MATLAB Parallel Computing

John Burkardt
Information Technology Department
Virginia Tech

....

FDI Summer Track V:
Using Virginia Tech High Performance Computing
http://people.sc.fsu.edu/∼jburkardt/presentations/fdi_2009_matlab.pdf

26-28 May 2009

MATLAB Parallel Computing: Things Change

"Why There Isn't Parallel MATLAB"

"There actually have been a few experimental versions of MATLAB for parallel computers... We have learned enough from these experiences to make us skeptical about the viability of a fully functional MATLAB running on today's parallel machines."

Cleve Moler, 1995.

MATLAB Parallel Computing: Things Change

(Let There Be) “Parallel MATLAB”

"We now have parallel MATLAB."

Cleve Moler, 2007.

MATLAB Parallel Computing

Introduction: MATLAB

MATLAB is a computing environment that is halfway between a programming language (where a user must do everything) and a menu-driven application (where the user only makes high level decisions).

Users always have the ability to lay out the precise details of an algorithm themselves.

They rely on MATLAB commands to access intelligent, flexible, and optimized versions of standard algorithms.

Introduction: MATLAB Adds Parallelism

MATLAB has recognized that parallel computing is necessary for scientific computation.

The underlying MATLAB core and algorithms are being extended to work with parallelism.

An explicit set of commands has been added to allow the user to request parallel execution or to control distributed memory.

New protocols and servers allow multiple copies of MATLAB to carry out the user's requests, to transfer data and to communicate.

MATLAB's parallelism can be enjoyed by novices and exploited by experts.

Introduction: Local Parallelism

MATLAB has developed a *Parallel Computing Toolbox* which is required for all parallel applications.

The Toolbox allows a user to run a job in parallel on a desktop machine, using up to 4 "workers" (additional copies of MATLAB) to assist the main copy.

If the desktop machine has multiple processors, the workers will activate them, and the computation should run more quickly.

This use of MATLAB is very similar to the shared memory parallel computing enabled by OpenMP; however, MATLAB requires much less guidance from the user.

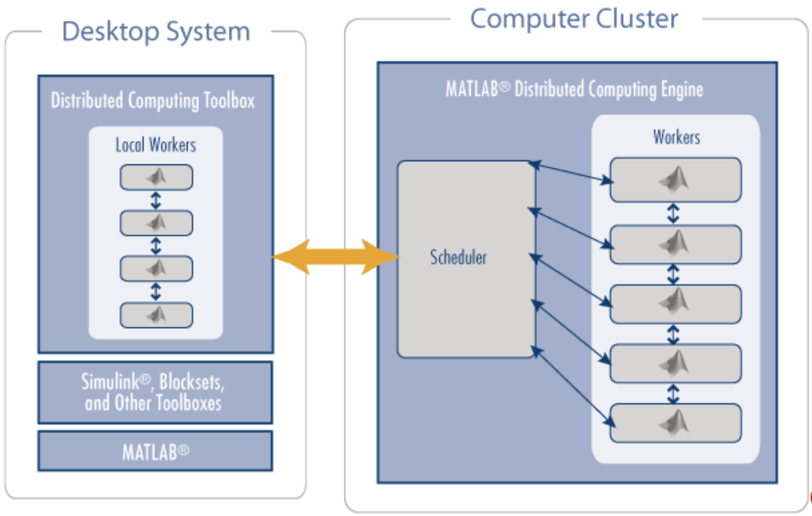
Introduction: Remote Parallelism

MATLAB has developed a *Distributed Computing Server* or **DCS**.

Assuming the user's code runs properly under the local parallel model, then it will also run under **DCS** with no further changes.

With the **DCS**, the user can start a job on the desktop that gets assistance from workers on a remote cluster.

Introduction: Local and Remote MATLAB Workers


Desktop System
Distributed Computing Toolbox
Local Workers
Simulink®, Blocksets, and Other Toolboxes
MATLAB®
Computer Cluster
MATLAB® Distributed Computing Engine
Workers
Scheduler

Introduction: SPMD for Distributed Data

If a cluster is available, the shared memory model makes less sense than a distributed memory model.

In such a computation, very large arrays can be defined and manipulated. Each computer does not have a copy of the same array, but instead a distinct portion of the array. In this way, the user has access to a memory space equal to the sum of the memories of all the participating computers.

MATLAB provides the **spmd** command to allow a user to declare such distributed arrays, and provides a range of operators that are appropriate for carrying out computations on such arrays.

Introduction: BATCH for Remote Jobs

MATLAB also includes a **batch** command that allows you to write a script to run a job (parallel or not, remote or local) as a separate process.

This means you can use your laptop or desktop copy of MATLAB to set up and submit a script for running a remote job. You can exit the local copy of MATLAB, turn off your laptop or do other work, and later check on the remote job status and retrieve your results.

Many computer clusters that have parallel MATLAB installed require users to submit their jobs in batch mode.

Introduction: PMODE: Interactive Parallel Mode

A typical parallel MATLAB user working interactively still sees the familiar MATLAB command window, which we may think of as being associated with the "master" copy of MATLAB.

However, MATLAB also allows a user to open a *parallel command window*. This is known as **pmode**.

Commands given in **pmode** are executed simultaneously on all the workers. Within **pmode**, the user has access to distributed arrays, parallel functions, and message-passing functions that are not visible or accessible in the normal command window.

Introduction: MATLAB + MPI

Parallel MATLAB uses a version of MPI (MPICH2).

In most cases, a user is happy not to see the underlying MPI activity that goes on.

However, MATLAB includes a rich set of calls that allow the user to employ the typical MPI activities of sending and receiving messages, broadcasting data, defining synchronization barriers and so on.

MATLAB Parallel Computing

Local Parallel Computing

If your desktop or laptop computer is fairly recently, it is probably a multicore machine.

"Regular" MATLAB only gets the power of one core.

But MATLAB's local parallel computing option engages up to 4 cores at the same time.

You'll need:

- the right version of MATLAB;
- a copy of the Parallel Computing Toolbox;
- the source code of your MATLAB program.

Local Parallel Computing: What Software Do You Have?

Your MATLAB must be **version 2008a** or later.

(Sadly, you may find that MathWorks has stopped issuing new releases for your slightly old computer, such as my desktop Apple G5 PowerPC machine.)

Your MATLAB must include the **Parallel Computing Toolbox.**

To get a list of *all* your toolboxes, type:

```
ver
```

Local Parallel Computing: Running A Program

Suppose you have a MATLAB M-file modified to compute in parallel (we'll explain that later!).

To do local parallel programming, start MATLAB the regular way.

This copy of MATLAB will be called the *client* copy; the extra copies created later are known as *workers* or sometimes as *labs*.

Running in parallel requires three steps:

1. request a number of (local) workers;
2. issue the normal command to run the program. The client MATLAB will call on the workers for help as needed;
3. release the workers.

Local Parallel Computing: Example of Running A Program

Suppose you have an M file named *samson.m*.

To run *samson.m* in parallel, type:

```
matlabpool open local 4

samson

matlabpool close
```

On a cluster, we can replace **local** by another argument that will allow us to create workers on multiple machines!

Local Parallel Computing: Running A Program

If all is well, the program runs the same as before... but faster.

Output will still appear in the command window in the same way, and the data will all be available to you.

What has happened is simply that some of the computations were carried out by other cores in a way that was hidden from you.

The program may seem like it ran faster, but it's important to **measure** the time exactly.

Local Parallel Computing: Timing A Program

To measure the **speed** of a program, you can use **tic** and **toc**:

```
matlabpool open local 4
tic
samson
toc
matlabpool close
```

Local Parallel Computing: Timing A Program

To measure the **speedup** of a program, you can try different numbers of workers:

```
for labs = 0 : 4
    if ( 0 < labs ) matlabpool ( 'open', 'local', labs )
    tic
    samson
    toc
    if ( 0 < labs ) matlabpool ( 'close' )
end
```

Because **labs** is a variable, we use the “function” form of **matlabpool**.

MATLAB Parallel Computing

The PARFOR Command

The simplest way of parallelizing a MATLAB program focuses on the **for** loops in the program.

If a **for** loop is suitable for parallel execution, replace the word **for** by the word **parfor** (meaning *"parallel for"*).

When the MATLAB program is run in parallel, the work in each **parfor** loop will be distributed among the workers.

The PARFOR Command: When Can It Be Used?

What determines whether a **for** loop is suitable for parallelization?

The crucial question that must be answered satisfactorily is this:

Can the iterations of the loop be performed in any order without affecting the results?

If the answer is "yes", then generally the loop can be parallelized.

The PARFOR Command: When Can It Be Used?

As a simple example of a loop that can be parallelized, think about the task of normalizing each column of a matrix. We find the largest entry in a column and divide the column by that value.

What happens in each column is independent.

```
for j = 1 : n
  c = 0.0
  for i = 1 : m
    c = max ( c, a(i,j) );
  end
  if ( c ~= 0 )
    for i = 1 : m
      a(i,j) = a(i,j) / c;
    end
  end
end
```

The PARFOR Command: Whan Can it Be Used?

As one example of when the **parfor** command cannot be used, think of the following difference equation:

```
u(1:m) = rand (1, m );

for j = 1 : n
  for i = 2 : m - 1
    v(i) = u(i-1) - 2 * u(i) + u(i+1);
  end
  u(2:m-1) = v(2:m-1);
end
```

The iterations (on **j**) are not independent. Each iteration needs results from the previous one.

The PARFOR Command: Data Sharing

Using **parfor** for parallel computing is very similar to the OpenMP shared memory model.

The MATLAB workers correspond to separate threads of execution.

In general, we think of all the data as being “shared” - that is, every worker can see or change any variable.

Some kinds of variables cannot be shared so simply. This includes **private variables** and **reduction variables**. OpenMP requires you to identify such variables, but MATLAB will attempt to do this for you automatically.

The PARFOR Command: Reduction Variables

In some parallelization systems, special care must be taken for what are called **reduction variables**.

Typically, these occur when certain functions such as the **max**, **min**, **sum** or **prod** are applied to an indexed loop value.

The problem is that the loop iterations are not completely independent in such a calculation - they must cooperate, but in a very simple way.

MATLAB can parallelize a loop even if it contains reduction variables; the user does not need to make any special action.

The PARFOR Command: Reduction Example

In this loop, the variables **total**, **big** and **fact** are reduction variables.

MATLAB is able to handle this calculation in parallel.
The user simply replaces **for** by **parfor**:

```
total = 0.0;
big = - Inf;
fact = 1;
for i = 1 : n
  total = total + x(i);
  big = max ( big, x(i) );
  fact = fact * x(i);
end
```

The PARFOR Command: BREAK and RETURN

A loop containing **break** or **return** cannot run in parallel.

There is no way to tell in advance when loop will break.

This simple prime check is an example.

```
function value = prime ( i )
  value = 1;
  for j = 2 : i - 1;
    if ( mod ( i, j ) == 0 )
      value = 0;
      break
    end
  end
  return
end
```

MATLAB Parallel Computing

The PRIME_NUMBER Example

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

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

If we want the program to run longer, we increase the variable **N**. Doubling **N** makes the run time increase by a factor of 4.

Notice that this program includes a loop that has a **break** statement. That's OK, because we do not parallelize that loop!

The PRIME_NUMBER Example

```
function total = prime_number ( n )

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

  total = 0;

  for i = 2 : n

    prime = 1;

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

    total = total + prime;

  end

  return
end
```

The PRIME_NUMBER Example

We can parallelize the loop whose index is **I**. The computations for two different values of **I** are completely independent.

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

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

(...OK, we also have to use the **matlabpool** command...)

We don't need to specify **local** in the **matlabpool** command, because that is the default value.

The PRIME_NUMBER Example

```
lab_num_array = [ 0, 1, 2, 4 ];
for lab_num = lab_num_array(1:4)

  fprintf ( 1, '\n' );

  if ( 0 < lab_num )
    matlabpool ( 'open', 'local', lab_num )
  end

  n = 100;

  for i = 1 : 3
    tic;
    primes = prime_number_parallel ( n );
    wtime = toc;
    fprintf ( 1, '  %8d  %8d  %8d  %14f\n', lab_num, n, primes, wtime );
    n = n * 100;
  end

  if ( 0 < lab_num )
    matlabpool ( 'close' )
  end

end
```

The PRIME_NUMBER Example: Timing

```
PRIME_NUMBER_PARALLEL_RUN
  Run PRIME_NUMBER_PARALLEL with 0, 1, 2, and 4 labs.

     N          1+0        1+1        1+2   1+4

    50        0.067      0.179      0.176      0.278
   500        0.008      0.023      0.027      0.032
  5000        0.100      0.142      0.097      0.061
 50000        7.694      9.811      5.351      2.719
500000      609.764    826.534    432.233    222.284
```

The PRIME_NUMBER Example: Timing Comments

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

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

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

This data suggests two conclusions:

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

AND

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

MATLAB Parallel Computing

- Introduction
- Local Parallel Computing
- The PARFOR Command
- The PRIME_NUMBER Example
- **More About PARFOR**
- The MD Example
- Virginia Tech Parallel MATLAB Resource
- Conclusion

More Abour PARFOR: Temporary Variables

Another feature of some parallelization systems involves the treatment of **temporary variables**. In the simplest case, a temporary variable is a scalar that is redefined and then used within each iteration of a loop.

A temporary variable is often simply a shorthand for a cumbersome expression.

Many parallelization systems require such variables to be declared as *private*, so that each worker has its own copy of the variable.

MATLAB does not require any such special treatment of such temporary variables.

More Abour PARFOR: Temporary Variables

In this loop, the variable **angle** is a temporary variable.

MATLAB is able to handle this calculation in parallel without any special action from the user, who simply has to replace **for** by **parfor**:

```
for i = 1 : n
  angle = ( i - 1 ) * pi / ( n - 1 );
  t(i) = cos ( angle );
end
```

More Abour PARFOR: WrapAround Variables

The most common programming practice that will make parallelization impossible occurs when the data used in one iteration of the loop is not available until a previous iteration has been computed.

Parallel MATLAB can't handle such a calculation properly, since the iteration where the variable is set may happen on a different processor than the iteration where the variable is needed.

Moreover, the iteration that needs the value might be computed **before** the iteration that sets the value.

Sometimes the loop can be rewritten; other times, the problem cannot be fixed because the computation is inherently recursive.

More Abour PARFOR: WrapAround Example

Suppose you compute the X locations of a set of nodes this way:

```
dx = 0.25;
x = zeros (1,n);
for i = 2 : n
  x(i) = x(i-1) + dx;
end
```

Parallel MATLAB's cannot be applied to this loop as it is written. This loop assumes the iterations will be done in exactly the usual order.

Luckily, this calculation can be rewritten to parallelize.

More Abour PARFOR: WrapAround Variables

Another example of temporary variable use involves counting how many times some condition has occurred, perhaps using this value as an index to store the corresponding value.

In this loop, we are looking for nonzero entries of the matrix **A**, and storing them in a compressed vector. The variable **k**, which counts how many such entries we have seen so far, is a "wraparound" temporary, whose value, set in one loop iteration, is needed in a later loop iteration.

It is not possible to ask MATLAB to carry out this operation in parallel simply by replacing the **for** loop by a **parfor**. A better choice might be to explore the **find** command!

More Abour PARFOR: WrapAround Variables

```
k = 0;
for i = 1 : m
  for j = 1 : n
    if ( a(i,j) ~= 0.0 )
      k = k + 1
      a2(k) = a(i,j);
      i2(k) = i;
      j2(k) = j;
    end
  end
end
```

More Abour PARFOR: Recursive Variables

Suppose we approximate the solution of a differential equation:

```
dt = 0.25;
u = zeros (1,n);
for i = 2 : n
  u(i) = u(i-1) + dt * f( t, u(i-1) );
end
```

There is no way to parallelize this loop. The value of **u(i)** cannot be computed until the value of **u(i-1)** is known, and that means the loop iterations cannot be executed in arbitrary order.

Similar issues arise when a Newton iteration is being carried out.

More Abour PARFOR: MATLAB Gets Confused

PARFOR is easy because MATLAB does a lot of work for you.

MATLAB is usually intelligent enough to determine how to classify and handle the variables in a loop.

But if it gets confused, there is no easy mechanism to help MATLAB!

More Abour PARFOR: MATLAB Gets Confused

```
n = 100;
Q = zeros(n,n);

parfor i = 1 : n
  for j = i : n

    if ( j == i )
      Q(i,j) = 1;
    else
      Q(i,j) = sqrt ( i / j );
    end

  end
end

Q
```

More Abour PARFOR: MATLAB Gets Confused

MATLAB 2008b refuses to run this program, complaining that it cannot determine the status of the variable Q.

It's not clear what MATLAB is concerned about, and there's no way to convince MATLAB to compile the code.

One way to work around such an error is to store the inner loop results in a temporary vector. When the loop is exited, copy the temporary vector into Q.

More Abour PARFOR: MATLAB Gets Confused

```
n = 100;
Q = zeros(n,n);

parfor i = 1 : n

  z = zeros(1,n);

  for j = i : n

    if ( j == i )
      z(j) = 1;
    else
      z(j) = sqrt ( i / j );
    end

  end

  Q(i,:) = z;

end

Q
```

MATLAB Parallel Computing

- Introduction
- Local Parallel Computing
- The PARFOR Command
- The PRIME_NUMBER Example
- More About PARFOR
- **The MD Example**
- Virginia Tech Parallel MATLAB Resource
- Conclusion

The MD Example

The MD program runs a simple molecular dynamics simulation.

The problem size **N** counts the number of molecules being simulated.

The program takes a long time to run, and it would be very useful to speed it up.

There are many for loops in the program, but it is a mistake to try to parallelize everything!

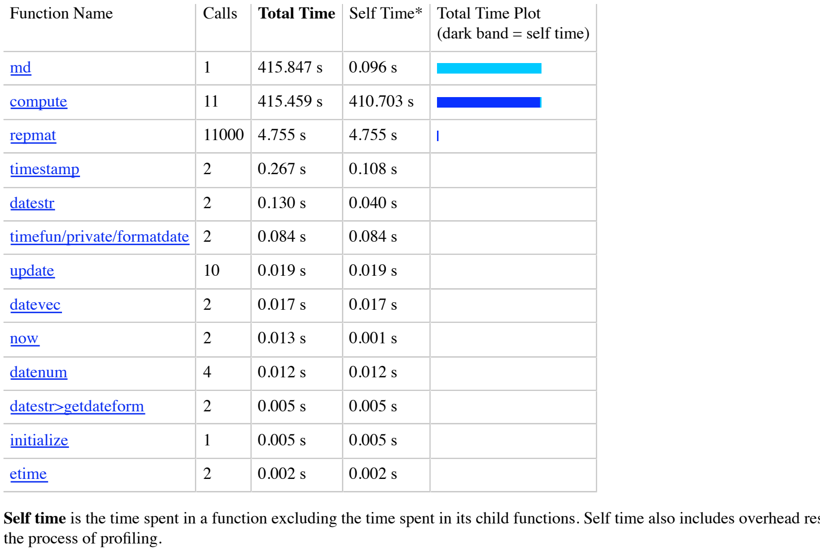
MATLAB has a **profile** command that can report where the CPU time was spent - which is where we should try to parallelize.

The MD Example: Run MATLAB's Profiler

```
profile on
md
profile viewer
```

```
Step        Potential        Kinetic          (P+K-E0)/E0
            Energy           Energy            Energy Error

   1    498108.113974        0.000000     0.000000e+00
   2    498108.113974        0.000009     1.794265e-11
 ...        ...                 ...            ...
   9    498108.111972        0.002011     1.794078e-11
  10    498108.111400        0.002583     1.793996e-11

     CPU time  = 415.740000 seconds.
     Wall time = 378.828021 seconds.
```

The MD Example



Function Name	Calls	**Total Time**	Self Time*	Total Time Plot (dark band = self time)
md	1	415.847 s	0.096 s	▬
compute	11	415.459 s	410.703 s	▬
repmat	11000	4.755 s	4.755 s	▏
timestamp	2	0.267 s	0.108 s	
datestr	2	0.130 s	0.040 s	
timefun/private/formatdate	2	0.084 s	0.084 s	
update	10	0.019 s	0.019 s	
datevec	2	0.017 s	0.017 s	
now	2	0.013 s	0.001 s	
datenum	4	0.012 s	0.012 s	
datestr>getdateform	2	0.005 s	0.005 s	
initialize	1	0.005 s	0.005 s	
etime	2	0.002 s	0.002 s	

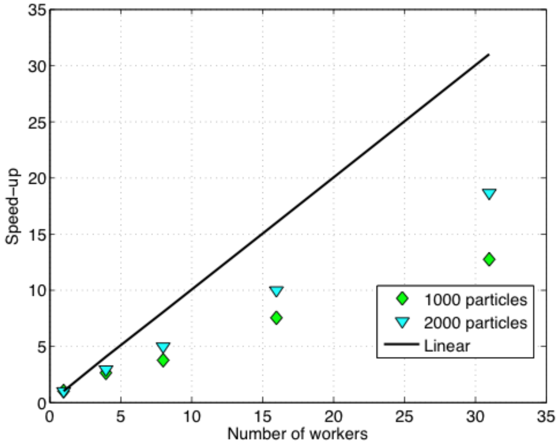
Self time is the time spent in a function excluding the time spent in its child functions. Self time also includes overhead res
the process of profiling.

The MD Example: The COMPUTE Function

```
function [ f, pot, kin ] = compute ( np, nd, pos, vel, mass )

  f = zeros( nd, np );
  pot = 0.0;
  pi2 = pi / 2.0;

  for i = 1 : np
    Ri = pos - repmat ( pos( :, i ), 1, np );     % array of vectors to 'i'
    D = sqrt ( diag ( Ri' * Ri ) );               % array of distances
    Ri = Ri( :, ( D > 0.0 ) );
    D = D( D > 0.0 );                             % save only pos values
    D2 = D .* ( D <= pi2 ) + pi2 * ( D > pi2 );   % truncate the potential.
    pot = pot + 0.5 * sum ( sin ( D2 ).^2 );      % accumulate pot. energy
    f( :, i ) = Ri * ( sin( 2*D2 ) ./ D );        % force on particle 'i'
  end

  kin = 0.5 * mass * sum ( diag ( vel' * vel) ); % kinetic energy

  return
end
```

The MD Example: Speedup

By inserting a PARFOR in COMPUTE, here is our speedup:



MATLAB Parallel Computing

VT Resources: The (Unsatisfactory) Past

Until recently, the central computing facilities at Virginia Tech had only very limited support for MATLAB.

MATLAB was only available on the Sun cluster known as **Dante**. (The MathWorks dropped support for the SGI Altix system some years ago).

Although the Parallel Computing Toolbox was available, there were some incompatibilities between the system and the software, which meant that parallel performance was poor.

Because of strong user interest, a better platform was needed.

VT Resources: The (Temporary) Present

The VT central computing facilities developed an experimental and temporary MATLAB cluster which shares file space with System X.

The cluster is known as **matlab1**.

It is relatively *small* and relatively *unknown*
It has been used to experiment with parallel MATLAB.

This cluster uses just 4 32-bit Xeon dual processors;
a user can get up to 8 parallel MATLAB workers.

A very limited number of users have access to **matlab1**.

VT Resources: Logging in to matlab1

Users access **matlab1** by first logging into a System X node: **sysx1**, **sysx2**, **sysx3** or **sysx4**.

They then can use **ssh** to log into **matlab1**.

```
ssh my_name@sysx1.arc.vt.edu <--from you to sysx1.

ssh matlab1                  <--from sysx1 to matlab1.
```

VT Resources: Logging in to matlab1

If you want MATLAB's command window to show up,
or you wish to have graphics displayed, then:

1. your terminal program must be an **xterm**;
2. your **ssh** connection to System X must use the -**X** switch;
3. your **ssh** connection to **matlab1** must use the -**X** switch;

```
ssh -X my_name@sysx1.arc.vt.edu <--from you to sysx1.

ssh -X matlab1                  <--from sysx1 to matlab1
```

VT Resources: Transferring files to matlab1

Since **matlab1** shares the **sysx** file system with the System X nodes, a user can move files back and forth simply by establishing an sftp connection to one of the System X nodes.

```
sftp my_name@sysx1.arc.vt.edu

cd sysx/my_project
put my_m_file.m
get my_output.txt

quit
```

VT Resources: Running MATLAB Interactively

On **matlab1**, MATLAB can be run interactively.

Connect from an X terminal; include -**X** on your **ssh** commands.

To access some of your M-files during the session, start the session from the directory that contains those files.

Start with the interactive command:

```
matlab
```

You will get the familiar interactive command window, which will stay until you enter **quit** or **exit**.

If you issue plot commands, the usual plot window will show up.

VT Resources: Running MATLAB Locally

Even though **matlab1** is a cluster, when you log in, you see only one machine. This particular machine has just 2 cores. You can run parallel MATLAB in the usual way, but you're only going to be able to ask for 2 workers!

```
matlabpool open local 2

samson

matlabpool close
```

VT Resources: Running MATLAB Over the Cluster

Since the cluster has 4 nodes, each with 2 processors, you can ask for 8 workers, but only if you replace the **local** switch by something that tells MATLAB how to access the other machines:

```
matlabpool open jmconfig1 8

samson

matlabpool close
```

VT Resources: Running MATLAB Indirectly

If your MATLAB program takes a long time to run, or is complicated, it may make sense to run MATLAB “indirectly”.

Technically, this is still an interactive session, but your input will now come from a file (perhaps **“input.m”**).

To keep **MATLAB** from trying to set up the command window you may want to include the **-nodisplay** switch.

The typical command gets a little complicated now:

```
matlab -nodisplay < input.m > output.txt &
```

The & at the end of the command line allows you to issue other commands while waiting for **matlab** to complete.

VT Resources: Moving to the Future

Experience with the experimental **matlab1** system has shown that parallel MATLAB runs efficiently, and that there is a good demand by users for more open access.

For these reasons, Virginia Tech has committed to creating a more powerful computer system to provide parallel MATLAB to the user community.

VT Resources: The (Immediate) Future

At Virginia Tech, a new computer cluster is arriving in June 2009, to be made available to users by July.

This IBM system is a cluster of 84 Nehalem processors, each having 8 cores, for a total of 672 cores.

The cluster will have 64 licenses for Parallel MATLAB.

This means that if there are several Parallel MATLAB jobs running on the cluster at the same time, the **total** number of workers, over all those jobs, can be no more than 64.

The number of simultaneous MATLAB licenses will be increased to 128 once there is enough usage.

MATLAB Parallel Computing

Conclusion: User Access

The new IBM cluster will encourage a parallel MATLAB community.

Any faculty, any graduate student will be able to request an account using:

IBM System Account Request Application Form

which will be available (once the system is ready) at:

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

You'll need your PID and password to access the form.

Conclusion: Job Submission

Short interactive debugging runs will probably be allowed on the login node.

To access multiple MATLAB workers, users will need to set up a configuration file, which will make it easy to submit batch jobs.

MATLAB's new **batch** feature will cooperate with VT's current queueing system.

Conclusion: MathWorks Training

The Mathworks will be presenting a class through the FDI which will present the features of parallel MATLAB in great detail.

This course is tentatively scheduled for June 2009.

Details about this class will be available soon.

Conclusion: Parallel MATLAB Is Ready to Try

Parallel hardware and communications software has become stable,

The MathWorks has realized that, in order to serve its engineering and science customers, it is time for it to offer access to parallel programming.

The **parfor** command is a simple way for many MATLAB programmers to get much of the benefit of parallel programming.

Conclusion: More Will Be Needed Soon

Right now, if MATLAB can't automatically handle a **parfor** loop, there is no easy way for the user to help.

Online documentation is very brief.

Distributed computing with **spmd** is somewhat "mysterious".

As Parallel MATLAB becomes popular, we hope there will be good textbooks, software packages and classes.