

Parallel MATLAB: Parallel FOR Loops

John Burkardt (ARC/ICAM) & Gene Cliff (AOE/ICAM)
3pm - 4pm, Friday, 04 June 2010,
3060 Torgersen Hall

.....

ARC: Advanced Research Computing
AOE: Department of Aerospace and Ocean Engineering
ICAM: Interdisciplinary Center for Applied Mathematics

.....

[https://people.sc.fsu.edu/~jburkardt/presentations/...
matlab_parfor_2010_vt.pdf](https://people.sc.fsu.edu/~jburkardt/presentations/...matlab_parfor_2010_vt.pdf)



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



INTRO: Parallel MATLAB

Parallel MATLAB is an extension of MATLAB that takes advantage of multicore desktop machines and clusters.

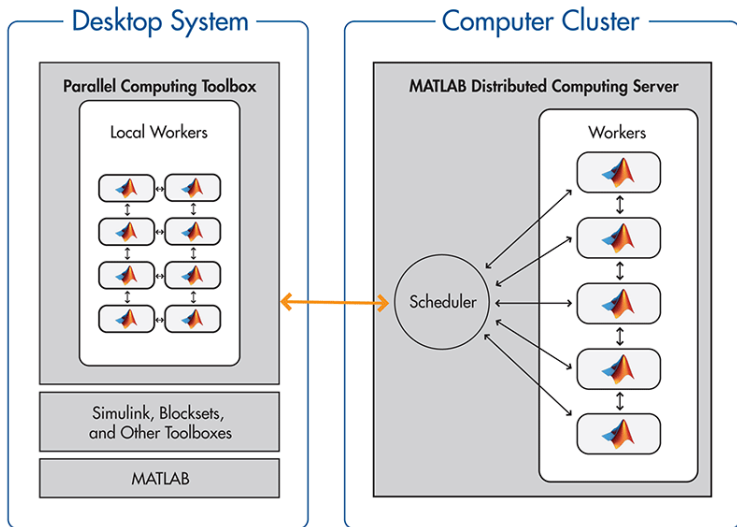
The *Parallel Computing Toolbox* or **PCT** runs on a desktop, and can take advantage of up to 8 cores there. Parallel programs can be run interactively or in batch.

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

Virginia Tech's Ithaca cluster allows parallel MATLAB to run on up to 64 cores simultaneously.



INTRO: Local and Remote MATLAB Workers



Three ways to write a parallel MATLAB program:

- suitable **for** loops can be made into **parfor** loops;
- the **spmatrix** 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. **spmatrix** 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 **createTask**);
- indirect remote, (**batch** or **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 a one-time setup of the user's machine and Ithaca account.



INTRO: PARFOR: Parallel FOR Loops

Today's Lecture: PARFOR - June 4th

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

Lecture #2: SPMD - June 25th

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 access and change worker data.

The workers can also send messages to other workers.



INTRO: "SPMD" 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.



Lecture #3: TASKS - July 2

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

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

In the *parallel task* version, the tasks must run at the same time, and they can communicate with each other.

In the *scattered task* version, the tasks can run at any time, in any order, and do not communicate.

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



INTRO: Direct Execution for PARFOR

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, though, because no one else can use the workers until you release them!

So...don't log into Ithaca interactively, and treat it as though it was your desktop machine! In our examples, we will indeed use Ithaca, but always through the indirect batch system.



INTRO: Indirect Execution for PARFOR

Parallel PARFOR 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 the Ithaca cluster.



Virginia Tech has installed the ITHACA cluster of 84 nodes. Each node is a separate computer with 2 quadcore processors.

This means each node can run 8 MATLAB workers.

At Virginia Tech, 8 nodes with 8 cores are dedicated to the parallel MATLAB cluster, so **theoretically** you can run a job with 64 workers.

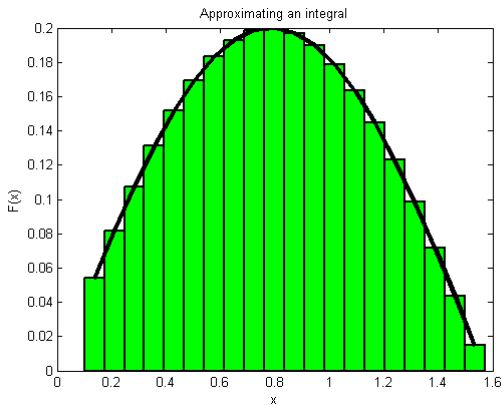
You should not routinely ask for all 64 workers. Occasionally, one node is down, so there are only 56. Moreover, if one job ties up all the workers, no one else can run. So we encourage the use of 24 or 32 workers at a time instead.



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



QUAD: Estimating an Integral



QUAD: The QUAD_FUN Function

```
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
```



QUAD: Comments

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.



QUAD: The Parallel 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
```



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.



- Introduction
- QUAD Example
- **Executing a PARFOR Program**
- 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 2008a** 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**.



EXECUTION: Ways to Run

Workers are gathered using the **matlabpool** command.

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

```
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:

```
job = batch ( 'quad_script', 'matlabpool', 4 )  
(or)  
job = batch ( 'quad_script', 'matlabpool', 4,  
             'configuration', 'local' )
```



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 Local 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 define the information needed to run the script:

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



EXECUTION: Indirect Remote BATCH

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:

```
job = batch ( 'quad_script', 'matlabpool', 4, ...  
    'Configuration', 'ithaca_2010b', ...  
    'FileDependencies', 'quad_fun' )
```



EXECUTION: Submitting a Job and Waiting

How do the results come back to you?

Whether run locally or remotely, the following commands send the job for execution, wait for it to finish, and then load the results into MATLAB's workspace:

```
job = batch ( ...information defining the job... )  
submit ( job );  
wait ( job );  
load ( job );
```

Doing this requires that you stay logged in so that the value of **job** can be used to identify output to the **load()** command.



EXECUTION: Submitting a Job and Coming Back Later

If you don't want to wait for a remote job to finish, you can exit after the **submit()**, turn off your computer, and go home.

However, when you think your job has run, you now have to try to retrieve the **job** identifier before you can load the results.

```
job = batch ( ...information defining the job... )  
submit ( job );
```

Exit MATLAB, turn off machine, go home.

Come back, restart machine, start MATLAB:

```
sched = findResource ( );  
jobs = findJob ( sched )
```

findJob() returns a cell array of all your jobs.

You pick out the one you want, say "k".

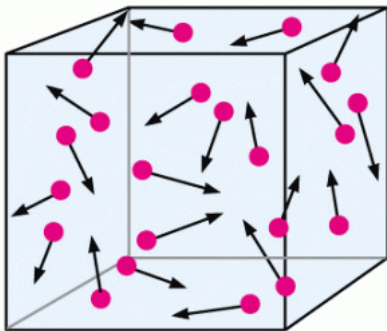
```
load ( jobs{k} );
```



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



MD: A Molecular Dynamics Simulation



Compute the positions and velocities of \mathbf{N} particles at a sequence of times. The particles exert a weak attractive force on each other.



MD: The Molecular Dynamics Example

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 **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: Profile the Sequential Code

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




Step	Potential Energy	Kinetic Energy	(P+K-E0)/E0 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: Where is Execution Time Spent?

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 from 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: Can We Use PARFOR?

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.

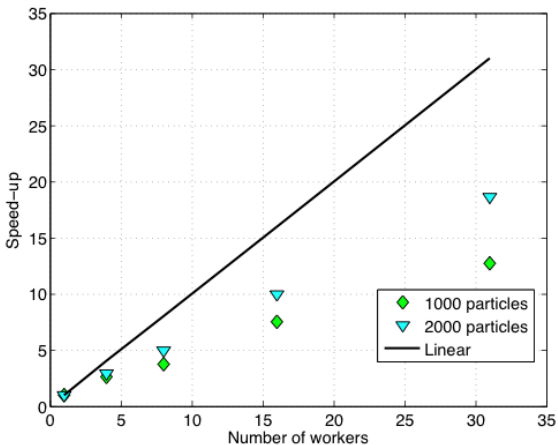
Tricky question: Could we parallelize the **j** loop instead?

Tricky question: Could we parallelize **both** loops?



MD: Speedup

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.



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: Why was the scalar value **POT** acceptable?



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



PRIME: The Prime Number Example

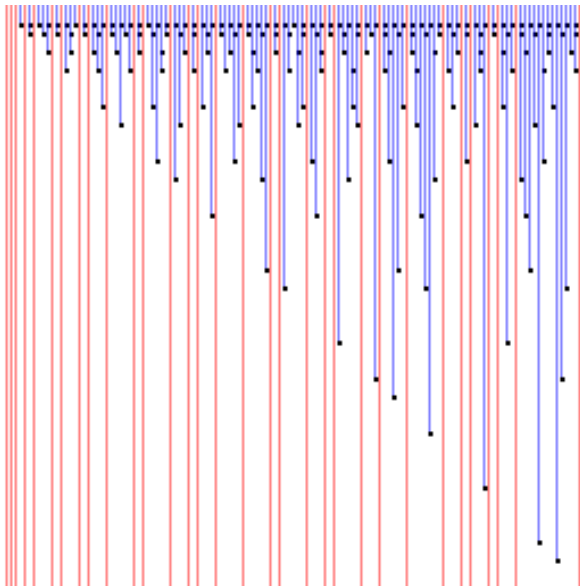
For our next example, we want a simple computation involving a loop which we can set up to run for a long time.

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** multiplies the run time roughly by 4.



PRIME: The Sieve of Eratosthenes



PRIME: Program Text

```
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
```



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: Local Execution With MATLABPOOL

```
matlabpool ( 'open', 'local', 4 )  
  
n = 50;  
  
while ( n <= 500000 )  
    primes = prime_parfor ( n );  
    fprintf ( 1, '\n%8d\n%8d\n', n, primes );  
    n = n * 10;  
end  
  
matlabpool ( 'close' )
```



PRIME: Timing

PRIME_PARFOR_RUN

Run PRIME_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.



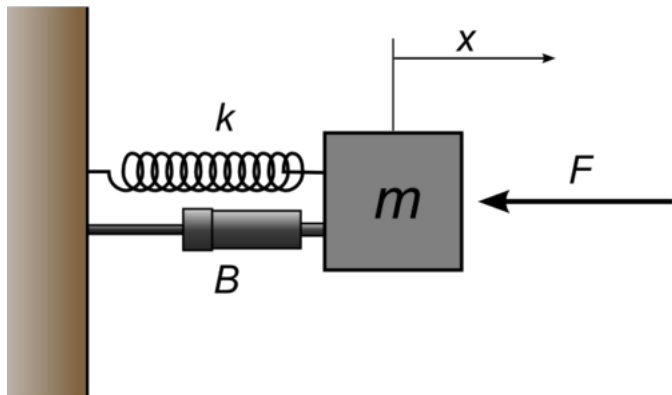
- Introduction
- QUAD Example
- Executing a PARFOR Program
- MD Example
- PRIME Example
- **ODE_SWEEP Example**
- FMINCON Example
- Conclusion



ODE: A Parameterized Problem

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

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



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.

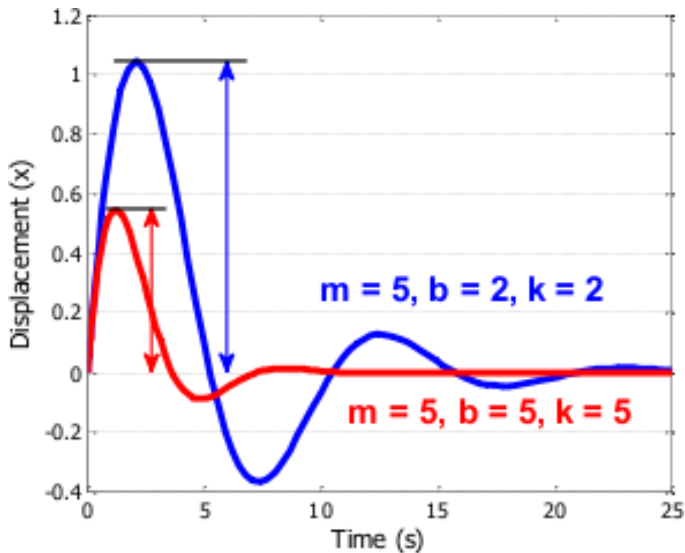
Each choice of parameters defines a solution, and let us suppose that the quantity of interest is the maximum deflection x_{\max} that occurs for each solution.

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 $x_{\max}(b, k)$.



ODE: Each Solution has a Maximum Value



ODE: A Parameterized Problem

Evaluating the implicit function $x_{max}(b, k)$ 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!



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_script', ...
    'Configuration', 'local', ...
    'FileDependencies', {'ode_system.m'}, ...
    'matlabpool', 4 );
wait ( job );
load ( job );
ode_sweep_display
destroy ( job )
```

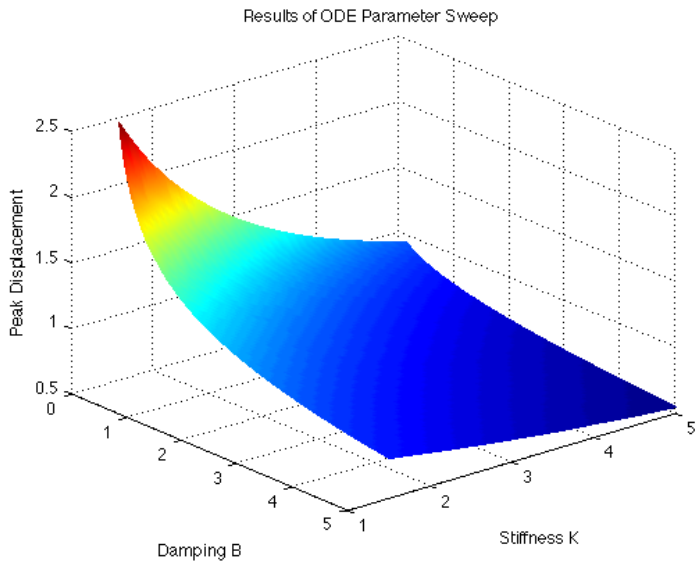


ODE: Display the Results

```
%  
% 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 $x_{max}(b, x)$ 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
- MD Example
- PRIME Example
- ODE_SWEEP Example
- **FMINCON Example**
- Conclusion



FMINCON: Hidden Parallelism

FMINCON is a popular MATLAB function available in the Optimization Toolbox. It 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.



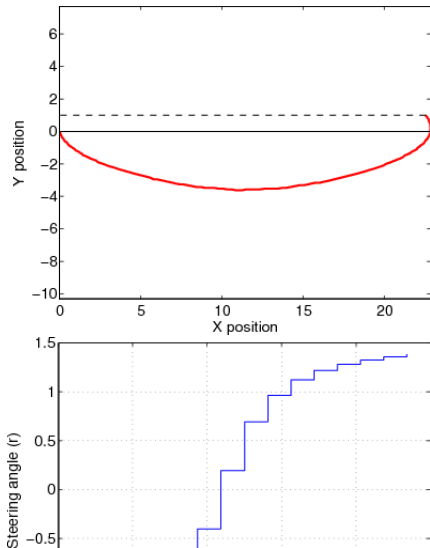
FMINCON: Path of a Boat Against a Current

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!



FMINCON: Riding the Helpful Current

The correct solution takes maximum advantage of the favorable current, and then steers back hard to the land on the line $y = 1$.



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!



- Introduction
- QUAD Example
- Executing a PARFOR Program
- 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 **spmatrix** 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: Desktop Experiments

Virginia Tech has a limited number of concurrent MATLAB licenses, which include the Parallel Computing Toolbox.

This is one way you can test parallel MATLAB on your desktop machine.

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: Cluster Experiments

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/index.php`

Under the item **Services & Support** select **User Accounts**.

On the new page, under **Ithaca Account Requests**, select **ARC Systems Account Request Form**. Fill in the information and submit it. Although you're asked to describe the research you want the account for, you can say that this account is to experiment with Ithaca to see if it is suitable for your work.



Conclusion: Desktop-to-Cluster Submission

If you want to use parallel MATLAB regularly, you may want to set up a way to submit jobs from your desktop to Ithaca, without logging in directly.

This requires defining a configuration file on your desktop, adding some scripts to your MATLAB directory, and setting up a secure connection to Ithaca.

The steps for doing this are described in the document:

```
https://portal.arc.vt.edu/matlab/...  
RemoteMatlabSubmission.pdf
```

We will be available to help you with this process.



Conclusion: VT MATLAB LISTSERV

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 User's Guide 4.3
www.mathworks.com/access/helpdesk/help/pdf_doc/distcomp/...distcomp.pdf
- 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/~jburkardt/m_src/m_src.html
 - **quad_parfor**
 - **md_parfor**
 - **prime_parfor**
 - **ode_sweep_parfor**
 - **fmincon_parallel**

