Parallel MATLAB at FSU: PARFOR and SPMD

John Burkardt
Department of Scientific Computing
Florida State University

.....

1:30 - 2:30
Thursday, 31 March 2011
499 Dirac Science Library

.....

http://people.sc.fsu.edu/~jburkardt/presentations/...
matlab_parallel_2011_fsu.pdf

- **Introduction**
- PARFOR: a Parallel FOR Loop
- QUAD Example (PARFOR)
- ODE Example (PARFOR)
- SPMD: Single Program, Multiple Data
- QUAD Example (SPMD)
- DISTANCE Example (SPMD)
- Conclusion

INTRO: Why Parallel Programming?

Why are we talking about parallel programming in the first place?

Many people have **no need** for parallel programming.
However, that assumes that they are:

- satisfied with the speed of their programs (forever!);
- not interested in solving problems with larger data sets (ever!);
- not interested in rewriting their programs.

At one time, I might also have added:

- *not willing to buy special parallel hardware,*

but most new desktops and laptops contain dual or quadcore processors, it's not hard to find 12 cores systems, and the number of cores will increase in the future.

INTRO: MultiCore Requires Parallel Programming

At one time, researchers needing faster execution for bigger problems simply bought the latest, fastest (sequential) processor.

Supercomputers of the past, such as the legendary series of Cray computers, were simply enormously expensive machines for feeding and cleaning up after a single really fast and powerful processor.

But the speed of a single processor has reached a permanent ceiling of 4 GigaHertz and will never get faster.

INTRO: MultiCore Requires Parallel Programming

To ensure a future for High Performance Computing:

- processor manufacturers have developed multicore systems;
- network companies created fast interprocessor connections;
- research universities have assembled computer clusters capable of running a single program;
- language committees have developed OpenMP and MPI for C and FORTRAN applications;
- researchers have developed new parallel algorithms;
- the MathWorks has developed parallel options in MATLAB.

INTRO: Parallel MATLAB

MATLAB's *Parallel Computing Toolbox* or **PCT** runs on a user's desktop, and can take advantage of up to 8 cores there.

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

The FSU HPC facility has a cluster of 526 compute nodes or "processors" controlling 2688 cores; the current license for MATLAB allows a program to run on up to 16 cores simultaneously there. (This limit could be increased if demand justified it.)

INTRO: Who Can Access Parallel MATLAB?

Anyone can do simple experiments with parallel MATLAB on their own desktop machine (if they have the Parallel Computing Toolbox);

They can do similar experiments on the Scientific Computing Department's hallway machines, although these only have two cores.

Any FSU researcher can request an account on the FSU HPC system, and hence run parallel MATLAB on 16 cores.

INTRO: How Does One Run Parallel MATLAB?

To run parallel MATLAB *on your desktop* involves:

- setting up the run;
- supplying a program which includes parallel commands.

and in some cases, both these steps are very simple.

As you can imagine, running on the cluster is a little more involved.

You set up the run by calling a command that submits your job to a queue. For short jobs, you might wait for the results, but for longer jobs, you actually log out and come back later to get your results.

This is one reason why it's helpful to do initial small experiments on a desktop machine.

INTRO: PARFOR and SPMD

In this talk, I will outline:

- two ways to create a parallel program in MATLAB
- the process of running a parallel program on desktop machine;
- how to run the same program on the FSU HPC cluster.
- what kinds of speedup (or slowdowns!) you can expect.

INTRO: PARFOR and SPMD

We will look at two ways to write a parallel MATLAB program:

- suitable **for** loops can be made into **parfor** loops;
- the **spmd** statement can define cooperating synchronized processing;

parfor approach is a simple way of making FOR loops run in parallel, and is similar to OpenMP.

spmd allows you to design almost any kind of parallel computation; it is powerful, but requires rethinking the program and data. It is similar to MPI.

- Introduction
- **PARFOR: a Parallel FOR Loop**
- QUAD Example (PARFOR)
- ODE Example (PARFOR)
- SPMD: Single Program, Multiple Data
- QUAD Example (SPMD)
- DISTANCE Example (SPMD)
- Conclusion

PARFOR: a Parallel FOR Command

parfor is MATLAB's simplest parallel programming command.

parfor replaces the **for** command in cases where a loop can be parallelized - *(and what does that mean?)*

The loop iterations are divided up among different "workers", executed in an unknown order, and the results gathered back to the main copy of MATLAB.

We think of a **for** loop as a small item of a program. It could, however, be one of the outermost statements, as in some kind of Monte Carlo calculation.

PARFOR: Computational Limitations

Because parallel loop iterations are carried out in an unknown order, all required data must exist before the loop begins.

An iteration cannot rely on results from a previous iteration.

```
x(1) = 0.0;
for i = 2 : n
  x(i) = x(i-1) + dt * f ( x(i), t );
end
```

PARFOR: Data Limitations

If any arrays are referenced by the loop iteration index, then this must be done in such a way that the array can be "sliced", that is, each reference array entry can be assigned to a unique loop iteration.

Such arrays are divided up among the workers, operated on, and put together afterwards.

```
parfor i = 2 : n
  a(i,2) = b(i+1) + a(i,2) * func ( c(i-3) ) + d(j);
end
```

So iteration 17, for example, gets A(17,2), B(18), C(14), and nobody else has that data during the loop.

PARFOR: Data Limitations

MATLAB's slicing requirement for **parfor** loops makes some computations impossible or awkward.

The loop may be *logically* parallelizable; it's just that MATLAB's parallel feature cannot divide the data in the way it wants.

The following loop wants to approximate the second derivative while invoking **parfor** to do the computation in parallel:

```
parfor i = 2 : n - 1
  d2(i) = ( a(i-1) - 2 * a(i) + a(i+1) ) / ( 2 * dx );
end
```

Because the array **a()** is indexed by **i-1**, **i**, and **i+1**, the array **a** cannot be sliced, and the parfor becomes illegal!

PARFOR: Desktop Execution

Let's assume that we have a MATLAB code that uses **parfor** in a legal way.

Let's also assume that the code is written as a MATLAB function, of the form

```
function [ y, z ] = calculator ( n, x )
```

and stored as the file **calculator.m**.

How can we execute these commands in parallel?

PARFOR: Interactive Execution

We have to start up an interactive MATLAB session, of course.

If we are interested in parallel execution, we must now request workers, using the **matlabpool** command. A typical form of this command is

```
matlabpool open local 4
```

or

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

The word **local** indicates that we are planning to run on the local machine, using the cores on this machine as the workers.

The value "4" is the number of workers you are asking for. It can be up to 8 on a local machine (but your "local" configuration may need to be adjusted from its default limit.)

PARFOR: Interactive Execution

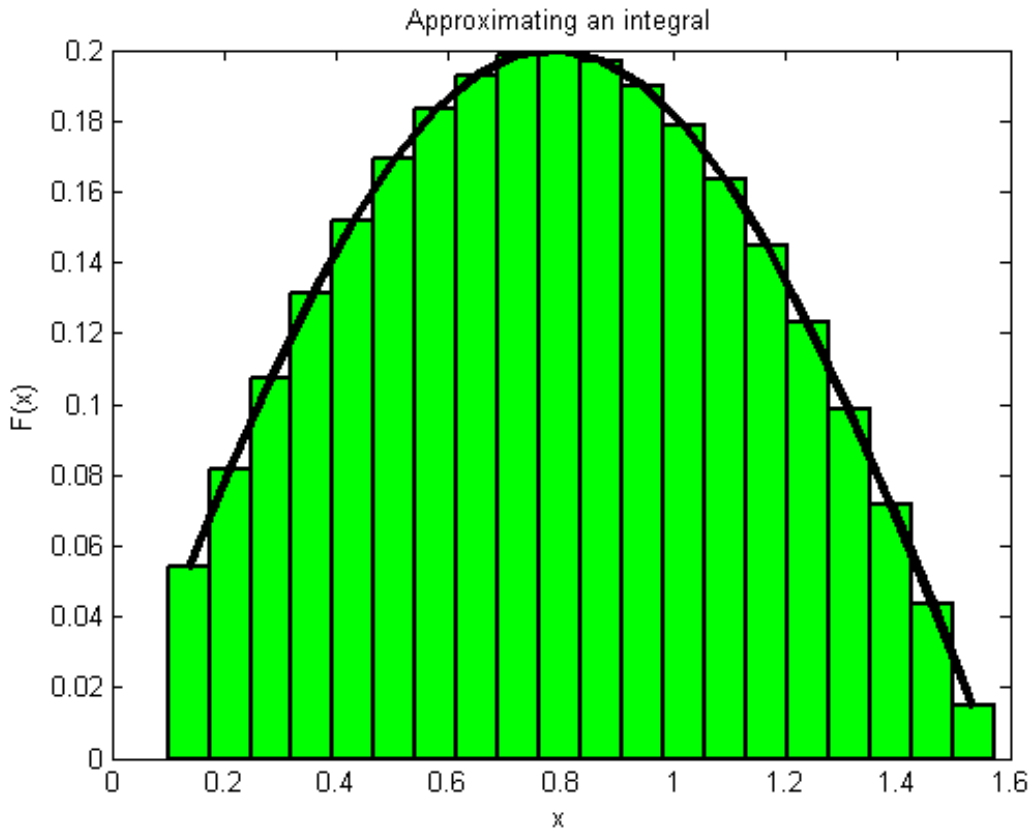
A MATLAB session in which parallel programming is involved looks the same as a regular session. But whenever a **parfor** command is encountered, the parallel workers are called upon.

```
n = 10000;
x = 2.5;

matlabpool open local 4  <-- Only needed once
[ y, z ] = calculator ( n, x );
...any further MATLAB commands you like ...

matlabpool close          <-- How to release the workers.
```

You only need to close the matlabpool if you want to issue another open command with more workers.

- Introduction
- PARFOR: a Parallel FOR Loop
- **QUAD Example (PARFOR)**
- ODE Example (PARFOR)
- SPMD: Single Program, Multiple Data
- QUAD Example (SPMD)
- DISTANCE Example (SPMD)
- Conclusion

QUAD: Estimating an Integral



QUAD: The QUAD_FUN Function

```
function q = quad_fun ( n, a, b )

  q = 0.0;
  dx = ( b - a ) / ( n - 1 );

  for i = 1 : n
    x = a + ( i - 1 ) * dx;
    fx = x^3 + sin ( x );
    q = q + fx;
  end

  q = q * ( b - a ) / n;

  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;
  dx = ( b - a ) / ( n - 1 );

  parfor i = 1 : n
    x = a + ( i - 1 ) * dx;
    fx = x^3 + sin ( x );
    q = q + fx;
  end

  q = q * ( b - a ) / n;

  return
end
```

QUAD: Comments

The function begins as one thread of execution, the client.

At the **parfor**, the client pauses, and the workers start up.

Each worker is assigned some iterations of the loop, and given any necessary input data. It computes its results which are returned to the client.

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.

QUAD: Reduction Operations

I stated that a loop cannot be parallelized if the results of one iteration are needed in order for the next iteration to carry on.

But this seems to be violated by the statement $\mathbf{q = q + fx}$.

To compute the value of **q** on the 10th iteration, don't we need the value from the 9th iteration?

MATLAB can see that **q** is not used in the loop except to accumulate a sum. It recognizes this as a reduction operation, and automatically parallelizes that calculation.

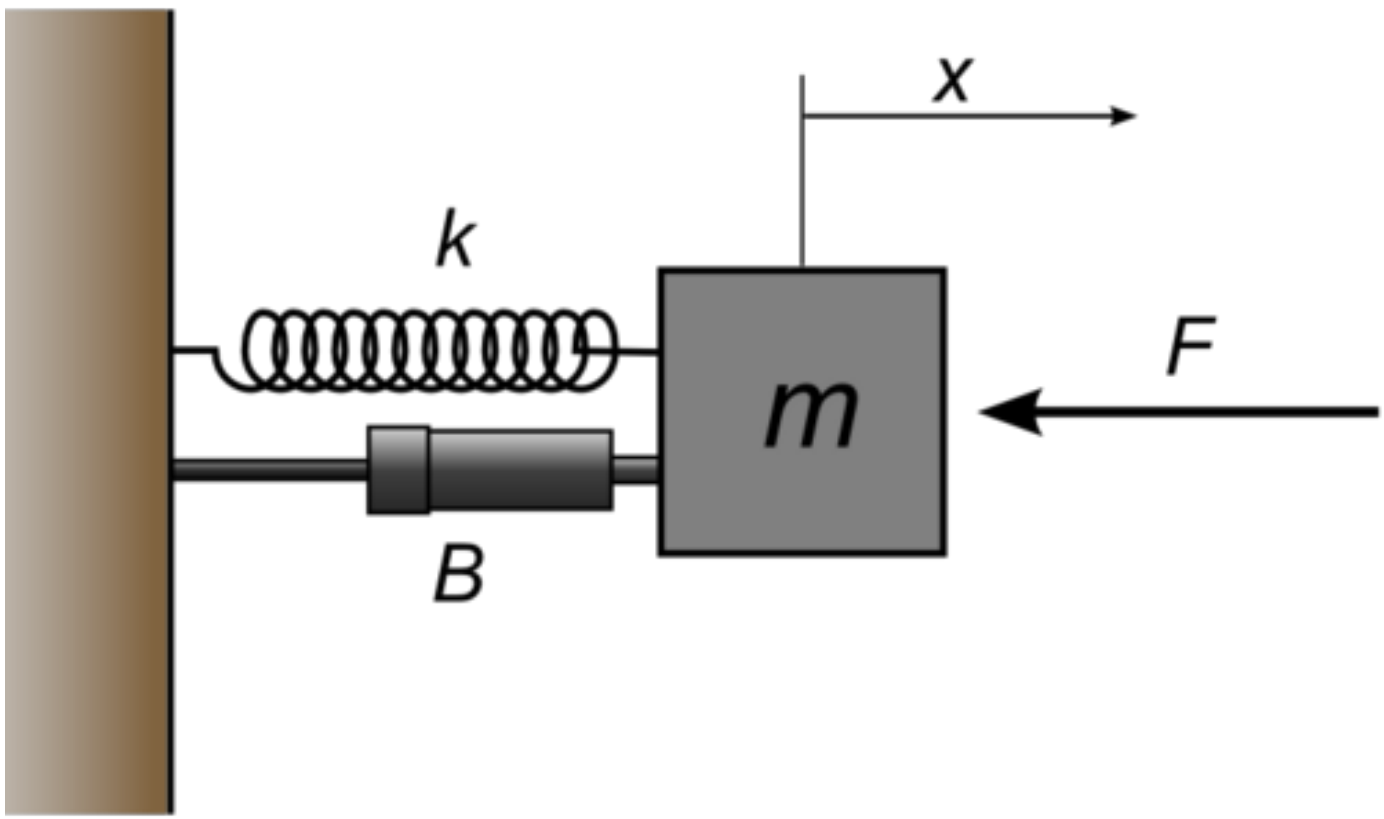
Such admissible reduction operations include iterated sums, products, logical sums and products, maximum and minimum calculations.

- Introduction
- PARFOR: a Parallel FOR Loop
- QUAD Example (PARFOR)
- **ODE Example (PARFOR)**
- SPMD: Single Program, Multiple Data
- QUAD Example (SPMD)
- DISTANCE Example (SPMD)
- Conclusion

ODE: A Parameterized Problem

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

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



ODE: A Parameterized Problem

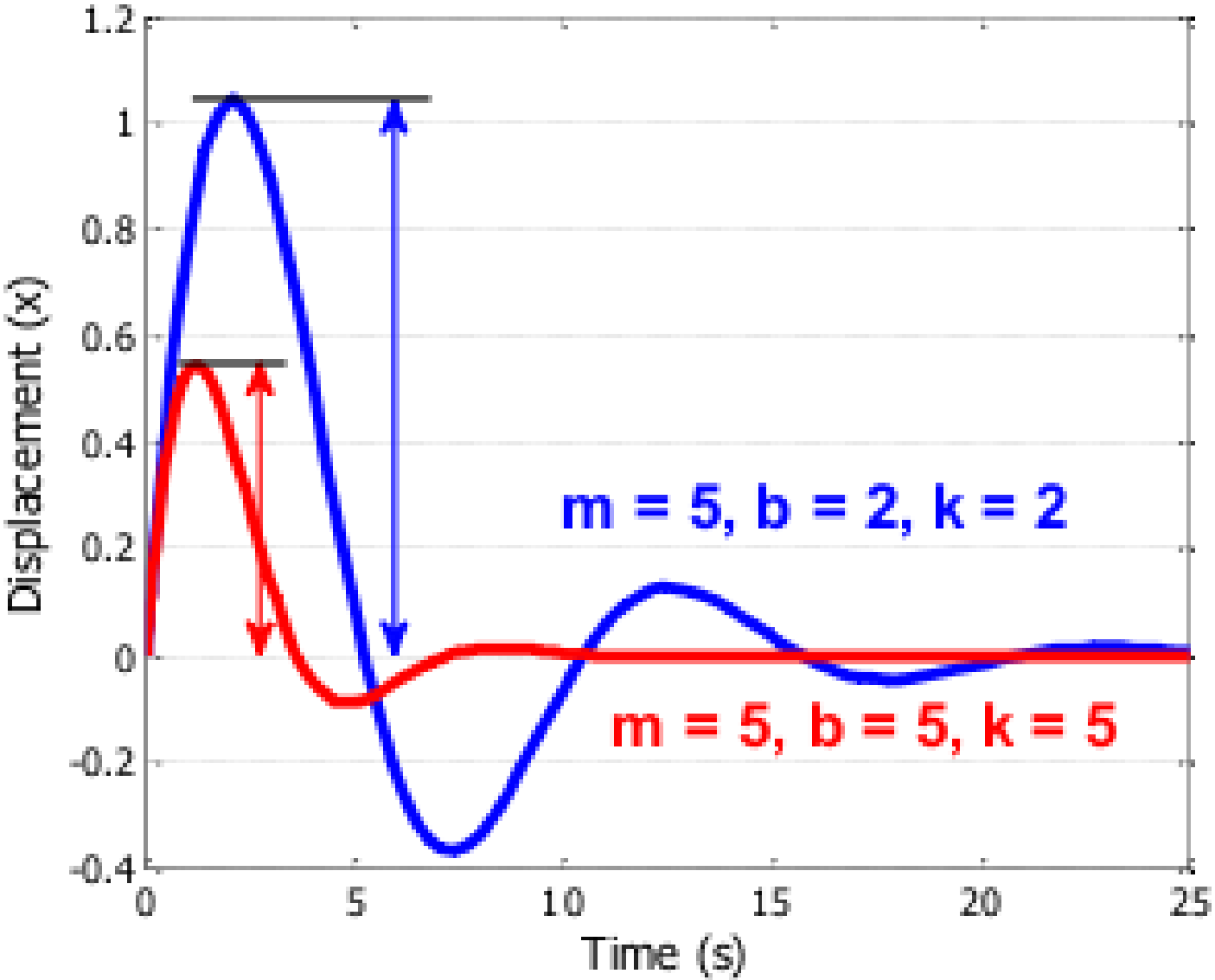
Solutions of this equation describe oscillatory behavior; $x(t)$ swings back and forth, in a pattern determined by the parameters m, b, k, f and the initial conditions.

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

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

So our computation might involve creating a plot of $xmax(b, k)$, and we could express this as a parallelizable loop over a list of pairs of b and k values.

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: ODE_FUN Computes the Peak Values

```
function peakVals = ode_fun ( bVals, kVals )

  [ kGrid, bGrid ] = meshgrid ( bVals, kVals );
  peakVals = nan ( size ( kGrid ) );
  m = 5.0;

  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
  return
end
```

ODE: ODE_DISPLAY Plots the Results

The function **ode_display.m** calls **surf** for a 3D plot, where **bVals** and **kVals** are X and Y, and **peakVals** plays the role of Z.

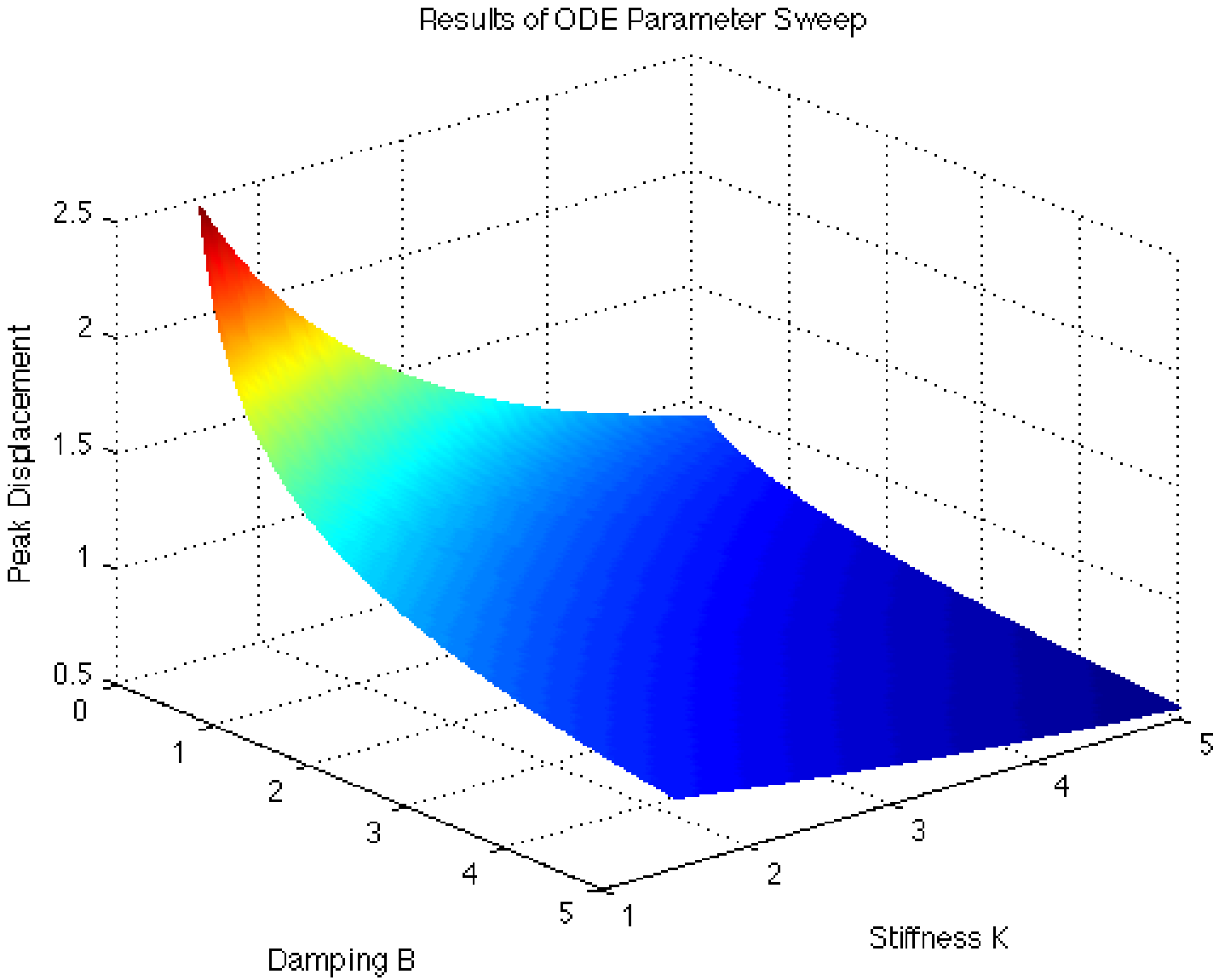
```
function ode_display ( bVals, kVals, peakVals )

  figure ( 1 );

  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: Interactive Execution

To run the program on our desktop, we set up the input, get the workers, and call the functions:

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

matlabpool open local 4

peakVals = ode_fun ( bVals, kVals );  <-- parallel

ode_display ( bVals, kVals, peakVals );
```

ODE: A Parameterized Problem



ODE: FSU HPC Cluster Execution

Now we will discuss the issue of running a parallel MATLAB program on the FSU HPC cluster, using the ODE program as our example.

It should be clear that we need to run **ode_fun** on the cluster, in order to get parallelism, but that we almost certainly don't want to run the **ode_display** program that way. We want to look at the results, zoom in on them, interactively explore, and so on.

And that means that somehow we have to get the **ode_fun** program and its input into the cluster, and then retrieve the results for interactive viewing on the cluster front-ends, or on our desktop.

ODE: fsuClusterMatlab

On the HPC cluster, we invoke the **fsuClusterMatlab** command:

```
bVals = 0.1 : 0.05 : 5;
kVals = 1.5 : 0.05 : 5;
results = fsuClusterMatlab([],[],'p','w',4,...
  @ode_fun, { bVals, kVals } )
```

- **[ ]** allows us to specify an output directory;
- **[ ]** allows us to specify queue arguments;
- **'p'** means this is a pool or parfor job;
- **'w'** means our MATLAB session waits til the job has run;
- **4** is the number of workers we request;
- **@ode_fun** names the function to evaluate;
- **bVals, kVals** are the input to **ode_fun**.

ODE: The RESULTS of fsuClusterMatlab

fsuClusterMatlab is a function, and it returns the output from **ode_fun** as a cell array which we have called **results**.

You need to copy items out of **results** in order to process them.

The first output argument is retrieved by copying **results**$\{\mathbf{1}\}$.

For our **ode_fun** function, that's all we will need:

```
peakVals = results{1};   <-- Notice the curly brackets!
```

ODE: fsuClusterMatlab

For an interactive session, we'd log in to the HPC front end, and move to the directory containing **ode_fun** and **ode_system**.

```
matlab

  bVals = 0.1 : 0.05 : 5;
  kVals = 1.5 : 0.05 : 5;
  results = fsuClusterMatlab([],[],'p','w',4,...
    @ode_fun, { bVals, kVals } )

  Wait while program is queued, then executes on cluster.

  peakvals = results{1};
  ode_display ( bVals, kVals, peakVals );

exit
```

ODE: fsuClusterMatlab with NOWAIT option

But you don't have to wait. You can send your job, log out, and return to your directory later. The details of this are a little gory!

1. Jobs are numbered. My last job is called **Job8**.
2. **cat Job8.state.mat** prints finished if the job is complete;
3. Now start MATLAB:

```
matlab
  cd Job8
  load ( 'Task1.out.mat' )
  peakvals = argsout{1}; <-- what we called "results"
  bVals = 0.1 : 0.05 : 5;
  kVals = 1.5 : 0.05 : 5;
  ode_display ( bVals, kVals, peakVals );
exit
```

- Introduction
- PARFOR: a Parallel FOR Loop
- QUAD Example (PARFOR)
- ODE Example (PARFOR)
- **SPMD: Single Program, Multiple Data**
- QUAD Example (SPMD)
- DISTANCE Example (SPMD)
- Conclusion

SPMD: Single Program, Multiple Data

The **SPMD** command 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 workers there are total, and can determine its behavior based on that ID.

- each worker runs on a separate core (ideally);
- each worker uses 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 worker 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' )
```

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.

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									

SPMD: When is Workspace Preserved?

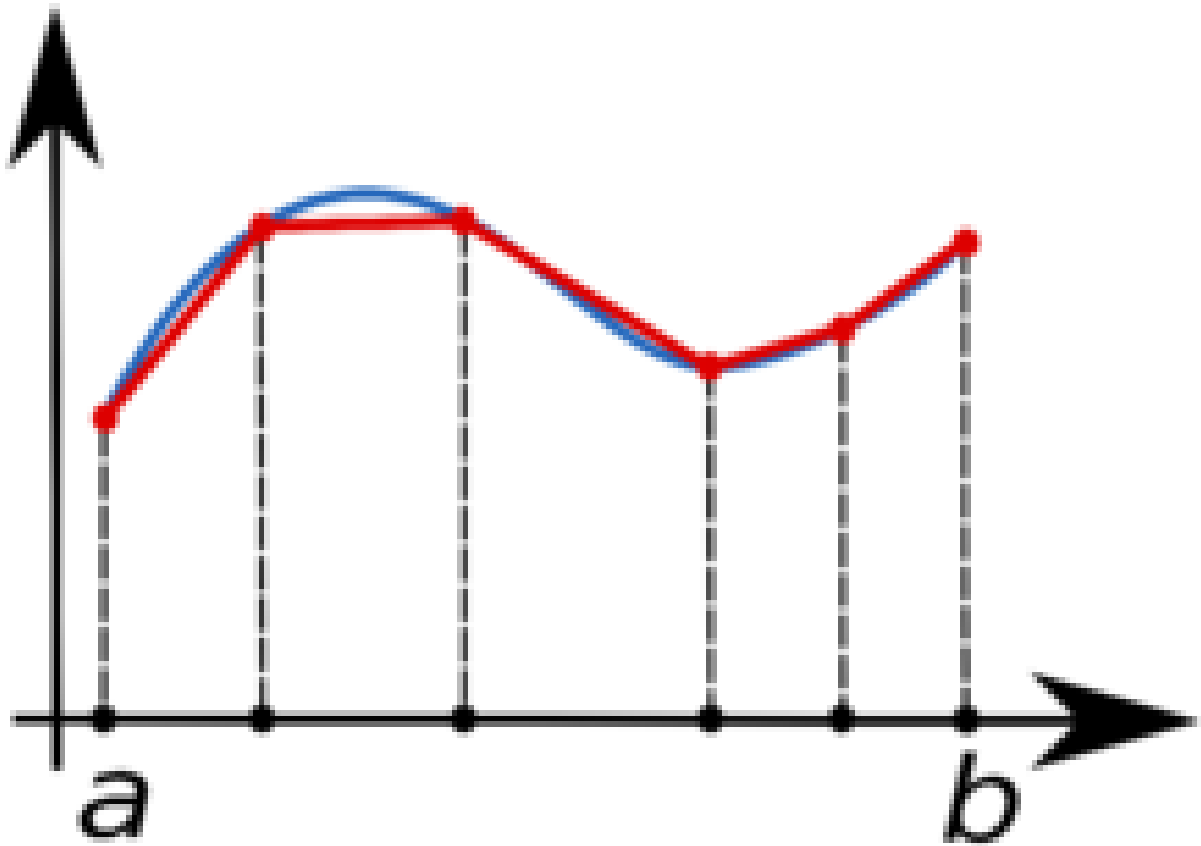
A program can contain several **spmd** blocks. When execution of one block is completed, the workers pause, but they do not disappear and their workspace remains intact. A variable set in one **spmd** block will still have that value if another **spmd** block is encountered.

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.

- Introduction
- PARFOR: a Parallel FOR Loop
- QUAD Example (PARFOR)
- ODE Example (PARFOR)
- SPMD: Single Program, Multiple Data
- **QUAD Example (SPMD)**
- DISTANCE Example (SPMD)
- 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:

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

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 approximate 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 is a program with 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 any worker by specifying the worker index. Thus $\mathbf{a}\{\mathbf{1}\}$ 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. It is very similar to a cell array.

The workers can "see" but not change the client's variables.

QUAD: Dealing with Composite Variables

So in QUAD, each worker could print its own **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
```

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

Having each worker print out its piece of the answer is not the right thing to do. (Imagine if we use 100 workers!)

The client should gather the values together into a sum:

```
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 )

  fprintf ( 1, 'Compute limits\n' );
  spmd
    a = ( labindex - 1 ) / numlabs;
    b =   labindex       / numlabs;
    fprintf ( 1, '  Lab %d works on [%f,%f].\n', labindex, a, b );
  end

  fprintf ( 1, 'Each lab estimates part of the integral.\n' );

  spmd
    x = linspace ( a, b, n );
    fx = f ( x );
    quad_part = ( b - a ) * ( fx(1) + 2 * sum ( fx(2:n-1) ) + fx(n) ) ...
      / 2.0 / ( n - 1 );
    fprintf ( 1, '  Approx %f\n', quad_part );
  end

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

  return
end
```

QUAD: Local Interactive Execution

SPMD programs execute locally just like PARFOR programs.

```
matlabpool open local 4

n = 10000;
value = quad_fun ( n );

matlabpool close
```

- Introduction
- PARFOR: a Parallel FOR Loop
- QUAD Example (PARFOR)
- ODE Example (PARFOR)
- SPMD: Single Program, Multiple Data
- QUAD Example (SPMD)
- **DISTANCE Example (SPMD)**
- Conclusion

DISTANCE: A Classic Problem

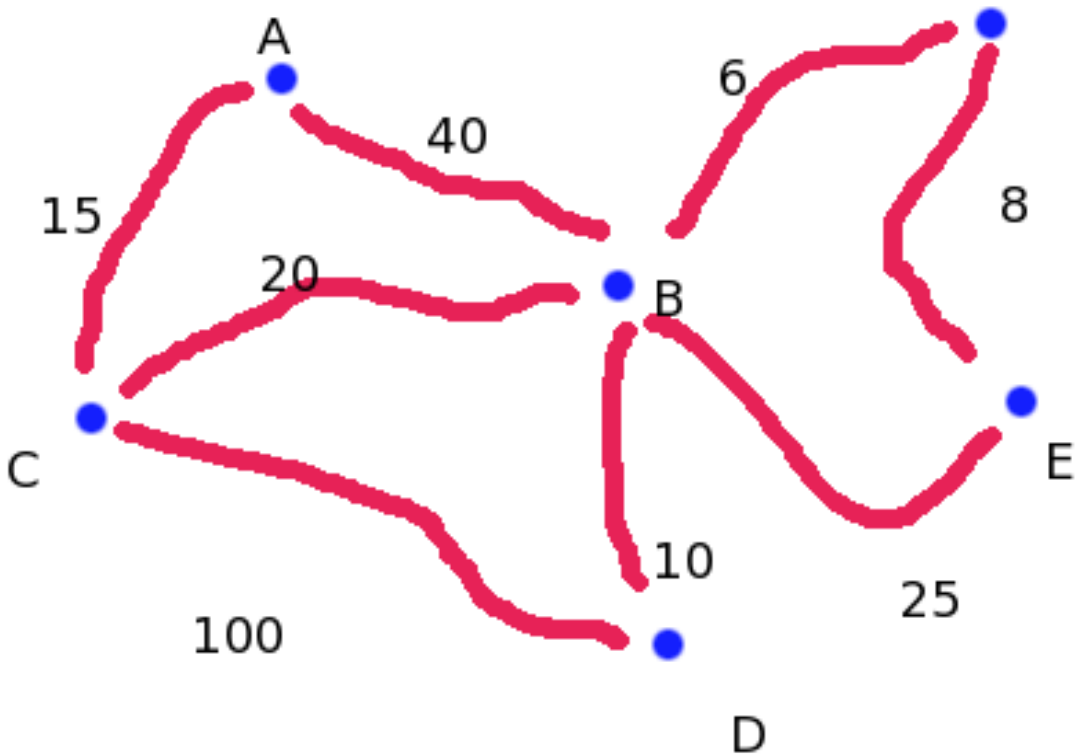
Consider the problem of determining the shortest path from a fixed city to all other cities on a road map, or more generally, from a fixed node to any other node on an abstract graph whose links have been assigned lengths.

Suppose the network of roads is described using a *one-hop distance matrix*; the value **ohd(i,j)** records the length of the direct road from city **i** to city **j**, or ∞ if no such road exists.

Our goal is, starting from the array **ohd**, to determine a new vector **distance**, which records the length of the shortest possible route from the fixed node to all others.

DISTANCE: An Intercity Map

Recall our example map with the intercity highway distances:



DISTANCE: An Intercity One-Hop Distance Matrix

Here is the "one-hop distance" matrix **ohd(i,j)**:

	A	B	C	D	E	F
A	0	40	15	∞	∞	∞
B	40	0	20	10	25	6
C	15	20	0	100	∞	∞
D	∞	10	100	0	∞	∞
E	∞	25	∞	∞	0	8
F	∞	6	∞	∞	8	0

An entry of ∞ means no direct route between the cities.

DISTANCE: Dijkstra's algorithm

Dijkstra's algorithm for the minimal distances:

Use two arrays, **connected** and **distance**.
Initialize **connected** to false except for A.
Initialize **distance** to the one-hop distance from A to each city.
Do N-1 iterations, to connect one more city at a time:

1. Find I, the unconnected city with minimum **distance[I]**;
2. Connect I;
3. For each unconnected city J, see if the trip from A to I to J is shorter than the current **distance[J]**.

The check we make in step 3 is:
distance[J] = min (**distance[J]**, **distance[I]** + **ohd[I][J]**)

DISTANCE: A Sequential Code

```
connected(1) = 1;
connected(2:n) = 0;

distance(1:n) = ohd(1,1:n);

for step = 2 : n

  [ md, mv ] = find_nearest ( n, distance, connected );

  connected(mv) = 1;

  distance = update_distance ( nv, mv, connected, ...
    ohd, distance );

end
```

DISTANCE: Parallelization Concerns

Although the program includes a loop, it is **not** a parallelizable loop! Each iteration relies on the results of the previous one.

However, let us assume we have a very large number of cities to deal with. Two operations are expensive and parallelizable:

- **find_nearest** searches for the nearest unconnected node;
- **update_distance** checks the distance of each unconnected node to see if it can be reduced.

These operations can be parallelized by using SPMD statements in which each worker carries out the operation for a subset of the nodes. The client will need to be careful to properly combine the results from these operations!

DISTANCE: Startup

We assign to each worker the node subset S through E.
We will try to preface worker data by "**my_**".

```
spmd
  nth = numlabs ( );
  my_s = floor ( ( ( labindex() - 1 ) * nv ) / nth ) + 1;
  my_e = floor (   ( labindex()       * nv ) / nth );
end
```

DISTANCE: FIND_NEAREST

Each worker uses **find_nearest** to search its range of cities for the nearest unconnected one.

But now each worker returns an answer. The answer we want is the node that corresponds to the smallest distance returned by all the workers, and that means the client must make this determination.

DISTANCE: FIND_NEAREST

lab_count = nth{1};

```
for step = 2 : n
  spmd
    [ my_md, my_mv ] = find_nearest ( my_s, my_e, n, ...
      distance, connected );
  end
  md = Inf;
  mv = -1;
  for i = 1 : lab_count
    if ( my_md{i} < md )
      md = my_md{i};
      mv = my_mv{i};
    end
  end
  distance(mv) = md;
```

DISTANCE: UPDATE_DISTANCE

We have found the nearest unconnected city.

We need to connect it.

Now that we know the minimum distance to this city, we need to check whether this decreases our estimated minimum distances to other cities.

DISTANCE: UPDATE_DISTANCE

```
  connected(mv) = 1;

  spmd
    my_distance = update_distance ( my_s, my_e, n, mv, ...
      connected, ohd, distance );
  end

  distance = [];
  for i = 1 : lab_count
    distance = [ distance, my_distance{:} ];
  end

end
```

DISTANCE: Desktop Run

To run the code on a desktop:

```
matlabpool open local 4

nv = 6;
ohd = initial_distance ( );

mind = dijkstra_fun ( nv, ohd );

disp ( mind );

matlabpool close
```

DISTANCE: fsuClusterMatlab

On the FSU HPC cluster, invoke **fsuClusterMatlab**:

```
nv = 6;
ohd = initial_distance ( );

results = fsuClusterMatlab([],[],'m','w',4,...
  @dijkstra_fun, { nv, ohd } );

mind = results{1};
disp ( mind );
```

Here, the 'm' argument indicates that this is an "MPI-like" job, that is, it invokes the **spmd** command.

DISTANCE: Comments

This example shows SPMD workers interacting with the client.

It's easy to divide up the work here. The difficulties come when the workers return their partial results, and the client must assemble them into the desired answer.

In one case, the client must find the minimum from a small number of suggested values.

In the second, the client must rebuild the **distance** array from the individual pieces updated by the workers.

Workers are not allowed to modify client data. This keeps the client data from being corrupted, at the cost of requiring the client to manage all such changes.

- Introduction
- PARFOR: a Parallel FOR Loop
- QUAD Example (PARFOR)
- ODE Example (PARFOR)
- SPMD: Single Program, Multiple Data
- QUAD Example (SPMD)
- DISTANCE Example (SPMD)
- **Conclusion**

Conclusion: A Parallel Version of MATLAB

Parallel MATLAB allows programmers to advance in the new world of parallel programming.

They can benefit from many of the same new algorithms, multicore chips and multi-node clusters that define parallel computing.

When you use the **parfor** command, MATLAB automatically determines from the form of your loop which variables are to be shared, or private, or are reduction variables; in OpenMP you must recognize and declare all these facts yourself.

When you use the **spmd** command, MATLAB takes care of all the data transfers that an MPI programmer must carry out explicitly.

Conclusion: Desktop Experiments

If you are interested in parallel MATLAB, the first thing to do is get access to the Parallel Computing Toolbox on your multicore desktop machine, so that you can do experimentation and practice runs.

You can begin with some of the sample programs we have discussed today.

You should then see whether the **parfor** or **spmd** approaches would help you in your own programming needs.

Conclusion: FSU HPC Cluster

If you are interested in serious parallel MATLAB computing, you should consider requesting an account on the FSU HPC cluster, which offers MATLAB on up to 16 cores.

To get an account, go to www.hpc.fsu.edu and look at the information under **Apply for an Account**.

Accounts on the general access cluster are available to any FSU faculty member, or to persons they sponsor.

CONCLUSION: Where is it?

- MATLAB Parallel Computing Toolbox User's Guide 5.0
  www.mathworks.com/access/helpdesk/help/pdf_doc/distcomp/...
  distcomp.pdf
- http://people.sc.fsu.edu/~jburkardt/presentations/...
  fsu_2011_matlab_parallel.pdf *these slides*;
- 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**
  - **ode_sweep_parfor**
  - **quad_spmd**
  - **dijkstra_spmd**