Parallel MATLAB:
Single Program Multiple Data

John Burkardt (ARC/ICAM) & Gene Cliff (AOE/ICAM)
3pm - 4pm, Friday, 25 June 2010,
3060 Torgersen Hall

....

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

....

http://people.sc.fsu.edu/~jburkardt/presentations/...
spmd_2010_vt.pdf

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

Lecture #1: PARFOR - June 4

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).

Lecture #3: TASKS - July 2

Task computing, described in a later lecture, allows us to define a computation as a single job that comprises many tasks. How these tasks are defined, run and collated is chosen by the user.

In **Parallel task computing**

- each program executes the same function;
- the programs must execute at the same time;
- the programs can communicate with each other;
- at completion, the user can examine the function output;

Parallel task computing can do *systolic programming*, where data is processed by passing from one program to the next.

Scattered Task computing allows us to run a lot of programs independently:

- each program might be executing the same function, or not;
- the programs may execute at any time, in any order;
- the programs do not communicate with each other;
- at completion, the user can examine the function output;

Scattered task computing could be used, for instance, to run 1,000 programs, each running a simulation with different input values.

SPMD: is Single Program, Multiple Data

Lecture #2: SPMD - June 25

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.

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_2010b', 'matlabpool', 7 )
```

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.

```
                    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.
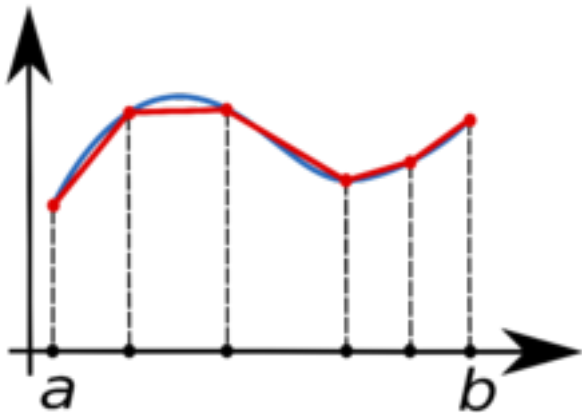
MATLAB Parallel Computing

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

Area of one trapezoid = average height * base.

QUAD: The Trapezoid Rule

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

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

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

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

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

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
```

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
```

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
```

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
```

```
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
```

MATLAB Parallel Computing

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

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 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 **spmd** 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.

Once the client has distributed the matrix by the command

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

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

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

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

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

Notice that local and global column indices will differ!

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};
```

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

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

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

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

or the client can access and concatenate local parts.

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.

```
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.

```
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
```

This conjugate gradient iteration code is the same, whether **A** is
an ordinary MATLAB array of doubles, or codistributed.

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 $x=A \backslash b$ 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

Another example of a program that combines SPMD and distributed data solves the 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 **fem2d_steady_heat_spmd.m**.
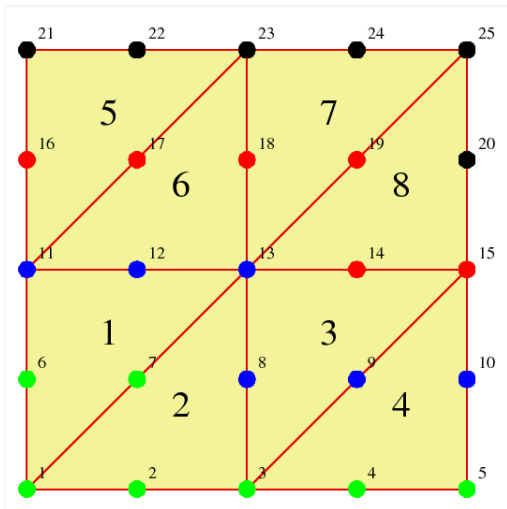
The discretized heat equation results in a linear system of the form

$$M z = F + b$$

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.

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.
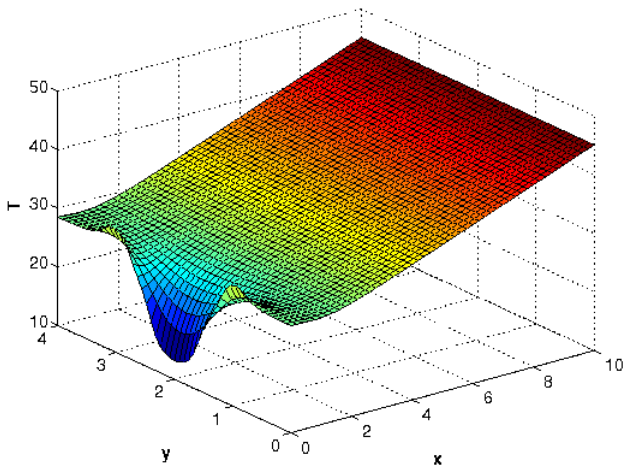
MATLAB Parallel Computing

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

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.

```matlab
%  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
- FACE Example
- Batch Computing
- Conclusion

```
%
%  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
```

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?

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
```

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.

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

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 );
```

```
spmd

  xl = getLocalPart ( xd );

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

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

```
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
```

```
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
```

Original Image

Filtered on Client

Filtered on 4 SPMD Workers

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.

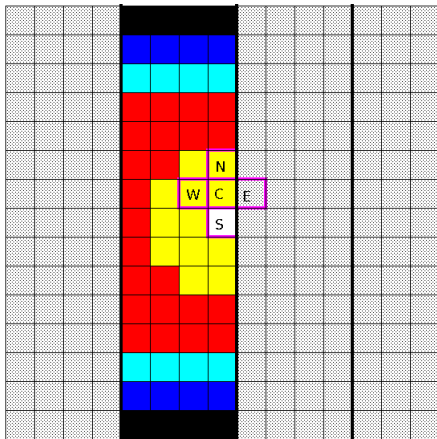The same operation applies in the **heat equation**, except that high contrasts (hot spots) tend to average out (cool off)!

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.

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

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.

MATLAB Parallel Computing

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

Many computations involve a geometric region. Often, this smooth and continuous region must be replaced by a collection of squares or triangles, cubes or tetrahedrons, in order to make the computation possible.

Geometric dissection is a very common task in computer graphics, computational physics, and computer-aided design.

Once the region has been dissected into many pieces, an important task is to determining which pieces lie on the boundary. This is because, as part of the computation, we will need to look at certain effects or forces that are transmitted through the boundary of the region.
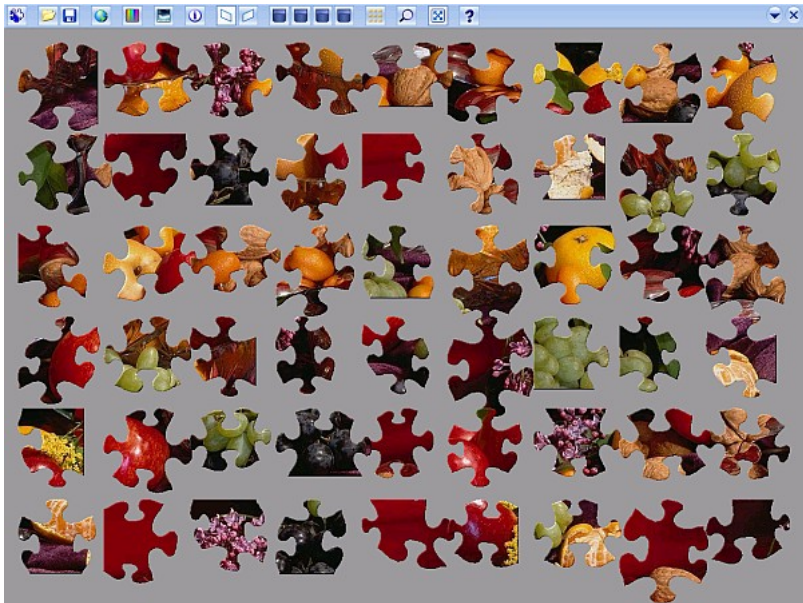
FACE: The Finite Element Data

Our problem is logically the same as the jigsaw puzzle problem.

Because we are using the finite element method, and our region is 3D, there are some differences:

- The region is a 3D box of dimensions 1x1x3;
- We have made a list of 3D points that lie in the box;
- Every piece of the region is the same shape, a tetrahedron;
- Each tetrahedron is described by listing 4 of the points;
- A "boundary" tetrahedron has a triangular face on the boundary;

We will simplify our problem by imagining we only need to find the tetrahedrons that touch the boundary face Z=3.

1x1x3 Channel
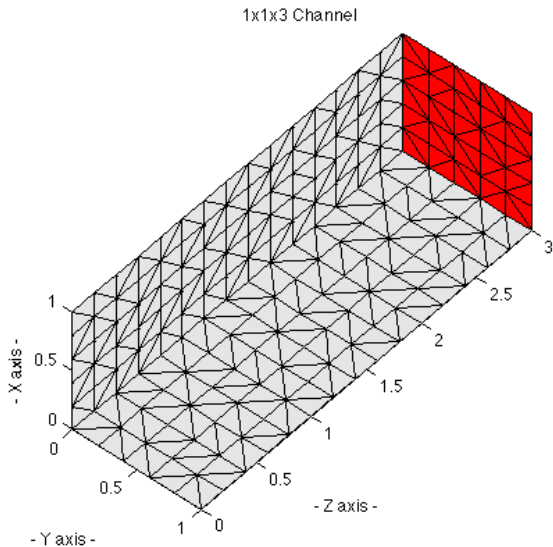
FACE: Parallel Solution

Just as a group can cooperate to find the edge pieces in a jigsaw puzzle, parallel MATLAB can be used to list the faces that lie on the boundaries.

Each SPMD worker:

- is assigned some elements by **getLocalPart**;
- starts a list for triangular faces (triples of points);
- examines each of the four faces of an element;
- a face whose three points lie in the plane is listed;

Then the client collects and concatenates the lists.

```
n_els = size ( e_conn, 2 );

spmd
  f_conn = zeros(3,floor(n_els/(20*numlabs());
  lab_els = getLocalPart( codistributed.colon(1,n_els) );
  for ne = lab_els
    nodes_loc = e_conn(:,ne);
    if all ( x_nodes([1 3 2], ii ) >= val_ii )
      if ( in_box ( xnodes([1 3 2],:), ...
        ii, xj_lb, xj_ub, xk_lb, xk_ub ) );
        nf = nf + 1;
        f_conn(:,nf) = nodes_loc([1 3 2] );
      end
    else Check [1 2 4], [1 4 3] and [2 3 4]
    end
  end
  f_conn = f_conn(:,1:nf);
end
```

FACE: The Client Assembles the Lists

Each worker made its own list called **f_conn**, of length **nf**. The client can sees *all* these partial lists; it specifies which one it is referring to by using an index.
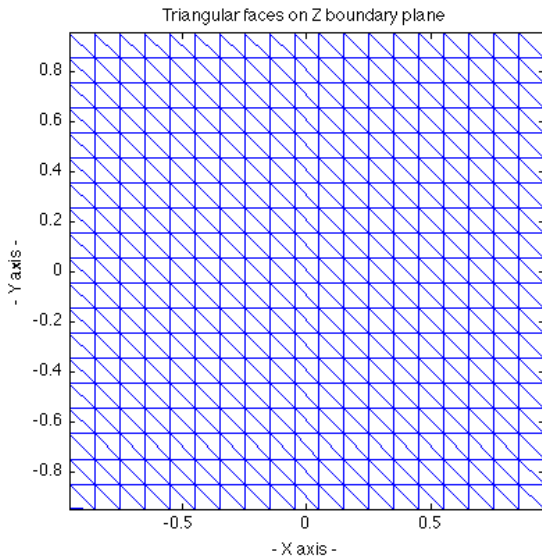
An easy way to figure out how many partial lists there are is simply to request the length of the **nf** array, that is, how many versions there are.

Although we preallocated space for each **f_conn** array, we "squeezed out" the unused entries, so now we can simply concatenate them to get the full list:

```
F_conn = [];
for j = 1 : length ( nf )
  F_conn = [ F_conn, f_conn(:, 1:nf{j} ) ];
end
```

Triangular faces on Z boundary plane

MATLAB Parallel Computing

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

```matlab
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 );
```

```
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
```

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' )
```

```
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.

BATCH: 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} );
```

MATLAB Parallel Computing

- SPMD: Single Program, Multiple Data
- QUAD Example
- Distributed Arrays
- IMAGE Example
- CONTRAST Example
- CONTRAST2: Messages
- FACE Example
- 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 DISTRIBUTED example showed that many MATLAB operations work for distributed arrays, a kind of array storage scheme associated with SPMD.

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 system.

The FACE example illustrated an interesting computation in which SPMD parallelism is the natural way to proceed.

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.

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.

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://people.sc.fsu.edu/∼jburkardt/m_src/m_src.html
  - **cg_distributed**
  - **contrast_spmd** and **contrast2_spmd**
  - **face_spmd**
  - **fd2d_heat_explicit_spmd**
  - **fem2d_heat_steady_spmd**
  - **image_spmd**
  - **prime_spmd**
  - **quad_spmd**