Parallel Programming with MATLAB

John Burkardt
Department of Scientific Computing
Florida State University

....

Hosted by Professor Eunjung Lee,
Department of Computational Science and Engineering,
Yonsei University, Seoul, Korea

....

http://people.sc.fsu.edu/∼jburkardt/presentations/. . .
matlab_parallel_2011_yonsei.pdf

28 September 2011

MATLAB Parallel Computing

- **Introduction**
- QUAD Example (PARFOR)
- MD Example
- PRIME Example
- ODE Example
- SPMD: Single Program, Multiple Data
- QUAD Example (SPMD)
- DISTANCE Example
- CONTRAST Example
- CONTRAST2: Messages
- Conclusion

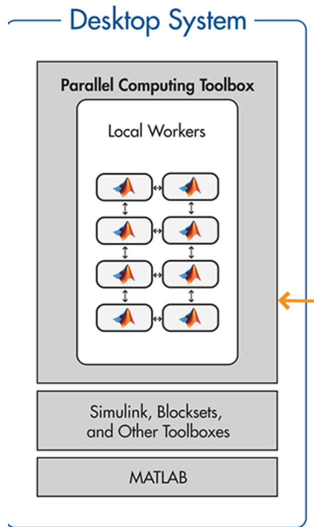
Parallel MATLAB is an extension of MATLAB that takes advantage of multicore desktop machines and clusters.

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

The user can:

- type in commands that will be executed in parallel, OR
- call an M-file that will run in parallel, OR
- submit an M-file to be executed in “batch” (not interactively).

INTRO: Local MATLAB Workers



INTRO: Parallel MATLAB

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

With a cluster running parallel MATLAB, a user can:

1. submit an M-file from a desktop, to run on the cluster, OR
2. log into the "front end" of the cluster, run interactively; OR
3. log into the "front end" of the cluster, and submit an M-file to be executed in "batch".

Options 1 and 3 allow the user to log out of the desktop or cluster, and come back later to check to see whether the computation has been completed.

For example, Virginia Tech's Ithaca cluster allows parallel MATLAB to run on up to 96 cores simultaneously.

INTRO: Local and Remote MATLAB Workers

Desktop System

Parallel Computing Toolbox

Local Workers

Simulink, Blocksets,
and Other Toolboxes

MATLAB

Computer Cluster

MATLAB Distributed Computing Server

Workers

Scheduler

INTRO: PARFOR and SPMD (and TASK)

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

- suitable **for** loops can be made into **parfor** loops;
- the **spmd** statement synchronizes cooperating processors;
- the **task** statement submits a program many times with different input, as in a Monte Carlo calculation; all the outputs can be analyzed together at the end.

parfor 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.

We won't have time to talk about the **task** *statement.*

INTRO: Execution

There are several ways to execute a parallel MATLAB program:

- interactive local (**matlabpool**), suitable for the desktop;
- indirect local, (**batch** or **createTask**);
- indirect remote, (**batch** or **createTask**), requires setup.

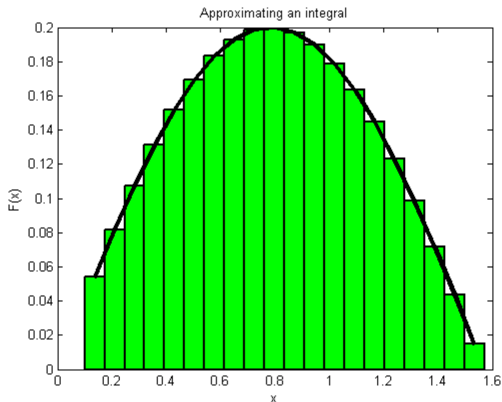
A cluster can accept parallel MATLAB jobs submitted from a user's desktop, and will return the results when the job is completed.

Making this possible requires a one-time setup of the user's machine, so that it "knows" how to interact with the cluster, and how to "talk" to the copy of MATLAB on the cluster.

MATLAB Parallel Computing

- Introduction
- **QUAD Example (PARFOR)**
- MD Example
- PRIME Example
- ODE Example
- SPMD: Single Program, Multiple Data
- QUAD Example (SPMD)
- DISTANCE Example
- CONTRAST Example
- CONTRAST2: Messages
- Conclusion

QUAD: Estimating an Integral



QUAD: The QUAD_FUN Function

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

  q = 0.0;
  w = ( b - a ) / n;

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

  return
end
```

QUAD: Comments

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

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

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

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

We could even compute the subareas at the same time, assuming there is some method to save the partial results and add them together in an organized way.

QUAD: The Parallel QUAD_FUN Function

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

  q = 0.0;
  w = ( b - a ) / n;

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

  return
end
```

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 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: What Do You Need For Parallel MATLAB?

1. Your machine should have multiple processors or cores:
   - On a PC: **Start** :: **Settings** :: **Control Panel** :: **System**
   - On a Mac: Apple Menu :: **About this Mac** :: **More Info...**
2. Your MATLAB must be **version 2008a** or later:
   - Go to the **HELP** menu, and choose **About Matlab**.
3. You must have the **Parallel Computing Toolbox**:
   - To list *all* your toolboxes, type the MATLAB command **ver**.

QUAD: Interactive Execution with MATLABPOOL

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

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

```
n = 10000; a = 0; b = 1;
matlabpool open local 4
q = quad_fun ( n, a, b );
matlabpool close
```

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

The value "4" is the number of workers you are asking for. It can be up to 8 on a local machine. It does not have to match the number of cores you have.

QUAD: Indirect Local Execution with BATCH

Indirect execution requires a script file, say **quad_script.m**:

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

Now we define the information needed to run the script:

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

QUAD: Indirect Remote Execution with BATCH

The **batch** command can send your job *anywhere*, and get the results back, if you have set up an account on the remote machine, and have defined a configuration on your desktop that describes how to access the remote machine.

For example, at Virginia Tech, a desktop computer can send a batch job to the cluster, requesting 32 cores:

```
job = batch ( 'quad_script', ...
  'matlabpool', 32, ...
  'Configuration', 'ithaca_2010b', ...
  'FileDependencies', { 'quad_fun' } )
```

QUAD: Submitting a Job and Waiting

How do the results come back to you?

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

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

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

QUAD: Submitting a Job and Coming Back Later

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

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

```
job = batch ( ...information defining the job... )
submit ( job );
  Exit MATLAB, turn off machine, go home.

  Come back, restart machine, start MATLAB:
sched = findResource ( );
jobs = findJob ( sched )
  findJob() returns a cell array of all your jobs.
  You pick out the one you want, say "k".
load ( jobs{k} );
```

QUAD: Reference

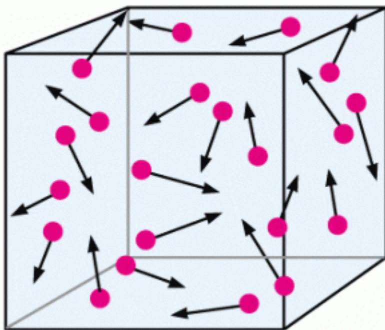
The **QUAD** example is available at:

http://people.sc.fsu.edu/∼jburkardt/...
m_src/quad_parfor/quad_parfor.html

This includes MATLAB source codes, script files, and the output.

- Introduction
- QUAD Example (PARFOR)
- **MD Example**
- PRIME Example
- ODE Example
- SPMD: Single Program, Multiple Data
- QUAD Example (SPMD)
- DISTANCE Example
- CONTRAST Example
- CONTRAST2: Messages
- Conclusion

MD: A Molecular Dynamics Simulation



Compute positions and velocities of **N** particles over time.
The particles exert a weak attractive force on each other.

MD: The Molecular Dynamics Example

How do you prepare a program to run in parallel?

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

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

  return
end
```

MD: Can We Use PARFOR?

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

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

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

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

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

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

MD: Speedup

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

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 other lectures we will see some alternatives.

MD: PARFOR is Particular

We were only able to parallelize the loop because the iterations were independent, that is, the results did not depend on the order in which the iterations were carried out.

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

This is a stronger requirement than independence of order!

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

MD: Reference

The **MD** example is available at:

http://people.sc.fsu.edu/∼jburkardt/...
m_src/md_parfor/md_parfor.html

This includes MATLAB source codes, script files, and the output.

- Introduction
- QUAD Example (PARFOR)
- MD Example
- **PRIME Example**
- ODE Example
- SPMD: Single Program, Multiple Data
- QUAD Example (SPMD)
- DISTANCE Example
- CONTRAST Example
- CONTRAST2: Messages
- Conclusion

PRIME: The Prime Number Example

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

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

If we want the program to run longer, we increase the variable **N**. Doubling **N** multiplies the run time roughly by 4.

PRIME: The Sieve of Erastosthenes

PRIME: Program Text

```
function total = prime ( n )

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

  total = 0;

  for i = 2 : n

    prime = 1;

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

    total = total + prime;

  end

  return
end
```

PRIME: We can run this in parallel

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

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

To make the program parallel, we replace **for** by **parfor**. That's all!

PRIME: Local Execution With MATLABPOOL

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

n = 50;

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

matlabpool ( 'close' )
```

PRIME: Timing

```
PRIME_PARFOR_RUN
  Run PRIME_PARFOR with 0, 1, 2, and 4 workers.
  Time is measured in seconds.
```

N	1+0	1+1	1+2	1+4
50	0.067	0.179	0.176	0.278
500	0.008	0.023	0.027	0.032
5000	0.100	0.142	0.097	0.061
50000	7.694	9.811	5.351	2.719
500000	609.764	826.534	432.233	222.284

PRIME: Timing Comments

There are many thoughts that come to mind from these results!

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

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

This data suggests two conclusions:

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

AND

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

PRIME: Reference

The **PRIME** example is available at:

http://people.sc.fsu.edu/∼jburkardt/...
m_src/prime_parfor/prime_parfor.html

This includes MATLAB source codes, script files, and the output.

MATLAB Parallel Computing

- Introduction
- QUAD Example (PARFOR)
- MD Example
- PRIME Example
- **ODE Example**
- SPMD: Single Program, Multiple Data
- QUAD Example (SPMD)
- DISTANCE Example
- CONTRAST Example
- CONTRAST2: Messages
- 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 $xmax(b, k)$.

ODE: Each Solution has a Maximum Value

m = 5, b = 2, k = 2

m = 5, b = 5, k = 5

Displacement (x)

Time (s)

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 Function

```
function peakVals = ode_fun_parfor ( 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: The Script

If we want to use the batch (indirect or remote execution) option, then we need to call the function using a script. We'll call this "ode_script_batch.m"

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

peakVals = ode_fun_parfor ( bVals, kVals );
```

ODE: Plot the Results

The script **ode_display.m** calls **surf** for a 3D plot.

Our parameter arrays **bVals** and **kVals** are X and Y, while the computed array **peakVals** plays the role of Z.

```
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: Interactive Execution

Using the interactive option, we set up the input, get the workers, and call the function:

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

matlabpool open local 4

peakVals = ode_fun_parfor ( bVals, kVals );

matlabpool close

ode_display    <-- Don't need parallel option here.
```

ODE: BATCH Execution

Using batch execution, our script computes the data on a cluster, (here, Virginia Tech's Ithaca cluster), but we plot that data back on the desktop:

```
job = batch ( ...
  'ode_sweep_script', ...
  'matlabpool', 64, ...
  'Configuration', 'ithaca_2011b', ...
  'FileDependencies', {'ode_fun_parfor', 'ode_system'} );

wait ( job );
load ( job );   <-- Load data from cluster
ode_display     <-- The desktop plots the data that
                    was computed on the cluster
```

ODE: A Parameterized Problem

Results of ODE Parameter Sweep

Peak Displacement

0
0.5
1
1.5
2
2.5

Damping B

1
2
3
4
5

Stiffness K

1
2
3
4
5

ODE: Reference

The **ODE** example is available at:

http://people.sc.fsu.edu/∼jburkardt/...
m_src/ode_sweep_parfor/ode_sweep_parfor.html

This includes MATLAB source codes, script files, and the output.

MATLAB Parallel Computing

- Introduction
- QUAD Example (PARFOR)
- MD Example
- PRIME Example
- ODE Example
- **SPMD: Single Program, Multiple Data**
- QUAD Example (SPMD)
- DISTANCE Example
- CONTRAST Example
- CONTRAST2: Messages
- 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: Getting Workers

Interactively, we get workers with **matlabpool**:

```
matlabpool open local 4
results = myfunc ( args );
```

or use **batch** to run in the background on your desktop:

```
job = batch ( 'myscript', ...
'matlabpool', 4, ...
'Configuration', 'local' )
```

or send the **batch** command to a cluster such as Virginia Tech's Ithaca:

```
job = batch ( 'myscript',
  'matlabpool', 31, ...
  'Configuration', 'ithaca_2011b' )
```

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?

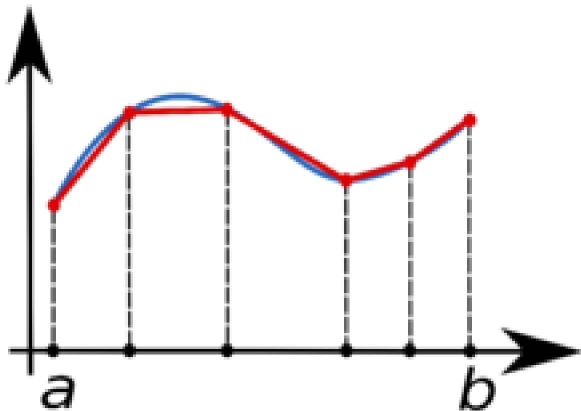
A program can contain several **spmd** blocks. When execution of one block is completed, the workers pause, but they do not disappear and their workspace remains intact. A variable set in one **spmd** block will still have that value if another **spmd** block is encountered.

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 regular MATLAB data does.

- Introduction
- QUAD Example (PARFOR)
- MD Example
- PRIME Example
- ODE Example
- SPMD: Single Program, Multiple Data
- **QUAD Example (SPMD)**
- DISTANCE Example
- CONTRAST Example
- CONTRAST2: Messages
- Conclusion

QUAD_SPMD: The Trapezoid Rule



Area of one trapezoid = average height * base.

QUAD_SPMD: 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 approximate to the integral of the function over the whole interval.

QUAD_SPMD: 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_SPMD: One Name References 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 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{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 "see" the client's variables and inherit a copy of their values, but cannot change the client's data.

QUAD_SPMD: 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
```

QUAD_SPMD: 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 handles 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_SPMD: Combining Partial Results

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

One way has the client gather the answers, and sum them:

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

with result:

```
Approximation 3.14159265
```

QUAD_SPMD: 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
```

The **QUAD_SPMD** example is available at:

http://people.sc.fsu.edu/∼jburkardt/...
m_src/quad_spmd/quad_spmd.html

This includes MATLAB source codes, script files, and the output.

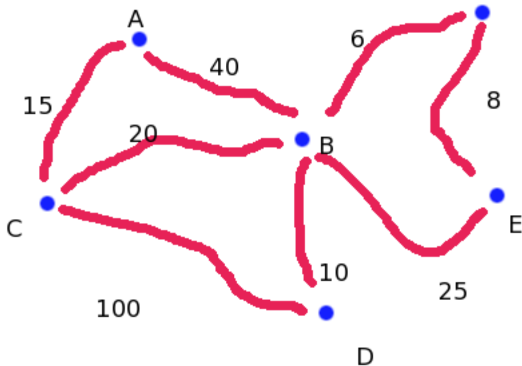
- Introduction
- QUAD Example (PARFOR)
- MD Example
- PRIME Example
- ODE Example
- SPMD: Single Program, Multiple Data
- QUAD Example (SPMD)
- **DISTANCE Example**
- CONTRAST Example
- CONTRAST2: Messages
- Conclusion

DISTANCE: A Classic Problem

Anyone doing highway traveling is familiar with the difficulty of determining the shortest route between points A and B. From a map, it's easy to see the distance between neighboring cities, but often the best route takes a lot of searching.

A **graph** is the abstract version of a network of cities. Some cities are connected, and we know the length of the roads between them. The cities are often called *nodes* or *vertices* and the roads are *links* or *edges*. Whereas cities are described by maps, we will describe our abstract graphs using a *one-hop distance matrix*, which is simply the length of the direct road between two cities, if it exists.

Here is an example map with the intercity highway distances:



DISTANCE: An Intercity One-Hop Distance Matrix

Supposing we live in city A, our question is, *"What is the shortest possible distance from A to each city on the map?"*

Instead of a map, we use a "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

where ∞ means there's no direct route between the two cities.

DISTANCE: The Shortest Distance

The map makes it clear that it's possible to reach every city from city A; we just have to take trips that are longer than "one hop". In fact, in this crazy world, it might also be possible to reach a city faster by taking two hops rather than the direct route. (Look at how to get from city A to city B, for instance!)

We want to use the information in the map or the matrix to come up with a **distance** vector, that is, a record of the shortest possible distance from city A to all other cities.

A method for doing this is known as *Dijkstra's algorithm*.

DISTANCE: Dijkstra's algorithm

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 all nodes for the nearest unconnected one;
- **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 ( )       * n ) / nth );
  my_e = floor ( ( ( labindex ( ) + 1 ) * n ) / nth ) - 1;
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: 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.

DISTANCE: Reference

The **DISTANCE** example is available at:

http://people.sc.fsu.edu/∼jburkardt/...
m_src/dijkstra_spmd/dijkstra_spmd.html

This includes MATLAB source codes, script files, and the output.

- Introduction
- QUAD Example
- MD Example
- PRIME Example
- ODE Example
- SPMD: Single Program, Multiple Data
- QUAD Example (SPMD)
- DISTANCE Example
- **CONTRAST Example**
- CONTRAST2: Messages
- Conclusion

CONTRAST: Image → Contrast Enhancement → Image2

```
%
%  Get 4 SPMD workers.
%
  matlabpool open local 4
%
%  Read an image.
%
  x = imread ( 'surfsup.tif' );
%
%  Since the image is black and white, it will be distributed by columns.
%
  xd = distributed ( x );
%
%  Have each worker enhance the contrast in its portion of the picture.
%
  spmd
    xl = getLocalPart ( xd );
    xl = nlfilter ( xl, [3,3], @adjustContrast );
    xl = uint8 ( xl );
  end
%
%  We are working with a black and white image, so we can simply
%  concatenate the submatrices to get the whole object.
%
  xf_spmd = [ xl{:} ];

  matlabpool close
```

CONTRAST: Image → Contrast Enhancement → Image2

Original image

Filtered on Client

Filtered on 4 SPMD Workers

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?

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
```

CONTRAST: Image → Contrast Enhancement → Image2

When the image is divided by columns among the workers, artificial internal boundaries are created. The 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!

The result is spurious lines on the processed image.

CONTRAST: Reference

The **CONTRAST** example is available at:

http://people.sc.fsu.edu/∼jburkardt/...
m_src/contrast_spmd/contrast_spmd.html

This includes MATLAB source codes, script files, and the output.

- Introduction
- QUAD Example (PARFOR)
- MD Example
- PRIME Example
- ODE Example
- SPMD: Single Program, Multiple Data
- QUAD Example (SPMD)
- DISTANCE Example
- CONTRAST Example
- **CONTRAST2: Messages**
- 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: Who 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
```

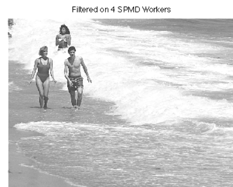
CONTRAST2: First Column Left, Last Column Right

```
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
```

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
```

CONTRAST2: Image → Enhancement → Image2



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

CONTRAST2: The Heat Equation

Image processing was used to illustrate this example, but consider that the contrast enhancement operation updates values by comparing them to their neighbors.

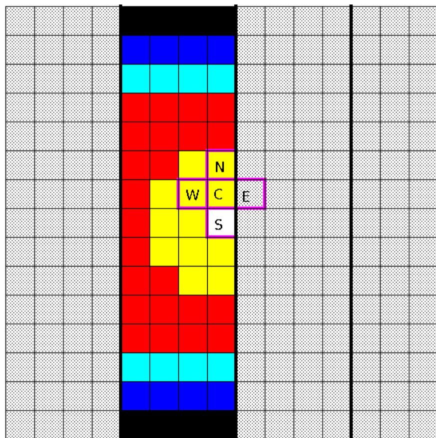
The same operation applies in the **heat equation**, except that high contrasts (hot spots) tend to average out (cool off)!

In a simple explicit method for a time dependent 2D heat equation, we repeatedly update each value by combining it with its north, south, east and west neighbors.

So we could do the same kind of parallel computation, dividing the geometry into strip, and avoiding artificial boundary effects by having neighboring SPMD workers exchange "boundary" data.

CONTRAST2: The Heat Equation

The "east" neighbor lies in the neighboring processor, so its value must be received by message in order for the computation to proceed.



CONTRAST2: The Heat Equation

So now it's time to modify the image processing code to solve the heat equation.

But just for fun, let's use our black and white image as the initial condition! Black is cold, white is hot.

In contrast to the contrast example, the heat equation tends to smooth out differences. So let's watch our happy beach memories fade away ... in parallel ... and with no artificial boundary seams.

CONTRAST2: The Heat Equation, Step 0

CONTRAST2: The Heat Equation, Step 10

CONTRAST2: The Heat Equation, Step 20

CONTRAST2: The Heat Equation, Step 40

CONTRAST2: The Heat Equation, Step 80

CONTRAST2: Reference

The **CONTRAST2** example is available at:

http://people.sc.fsu.edu/∼jburkardt/...
m_src/contrast2_spmd/contrast2_spmd.html

This includes MATLAB source codes, script files, and the output.

MATLAB Parallel Computing

- Introduction
- QUAD Example (PARFOR)
- MD Example
- PRIME Example
- ODE Example
- SPMD: Single Program, Multiple Data
- QUAD Example (SPMD)
- DISTANCE Example
- CONTRAST Example
- CONTRAST2: Messages
- **Conclusion**

Conclusion: A Parallel Version of MATLAB

MATLAB's Parallel Computing Toolbox allows programmers to take advantage of parallel architecture (multiple cores, cluster computers) and parallel programming techniques, to solve big problems efficiently.

MATLAB controls the programming environment; that makes it possible to send jobs to a remote computer system without the pain of logging in, transferring files, running the program and bringing back the results. MATLAB automates all this for you.

Moreover, 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.

CONCLUSION: Thanks to my Host!

CONCLUSION: Where is it?

- MATLAB Parallel Computing Toolbox User's Guide 5.2, available on the MathWorks website;
- 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/... presentations/**yonsei_matlab_2011.pdf** *(these slides)*
- http://people.sc.fsu.edu/∼jburkardt/m_src/m_src.html
    - **quad_parfor**
    - **md_parfor**
    - **prime_parfor**
    - **ode_sweep_parfor**
    - **quad_spmd**
    - **dijkstra_spmd**
    - **contrast_spmd**
    - **contrast2_spmd**