Parallel MATLAB at FSU: Parallel FOR Loops

John Burkardt
Virginia Tech

....

https://people.sc.fsu.edu/∼jburkardt/presentations/...
matlab_parfor_2010_fsu.pdf

12 April 2010

MATLAB Parallel Computing

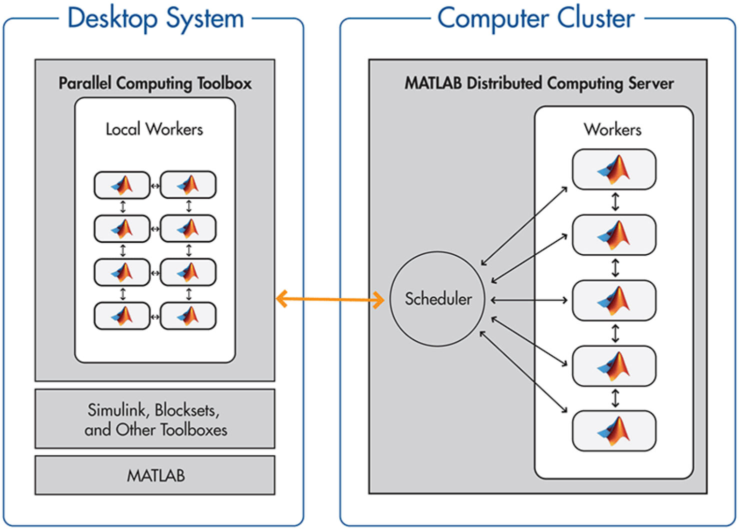
INTRO: Parallel MATLAB

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. Parallel programs can be run interactively or in batch.

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

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
Scheduler
Workers

INTRO: PARFOR: Parallel FOR Loops

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.

Using **parfor** is similar to OpenMP.

INTRO: "SPMD" Single Program Multiple Data

Tuesday's Lecture (SPMD and...)

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 by special **spmd** statements.

Workers can "see" the client's data; the client can see and change worker data.

The workers can also send messages to other workers.

INTRO: "SPMD" Distributed Arrays

Tuesday's Lecture (...and Distributed Arrays)

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.

INTRO: "TASK" Computing

Friday's Lecture (Tasks)

MATLAB can generate and manage a computation which has been divided up into many independent tasks.

Typically, these tasks can be though of as evaluating the same MATLAB function, but using different input.

The tasks are presumed to be executable in any order, at any time, on any available processor.

Once all the tasks are completed, the results are available as a single output object.

INTRO: Direct Execution for PARFOR/SPMD

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 containing **parfor** or **spmd** 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, though, because no one else can use the workers until you release them!

INTRO: Indirect Execution for PARFOR/SPMD

Parallel (PARFOR and SPMD) MATLAB jobs can be run indirectly.

The **batch** command is used to specify a MATLAB script 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 clusters.

(FSU HPC cluster access is **not** done with the **batch** command!)

INTRO: Indirect Execution for TASKS

Parallel (TASK) MATLAB jobs can be run indirectly.

The **create_job** command defines a job, which is then filled up with tasks, by the **create_task** command.

Once all the tasks have been defined, a **submit** command sends the job to be run indirectly.

These commands work on the desktop, and can be set up to access clusters.

(FSU HPC cluster access is **not** done with the **submit** command!)

INTRO: FSU Cluster PARFOR/SPMD/TASKS Execution

To submit a parallel MATLAB job to the FSU HPC cluster, you start a MATLAB session on one of the login nodes, and after you've set up any necessary data, you issue the command

```
results = fsuClusterMatlab(*, *, *, * , *, *, *);
```

The stars will be explained later. They specify the MATLAB function to run, the kind of job (PARFOR/SPMD/TASKS), where the output should go, and how many workers are requested.

Once you issue this command on the login node, your job is placed in a queue on the cluster, executed, and the results are returned.

You can wait for your results, or log in later and examine them.

INTRO: Parallel MATLAB on the FSU HPC Cluster

FSU's Department of Scientific Computing maintains an HPC cluster on which parallel MATLAB is available.

There are 10 login nodes, each with 2 PCT licenses. So as many as 20 people could be "talking" parallel MATLAB at the same time.

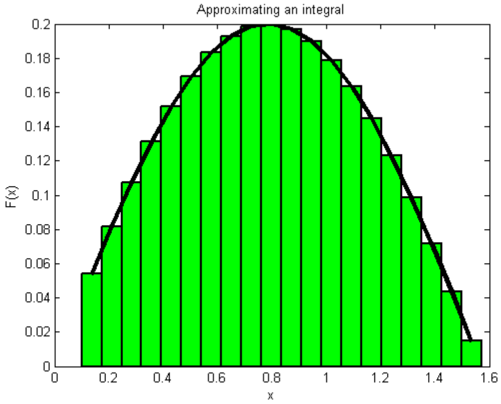
The computing cluster has 128 nodes with dual socket dual core AMD's, and 256 nodes with dual socket quad core AMD'S:

128 * (2 * 2) + 256 * (2 * 4) = 2,560 cores.

The HPC cluster has 16 DCS licenses, which means no job can get more than 16-fold parallelism, and those 20 potential users must contend for the 16 processors. One thing the fsuClusterMatlab command does is try to manage this contention.

MATLAB Parallel Computing

QUAD: Estimating an Integral



QUAD: The QUAD Function

```
function q = quad ( 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
```

QUAD: Comments

The function **quad** 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.

QUAD: The QUAD_FUN Function

```
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
```

QUAD: Comments

quad_fun does the same calculation as **quad**.

The **parfor** statement changes how this program does the calculation. 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

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 2008a** or later:
    - Go to the **HELP** menu, and choose **About Matlab**.
3. You must have the **Parallel Computing Toolbox**:
    - To list *all* your toolboxes, type the MATLAB command **ver**.

EXECUTION: Three Ways to Run

A **parfor** program needs MATLAB to gather “workers” to cooperate on the program.

On a desktop, we issue an interactive **matlabpool** request:

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

or the **batch** command with a script and a **matlabpool** argument:

```
batch ( 'quad_script', 'matlabpool', 4 )
```

or, on the FSU HPC cluster:

```
fsuClusterMatlab([],[],'p','w',4,...
  @quad_fun, { n, a, b } )
```

EXECUTION: Interactive MATLABPOOL

To run *quad_fun.m* in parallel on your desktop, type:

```
n = 10000; a = 0; 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.

EXECUTION: Indirect BATCH

The batch command, for indirect execution, only accepts scripts. We can make a suitable script called **quad_script.m**:

```
n = 10000; a = 0; b = 1;
q = quad_fun ( n, a, b )
```

Now we create a job:

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

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

```
submit ( job );
wait ( job );
load ( job );
destroy ( job );
```

EXECUTION: fsuClusterMatlab

On the HPC cluster, we could run the program by the command

```
n = 10000; a = 0; b = 1;
fsuClusterMatlab([],[],'p','w',4,...
  @quad_fun, { n, a, b } )
```

- **[ ]** allows us to specify an output directory;
- **[ ]** allows us to specify queue arguments;
- **'p'** means this is a "pool" (parfor) job;
- **'w'** means our MATLAB session pauses til the job has run;
- **4** is the number of workers we request;
- **@quad_fun** names the function to evaluate;
- { **n, a, b**} holds the input arguments to **quad_fun**.

MATLAB Parallel Computing

- Introduction
- QUAD Example
- Executing a PARFOR Program
- **MD Example**
- PRIME_NUMBER Example
- ODE_SWEEP Example
- FMINCON Example
- Conclusion

MD: The Molecular Dynamics Example

The MD program runs a simple molecular dynamics simulation.

N counts the number of molecules being simulated.

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

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

MATLAB has a **profile** command that can report where the CPU time was spent - which is where we should try to parallelize.

MD: Run MATLAB's Profiler

```
>> profile on
>> md
>> profile viewer
```

```
Step     Potential          Kinetic       (P+K-E0)/E0
         Energy             Energy        Energy Error

   1   498108.113974         0.000000    0.000000e+00
   2   498108.113974         0.000009    1.794265e-11
 ...      ...                  ...           ...
   9   498108.111972         0.002011    1.794078e-11
  10   498108.111400         0.002583    1.793996e-11

    CPU time  = 415.740000 seconds.
    Wall time = 378.828021 seconds.
```

MD: Profile Results

Function Name	Calls	**Total Time**	Self Time*	Total Time Plot (dark band = self time)
md	1	415.847 s	0.096 s	
compute	11	415.459 s	410.703 s	
repmat	11000	4.755 s	4.755 s	
timestamp	2	0.267 s	0.108 s	
datestr	2	0.130 s	0.040 s	
timefun/private/formatdate	2	0.084 s	0.084 s	
update	10	0.019 s	0.019 s	
datevec	2	0.017 s	0.017 s	
now	2	0.013 s	0.001 s	
datenum	4	0.012 s	0.012 s	
datestr>getdateform	2	0.005 s	0.005 s	
initialize	1	0.005 s	0.005 s	
etime	2	0.002 s	0.002 s	

Self time is the time spent in a function excluding the time spent in its child functions. Self time also includes overhead res
the process of profiling.

MD: The COMPUTE Function

```
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 )
        rij(1:nd) = pos(1:d,i) - pos(1:nd,j);
        d = sqrt ( sum ( rij(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) - rij(1:nd) * sin ( 2.0 * d2 ) / d;
      end
    end
  end

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

MD: Speedup

In the **compute** function, the important quantity is the force **f**. For each particle **i**, **f(i)** is computed by determining the distance to all other particles, squaring, truncating, and taking the sine.

Notice that the computation for each particle is independent. We could compute each value on a separate worker, at the same time.

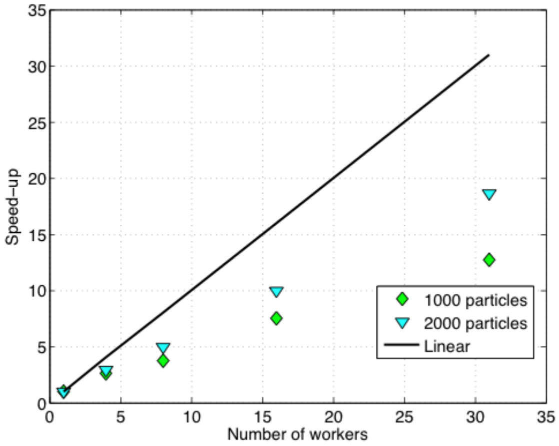
The MATLAB command **parfor** will distribute the iterations of this loop across the available workers.

Tricky question: Could we parallelize J instead? Both loops?

Tricky question: Could we parallelize both I and J loops?

MD: Speedup

Replacing **for i** by **parfor i**, here is our speedup:



MD: 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 we will see some alternatives as well.

MD: PARFOR is Particular

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: How come the scalar value **POT** was acceptable?

MATLAB Parallel Computing

- Introduction
- QUAD Example
- Executing a PARFOR Program
- MD Example
- **PRIME_NUMBER Example**
- ODE_SWEEP Example
- FMINCON Example
- Conclusion

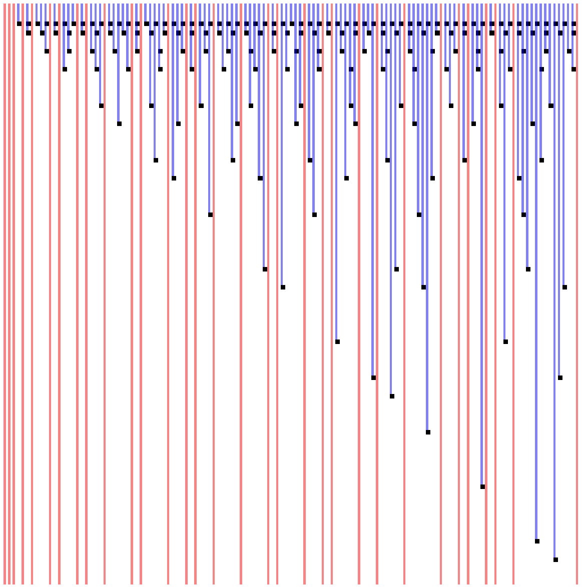
PRIME: The Prime Number Example

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** makes the run time increase by a factor of 4.

PRIME: The Sieve of Erastosthenes



PRIME: Program Text

```
function total = prime_number ( n )

%% PRIME_NUMBER 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
```

PRIME: We can run this in parallel

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!

PRIME: Execution Commands

```
matlabpool ( 'open', 'local', 4 )

n = 50;

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

matlabpool ( 'close' )
```

PRIME: Timing

```
PRIME_NUMBER_PARFOR_RUN
  Run PRIME_NUMBER_PARFOR with 0, 1, 2, and 4 labs.

     N           1+0        1+1       1+2       1+4

    50         0.067      0.179     0.176     0.278
   500         0.008      0.023     0.027     0.032
  5000         0.100      0.142     0.097     0.061
 50000         7.694      9.811     5.351     2.719
500000       609.764    826.534   432.233   222.284
```

PRIME: Timing Comments

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

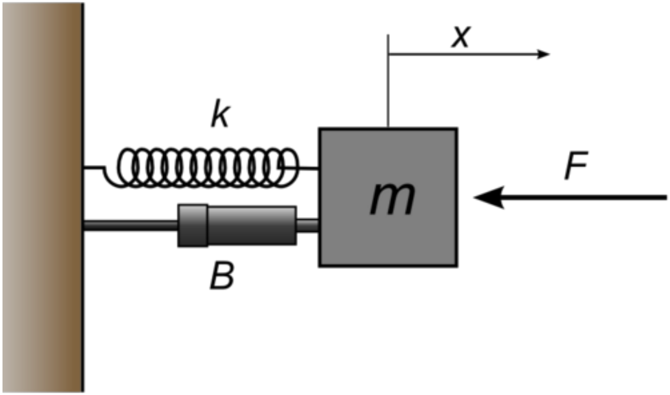
Parallelism doesn't pay until you have a decent number of workers.

MATLAB Parallel Computing

ODE: A Parameterized Problem

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

$$m\frac{d^2x}{dt^2} + b\frac{dx}{dt} + k\,x = f(t)$$



ODE: A Parameterized Problem

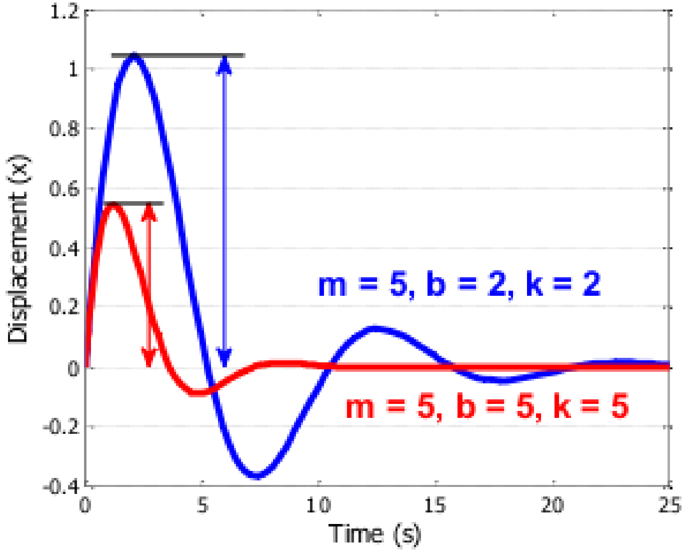
Solutions of this equation describe oscillatory behavior; $x(t)$ swings back and forth, in a pattern determined by the parameters m, b, k, f and the initial conditions.

Each choice of parameters defines a solution, and let us suppose that the quantity of interest is the maximum deflection **xmax** 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 $xmax(b, k)$.

ODE: Each Solution has a Maximum Value



ODE: A Parameterized Problem

Evaluating the implicit function $xmax(b, k)$ 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!

ODE: The Parallel Code

```
m = 5.0;
bVals = 0.1 : 0.05 : 5;
kVals = 1.5 : 0.05 : 5;

[ kGrid, bGrid ] = meshgrid ( bVals, kVals );

peakVals = nan ( size ( kGrid ) );

tic;

parfor ij = 1 : numel(kGrid)

  [ T, Y ] = ode45 ( @(t,y) ode_system ( t, y, m, bGrid(ij), kGrid(ij) ), ...
    [0, 25], [0, 1] );

  peakVals(ij) = max ( Y(:,1) );

end

toc;
```

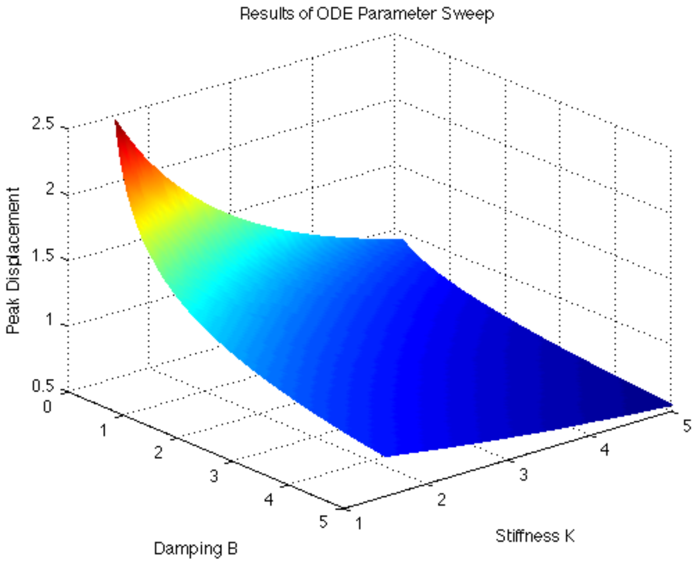
ODE: MATLABPOOL or BATCH Execution

```
  matlabpool open local 4
  ode_sweep_parfor
  matlabpool close
  ode_sweep_display

----------------------

  job = batch ( ...
    'ode_sweep_parfor', ...
    'FileDependencies', {'ode_system.m'}, ...
    'matlabpool', 4 );
  wait ( job );
  load ( job );
  ode_sweep_display
  destroy ( job )
```

ODE: Display the Results

```
%
%  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



ODE: A Very Loosely Coupled Calculation

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 $xmax(b, x)$ 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.

MATLAB Parallel Computing

FMINCON: Hidden Parallelism

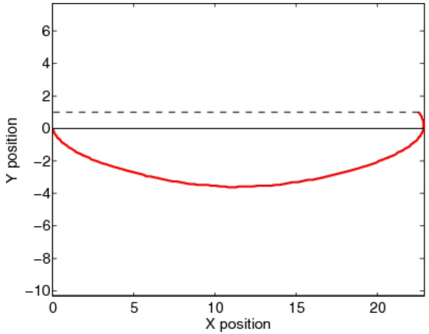
FMINCON is a popular built-in MATLAB function which finds the minimizer of a function of several variables with constraints:

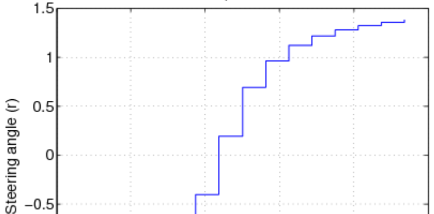
```
min F(X)  subject to:

A*X   <= B,
Aeq*X  = Beq  (linear constraints)
C(X)  <= 0,
Ceq(X) = 0    (nonlinear constraints)
LB <= X <= UB (bounds)
```

If no derivative or Hessian information is supplied by the user, then **FMINCON** uses finite differences to estimate these quantities. If **fun** is expensive to evaluate, the finite differencing can dominate the execution.

FMINCON: Path of a Boat Against a Current

Professor Gene Cliff created an example using FMINCON, in which a boat is steering against a current, and trying to reach the shore.





FMINCON: Hidden Parallelism

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:

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

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

...and uses the **matlabpool** command to make workers available!

MATLAB Parallel Computing

- Introduction
- QUAD Example
- Executing a PARFOR Program
- MD Example
- PRIME_NUMBER 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).

CONCLUSION: Summary of Examples

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 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.

CONCLUSION: Where is it?

The Parallel Computing Toolbox is installed on the HPC login nodes (2 licenses each) and there are 16 DCE licenses on the HPC compute nodes.

FSU's Department of Scientific Computing has received 20 extra, temporary licenses for the Parallel Computing Toolbox.

It's available on classroom machines **class01** through **class10** and the public machines **hallway-b** through **hallway-f**, and valid through April 21.

Run it by typing **/scratch/R2010aTrial/bin/matlab**

CONCLUSION: Where is it?

The temporary license includes lots of extras!:

- Curve Fitting Toolbox
- Image Processing Toolbox
- Optimization Toolbox
- Parallel Computing Toolbox
- Signal Processing Blockset
- Signal Processing Toolbox
- Statistics Toolbox
- Symbolic Math Toolbox

CONCLUSION: Where is it?

- MATLAB Parallel Computing Toolbox User's Guide 4.3 www.mathworks.com/access/helpdesk/help/pdf_doc/distcomp/... distcomp.pdf
- FSU HPC web site: www.hpc.fsu.edu/
- Gaurav Sharma, Jos Martin, *MATLAB: A Language for Parallel Computing*, International Journal of Parallel Programming, Volume 37, Number 1, pages 3-36, February 2009.
- http://people.sc.fsu.edu/∼burkardt/m_src/m_src.html
  - **quad_parfor**
  - **md_parfor**
  - **prime_number_parfor**
  - **ode_sweep_parfor**
  - **fmincon_parallel**