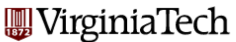
Parallel MATLAB at VT

John Burkardt (ARC/ICAM - jvburkardt@gmail.com)
Gene Cliff (AOE/ICAM - ecliff@vt.edu)
Justin Krometis (ARC/ICAM - jkrometis@vt.edu)
11:30am-1:30pm Thursday, 19 January 2017
.... NLI ....

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

Slide images at: arc.vt.edu/resources/software/matlab

- **Introduction**
- Programming Models
- Execution
- Example: Quadrature
- Conclusion



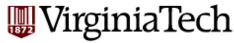
INTRO: Parallel MATLAB

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 cores (R2014a has no limit, R2013b limit is 12, ...). 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.

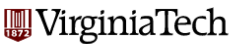
ARC's clusters (Cascades, DragonsTooth, NewRiver, BlueRidge) provides **MDCS** services for up to 224 cores. Currently, single users are restricted to 96 cores.



INTRO: 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 2012a** or later:
    - Go to the **HELP** menu, and choose **About Matlab**.
3. You must have the **Parallel Computing Toolbox**:
    - At VT, the concurrent (& student) license includes the PCT.
    - The *standalone* license does not include 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.
    - ARC Clusters currently support R2015a-R2016b.

- Introduction
- **Programming Models**
- Execution
- Example: Quadrature
- Conclusion



PROGRAMMING: Obtaining Parallelism

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 may require rethinking the program/data. The **task** approach is simple, but suitable only for computations that need almost no communication.

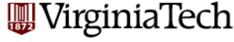
Lecture #2: PARFOR

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 array-data access.

OpenMP implements a `directive` for 'parallel for loops'



PROGRAMMING: "SPMD" Single Program Multiple Data

Lecture #3: 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**.

PROGRAMMING: "SPMD" Distributed Arrays

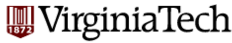
SPMD programming includes distributed arrays.

A distributed array is logically one array, and a large set of MATLAB commands can treat it that way (*e.g.* 'backslash').

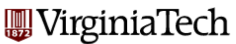
However, portions of the array are scattered across multiple processors. This means such an array can be really large.

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.



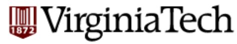
- Introduction
- Programming Models
- **Execution**
- Example: Quadrature
- Conclusion



EXECUTION: Models

There are several ways to execute a parallel MATLAB program:

Model	Command	Where It Runs
Interactive	`matlabpool`	This machine
Interactive	`parpool` (R2013b)	This machine
Indirect local	`batch`	This machine
Indirect remote	`batch`	Remote machine



EXECUTION: Direct using `parpool`

Parallel MATLAB jobs can be run *directly*, that is, interactively.

The **parpool** (previously **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.

Note: Starting in R2013b, if you try to execute a parallel program and a pool of workers is not already open, MATLAB will open it for you. The pool of workers will then remain open for a time that can be specified under Parallel → Parallel Preferences (default = 30 minutes).

VirginiaTech

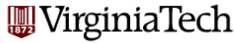
EXECUTION: Indirect Local using `batch`

Parallel MATLAB jobs can be run indirectly.

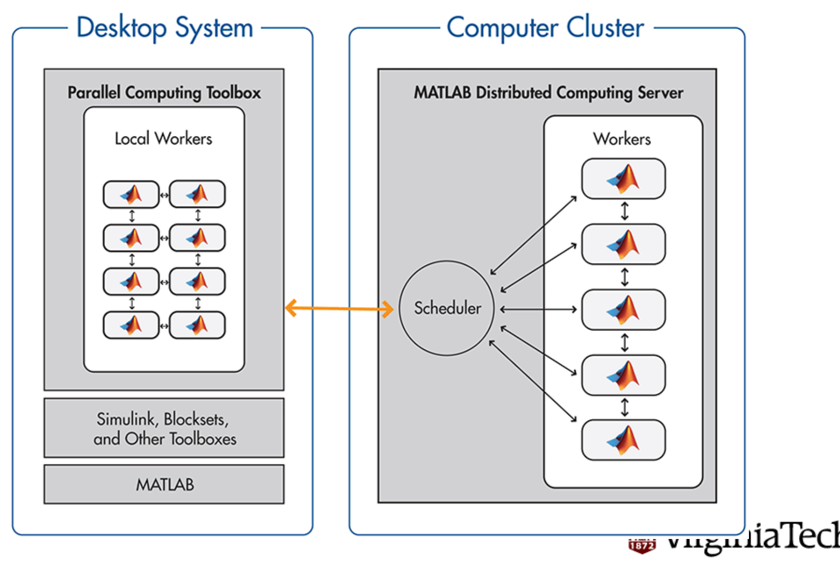
The **batch** command is used to specify a MATLAB code to be executed, to indicate any files that will be needed, and how many workers are requested.

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 ARC clusters (e.g. NewRiver).

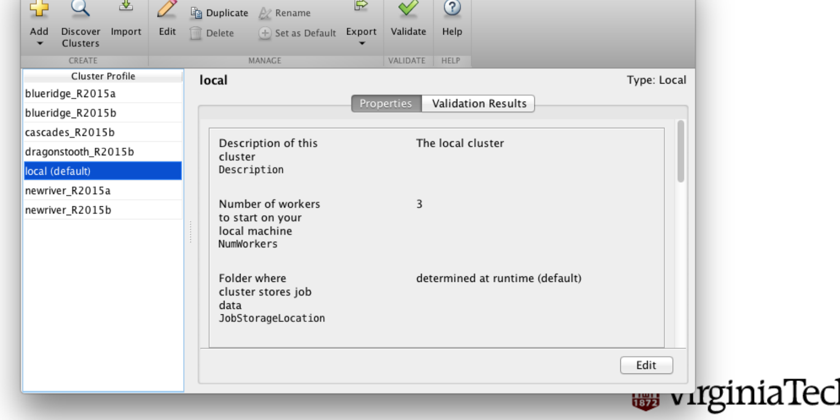


EXECUTION: Local and Remote MATLAB Workers



EXECUTION: Managing Cluster Profiles

MATLAB uses Cluster Profiles (previously called “configurations”) to set the location of a job. ‘local’ is the default. Others can be added to send jobs to other clusters (e.g. NewRiver).



VirginiaTech

EXECUTION: Ways to Run

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

```
mypool = parpool ( 'local', 4 )
q = quad_fun ( n, a, b );
delete(mypool)
```

'local' is a default Cluster Profile defined as part of the PCT.
The **batch** command runs a script, with a **Pool** argument:

```
job = batch ( 'quad_script', 'Pool', 4 )
(or)
job = batch ( 'Profile','local', 'quad_script', ...
                                        'Pool', 4 )
```

EXECUTION: ARC Clusters

ARC offers resources with Matlab installed, including:

System	Usage	Nodes	Node Description	Special Features
Cascades	Large-scale CPU	196	32 cores, 128 GB (2× Intel Broadwell)	16 K80 GPGPU 2 3TB nodes
DragonsTooth	Single-node jobs	48	24 cores, 256GB (2× Intel Haswell)	
NewRiver	Data Intensive	126	24 cores, 128 GB (2× Intel Haswell)	8 K80 GPGPU 24 512GB nodes 2 3TB nodes
BlueRidge	Large-scale CPU, MIC	408	16 cores, 64 GB (2× Intel Sandy Bridge)	260 Intel Xeon Phi 4 K40 GPU 18 128GB nodes

ARC has a MDCS that can currently accommodate a combination of jobs with a total of 224 workers. At this time the queueing software imposes a limit of 96 workers per user.

VirginiaTech

EXECUTION: Configuring Desktop-to-Cluster Submission

- If you want to work with parallel MATLAB on ARC resources, you must first get an account. Go to

  `http://www.arc.vt.edu/account`

  Log in (PID and password), select the systems you want to work with and MATLAB in the Software section, and submit.

- Steps to set up submission from your desktop include:
  1. Download and add some files to your MATLAB directory
  2. Run a script to create a new profile on your desktop.

  A new cluster profile (e.g. `newriver_R2015b`) will be created that can be used in `batch()`.
  These steps are described in detail here:

  `http://www.arc.vt.edu/matlabremote`

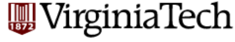
EXECUTION: Intracluster Submission

You can also submit jobs to ARC clusters from a cluster login node.

- Pros: Easier to set up. Only one file system to manage.
- Cons: Requires logging into the cluster (e.g., with SSH). Have to use Matlab command line (except on NewRiver).

Setting up intracluster submission is very simple - running a one-question script at the Matlab command line on the ARC cluster.
The full steps are described here:

`http://www.arc.vt.edu/matlabremote#intracluster`



EXECUTION: Job Monitor

Matlab's Job Monitor provides a convenient way to track jobs that are running in the background locally or remotely on ARC's machines.

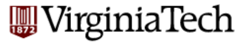
Job Monitor

Select Profile: newriver_R2015b

ID	Username	Submit Time	Finish Time	Tasks	State	Description
1	jkrometis	Fri Jul 22 14:52:05 EDT 2016		12	finished	Batch job running script: prime_script.
2	jkrometis	Fri Jul 22 15:36:39 EDT 2016		12	finished	Batch job running script: prime_script.
3	jkrometis	Fri Jul 22 15:39:55 EDT 2016	Fri Jul 22 15:40:50 EDT 2016	12	finished	Batch job running script: prime_script.

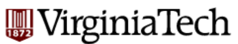
Last updated at Mon Sep 26 06:16:55 EDT 2016

Auto update: Every 5 minut

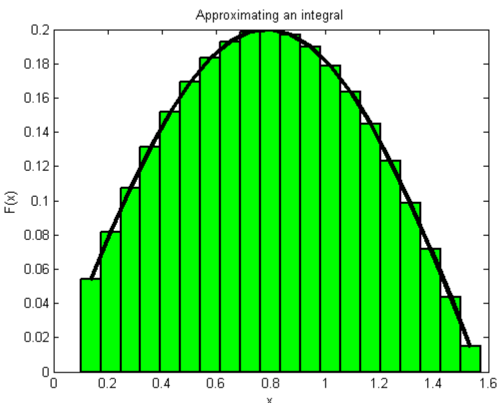


MATLAB Parfor

- Introduction
- Programming Models
- Execution
- **Example: Quadrature**
- 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= 4./(1+x.^2);
        q = q+w*fx;
    end

    return
end
```

QUAD: Comments

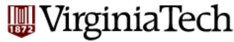
The function **quad_fun** estimates the integral of a particular function over the interval $[a, b]$.

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;
%   for i=1:n  % avoid starting pool
    parfor i=1:n
      x = ((n-i)*a+(i-1)*b)/(n-1);
      fx= 4./(1+x.^2);
      q = q+w*fx;
    end

    return
end
```

QUAD: Comments

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 (with exceptions) 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.

VirginiaTech

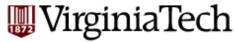
QUAD: Interactive

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

```
n = 10000; a = 0.5; b = 1;
pool = parpool('local',4)
q = quad_fun ( n, a, b );
delete(pool)
```

The word **local** is choosing the local profile, 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 12 on a local machine. It does not have to match the number of cores you have.



QUAD: Indirect Local BATCH

The batch command, for indirect execution, accepts scripts (and since R2010b functions). We can make a suitable script called **quad_script.m**:

```
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:

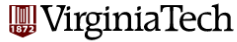
```
job = batch ( 'quad_script', 'Pool', 4, ...
  'Profile', 'local', ...
  'AttachedFiles', { 'quad_fun' } )
```

QUAD: Indirect Local BATCH

After issuing batch(), the following commands wait for the job to finish, gather the results, and clear out the job information:

```
wait ( job );   % no prompt until the job is finished
load ( job );   % load data from the job's Workspace
delete ( job ); % clean up (destroy prior to R2012a)
```

Note: You may not want to have Matlab wait for long research runs. Rather, you may want to submit (perhaps a few times) and come back and check the results later (e.g., with Job Monitor).



QUAD: Indirect Remote BATCH

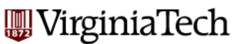
The batch command can send your job *anywhere*, and get the results back, as long as you have set up an account on the remote machine, and you have defined a Cluster Profile on your desktop that tells it how to access the remote machine.

At Virginia Tech, with proper set up, your desktop can send a batch job to an ARC cluster as easily as running locally:

```
job = batch ( 'quad_script', 'Pool', 4, ...
  'Profile', 'newriver_R2015a, ...
  'AttachedFiles', { 'quad_fun' } )
```

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

- Introduction
- Programming Models
- Execution
- Example: Quadrature
- **Conclusion**



CONCLUSION: Summary

- Introduction: Parallel Computing Toolbox
- Models of parallelism: parfor, spmd, distributed
- Models of execution: Interactive vs. Indirect, Local vs. Remote
    - ARC clusters
- Quadrature example: Parallelizing and Running

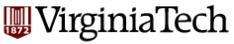
CONCLUSION: Desktop Experiments

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.



- **Introduction**
- FMINCON Example
- Executing a PARFOR Program
- PRIME Example
- Classification of variables
- ODE_SWEEP Example
- MD Example
- Conclusion

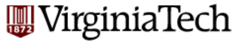
INTRO: Parallel Loops in MATLAB

In a previous lecture we discussed MATLAB's *Parallel Computing Toolbox* (**PCT**), and the *Distributed Computing Server* (**MDCS**) that runs on Virginia Tech's cluster(s).

As noted previously there are 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.

Here we focus on **parfor** loops and on **options** for parallelism in MATLAB toolboxes.



INTRO: PCT Disables Multithreading

In your (non-parallel) codes, MATLAB will automatically use multithreading; although your code is running on a single core, there may be several threads of execution that can be carried out simultaneously.

This is especially true in linear algebra functions, such as LU, QR and SVD factorizations.

However, when the PCT is in use, multithreading is disabled. If your nonparallel code was getting the advantage of multithreading, then the parallel version might run slower if only a few parallel cores are used.

For details, refer to:
http://www.mathworks.com/support/solutions/en/data/1-4PG4AN/index.html?solution=1-4PG4AN

INTRO: PCT has an "Overhead" Charge

Having twice as many helpers on a job ought to cut the time in half. But parallelism has an overhead; it takes some time and space to set it up, to synchronize, to distribute and collect data.

This means that parallelism might not work well for:

- a problem that only uses a few workers;
- a problem in which each worker only does a small amount of work;

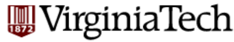
As you increase the number of workers, the improvement will be less than expected at first; then it may be linear for a while (the "sweet spot"), but then it will drop off again as there is not enough work.

BUILTIN: Many Functions have Parallelism Built In

Many programmers feel that the **parfor** command is the simplest way to get parallel performance. But in fact, there's a way that's even simpler, because now many MATLAB functions include a built-in parallelism option to take advantage of the PCT.

Such functions include an input option or option structure that allows you to request parallel execution. On invoking this, MATLAB will check your default parallel configuration and take care of everything for you.

This is the "Royal Road to Parallelism", and you may find it the quickest way to profit from the PCT.



BUILTIN: FMINCON Example

FMINCON is a popular MATLAB function available in the Optimization Toolbox. It finds a local minimizer of a real-valued function of several real variables with constraints:

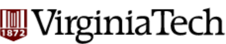
```
min F(X)  subject to:

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

If no derivative (Jacobian) information is supplied by the user, then **FMINCON** uses finite differences to estimate these quantities. If **F, C** or **Ceq** are expensive to evaluate, the finite differencing can dominate the execution time.

BUILTIN: Path of a Boat Against a Current

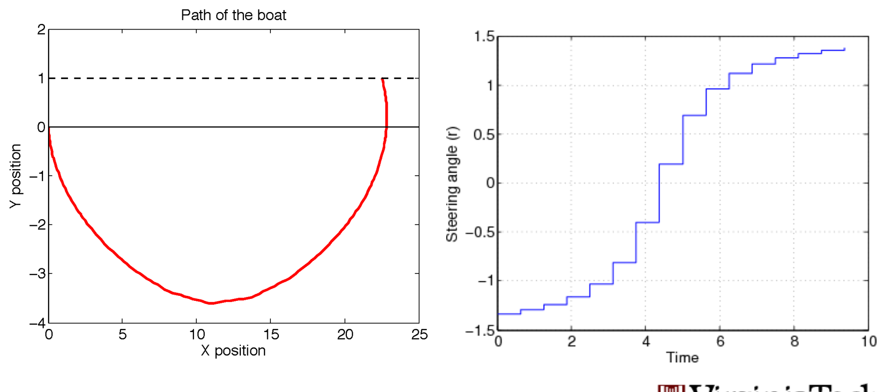
A boat at (x,y)=0 is trying to reach the riverbank at y=1.

The boat is given 10 minutes, and must try to land as far as possible upstream. In this unusual river, the current is zero midstream (x=0), increasingly negative above the x axis, and increasingly positive (helpful) below the x axis!



BUILTIN: 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 $\mathbf{y = 1}$.



BUILTIN: The UseParallel Option

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 );
```

BUILTIN: Toolboxes with Builtin Parallel Functions

- Simulink
- Code Generation
- Computational Biology
- Control System Design and Analysis
- Image Processing and Computer Vision
- Global Optimization
- Model-Based Calibration
- Optimization
- Statistics and Machine Learning
- Signal Processing and Communicatons
- Verification, Validation and Test

For details, see
https://www.mathworks.com/products/parallel-computing/parallel-support.html

PARFOR: Parallel For Loops

The **parfor** statement 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.

Why not replace every **for** by **parfor**?

A loop can only be parallelized if the work of each iteration is logically independent; the results should be the same even if we were to permute the loop indices.

MATLAB also imposes some restrictions on how arrays can be read and written within a parallel loop.

(And, of course, some loops may have too little work in them to be worth parallelizing in the first place!)

VirginiaTech

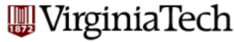
PRIME: The Prime Number Example

Let us look at an example in which we want to write a function that can tell us how many prime numbers there are between 1 and a user-specified limit **N**.

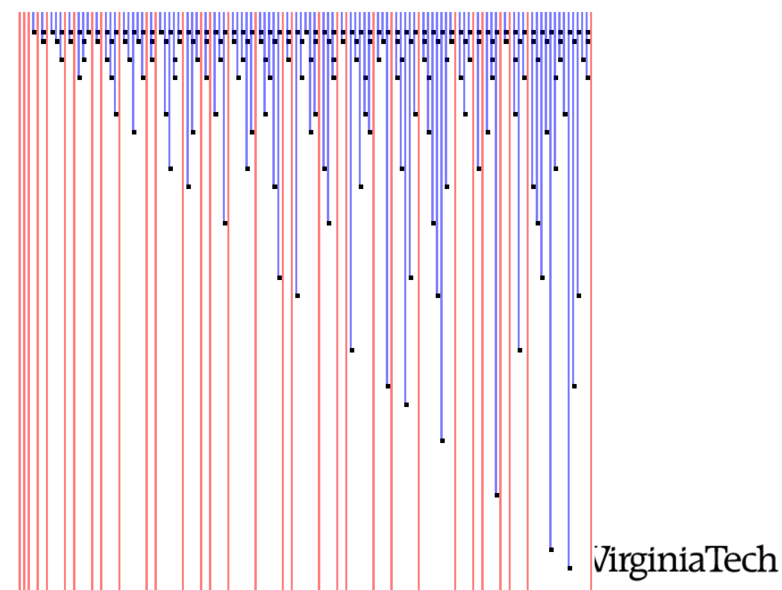
There are many clever ways to do such a calculation, but let's concentrate on a simple approach, and see how it can be converted to a parallel approach.

To determine how many primes there are, we need to check each integer **I** in the range. To check an integer **I**, we simply see whether it is divisible by any integer **J** strictly between 1 and **I**.

Notice that the work our program has to do increases nonlinearly as **N** increases. Doubling **N** multiplies the run time roughly by 4.



PRIME: The Sieve of Eratosthenes



PRIME: Program Text

```
function total = prime ( n )

  total = 0;

  for i = 2 : n             <-- Change this for to parfor?

    prime = 1;

    for j = 2 : i - 1      <-- Don't change this for to parfor!
      if ( mod ( i, j ) == 0 )
        prime = 0;
      end
    end

    total = total + prime;

  end

  return
end
```

VirginiaTech

PRIME: We can run this in parallel

We can parallelize the loop over **i**, replacing **for** by **parfor**.

However, we can't parallelize both **i** and **j**! MATLAB doesn't allow nested parallel loops

Why would it be inefficient AND wrong to parallelize the **j** loop instead of the **i** loop?

Another worry: there is a single variable **total** which is accessed by each worker. So each worker will start out with a copy of this variable set to 0, and will increment it independently. At the end of the loop execution, what do we do with multiple values for a single variable?

MATLAB is smart enough to realize that the right thing to do here is compute the sum of these values and return that as the final value of **total**. This is known as a *reduction variable*.

VirginiaTech

PRIME: Run the function on your machine

```
pool_obj = parpool ( 'local', 4 ); <=== Explicitly request
                                        4 workers
n = 50;

while ( n <= 5000000 )
  total = prime ( n );             <=== parallel code
  fprintf ( 1, '%8d  %8d\n', n, total );
  n = n * 10;
end

delete ( pool_obj );               <=== Release workers
```

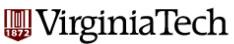
VirginiaTech

PRIME: Timing

We can use **tic** and **toc** to measure the time required.

`Run with 1 client and 0, 2, 4, 8 workers.`

N	1+0	1+2	1+4	1+8
50,000	0.13	0.10	0.08	0.06
500,000	3.25	1.66	0.85	0.44
5,000,000	84.9	43.3	21.8	10.09



PRIME: Timing Comments

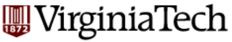
The timing data suggests two conclusions:

Parallelism doesn't pay until your problem is big enough;

AND

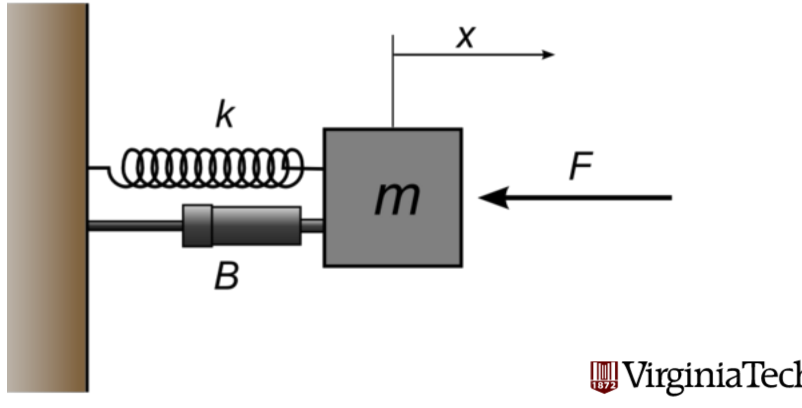
Parallelism doesn't pay until you have a decent number of workers.

(By the way, it is also possible to have **too many** workers for a given problem!)



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

$$m\frac{d^2x}{dt^2} + b\frac{dx}{dt} + k\,x = f(t), \quad x(0) = 0,\ \dot{x}(0) = v\ .$$



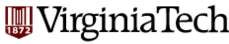
ODE: A Parameterized Problem

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.

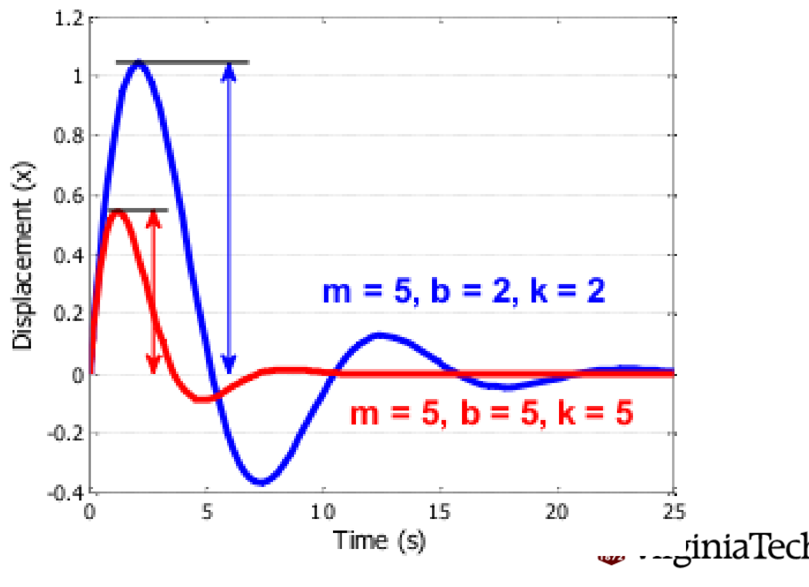
Each choice of parameters defines a solution, and let us suppose that the quantity of interest is the maximum deflection **xmax** that occurs for each solution.

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;

[ bGrid, kGrid ] = 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;
```

VirginiaTech

ODE: PARPOOL or BATCH Execution

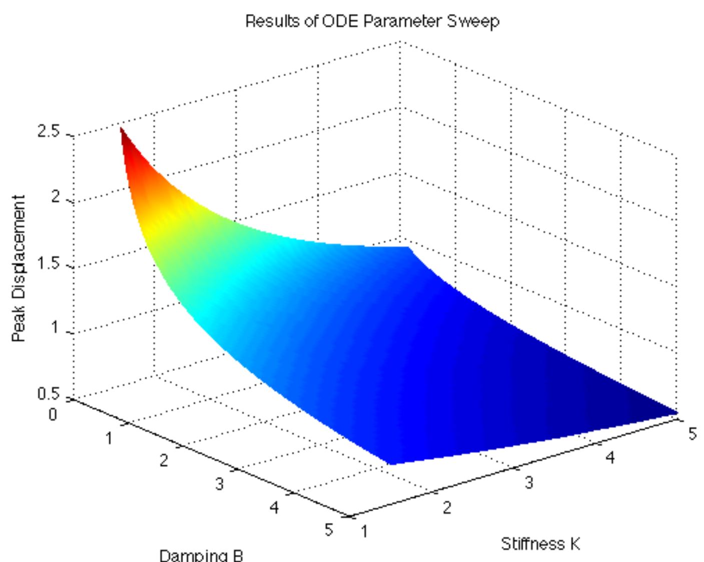
```
pool_obj = parpool('local', 4);
ode_sweep_parfor
delete(pool_obj)
ode_sweep_display

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

job = batch ( ...
  'ode_sweep_script', ...
  'Profile', 'local', ...
  'AttachedFiles', {'ode_system.m'}, ...
  'pool', 4 );
wait ( job );
load ( job );
ode_sweep_display
```

VirginiaTech

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



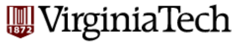
aTech

ODE: A Very Loosely Coupled Calculation

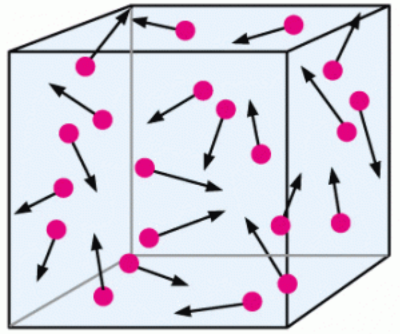
In the PRIME program, the **parfor** loop was invoked in the outer loop; in each iteration there is a reasonable workload/.

In the ODE parameter sweep, we have several thousand IVP'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.

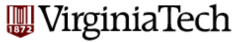
This kind of loosely-coupled problem can be treated as a *task computing* problem, wherein MATLAB can treat this problem as a collection of many little tasks to be computed in an arbitrary fashion and assembled at the end.



MD: A Molecular Dynamics Simulation



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



MD: The Molecular Dynamics Example

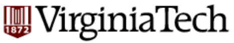
The MD program runs a simple molecular dynamics simulation.

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         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: 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 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:nd, 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
```

VirginiaTech

MD: Can We Use PARFOR?

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

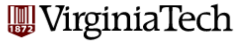
Iteration **i** computes the force on particle **i**, determining the distance to each particle **j**, squaring, truncating, taking the sine.

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.

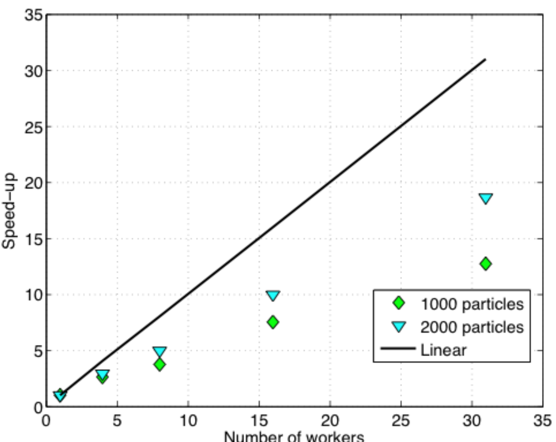
Tricky question: Could we parallelize the **j** loop instead?

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



MD: Speedup

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



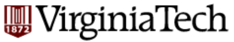
MD: Speedup

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 in the next lecture we will see some alternatives.



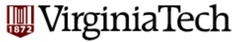
MD: PARFOR is Particular

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' **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?



PARFOR: Direct Execution

Parallel MATLAB jobs can be run *directly*, that is, interactively.

The **parpool** 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 an ARC Cluster, from your desktop machine or from an interactive session on an ARC Cluster login node! In our examples, we will indeed use `NewRiver`, but always through the indirect batch system.

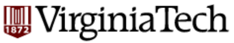
PARFOR: Indirect Execution

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.

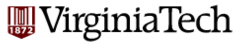
The **batch** command starts the computation in the background. The user can work on other things, and collect the results when the job is completed.

The **batch** command works on the desktop, and can be set up



PARFOR: Variable Classification

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



PARFOR: A Variable Classification 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
```

temporary variable → `a`
reduction variable → `z`
sliced output variable → `b(i)`
loop variable → `i`
sliced input variable → `r(i)`
broadcast variable → `c`

NB: "The range of a parfor statement must be increasing consecutive integers"
Trick Ques: What values to **a**, **i**, and **d** have after exiting the loop ?

PARFOR: 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).

PARFOR: Allocating the loop indices

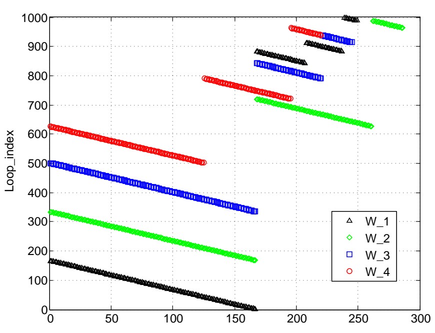
How are the *loop indices* distributed among the workers ?

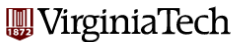
Run *ii=1:1000* on 4 workers.

$\approx$ 63% indices allocated in the first 4 *chunks*.

Indices then assigned in smaller chunks as a worker finishes

Similar to **OpenMP**'s **Dynamic** assignment





PARFOR: `batch` options

- `'Workspace'` — A 1-by-1 struct to define the workspace on the worker just before the script is called. The field names of the struct define the names of the variables, and the field values are assigned to the workspace variables. By default this parameter has a field for every variable in the current workspace where batch is executed. This parameter supports only the running of scripts.
- `'Profile'` — The name of a cluster profile used to identify the cluster. If this option is omitted, the default profile is used to identify the cluster and is applied to the job and task properties.
- `'AdditionalPaths'` — A character vector or cell array of character vectors that defines paths to be added to the MATLAB® search path of the workers before the script or function executes. The default search path might not be the same on the workers as it is on the client; the path difference could be the result of different current working folders (pwd), platforms, or network file system access. The `'AdditionalPaths'` property can assure that workers are looking in the correct locations for necessary code files, data files, model files, etc.
- `'AttachedFiles'` — A character vector or cell array of character vectors. Each character vector in the list identifies either a file or a folder, which gets transferred to the worker.
- `'AutoAttachFiles'` — A logical value to specify whether code files should be automatically attached to the job. If `true`, the batch script or function is analyzed and the code files that it depends on are automatically transferred to the worker. The default is `true`.
- `'CurrentFolder'` — A character vector indicating in what folder the script executes. There is no guarantee that this folder exists on the worker. The default value for this property is the cwd of MATLAB when the `batch` command is executed. If the argument is `'.'`, there is no change in folder before batch execution.
- `'CaptureDiary'` — A logical flag to indicate that the toolbox should collect the diary from the function call. See the `diary` function for information about the collected data. The default is `true`.
- `'Pool'` — An integer specifying the number of workers to make into a parallel pool for the job *in addition* to the worker running the batch job itself. The script or function uses this pool for execution of statements such as `parfor` and `spmd` that are inside the batch code. Because the pool requires N workers in addition to the worker running the batch, there must be at least N+1 workers available on the cluster. You do not need a parallel pool already running to execute batch; and the new pool that batch creates is not related to a pool you might already have open. (See Run a Batch Parallel Loop.) The default value is 0, which causes the script or function to run on only a single worker without a parallel pool.

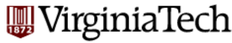
CONCLUSION: Summary of PARFOR Examples

In the **FMINCON** example, all we had to do to take advantage of parallelism was set an option (and *possibly* make sure some workers were available).

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



CONCLUSION: Summary of PARFOR 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.

MATLABs 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.

- **SPMD: Single Program, Multiple Data**
- QUAD Example
- Distributed Arrays
- LBVP & FEM_2D_HEAT Examples
- IMAGE Example
- CONTRAST Example
- CONTRAST2: Messages
- Batch Computing
- Conclusion

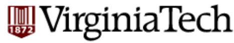
SPMD: is not PARFOR

Previous Lecture: PARFOR

The **parfor** command, described earlier, is easy to use, but it only lets us do parallelism in terms of loops. The only choice we make is whether a loop is to run in parallel.

- We can't determine how the loop iterations are divided up;
- we can't be sure which worker runs which iteration;
- workers cannot exchange data.

Using **parfor**, the individual workers are *anonymous*, and all the data are shared (or copied and returned).



Lecture: SPMD

The **SPMD** construct is like a very simplified version of **MPI**. There is one client process, supervising workers who cooperate on a single program. Each worker (sometimes also called a "lab") has an identifier, knows how many total workers there are, and can determine its behavior based on that identifier.

- each worker runs on a separate core (ideally);
- each worker uses a separate workspace;
- a common program is used;
- workers meet at synchronization points;
- the client program can examine or modify data on any worker;
- any two workers can communicate directly via messages.

SPMD: The SPMD Environment

MATLAB sets up one special agent called the client.

MATLAB sets up the requested number of workers, each with a copy of the program. Each worker "knows" it's a worker, and has access to two special functions:

- **numlabs()**, the number of workers;
- **labindex()**, a unique identifier between 1 and **numlabs()**.

The empty parentheses are usually dropped, but remember, these are functions, not variables!

If the client calls these functions, they both return the value 1! That's because when the client is running, the workers are not. The client could determine the number of workers available by

```
n = matlabpool ( 'size' )   or
n = pool_obj.NumWorkers
```

SPMD: The SPMD Command

The client and the workers share a single program in which some commands are delimited within blocks opening with **spmd** and closing with **end**.

The client executes commands up to the first **spmd** block, when it pauses. The workers execute the code in the block. Once they finish, the client resumes execution.

The client and each worker have separate workspaces, but it is possible for them to communicate and trade information.

The value of variables defined in the "client program" can be referenced by the workers, but not changed.

Variables defined by the workers can be referenced or changed by the client, but a special syntax is used to do this.

VirginiaTech

SPMD: How SPMD Workspaces Are Handled

```
                    Client       Worker 1    Worker 2
                    a  b  e  |  c  d  f  |  c  d  f
                    -------------------------------
a = 3;              3  -  -  |  -  -  -  |  -  -  -
b = 4;              3  4  -  |  -  -  -  |  -  -  -
spmd                         |           |
  c = labindex();   3  4  -  |  1  -  -  |  2  -  -
  d = c + a;        3  4  -  |  1  4  -  |  2  5  -
end                          |           |
e = a + d{1};       3  4  7  |  1  4  -  |  2  5  -
c{2} = 5;           3  4  7  |  1  4  -  |  5  6  -
spmd                         |           |
  f = c * b;        3  4  7  |  1  4  4  |  5  6 20
end
```

VirginiaTech

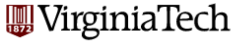
SPMD: When is Workspace Preserved?

A program can contain several **spmd** blocks. When execution of one block is completed, the workers pause, but they do not disappear and their workspace remains intact. A variable set in one **spmd** block will still have that value if another **spmd** block is encountered. Unless the *client* has changed it, as in our example. You can imagine the client and workers simply alternate execution. In MATLAB, variables defined in a function "disappear" once the function is exited. The same thing is true, in the same way, for a MATLAB program that calls a function containing **spmd** blocks. While inside the function, worker data is preserved from one block to another, but when the function is completed, the worker data defined there disappears, just as the regular MATLAB data does. It's not legal to nest an **smpd** block within another **spmd** block or within a **parfor** loop. Some additional limitations are discussed at

http://www.mathworks.com/help/distcomp/programming-tips_brukbnp-9.html?searchHighlight=nested+spmd

MATLAB Parallel Computing

- SPMD: Single Program, Multiple Data
- **QUAD Example**
- Distributed Arrays
- LBVP & FEM_2D_HEAT Examples
- IMAGE Example
- CONTRAST Example
- CONTRAST2: Messages
- Batch Computing
- Conclusion



QUAD: The Trapezoid Rule

Area of one trapezoid = average height * base.

QUAD: The Trapezoid Rule

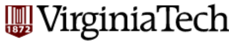
To estimate the area under a curve using one trapezoid, we write

$$\int_a^b f(x)\,dx \approx (\frac{1}{2}f(a) + \frac{1}{2}f(b)) * (b-a)$$

We can improve this estimate by using $n-1$ trapezoids defined by equally spaced points x_1 through x_n:

$$\int_a^b f(x)\,dx \approx (\frac{1}{2}f(x_1) + f(x_2) + ... + f(x_{n-1}) + \frac{1}{2}f(x_n)) * \frac{b-a}{n-1}$$

If we have several workers available, then each one can get a part of the interval to work on, and compute a trapezoid estimate there. By adding the estimates, we get an approximation to the integral of the function over the whole interval.



QUAD: Use the ID to assign work

To simplify things, we'll assume our original interval is [0,1], and we'll let each worker define *a* and *b* to mean the ends of its subinterval. If we have 4 workers, then worker number 3 will be assigned $[\frac{1}{2}, \frac{3}{4}]$.

To start our program, each worker figures out its interval:

```
fprintf ( 1, '  Set up the integration limits:\n' );

spmd
  a = ( labindex() - 1 ) / numlabs();
  b =   labindex()       / numlabs();
end
```

QUAD: One Name Must Reference Several Values

Each worker has its own workspace. It can "see" the variables on the client, but it usually doesn't know or care what is going on on the other workers.

Each worker defines **a** and **b** but stores *different values* there.

The client can "see" the workspace of all the workers. Since there are multiple values using the same name, the client must specify the index of the worker whose value it is interested in. Thus **a**$\{\mathbf{1}\}$ is how the client refers to the variable **a** on worker 1. The client can read or write this value.

MATLAB's name for this kind of variable, indexed using curly brackets, is a composite variable. The syntax is similar to a cell array.

The workers can "see" the client's variables and inherits a copy of their values, but cannot change the client's data.

QUAD: Dealing with Composite Variables

So in QUAD, each worker could print **a** and **b**:

```
spmd
  a = ( labindex() - 1 ) / numlabs();
  b =   labindex()       / numlabs();
  fprintf ( 1, '  A = %f, B = %f\n', a, b );
end
```

——— or the client could print them all ———

```
spmd
  a = ( labindex() - 1 ) / numlabs();
  b =   labindex()        / numlabs();
end
for i = 1 : 4   <-- "numlabs" wouldn't work here!
  fprintf ( 1, '  A = %f, B = %f\n', a{i}, b{i} );
end
```

VirginiaTech

QUAD: The Solution in 4 Parts

Each worker can now carry out its trapezoid computation:

```
spmd
  x = linspace ( a, b, n );
  fx = f ( x );    (Assume f can handle vector input.)
  quad_part = ( b - a ) / ( n - 1 ) *
    * ( 0.5 * fx(1) + sum(fx(2:n-1)) + 0.5 * fx(n) );
  fprintf ( 1, '  Partial approx %f\n', quad_part );
end
```

with result:

```
2   Partial approx 0.874676
4   Partial approx 0.567588
1   Partial approx 0.979915
3   Partial approx 0.719414
```

QUAD: Combining Partial Results

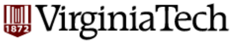
We really want one answer, the sum of all these approximations.

One way to do this is to gather the answers back on the client, and sum them:

```
quad = sum ( quad_part{1:4} );
fprintf ( 1, '  Approximation %f\n', quad );
```

with result:

```
Approximation 3.14159265
```



QUAD: Source Code for QUAD_FUN

```
function value = quad_fun ( n )

%*****************************************************************************80
%
%% QUAD_FUN demonstrates MATLAB's SPMD command for parallel programming.
%
%  Discussion:
%
%    A block of statements that begin with the SPMD command are carried
%    out in parallel over all the LAB's.  Each LAB has a unique value
%    of LABINDEX, between 1 and NUMLABS.
%
%    Values computed by each LAB are stored in a composite variable.
%    The client copy of MATLAB can access these values by using an index.
%
%  Licensing:
%
%    This code is distributed under the GNU LGPL license.
%
%  Modified:
%
%    18 August 2009
%
%  Author:
%
%    John Burkardt
%
%  Parameters:
%
%    Input, integer N, the number of points to use in each subinterval.
%
%    Output, real VALUE, the estimate for the integral.
%
```

VirginiaTech

QUAD: Source Code for QUAD_FUN (cont'd)

```
  fprintf ( 1, '\n' );
  fprintf ( 1, 'QUAD_FUN\n' );
  fprintf ( 1, '  Demonstrate the use of MATLAB''s SPMD command\n' );
  fprintf ( 1, '  to carry out a parallel computation.\n' );
%
%  The entire integral goes from 0 to 1.
%  Each LAB, from 1 to NUMLABS, computes its subinterval [A,B].
%
  fprintf ( 1, '\n' );

  spmd
    a = ( labindex - 1 ) / numlabs;
    b =   labindex       / numlabs;
    fprintf ( 1, '  Lab %d works on [%f,%f].\n', labindex, a, b );
  end
%
%  Each LAB now estimates the integral, using N points.
%
  fprintf ( 1, '\n' );

  spmd
    if ( n == 1 )
      quad_local = ( b - a ) * f ( ( a + b ) / 2 );
    else
      x = linspace ( a, b, n );
      fx = f ( x );
      quad_local = ( b - a ) * ( fx(1) + 2 * sum ( fx(2:n-1) ) + fx(n) ) ...
        / (2.0 * ( n - 1 ));
    end
    fprintf ( 1, '  Lab %d computed approximation %f\n', labindex, quad_local );
  end
```

VirginiaTech

QUAD: Source Code for QUAD_FUN (cont'd)

```
%
%  The variable quad_local has been computed by each LAB.
%  Variables computed inside an SPMD statement are stored as "composite"
%  variables, similar to MATLAB's cell arrays.  Outside of an SPMD
%  statement, composite variable values are accessible to the
%  client copy of MATLAB by index.
%
%  The GPLUS function adds the individual values, returning
%  the sum to each LAB -- so QUAD is also a composite value,
%  but all its values are equal.
%
  spmd
    quad = gplus ( quad_local ); % Note use of a gop
  end
%
%  Outside of an SPMD statement, the client copy of MATLAB can
%  access any entry in a composite variable by indexing it.
%
  value = quad{1};

  fprintf ( 1, '\n' );
  fprintf ( 1, '  Result of quadrature calculation:\n' );
  fprintf ( 1, '  Estimate QUAD = %24.16f\n', value );
  fprintf ( 1, '  Exact value   = %24.16f\n', pi );
  fprintf ( 1, '  Error          = %e\n', abs ( value - pi ) );
  fprintf ( 1, '\n' );
  fprintf ( 1, 'QUAD_FUN\n' );
  fprintf ( 1, '  Normal end of execution.\n' );

  return
end
```

VirginiaTech

- SPMD: Single Program, Multiple Data
- QUAD Example
- **Distributed Arrays**
- LBVP & FEM_2D_HEAT Examples
- IMAGE Example
- CONTRAST Example
- CONTRAST2: Messages
- Batch Computing
- Conclusion

DISTRIBUTED: The Client Can Distribute

If the client process has a 300x400 array called **a**, and there are 4 SPMD workers, then the simple command

```
ad = distributed ( a );
```

distributes the elements of **a** by columns:

```
          Worker 1   Worker 2   Worker 3   Worker 4
Col:     1:100 | 101:200 | 201:300 | 301:400 ]
Row
  1 [ a b c d | e f g h | i j k l | m n o p ]
  2 [ A B C D | E F G H | I J K L | M N O P ]
... [ * * * * | * * * * | * * * * | * * * * ]
300 [ 1 2 3 4 | 5 6 7 8 | 9 0 1 2 | 3 4 5 6 ]
```

By default, the last dimension is used for distribution.

VirginiaTech

DISTRIBUTED: Workers Can Get Their Part

Once the client has distributed the matrix by the command

```
ad = distributed ( a );
```

then each worker can make a local variable containing its part:

```
spmd
  al = getLocalPart ( ad );
  [ ml, nl ] = size ( al );
end
```

On worker 3, [ml, nl] = (300, 100), and **al** is

```
[ i j k l ]
[ I J K L ]
[ * * * * ]
[ 9 0 1 2 ]
```

Notice that local and global column indices will differ!

VirginiaTech

DISTRIBUTED: The Client Can Collect Results

The client can access any worker's local part by using curly brackets. Thus it could copy what's on worker 3 by

```
worker3_array = al{3};
```

However, it's likely that the client simply wants to collect all the parts and put them back into one normal MATLAB array. If the local arrays are simply column-sections of a 2D array:

```
a2 = [ al{:} ]
```

Suppose we had a 3D array whose third dimension was 3, and we had distributed it as 3 2D arrays. To collect it back:

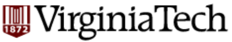
```
a2 = al{1};
a2(:,:,2) = al{2};
a2(:,:,3) = al{3};
```

DISTRIBUTED: Methods to Gather Data

Instead of having an array created on the client and distributed to the workers, it is possible to have a distributed array constructed by having each worker build its piece. The result is still a distributed array, but when building it, we say we are building a **codistributed** array.

Codistributing the creation of an array has several advantages:

1. The array might be too large to build entirely on one core (or processor);
2. The array is built faster in parallel;
3. You avoid the communication cost of distributing it.



DISTRIBUTED: Accessing Distributed Arrays

The command **al = getLocalPart (ad)** makes a local copy of the part of the distributed array residing on each worker. Although the workers could reference the distributed array directly, the local part has some uses:

- references to a local array are faster;
- the worker may only need to operate on the local part; then it's easier to write **al** than to specify **ad** indexed by the appropriate subranges.

The client can copy a distributed array into a "normal" array stored entirely in its memory space by the command

```
a = gather ( ad );
```

or the client can access and concatenate local parts.

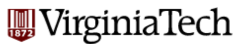
DISTRIBUTED: Conjugate Gradient Setup

Because many MATLAB operators and functions can automatically detect and deal with distributed data, it is possible to write programs that carry out sophisticated algorithms in which the computation never explicitly worries about where the data is!

The only tricky part is distributing the data initially, or gathering the results at the end.

Let us look at a conjugate gradient code which has been modified to deal with distributed data.

Before this code is executed, we assume the user has requested some number of workers, using the interactive **parpool** or indirect **batch** command. (**R2013b** and later can do this automatically).



DISTRIBUTED: Conjugate Gradient Setup

```
% Script to invoke conjugate gradient solution
% for sparse distributed (or not) array
%

    N      = 1000;
    nnz    = 5000;
    rc     = 1/10;   % reciprocal condition number

    A = sprandsym(N, nnz/N^2, rc, 1); % symmetric, positive definite
    A = distributed(A);%A = distributed.sprandsym() is not available

    b = sum(A, 2);
%   fprintf( 1, '\n isdistributed(b): %2i  \n', isdistributed(b) );

    [x, e_norm ] = cg_emc( A, b);

    fprintf( 1, 'Error residual :   %8.4e \n', e_norm{1});

    np = 10;
    fprintf( 1, 'First few x values: \n');
    fprintf( 1, ' x( %02i )  = %8.4e \n', [ 1:np ; gather(x(1:np))']);
```

`sprandsym` sets up a sparse random symmetric array **A**.

`distributed` ‘casts’ **A** to a distributed array on the workers.

Why do we write `e_norm{1}` & `gather(x)` ?

VirginiaTech

DISTRIBUTED: Conjugate Gradient Iteration

```
function [x, resnrm ] = cg_emc( A, b, x0, tol, itmax)
% Conjugate gradient iteration for A x = b,
% (from Gill, Murray and Wright, p 147)

% Possibly supply missing input parameters (omitted)
  spmd
% initialization
     p     = codistributed.zeros(size(x0));
     beta  = 0;
     r     = A*x0 - b;
     rknrm= r'*r;
     x     = x0;
     iter  = 0;
%  CG loop
     while 1
         p      = beta*p - r;
         tmp    = A*p;
         alpha  = rknrm/(p'*tmp);
         x      = x + alpha*p;
         r      = r + alpha*tmp;
         rkpnrm= r'*r;
         beta   = rkpnrm/rknrm;
         rknrm  = rkpnrm;
         iter   = iter + 1;
         resnrm= norm(A*x - b);
         if iter >= itmax || resnrm <= tol
             break
         end
     end % while 1
  end % spmd

end % function
```

DISTRIBUTED: Comment

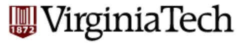
In `cg_emc.m`, we can remove the `spmd` block and simply invoke **distributed()**; the operational commands don't change.

There are several comments worth making:

- The communication overhead can be severely increased
- Not all MATLAB operators have been extended to work with distributed memory. In particular, (the last time we asked), the backslash or "linear solve" operator **x=A\b** can't be used yet for sparse distributed arrays.
- Getting "real" data (as opposed to matrices full of random numbers) properly distributed across multiple processors involves more choices and more thought than is suggested by the `conjugate gradient` example !

- **SPMD: Single Program, Multiple Data**
- QUAD Example
- Distributed Arrays
- **LBVP & FEM_2D_HEAT Examples**
- IMAGE Example
- CONTRAST Example
- CONTRAST2: Messages
- Batch Computing
- Conclusion

In the next example, we demonstrate a mixed approach wherein the `stiffness matrix` (**K**) is initially constructed as a **codistributed** array on the workers. Each worker then modifies its `localPart`, and also assembles the local contribution to the forcing term (**F**). The *local* forcing arrays are then used to build a **codistributed** array.



DISTRIBUTED: FD_LBVP_script

```
%% FD_LBVP_SCRIPT invokes the function fd_lbvp_fun
%
%  Licensing:
%
%    This code is distributed under the GNU LGPL license.
%
%  Author:
%
%%     Gene Cliff

      n = 100;              % grid parameter

% Define coefficient functions and boundary data for LBVP
      hndl_p = @(x)  0;
      c_q    = 4;           % positive for exact solution match
      hndl_q = @(x)  c_q;
      c_r    = -4;
      hndl_r = @(x) c_r*x; % linear for exact solution match

      alpha = 0;
      beta  = 2;

%  Invoke solver
      fprintf( 1, '\n Invoke fd_lbvp_fun \n');
      T  = fd_lbvp_fun(n, hndl_p, hndl_q, hndl_r, alpha, beta);

% gather the distributed solution on the client process

      Tg = gather(T); % When 'batch' finishes the 'pool' is closed
                      % and distributed data is lost
```

VirginiaTech

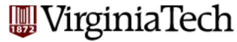
DISTRIBUTED: FD_LBVP code

```
function T =  fd_lbvp_fun(n, hndl_p, hndl_q, hndl_r, alpha, beta)
% Finite-difference approximation to the BVP
%  T''(x) =  p(x) T'(x) + q(x) T(x) + r(x),        0 \le x \le 1
%  with    T(0) = alpha,   T(1) = beta

%  Modified:
%
%     2 March 2012
%
%  Author:
%
%     Gene  Cliff

% We use a uniform grid with n interior grid points
% From Numerical Analysis, Burden & Faires, 2001, \S~11.3

  h = 1/(n+1); ho2 = h/2; h2 = h*h;
  spmd
     A    = codistributed.zeros(n,n);
     locP = getLocalPart(codistributed.colon(1, n)); %index vals on lab
     locP = locP(:);% make it a column array

% Loop over columns entering unity above/below the diagonal entry
% along with 2 plus the appropriate q function values
% Note that columns 1 and n are exceptions
     for jj=locP(1):locP(end)
         % on the diagonal
         A(jj, jj) =    2 + h2*feval(hndl_q, jj*h);
         % above the diagonal
        if jj ~= 1;  A(jj-1, jj) = -1+ho2*feval(hndl_p, (jj-1)*h); end
        % below the diagonal
        if jj ~= n;  A(jj+1, jj) = -1+ho2*feval(hndl_p,    jj  *h); end
     end
```

VirginiaTech

DISTRIBUTED: FD_LBVP code (cont'd)

```
    locF   = -h2*feval(hndl_r, locP*h); % hndl_r okay for vector input

    if labindex() == 1
        locF( 1 ) = locF( 1 ) + alpha*(1+ho2*feval(hndl_p,  h ));
    end
    if labindex() == numlabs();
        locF(end) = locF(end) +  beta*(1-ho2*feval(hndl_p, 1-h));
    end

%   codist = codistributor1d(dim,        partition,          global_size);
    codist = codistributor1d(1,  codistributor1d.unsetPartition, [n, 1]);
    F      = codistributed.build(locF, codist); % distribute the array(s)
  end % spmd block

   T = A\F;   % mldivide is defined for codistributed arrays
```



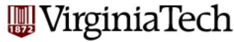
DISTRIBUTED: 2D Finite Element Heat Model

Next, we consider an example that combines SPMD and distributed data to solve a steady state heat equations in 2D, using the finite element method. Here we demonstrate a different strategy for assembling the required arrays.

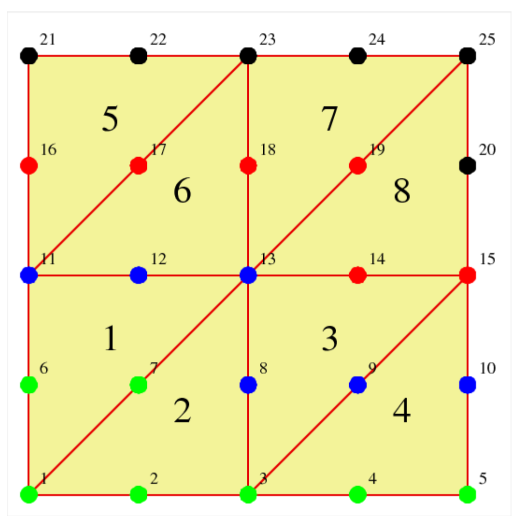
Each worker is assigned a subset of the finite element nodes. That worker is then responsible for constructing the columns of the (sparse) finite element matrix associated with those nodes.

Although the matrix is assembled in a distributed fashion, it has to be gathered back into a standard array before the linear system can be solved, because sparse linear systems can't be solved as a distributed array (yet).

This example is available as in the **fem_2D_heat** folder.



DISTRIBUTED: The Grid & Node Coloring for 4 labs



VirginiaTech

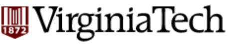
DISTRIBUTED: Finite Element System matrix

The discretized heat equation results in a linear system of the form

$$K\,z = F + b$$

where **K** is the stiffness matrix, **z** is the unknown finite element coefficients, **F** contains source terms and **b** accounts for boundary conditions.

In the parallel implementation, the system matrix **K** and the vectors **F** and **b** are distributed arrays. The default distribution of **K** by columns essentially associates each SPMD worker with a group of finite element nodes.



DISTRIBUTED: Finite Element System Matrix

To assemble the matrix, each worker loops over all elements. If element E contains *any* node associated with the worker, the worker computes the entire local stiffness matrix K. Columns of K associated with worker nodes are added to the local part of $\mathbf{K}$. The rest are discarded (which is OK, because they will also be computed and saved by the worker responsible for those nodes).

When element 5 is handled, the “blue”, “red” and “black” processors each compute K. But blue only updates column 11 of $\mathbf{K}$, red columns 16 and 17, and black columns 21, 22, and 23.

At the cost of some redundant computation, we avoid a lot of communication.

Assemble Codistributed Arrays - code fragment

```
    spmd
%
%  Set up codistributed structure
%
%  Column pointers and such for codistributed arrays.
%
     Vc   = codistributed.colon(1, n_equations);
     lP   = getLocalPart(Vc);
     lP_1= lP(1); lP_end = lP(end); %first and last columns of K on this lab
     co_dist_Vc = getCodistributor(Vc); dPM = co_dist_Vc.Partition;
   ...
% sparse arrays on each lab
%
     K_lab = sparse(n_equations, dPM(labindex));
   ...
%  Build the finite element matrices - Begin loop over elements
%
     for n_el=1:n_elements
       nodes_local     = e_conn(n_el,:);% which nodes are in this element
     % subset of nodes/columns on this lab
       lab_nodes_local = extract( nodes_local, lP_1, lP_end);
   ... if empty do nothing, else accumulate K_lab, etc end
     end % n_el
%
% Assemble the 'lab' parts in a codistributed format.
%  syntax for version R2009b
     codist_matrix = codistributor1d( 2, dPM, [n_equations, n_equations]);
     K = codistributed.build(K_lab, codist_matrix );

    end % spmd
```

DISTRIBUTED: 2D Heat Equation - The Results

- SPMD: Single Program, Multiple Data
- QUAD Example
- Distributed Arrays
- LBVP & FEM_2D_HEAT Examples
- **IMAGE Example**
- CONTRAST Example
- CONTRAST2: Messages
- Batch Computing
- Conclusion

IMAGE: Image Processing in Parallel

Here is a mysterious SPMD program to be run with 3 workers:

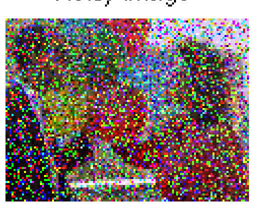
```
x = imread ('balloons.tif');

y = imnoise ( x, 'salt & pepper', 0.30 );

yd = distributed ( y );

spmd
  yl = getLocalPart ( yd );
  yl = medfilt2 ( yl, [ 3, 3 ] );
end

z(1:480,1:640,1) = yl{1};
z(1:480,1:640,2) = yl{2};
z(1:480,1:640,3) = yl{3};

figure;
subplot ( 1, 3, 1 );imshow ( x );title ( 'X' );
subplot ( 1, 3, 2 );imshow ( y );title ( 'Y' );
subplot ( 1, 3, 3 );imshow ( z );title ( 'Z' );
```

Without comments, what can you guess about this program?

IMAGE: Image → Noisy Image → Filtered Image

Original image

Noisy Image



Median Filtered Image

This filtering operation uses a 3x3 pixel neighborhood.
We could blend *all* the noise away with a larger neighborhood.

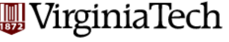


IMAGE: Image → Noisy Image → Filtered Image

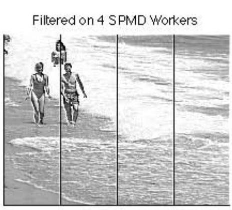
```
%  Read a color image, stored as 480x640x3 array.
%
  x = imread ('balloons.tif');
%
%  Create an image Y by adding "salt and pepper" noise to X.
%
  y = imnoise ( x, 'salt & pepper', 0.30 );
%
%  Make YD, a distributed version of Y.
%
  yd = distributed ( y );
%
%  Each worker works on its "local part", YL.
%
  spmd
    yl = getLocalPart ( yd );
    yl = medfilt2 ( yl, [ 3, 3 ] );
  end
%
%  The client retrieves the data from each worker.
%
  z(1:480,1:640,1) = yl{1};
  z(1:480,1:640,2) = yl{2};
  z(1:480,1:640,3) = yl{3};
%
%  Display the original, noisy, and filtered versions.
%
  figure;
  subplot ( 1, 3, 1 ); imshow ( x ); title ( 'Original image' );
  subplot ( 1, 3, 2 ); imshow ( y ); title ( 'Noisy Image' );
  subplot ( 1, 3, 3 ); imshow ( z ); title ( 'Median Filtered Image' );
```

- SPMD: Single Program, Multiple Data
- QUAD Example
- Distributed Arrays
- LBVP & FEM_2D_HEAT Examples
- IMAGE Example
- **CONTRAST Example**
- CONTRAST2: Messages
- Batch Computing
- Conclusion

CONTRAST: Image → Contrast Enhancement → Image2

```
%
% Get 4 SPMD workers
%
  parpool open 4
%
% Read an image
%
  x = imageread( 'surfsup.tif' );
%
% Since the image is black and white, it will be distributed by columns
%
  xd = distributed(x);
%
%  Each worker enhances the contrast on its portion of the picture
%
  spmd
      xl = getLocalPart(xd);
      xl = nlfilter( xl, [3, 3], @contrast_enhance);
      xl = uint8(xl);
  end
%
% Concatenate the submatrices to assemble the whole image
%
  xf_spmd = [ xl{:} ];

  parpool/delete
```

CONTRAST: Image → Contrast Enhancement → Image2



When a filtering operation is done on the client, we get picture 2. The same operation, divided among 4 workers, gives us picture 3. What went wrong?

VirginiaTech

CONTRAST: Image → Contrast Enhancement → Image2

Each pixel has had its contrast enhanced. That is, we compute the average over a 3x3 neighborhood, and then increase the difference between the center pixel and this average. Doing this for each pixel sharpens the contrast.

```
+-----+-----+-----+
| P11 | P12 | P13 |
+-----+-----+-----+
| P21 | P22 | P23 |
+-----+-----+-----+
| P31 | P32 | P33 |
+-----+-----+-----+


P22 <- C * P22 + ( 1 - C ) * Average
with   C > 1 (specified)
```

CONTRAST: Image → Contrast Enhancement → Image2

When the image is divided by columns among the workers, artificial internal boundaries are created. The **nlfilter** algorithm turns any pixel lying along the boundary to white. (The same thing happened on the client, but we didn't notice!)

```
        Worker 1                Worker 2
   +-----+-----+-----+     +-----+-----+-----+     +----
   | P11 | P12 | P13 |     | P14 | P15 | P16 |     | P17
   +-----+-----+-----+     +-----+-----+-----+     +----
   | P21 | P22 | P23 |     | P24 | P25 | P26 |     | P27
   +-----+-----+-----+     +-----+-----+-----+     +----
   | P31 | P32 | P33 |     | P34 | P35 | P36 |     | P37
   +-----+-----+-----+     +-----+-----+-----+     +----
   | P41 | P42 | P43 |     | P44 | P45 | P46 |     | P47
   +-----+-----+-----+     +-----+-----+-----+     +----
```

Dividing up the data has created undesirable artifacts!

VirginiaTech

CONTRAST: Image → Contrast Enhancement → Image2

The result is spurious lines on the processed image.

MATLAB Parallel Computing

- SPMD: Single Program, Multiple Data
- QUAD Example
- Distributed Arrays
- LBVP & FEM_2D_HEAT Examples
- IMAGE Example
- CONTRAST Example
- **CONTRAST2: Messages**
- Batch Computing
- Conclusion

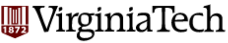
CONTRAST2: Workers Need to Communicate

The spurious lines would disappear if each worker could just be allowed to peek at the last column of data from the previous worker, and the first column of data from the next worker.

Just as in MPI, MATLAB includes commands that allow workers to exchange data.

The command we would like to use is **labSendReceive()** which controls the simultaneous transmission of data from all the workers.

```
data_received = labSendReceive ( to, from, data_sent );
```



CONTRAST2: Whom Do I Want to Communicate With?

```
spmd

  xl = getLocalPart ( xd );

  if ( labindex() ~= 1 )
    previous = labindex() - 1;
  else
    previous = numlabs();
  end

  if ( labindex() ~= numlabs())
    next = labindex() + 1;
  else
    next = 1;
  end
```

CONTRAST2: First Column Left, Last Column Right

```
column = labSendReceive ( previous, next, xl(:,1) );

if ( labindex() < numlabs() )
  xl = [ xl, column ];
end

column = labSendReceive ( next, previous, xl(:,end) );

if ( 1 < labindex() )
  xl = [ column, xl ];
end
```

VirginiaTech

CONTRAST2: Filter, then Discard Extra Columns

```
  xl = nlfilter ( xl, [3,3], @enhance_contrast );

  if ( labindex() < numlabs() )
    xl = xl(:,1:end-1);
  end

  if ( 1 < labindex() )
    xl = xl(:,2:end);
  end

  xl = uint8 ( xl );

end
```

VirginiaTech

CONTRAST2: Image $\rightarrow$ Enhancement $\rightarrow$ Image2

Original image

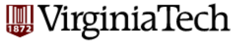
Filtered on Client

Filtered on 4 SPMD Workers

Four SPMD workers operated on columns of this image.
Communication was allowed using **labSendReceive**.

MATLAB Parallel Computing

- SPMD: Single Program, Multiple Data
- QUAD Example
- Distributed Arrays
- LBVP & FEM_2D_HEAT Examples
- IMAGE Example
- CONTRAST Example
- CONTRAST2: Messages
- **Batch Computing**
- Conclusion



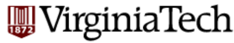
BATCH: Indirect Execution

We can run quick, local interactive jobs using the **matlabpool** or **parpool** command to get parallel workers.

The **batch** command is an alternative which allows you to execute a MATLAB script (using either **parfor** or **spmd** statements) in the background on your desktop...or on a remote machine.

The **batch** command includes a **matlabpool** or **pool** argument that allows you to request a given number of workers.

For remote jobs, the number of cores or processors you are asking for is the matlabpool **plus one**, because of the client.



BATCH: Contrast2

Running contrast2 on NewRiver and locally:

```
% Run on NewRiver
job = batch ( 'contrast2_script', ...
  'Profile', 'newriver_R2015b', ...
  'CaptureDiary', true, ...
  'AttachedFiles', { 'contrast2_fun', 'contrast_enhance', 'surfsup.tif' }, ...
  'CurrentDirectory', '.', ...
  'pool', n );

% Run locally
x = imread ( 'surfsup.tif' );
xf = nlfilter ( x, [3,3], @contrast_enhance );
xf = uint8 ( xf );

% Wait for NewRiver job to complete
wait ( job );
% Load results from NewRiver job
load ( job );
```

BATCH: Contrast2

Notes:

- We need to include both scripts and the input file `surfsup.tif` in the `AttachedFiles` flag
- We can do some work before issuing `wait()`
- We leverage two kinds of parallelism:
  - Parallel (using spmd) on an ARC Cluster
  - Run locally while job is running on an ARC Cluster

- SPMD: Single Program, Multiple Data
- QUAD Example
- Distributed Arrays
- LBVP & FEM_2D_HEAT Examples
- IMAGE Example
- CONTRAST Example
- CONTRAST2: Messages
- Batch Computing
- **Conclusion**

CONCLUSION: Summary of Examples

The QUAD example showed a simple problem that could be done as easily with SPMD as with PARFOR. We just needed to learn about composite variables!

The CONJUGATE GRADIENT example showed that many MATLAB operations work for distributed arrays, a kind of array storage scheme associated with SPMD.

The LBVP & FEM_2D_HEAT examples show how to construct local arrays and assemble these to **codistributed** arrays. This enables treatment of very large problems.

The IMAGE and CONTRAST examples showed us problems which can be broken up into subproblems to be dealt with by SPMD workers. We also saw that sometimes it is necessary for these workers to communicate, using a simple message-passing system.

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 this email list, send a blank email to

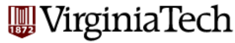
`mathworks-g+subscribe@vt.edu`

The subject and body of the message should both be empty.

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 ARC website at

  `http://www.arc.vt.edu/matlab#resources`

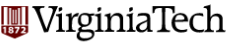


AFTERWORD: **PMODE**

PMODE allows interactive parallel execution of MATLAB commands. PMODE achieves this by defining and submitting a parallel job, and it opens a Parallel Command Window connected to the labs running the job. The labs receive commands entered in the Parallel Command Window, process them, and send the command output back to the Parallel Command Window.

`pmode start 'local' 2` will initiate **pmode**; `pmode exit` will delete the parallel job and end the **pmode** session

This may be a useful way to experiment with computations on the **labs**.



Please complete the evaluation form

Thanks