0.05 : 5;$
$kVals = 1.5 : 0.05 : 5;$

$[kGrid, bGrid] = \text{meshgrid}(bVals, kVals);$;

$\text{peakVals} = \text{nan}(\text{size}(kGrid));$

$\text{tic;}$

$\text{parfor } ij = 1 : \text{numel}(kGrid)$

$\quad [T, Y] = \text{ode45}(@(t,y) \text{ode\_system}(t, y, m, bGrid(ij), kGrid(ij)), ... [0, 25], [0, 1]);$

$\quad \text{peakVals}(ij) = \text{max}(Y(:,1));$

$\text{end}$

$\text{toc;}$
matlabpool open local 4
ode_sweep_parfor
matlabpool close
ode_sweep_display

----------

job = batch ( ... 
    'ode_sweep_script', ... 
    'Configuration', 'local', ... 
    'FileDependencies', {'ode_system.m'}, ... 
    'matlabpool', 4 );
wait ( job );
load ( job );
ode_sweep_display
destroy ( job )
% Display the results.

figure;

surf (bVals, kVals, peakVals, 'EdgeColor', 'Interp', 'FaceColor', 'Interp');

title ('Results of ODE Parameter Sweep');
xlabel ('Damping B');
ylabel ('Stiffness K');
zlabel ('Peak Displacement');
view (50, 30)
ODE: A Parameterized Problem

Results of ODE Parameter Sweep

Peak Displacement

Damping B

Stiffness K
In the MD program, the `parfor` loop was only a part of the calculation; other parts of the calculation had to run in order, and the loop itself was called several times, but each time the input depended on previous computations.

In the ODE parameter sweep, we have several thousand ODE’s to solve, but we could solve them in any order, on various computers, or any way we wanted to. All that was important was that when the computations were completed, every value $x_{max}(b, x)$ had been computed.

This kind of loosely-coupled problem can be treated as a task computing problem, and we will see later on how MATLAB can treat this problem as a collection of many little tasks to be computed in an arbitrary fashion and assembled at the end.
Introduction

QUAD Example

Executing a PARFOR Program

Classification of variables

MD Example

PRIME Example

ODE_SSWEP Example

FMINCON Example

Conclusion
**FMINCON** is a popular MATLAB function available in the Optimization Toolbox. It finds the minimizer of a function of several variables with constraints:

\[
\text{min } F(X) \quad \text{subject to:}
\]

\[
\begin{align*}
A\cdot X & \leq B, \\
A_{eq}\cdot X & = B_{eq} \quad \text{(linear constraints)} \\
C(X) & \leq 0, \\
C_{eq}(X) & = 0 \quad \text{(nonlinear constraints)} \\
LB & \leq X \leq UB \quad \text{(bounds)}
\end{align*}
\]

If no derivative or Hessian information is supplied by the user, then **FMINCON** uses finite differences to estimate these quantities. If \textit{fun} is expensive to evaluate, the finite differencing can dominate the execution.
An example using FMINCON involves a boat trying to cross a river against a current. The boat is given 10 minutes to make the crossing, and must try to land as far as possible upstream. In this unusual river, the current is zero midstream, negative above the x axis, and positive (helpful) below the x axis!
The correct solution takes maximum advantage of the favorable current, and then steers back hard to the land on the line $y = 1$. 
FMINCON uses an options structure that contains default settings. The user can modify these by calling the procedure optimset. The finite differencing process can be done in parallel if the user sets the appropriate option:

```matlab
options = optimset( optimset( 'fmincon' ), ...
    'LargeScale','off', ...
    'Algorithm', 'active-set', ...
    'Display', 'iter', ...
    'UseParallel', 'Always' );
```

```matlab
[ x_star, f_star, exit ] = fmincon ( h_cost, z0, ... ...
    [], [], [], [], LB, UB, h_cnst, options );
```

...and uses the matlabpool command to make workers available!
Introduction
QUAD Example
Executing a PARFOR Program
Classification of variables
MD Example
PRIME Example
ODE_SWEEP Example
FMINCON Example
Conclusion
CONCLUSION: Summary of Examples

The QUAD example showed the simplest use of parfor. We will see new versions of this example again when we talk about spmd and task programming.

In the MD example, we did a profile first to identify where the work was.

By timing the PRIME example, we saw that it is inefficient to work on small problems, or with only a few processors.

In the ODE_SWEEP example, the loop we modified was not a small internal loop, but a big “outer” loop that defined the whole calculation.

In the FMINCON example, all we had to do to take advantage of parallelism was set an option (and then make sure some workers were available).
We only briefly mentioned the limitations of the `parfor` statement.

You can look in the User’s Guide for some more information on when you are allowed to turn a `for` loop into a `parfor` loop. It’s not as simple as just knowing that the loop iterations are independent. MATLAB has concerns about data usage as well.

MATLAB’s built in program editor (`mlint`) knows all about the rules for using `parfor`. You can experiment by changing a `for` to `parfor`, and the editor will immediately complain to you if there is a reason that MATLAB will not accept a `parfor` version of the loop.
Virginia Tech has a limited number of concurrent MATLAB licenses, and including the Parallel Computing Toolbox.

Since Fall 2011, the PCT is included with the student license.

Run `ver` in the Matlab Command Window to see what licenses you have available.

If you don’t have a multicore machine, you won’t see any speedup, but you may still be able to run some ‘parallel’ programs.
CONCLUSION: Desktop-to-Ithaca Submission

- If you want to work with parallel MATLAB on Ithaca, you must first get an account, by going to this website:
  
  http://www.arc.vt.edu/forms/account_request.php

- For remote submission you must define appropriate configuration file(s) on your desktop:
  
  http://www.arc.vt.edu/resources/software/matlab/remotesub.php

- For information on Passwordless login:
  
  http://www.arc.vt.edu/resources/software/matlab/passwordless.php
There is a local LISTSERV for people interested in MATLAB on the Virginia Tech campus. We try not to post messages here unless we really consider them of importance!

Important messages include information about workshops, special MATLAB events, and other issues affecting MATLAB users.

To subscribe to the mathworks listserver, send email to:

listserv@listserv.vt.edu.

The body of the message should simply be:

subscribe mathworks firstname lastname
CONCLUSION: Where is it?

- MATLAB Parallel Computing Toolbox Product Documentation
  http://www.mathworks.com/help/toolbox/distcomp/

- Gaurav Sharma, Jos Martin,
  *MATLAB: A Language for Parallel Computing*,
  International Journal of Parallel Programming,
  Volume 37, Number 1, pages 3-36, February 2009.

- An Adobe PDF with these notes, along with a zipped-folder containing the MATLAB codes can be downloaded from the ICAM website: http://www.icam.vt.edu/
  (click on the Computing link)
or from the VT Scholar-site
  https://scholar.vt.edu/portal/site/0f07fbda-56fa-4c66-9d25-61a7887b39c6