

Parallel MATLAB: Single Program Multiple Data

John Burkardt (FSU) & Gene Cliff (AOE/ICAM)

1:30pm - 3:30pm, Thursday, 02 June 2011

Mathematics Commons Room

.....

vt_2011_spmd.pdf

.....

FSU: Florida State University

AOE: Department of Aerospace and Ocean Engineering

ICAM: Interdisciplinary Center for Applied Mathematics

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

Lecture #1: PARFOR - 31 May

The **parfor** command, described in a previous lecture, 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;
- we can't examine the work of any individual worker.

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

SPMD: is Single Program, Multiple Data

Lecture #2: SPMD - 2 June

The **SPMD** command (**today's lecture**) 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

An **spmd** program needs workers to cooperate on the program.

So on a desktop, we issue an interactive **matlabpool** request:

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

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

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

or send the **batch** command to the Ithaca cluster:

```
job = batch ( 'myscript', 'Configuration', ...
             'ithaca_2009b', 'matlabpool', 7 )
```

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?

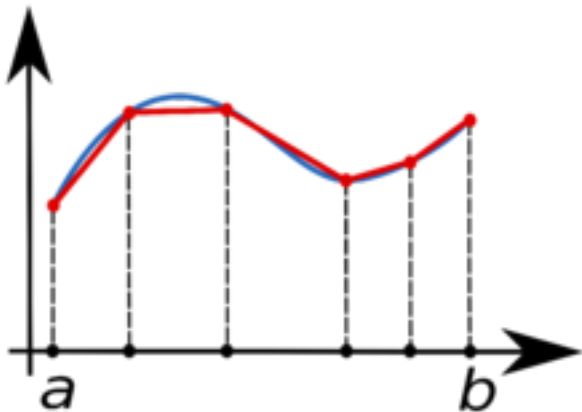
A program can contain several **spmd** blocks. When execution of one block is completed, the workers pause, but they do not disappear and their workspace remains intact. A variable set in one **spmd** block will still have that value if another **spmd** block is encountered.

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.

- SPMD: Single Program, Multiple Data
- **QUAD Example**
- Distributed Arrays
- 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

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

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

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

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 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 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
```

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 )

    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
```

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

DISTRIBUTED: Conjugate Gradient Setup

It is possible to use what amounts to **SPMD** programming without explicitly using the **spm** statement. That's because many MATLAB functions and operators are capable of carrying out algorithms that involve the cooperation of multiple workers with separate workspaces.

The user might only see the “client” copy of MATLAB; special commands or options distribute the data to the available workers, who then cooperate to carry out the computation.

Again, this is “really” **SPMD** programming, except that the MathWorks staff had to write the **spm** blocks, hidden inside MATLAB's functions.


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:

```
spmatrix
  a1 = getLocalPart ( ad );
  [ m1, n1 ] = size ( a1 );
end
```

On worker 3, $[m1, n1] = (300, 100)$, and **a1** is

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

Notice that local and global **column** indices will differ!

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 = a1{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 = [ a1{:} ]
```

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 = a1{1};  
a2(:, :, 2) = a1{2};  
a2(:, :, 3) = a1{3};
```

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 skip the communication cost of distributing it.

DISTRIBUTED: Accessing Distributed Arrays

The command `a1 = 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 `a1` 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 **matlabpool** or indirect **batch** command.

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);

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

fprintf(1, 'Error_residual: %8.4e\n', e_norm);

np = 8;
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

DISTRIBUTED: Conjugate Gradient Iteration

```
function [x, resnorm] = cg_emc( A, b, x0, tol, itmax)
% Conjugate gradient iteration for  $Ax = b$ ,
% with symmetric, positive-definite  $A$ 
% (from Gill, Murray and Wright, p 147

% Possibly supply missing input parameters ((omitted)

% initialization
    p = 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;
        resnorm = norm(A*x - b);
        if iter >= itmax || resnorm <= tol
            break
        end
    end % while 1 > 0

end % function
```

In the conjugate gradient example, we have emphasized how trivial it is to extend a MATLAB algorithm to a distributed memory problem. Essentially, all you have to do is invoke **distributed()**; the operational commands don't change.

There are two comments worth making, in the interest of honesty:

- 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 $\mathbf{x}=\mathbf{A}\backslash\mathbf{b}$ 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 example we have shown!

In the next example, we demonstrate a mixed approach wherein the stiffness matrix (\mathbf{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 (\mathbf{F}). The *local* forcing arrays are then used to build a **codistributed** array.

DISTRIBUTED: 1D Finite Difference Heat code

```
%Script to invoke codistributed solve for 1D heat equn by finite difference

N = 100;

spmd
    K = -2*codistributed.eye( N ); % diagonal part
    locP = getLocalPart(codistributed.colon(1, N));
% Loop over columns entering unity above/below the diagonal entry
% Note that columns 1 and N are exceptions
    for jj=locP(1):locP(end)
        if jj ~= 1; K(jj-1, jj) = 1; end % above the diagonal
        if jj ~= N; K(jj+1, jj) = 1; end % below the diagonal
    end
    locF = (10*labindex())*ones(size(locP'))/N^2;% funky rhs
    codist = codistributor1d(1, codistributor1d.unsetPartition, [N, 1]);
    F = codistributed.build(locF, codist); % distribute the array(s)
end

T = K\F; % mldivide is defined for distributed arrays

plot((1:N)/N, T, 'ro')
```

DISTRIBUTED: 2D Finite Element Heat Model

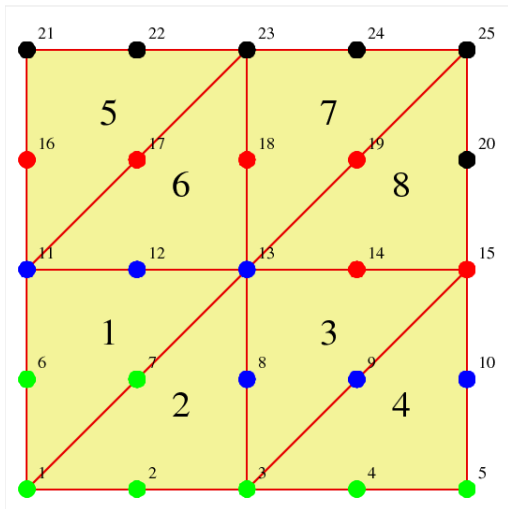
Finally, we consider an example that combines SPMD and distributed data to solve a steady state heat equations in 2D, using the finite element method.

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



DISTRIBUTED: Finite Element System matrix

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

$$Kz = F + b$$

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 $K_{i,j}$. Columns of \mathbf{K} associated with worker nodes are added to the local part of $K_{i,j}$. 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.

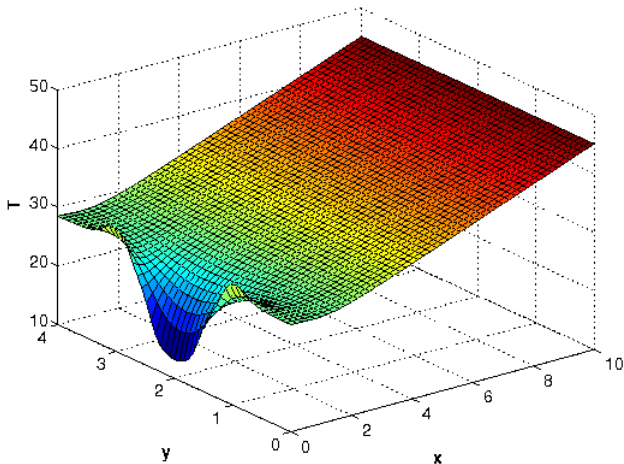
At the cost of some redundant computation, we avoid a lot of communication.

Assemble Codistributed Arrays - code fragment

```
spmf
%
% Set up codistributed structure
%
% Column pointers and such for codistributed arrays.
%
Vc = codistributed.colon(1, n.equations);
IP = getLocalPart(Vc);
IP_1= IP(1); IP_end = IP(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, IP_1, IP_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 % spmf
```

DISTRIBUTED: The Results



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

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 \rightarrow Noisy Image \rightarrow 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 \rightarrow Noisy Image \rightarrow 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
- IMAGE Example
- **CONTRAST Example**
- CONTRAST2: Messages
- Batch Computing
- Conclusion

CONTRAST: Image \rightarrow Contrast Enhancement \rightarrow Image2

```
%  
% Get 4 SPMD workers  
%  
matlabpool open 4  
%  
% Read an image  
%  
x = imread( '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 = nifilter( xl, [3, 3], @adjustContrast);  
    xl = uint8(xl);  
end  
%  
% Concatenate the submatrices to assemble the whole image  
%  
xf_spmd = [ xl{:} ];  
  
matlabpool close
```

CONTRAST: Image \rightarrow Contrast Enhancement \rightarrow 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?

CONTRAST: Image \rightarrow Contrast Enhancement \rightarrow 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 |
+-----+-----+-----+
```

$$P22 \leftarrow C * P22 + (1 - C) * \text{Average}$$

CONTRAST: Image \rightarrow Contrast Enhancement \rightarrow 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!

CONTRAST: Image \rightarrow Contrast Enhancement \rightarrow Image2



The result is spurious lines on the processed image.

- SPMD: Single Program, Multiple Data
- QUAD Example
- Distributed Arrays
- 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: 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

```
x1 = nlfilter ( x1, [3,3], @enhance_contrast );

if ( labindex < numlabs )
    x1 = x1(:,1:end-1);
end

if ( 1 < labindex )
    x1 = x1(:,2:end);
end

x1 = uint8 ( x1 );

end
```

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**.

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

BATCH: Indirect Execution

We can run quick, local interactive jobs using the **matlabpool** 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** 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.

Since Ithaca allocates cores in groups of 8, it makes sense to ask for 7, or 15, or 23 or 31 workers, for instance.

BATCH: PRIME_FUN is the function

```
function total = prime_fun ( n )  
  
    spmd  
  
        nlo = ( n * ( labindex - 1 ) ) / numlabs + 1;  
        nhi = ( n * labindex ) / numlabs;  
        if ( nlo == 1 )  
            nlo = 2;  
        end  
  
        total_part = 0;  
  
        for i = nlo : nhi  
  
            prime = 1;  
  
            for j = 2 : i - 1  
                if ( mod ( i, j ) == 0 )  
                    prime = 0;  
                    break  
                end  
            end  
  
            total_part = total_part + prime;  
  
        end  
  
        total_spmd = gplus ( total_part );  
    end  
  
    total = total_spmd {1};  
    return  
end
```


BATCH: PRIME_SCRIPT runs the function

```
%% PRIME_SCRIPT is a script to call PRIME_FUN.
%
% Discussion:
%
%   The BATCH command runs scripts, not functions. So we have to write
%   this short script if we want to work with BATCH!
%
% Licensing:
%
%   This code is distributed under the GNU LGPL license.
%
% Modified:
%
%   27 March 2010
%
% Author:
%
%   John Burkardt
%
n = 10000;

fprintf ( 1, '\n' );
fprintf ( 1, 'PRIME_SCRIPT\n' );
fprintf ( 1, 'Count_prime_numbers_from_1_to_%d\n', n );

total = prime_fun ( n );
```

BATCH: Using the BATCH Command

```
job = batch ( 'prime_script', ...  
    'configuration', 'local', ... <-- Run it locally.  
    'matlabpool', 7 )           <-- 7 workers, 1 client.  
  
wait ( job ); <-- One way to find out when job is done.  
  
load ( job ); <-- Load the output variables from  
               the job into the MATLAB workspace.  
  
total          <-- We can examine the value of TOTAL.  
  
destroy ( job ); <-- Clean up
```

BATCH: Using the BATCH Command

The **wait** command pauses your MATLAB session.

Using **batch**, you can submit multiple jobs:

```
job1 = batch ( ... )  
job2 = batch ( ... )
```

Using **get**, you can check on any job's status:

```
get ( job1, 'state' )
```

Using **load**, you can get the whole workspace, or you can examine just a single output variable if you specify the variable name:

```
total = load ( job2, 'total' )
```

BATCH: The BATCH Command

```
job_id = batch (  
    'script_to_run', ...  
    'configuration', 'local' or 'ithaca_2009b', ...  
    'FileDependencies', 'file' or {'file1','file2'}, ...  
    'CaptureDiary', 'true', ... (the default)  
    'CurrentDirectory', '/home/burkardt/matlab', ...  
    'PathDependencies', 'path' or {'path1','path2'}, ...  
    'matlabpool', number of workers (can be zero!) )
```

Note that you do not include the file extension when naming the script to run, or the files in the FileDependencies.

- SPMD: Single Program, Multiple Data
- QUAD Example
- Distributed Arrays
- 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 HEAT example shows how to construct local arrays and assemble these to **distributed** 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: Desktop Experiments

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

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

If you don't have a multicore machine, you won't see any speedup, but you may still be able to run some "parallel" programs.

Conclusion: Ithaca Experiments

If you want to work with parallel MATLAB on Ithaca, you must first get an account, by going to this website:

`http://www.arc.vt.edu/index.php`

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

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

Conclusion: Desktop-to-Ithaca Submission

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

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

The steps for doing this are described in the document:

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

We will be available to help you with this process.

Conclusion: VT MATLAB LISTSERV

There is a local LISTSERV for people interested in MATLAB on the Virginia Tech campus. We try **not** to post messages here unless we really consider them of importance!

Important messages include information about workshops, special MATLAB events, and other issues affecting MATLAB users.

To subscribe to the mathworks listserver, send email to:

```
listserv@listserv.vt.edu.
```

The body of the message should simply be:

```
subscribe mathworks firstname lastname
```

CONCLUSION: Where is it?

- MATLAB Parallel Computing Toolbox User's Guide 4.3
www.mathworks.com/access/helpdesk/help/pdf_doc/distcomp/...distcomp.pdf
- Gaurav Sharma, Jos Martin,
MATLAB: A Language for Parallel Computing,
International Journal of Parallel Programming,
Volume 37, Number 1, pages 3-36, February 2009.
- http://scholar.vt.edu/.../spmd_codes
 - **quad**
 - **conjugate_gradient**
 - **fd_1D_heat**
 - **fem_2D_heat**
 - **image**
 - **contrast** and **contrast2**
 - **batch_prime**

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 destroy the parallel job and end the **pmode** session

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