Parallel MATLAB at FSU:
Single Program Multiple Data

John Burkardt
Virginia Tech

....

https://people.sc.fsu.edu/∼jburkardt/presentations/...
matlab_spmd_2010_fsu.pdf

13 April 2010

MATLAB Parallel Computing

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

SPMD: is not PARFOR

The **parfor** command (Monday's 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 not TASK PROGRAMMING

Task computing (Friday's lecture) allows us to run many copies of one program, each time with a different set of input, each copy on one processor.

- the program copies don't necessarily run at the same time;
- no task can communicate with any other task;
- no task can provide results to another task;

Using **task computing**, your job is broken up into many "atomic" tasks. Only when all tasks are completed can the results be combined.

SPMD: is Single Program, Multiple Data

The **SPMD** command (today's lecture) is like working with a very simplified version of **MPI**. There is one client process, supervising workers who cooperate on a single program. Each worker 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 processor;
- each worker uses separate memory;
- 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

The **spmd** program needs MATLAB to gather 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 the **batch** command with a **matlabpool** argument:

```
batch ( 'myscript', 'matlabpool', 4 )
```

On the FSU HPC cluster, we say:

```
results = fsuClusterMatlab([],[],'m','w',4, ...
  @myfunc, { args } )
```

(The **'m'** *indicates that this is an "mpi-like" job.)*

SPMD: The SPMD Environment

MATLAB sets up one processor which “knows” it’s 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 variables:

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

Oddly enough, the client does *not* know the value of **numlabs**. It could determine this by the command

```
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.

Each worker has its own memory space, in which it stores the data it is working on. When an **spmd** block is completed, the worker pauses, but the data is not lost. The names and their related values can all be accessed as soon as another **spmd** block is encountered.

MATLAB Parallel Computing

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

QUAD: Setting Up Integration Limits

Here is the start of a program to estimate an integral on [0,1].

Notice that the variables **a** and **b** will have different values on each worker...and no value on the client!

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

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

QUAD: Each worker has an ID

The **spmd** delimiter marks a section of code which is to be carried out by each lab or worker, and *not* by the client.

The fact that the MATLAB program can be marked up into instructions for the client and instructions for the workers explains the **single program** part of **SPMD**.

But how do multiple workers do different things if they see the same instructions? Luckily, each worker is assigned a unique identifier, the value of the variable **labindex**.

The worker also gets the value of **numlabs**, the total number of workers. This information is enough to ensure that each worker can be assigned different tasks. This explains the **multiple data** part of **SPMD**!

QUAD: Use the ID to assign work

Now let’s go back to our program fragment. But first we must explain that we are trying to approximate an integral over the interval [0,1]. Using **SPMD**, we are going to have each worker pick a portion of that interval to work on, and we’ll sum the result at the end. Now let’s look more closely at the statements:

```
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 runs on a separate processor with its own memory. It can "see" the variables on the client, but it 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 memory of all the workers. Since there are multiple values using the same name, the client can refer to them by 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.

For the client, these worker variables are composite variables. Their indexing is similar to cell arrays.

The workers can "see" variables in the client's memory. They can read those values, but are not allowed to change them.

QUAD: How Names and Values are Shared (or not)

```
a = 1;
b = 2;
spmd
  a = b + 1;  <-- Illegal to write client variables.
  c = a + b;    <-- OK to "read" client's variables.
  d = labindex;
  e = numlabs;
end
c = 1;  <-- Illegal to use same name as worker variable.
c{2} = 1;    <-- OK to "write" worker variables!
f = d{3};  <-- OK to "read" worker variables.
c_sum = c{1} + c{2} + c{3} + c{4};
h = ( b - a ) / numlabs;
h = ( b - a ) / numlabs{4};
numlabs = matlabpool ( 'size' );
n = matlabpool ( 'size' );
```

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

Assuming we've defined our limits of integration, we now want to carry out the trapezoid rule for integration:

```
spmd
  x = linspace ( a, b, n );
  fx = f ( x );
  quad_part = ( fx(1) + 2 * sum(fx(2:n-1)) + fx(n) )
    /2 /(n-1);
  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_SPMD

```
function value = quad_spmd ( 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
```

MATLAB Parallel Computing

- 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 **spmd** statement. That's because many MATLAB functions and operators are capable of carrying out algorithms that involve the cooperation of multiple workers with separate memory spaces.

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 **spmd** blocks, hidden inside MATLAB's functions.

DISTRIBUTED: The Client Can Distribute

Where does distributed data come from? MATLAB provides several ways to get started. 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 the one used for distribution.

DISTRIBUTED: Workers Can Get Their Part

If the client has distributed the matrix by the command

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

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

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

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

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

Notice that local and global column indices will differ!

DISTRIBUTED: The Client Can Collect Results

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

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

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

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

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

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

DISTRIBUTED: Methods to Gather Data

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

If you have a choice, building a distributed array in the codistributed way has several advantages over building it on the client and then distributing it:

1. The array might be too large to build entirely on one processor;
2. The array can be build much faster if each processor can set up its part;
3. Once the array is built, you skip the communication cost of distributing it.

DISTRIBUTED: Methods to Gather Data

When the **getLocalPart** command is used, a local copy is made of that part of the distributed array. This is a separate object from the distributed array!

Changes to the local part don't immediately affect the distributed array. To overwrite the distributed array with the (modified) local parts, the workers can issue the command:

```
ad = gather ( al )
```

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 the 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.

DISTRIBUTED: Conjugate Gradient Setup

```
n = 1400;
nonzer = 7;
lambda = 20;
niter = 15;
nz = n * ( nonzer + 1 ) * ( nonzer + 1 ) + n * ( nonzer + 2 );

A = sprand ( n, n, 0.5 * nz / n^2, codistributor ( ) );
A = 0.5 * ( A + A' );

I = speye ( n, codistributor ( ) );
A = A - lambda * I;

x = ones ( n, 1 );

for iter = 1 : niter
  [ z, rnorm ] = cgit ( A, x );
  zeta = lambda + 1 / ( x' * z )
  x = z / norm ( z );
end
```

sprand sets up a sparse random array **A**.
speye sets up a sparse identity matrix **I**.
The **codistributor()** qualifier means **A** and **I** are distributed across the workers, and built in a codistributed way.

DISTRIBUTED: Conjugate Gradient Iteration

```
function [ z, rnorm ] = cgit ( A, x )

  z = zeros ( size ( x ) );
  r = x;
  rho = r' * r;
  p = r;

  for i = 1 : 15
    q = A * p;
    alpha = rho / ( p' * q );
    z = z + alpha * p;
    rho0 = rho;
    r = r - alpha * q;
    rho = r' * r;
    beta = rho / rho0;
    p = r + beta * p;
  end

  rnorm =  norm ( x - A * z );

  return
end
```

The conjugate gradient iteration code is identical to code that would be used if **A** was an ordinary MATLAB array.

DISTRIBUTED: Comment

In this 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 figure out what that **codistributor()** call is doing; 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 **x=A\b** can't be used yet for sparse distributed matrices.
- 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!

DISTRIBUTED: a Finite Element Heat Code

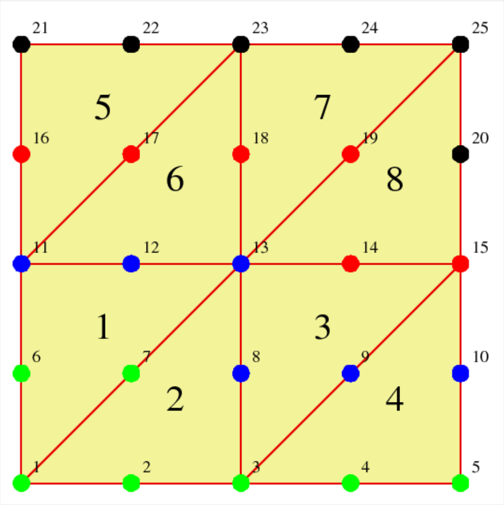
Professor Gene Cliff at Virginia Tech has prepared a program that combines SPMD and distributed data to solve the steady state heat equations in 2D, using the finite element method.

He assigns a subset of the finite element nodes to each worker. 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 **fem2d_steady_heat_spmd**.

DISTRIBUTED: The Grid



DISTRIBUTED: Finite Element System matrix

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

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

where **M** 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 **M** and the vectors **F** and **b** are distributed arrays. The default distribution of **M** by columns essentially associates each SPMD worker with a group of finite element nodes.

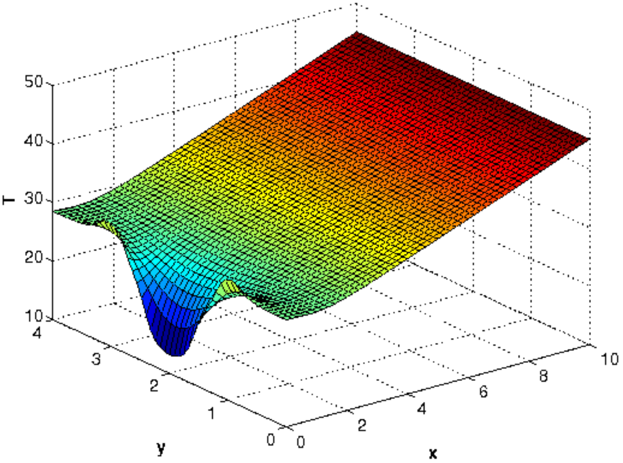
DISTRIBUTED: Finite Element System Matrix

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 **K** associated with worker nodes are added to the local part of $M_{i,j}$. 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 **M**, 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.

DISTRIBUTED: The Results



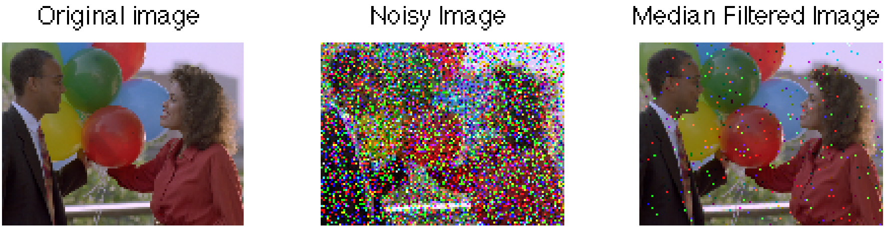
MATLAB Parallel Computing

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

IMAGE: Image Processing in Parallel

```
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' );
```

IMAGE: Image → Noisy Image → Filtered Image



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

IMAGE: Image → Noisy Image → Filtered Image

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

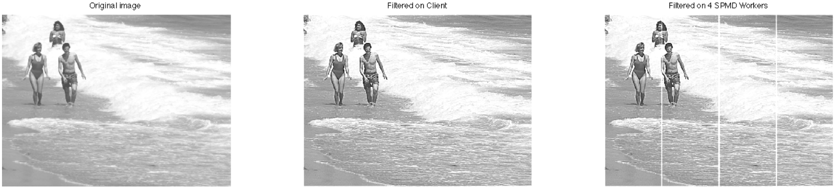
MATLAB Parallel Computing

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



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
```

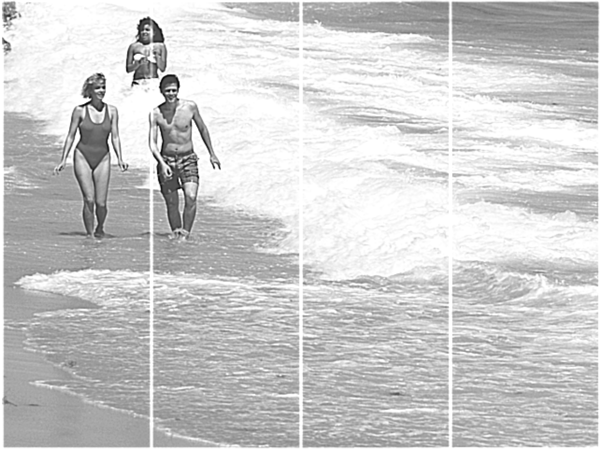
CONTRAST: Image → Contrast Enhancement → Image2

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!

CONTRAST: Image → Contrast Enhancement → Image2



The result is the spurious lines on the processed image.

MATLAB Parallel Computing

- 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
```

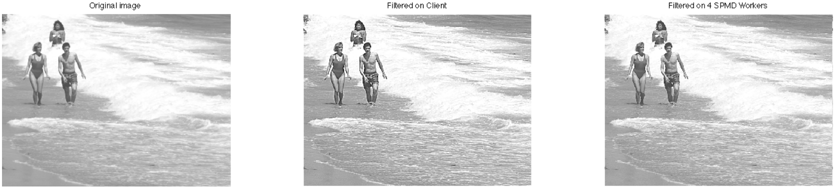
CONTRAST2: First Column Left, Last Column Right

```
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

```
  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

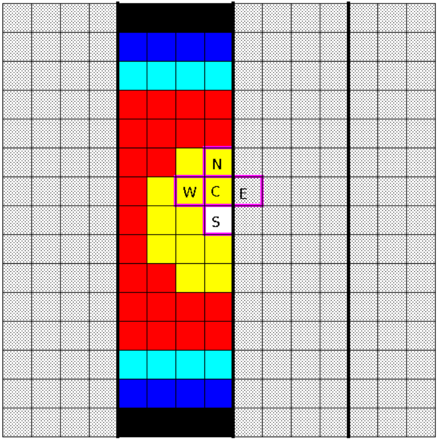
I have used image processing to illustrate this example, but consider the fact that the contrast enhancement operation updates values by comparing them to their neighbors.

The very same kind of operation applies in the **heat equation**, except, of course that there, heat differences tend to average out!

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.

That means 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 sent by message in order for the computation to proceed.

CONTRAST2: The Heat Equation

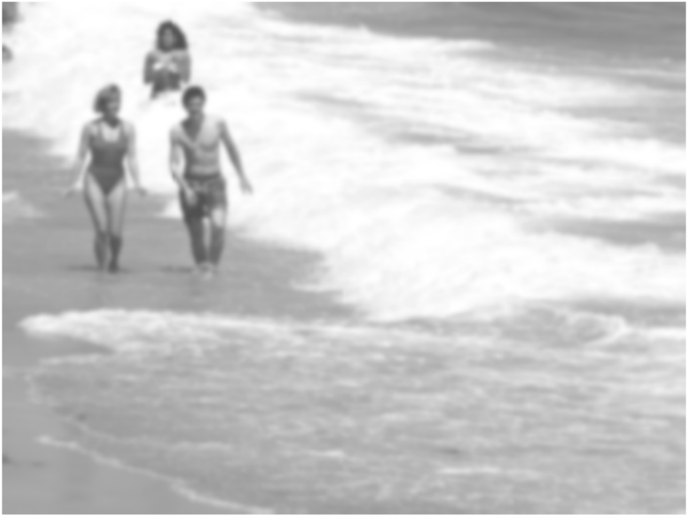
So now it's time to modify the image processing code to solve the heat equation.

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

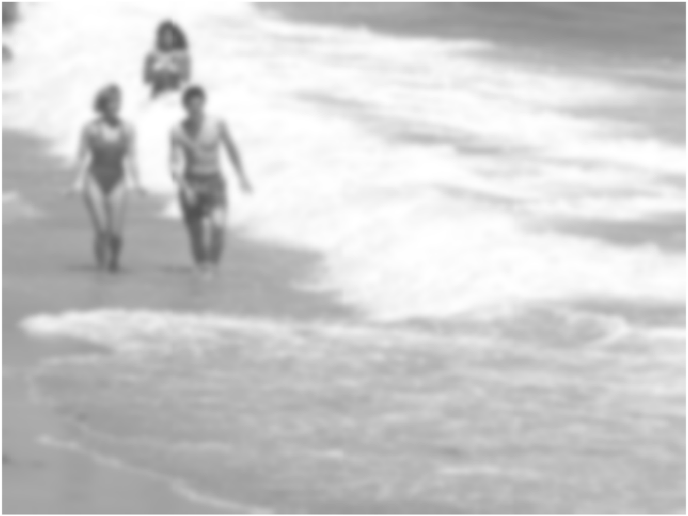
In contrast to the contrast example, the heat equation tends to smooth out differences. So let's watch our happy beach memories fade away ... in parallel ... and with no artificial boundary seams.

CONTRAST2: The Heat Equation, Step 0

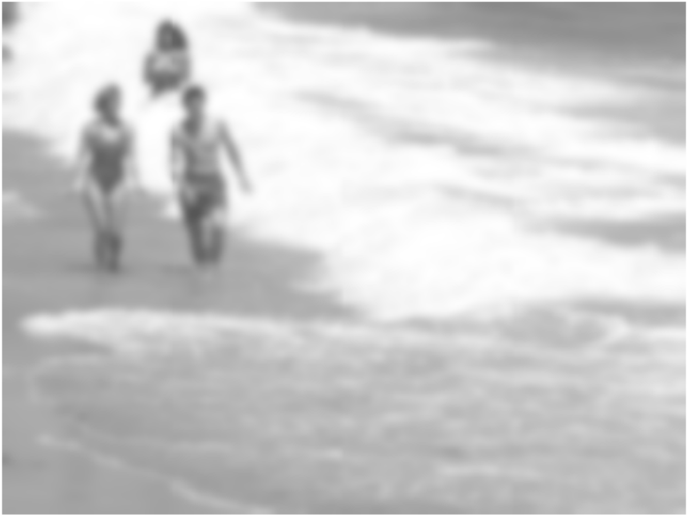
CONTRAST2: The Heat Equation, Step 10



CONTRAST2: The Heat Equation, Step 20



CONTRAST2: The Heat Equation, Step 40



CONTRAST2: The Heat Equation, Step 80

MATLAB Parallel Computing

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

BATCH: Indirect Execution

In the previous lecture, we looked at interactive execution with the **matlabpool** command. The **batch** command is an alternative approach which allows you to execute a MATLAB script in the background on your own machine.

The script can use **parfor** or **spmd** statements. The **batch** command includes a **matlabpool** argument that allows you to request a given number of workers.

Since the batch job will not use your current MATLAB session, your **matlabpool** request must ask for one more worker than normal, to play the role of the client!

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!
%
  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', 5 )             <-- 4 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 perhaps remote, ...
  'FileDependencies', 'file' or {'file1','file2'}, ...
  'CaptureDiary', 'true', ...
  '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.

MATLAB Parallel Computing

- 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 CG example showed that many MATLAB operations work for distributed arrays, a kind of array storage scheme associated with SPMD. It takes some effort to understand and use distributed arrays, but this is a key idea for sophisticated SPMD programming.

The IMAGE examples showed us interesting cases in which a problem 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 kind of MPI message system.

CONCLUSION: Where is it?

The Parallel Computing Toolbox is installed on the HPC login nodes (2 licenses each) and there are 16 DCE licenses on the HPC compute nodes.

FSU's Department of Scientific Computing has received 20 extra, temporary licenses for the Parallel Computing Toolbox.

It's available on classroom machines **class01** through **class10** and the public machines **hallway-b** through **hallway-f**, and valid through April 21.

Run it by typing **/scratch/R2010aTrial/bin/matlab**

CONCLUSION: Where is it?

The temporary license includes lots of extras!:

- Curve Fitting Toolbox
- Image Processing Toolbox
- Optimization Toolbox
- Parallel Computing Toolbox
- Signal Processing Blockset
- Signal Processing Toolbox
- Statistics Toolbox
- Symbolic Math Toolbox

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.
- FSU HPC web site: www.hpc.fsu.edu/
- http://people.sc.fsu.edu/∼burkardt/m_src/m_src.html
    - **quad_spmd**
    - **cg_distributed**
    - **fd2d_heat_explicit_spmd**
    - **fem2d_heat_steady_spmd**
    - **image_spmd**
    - **contrast_spmd** and **contrast2_spmd**