

# MATLAB Parallel Computing

John Burkardt (ARC/ICAM) & Gene Cliff (AOE/ICAM)  
Virginia Tech

.....

FDI Fall Short Course:

Introduction to Parallel MATLAB at Virginia Tech

[http://people.sc.fsu.edu/~jburkardt/presentations/  
fdi\\_matlab\\_2009.pdf](http://people.sc.fsu.edu/~jburkardt/presentations/fdi_matlab_2009.pdf) .....

**ARC:** Advanced Research Computing

**AOE:** Department of Aerospace and Ocean Engineering

**ICAM:** Interdisciplinary Center for Applied Mathematics

09 September 2009



# MATLAB Parallel Computing: Some Announcements



While we have your attention...



# MATLAB Parallel Computing: Some Announcements

**ITHACA** is an IBM iDataPlex cluster recently installed by Virginia Tech's Advanced Research Computing facility.

It is intended to gradually take over the high performance computing load from System X.

**ITHACA** supports OpenMP, MPI and Parallel MATLAB programming.

- **Right Now:** Friendly users accepted for Ithaca (*talk to John Burkardt if you are interested,*);
- **Mid October:** Ithaca opened to general users (*accounts given out through online application.*).

# MATLAB Parallel Computing: Some Announcements

**MATLAB Training** is available this fall.

These classes will be presented by the MathWorks. Some of these classes are tentative. Check the FDI website for details.

- **2 October:** Simulink (daylong);
- **3 October:** SimMechanics (daylong);
- **8 October:** MATLAB Programming Techniques.
- **8 October:** Parallel Computing with MATLAB.
- **29 October:** Parallel Computing with MATLAB (daylong).
- **19 November:** Real-time Data Acquisition and Control.
- **19 November:** Statistical Methods in MATLAB.



VirginiaTech

## “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.**



## (Let There Be) “Parallel MATLAB”

*“We now have parallel MATLAB.”*

**Cleve Moler, 2007.**

# MATLAB Parallel Computing

- **Introduction**
- Local Parallel Computing
- Parallelism with PARFOR
- The PRIME\_NUMBER Example
- Behind the Magic
- Bumps in the Road
- The MD Example
- Parallelism with SPMD
- Parallel Computing with `fmincon`
- Codistributed Arrays
- A 2D Heat Equation
- Parallel MATLAB at Virginia Tech
- Conclusion

# 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

The MathWorks 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 8 "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.

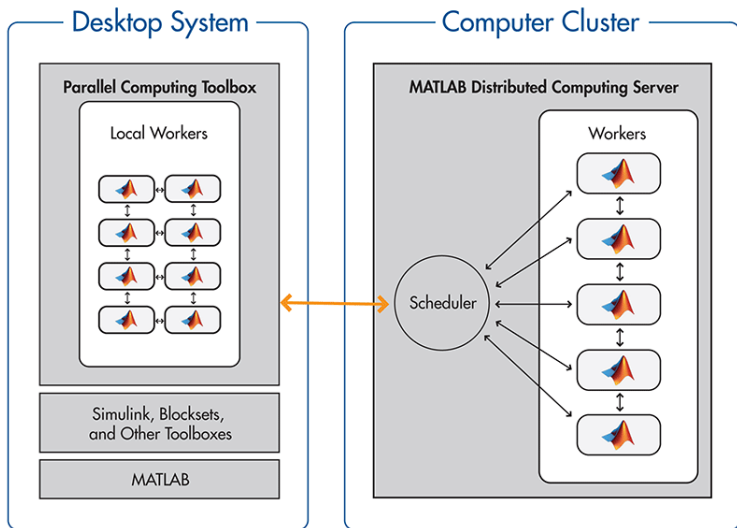
# Introduction: Remote Parallelism

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



# 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

- Introduction
- **Local Parallel Computing**
- Parallelism with PARFOR
- The PRIME\_NUMBER Example
- Behind the Magic
- Bumps in the Road
- The MD Example
- Parallelism with SPMD
- Parallel Computing with `fmincon`
- Codistributed Arrays
- A 2D Heat Equation
- Parallel MATLAB at Virginia Tech
- Conclusion



VirginiaTech

# Local Parallel Computing

If your desktop or laptop computer is fairly recent, it may have more than one processor; the processors may have multiple cores.

Executing MATLAB in the regular way only engages one core, leaving the others idle. (However, some MATLAB linear algebra routines will notice these idle cores and engage them for subtasks).

The Parallel Computing Toolbox runs up to 8 cooperating copies of MATLAB, taking direct advantage of the extra power.

You'll need:

- the right version of MATLAB;
- the Parallel Computing Toolbox;
- a MATLAB M-file that uses new parallel keywords.

# Local Parallel Computing: What Do You Need?

Your machine must have multiple cores.

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

\*\*\*To check MATLAB's version, go to the **HELP** menu, and choose **About Matlab**.

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

\*\*\*To list *all* your toolboxes, type the MATLAB command **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
```

When we want to run on a cluster, we'll only have to replace the word **local** by another suitable word, which defines the “configuration” (how MATLAB rounds up the workers.)

# 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 time a program, you can use **tic** and **toc**:

```
matlabpool open local 4
```

```
tic
```

```
samson
```

```
toc
```

```
matlabpool close
```

**tic** starts the clock, **toc** stops the clock and prints the time.



VirginiaTech

# 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
end
```

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



VirginiaTech



# MATLAB Parallel Computing

- Introduction
- Local Parallel Computing
- **Parallelism with PARFOR**
- The PRIME\_NUMBER Example
- Behind the Magic
- Bumps in the Road
- The MD Example
- Parallelism with SPMD
- Parallel Computing with `fmincon`
- Codistributed Arrays
- A 2D Heat Equation
- Parallel MATLAB at Virginia Tech
- Conclusion



VirginiaTech

# 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.

## REMARK: These Are Just Examples!

We're going to look at some examples of the use of the **parfor** keyword to make loops execute in parallel.

These are not realistic examples; the loops are short, and the operations are simple, and it would be easier to use MATLAB's built in operations to replace the entire loop by a simple command.

But these examples are for illustration, only. They give you an idea of how **parfor** can be used.

# 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.

# The PARFOR Command: When Can It Be Used?

```
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
end
```

# The PARFOR Command: When Can it Be Used?

As an example of when the **parfor** command *cannot* be used, consider 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. (We could, however, use a **parfor** on the **i** loop.)



# 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.

Variables that cannot be shared so simply include **private variables** and **reduction variables**.

OpenMP requires you to identify such variables; MATLAB tries to determine what do do implicitly, based on how the variables are used in the loop.



# The PARFOR Command: BREAK and RETURN

A loop containing **break** or **return** cannot run in parallel; the sequential code breaks at the first occurrence of the test. The parallel code can't do the same.

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

You cannot replace **for** by **parfor** here!



VirginiaTech

# MATLAB Parallel Computing

- Introduction
- Local Parallel Computing
- Parallelism with PARFOR
- **The PRIME\_NUMBER Example**
- Behind the Magic
- Bumps in the Road
- The MD Example
- Parallelism with SPMD
- Parallel Computing with `fmincon`
- Codistributed Arrays
- A 2D Heat Equation
- Parallel MATLAB at Virginia Tech
- Conclusion



VirginiaTech

# 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**, replacing **for** by **parfor**. The computations for different values of **i** are independent.

*But there's a break inside this loop! Why is that not a problem?*

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!

# The PRIME\_NUMBER Example: How to Run It

```
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 = 50;  
  
    while ( n <= 500000 )  
        tic;  
        primes = prime_number_parallel ( n );  
        wtime = toc;  
        fprintf ( 1, '___%8d___%8d___%8d___%14f\n', lab_num, n, primes, wtime );  
        n = n * 10;  
    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
- Parallelism with PARFOR
- The PRIME\_NUMBER Example
- **Behind the Magic**
- Bumps in the Road
- The MD Example
- Parallelism with SPMD
- Parallel Computing with `fmincon`
- Codistributed Arrays
- A 2D Heat Equation
- Parallel MATLAB at Virginia Tech
- Conclusion



VirginiaTech

# Behind the Magic:



What is behind the curtain of MATLAB's Magic?



# Behind the Magic: MATLAB's Implicit Parallelization

In other parallel languages, the user declares how variables are to be handled. This means more work, but also indicates that some ambiguous situations are safe to parallelize.

MATLAB figures out how to parallelize the user's code **implicitly**; that is, it looks at the code, and figures out by itself how the variables are used, and which ones must be treated in special ways. This is easy for the user, but it also means that some parallelizable operations **will not be parallelized** by MATLAB!

In particular, if an array is involved, MATLAB will only parallelize the loop if the array can be “sliced”, that is, if each iteration of the loop only refers to a distinct, separate portion of the array.

Let us look at how MATLAB's classification of your variables determines whether you can run in parallel.



# Behind the Magic: Variable Classification

```
a = 0;
c = pi;
z = 0;
r = rand ( 1, 10 );

parfor i = 1 : 10 <-- i is a "(parfor) loop index"
    a = i;           <-- a is a "temporary";
    z = z + i;       <-- z is a "reduction"
    b(i) = r(i);     <-- b and r are "sliced";
    if ( i <= c )    <-- c is "broadcast"
        d = 2 * a;   <-- d is "trouble"
    end
end
```



# Behind the Magic: Variable Classification

When MATLAB encounters a **parfor** loop, it must classify all the variables used in the loop, in order to determine how to carry out the operations in parallel.

MATLAB's variable classes are:

- **loop index**, the **parfor** index;
- **sliced**, any array uniquely indexed by the loop index;
- **broadcast**, defined before the loop, not set inside the loop;
- **reduction**, accumulates a result in a standard way;
- **temporary**, assigned, then used, in **each** iteration.

# Behind the Magic: The (parfor) Loop Index

The **loop index** is the index of the loop to which the **parfor** has been applied. The range of this index is divided up (in an unpredictable way) among the workers.

The loop index is also used to determine which arrays used in the loop must be divided among the workers.

Of course, the parfor loop may occur inside a loop, and may contain loops. The indices of those loops are treated as regular variables (broadcast or temporary).

MATLAB requires that the range of a parfor loop index be *consecutive integers*.

# Behind the Magic: Sliced Variables

**Sliced variables** are vectors or arrays whose entries can be uniquely associated with a particular loop iteration.

Assume the parfor loop index is **i**. Then the following are examples of sliced vectors:

```
parfor i = 1 : n
    a(i) = b(i) + c(i+1) + d(i-2) + e(n+1-i);
end
```

# Behind the Magic: Sliced Variables

Unfortunately, in many calculations, the way we use arrays does not correspond to what MATLAB wants. In particular, if loop iteration  $i$  works with vector entries  $i$  and  $i+1$ , then MATLAB cannot slice the variable the way it wants.

For example, entry  $x(45)$  is needed by two distinct loop iterations, with  $i=44$  and 45:

```
for i = 1 : n - 1
    dudx(i) = ( u(i+1) - u(i) ) / dx;
end
```



## Behind the Magic: Sliced Variables

Here are two ways to get around the problem:

```
up1(1:n-1) = u(2:n);  
parfor i = 1 : n - 1  
    dudx(i) = ( up1(i) - u(i) ) / dx;  
end
```

or

```
parfor i = 1 : n - 1  
    dudx(i) = u(i+1);  
end  
parfor i = 1 : n - 1  
    dudx(i) = ( dudx(i) - u(i) ) / dx;  
end
```

# Behind the Magic: Broadcast Variables

**Broadcast variables** are defined by what they are not:

- ❶ *not* the loop index;
- ❷ *not* sliced variables;
- ❸ *not* assigned a value inside the loop.

Broadcast variables are only used on the right hand side of assignment statements, and don't change during the loop.

So their initial values can be broadcast to all workers before the iterations begin, and there's nothing more to worry about.

## Behind the Magic: Broadcast Variables

In this loop, **x**, **dx**, **dy**, **j** and **c(j)** are broadcast variables.

*(Meanwhile, **i** is the (parfor) loop index, **u(i)** and **d(i)** are sliced variables, and **y** is a temporary.)*

```
dx = 0.25;
for j = 1 : n
    x = j * dx;
    dy = j * 0.01;
    parfor i = 1 : n
        y = i * dy;
        u(i) = c(j) * d(i) * f ( x, y ) / dx / dy;
    end
end
```



# Behind the Magic: Reduction Variables

**Reduction variables** occur when certain functions such as **max**, **min**, **sum** or **prod** are used iteratively.

These operations are “*semi-parallel*”, that is, each worker can compute part of the result, as long as the partial results are put together in the end in the correct way.

During parallel execution of the loop, a reduction variable **does not have a definable value** - it is really just a set of partial results.

MATLAB recognizes and automatically parallelizes many reductions.

# Behind the Magic: Reduction Variables

In this loop, **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
```

## Behind the Magic: Reduction Variables

Because they are shared over many workers, reduction variables do not have a definable value inside a parallel loop.

If your loop tries to test, use or print a reduction variable while it is being formed, MATLAB can't parallelize the loop.

```
total = 0.0;
for i = 1 : n
    total = total + x(i);
    if ( 1.0 < total )
        large_enough = 1;
    end
    cum(i) = total;
    fprintf ( 1, ' Current total is %f\n', total );
end
```

You cannot replace **for** by **parfor** here!



VirginiaTech

# Behind the Magic: Temporary Variables

**Temporary variables** are often shorthand for a long expression. Such a variable is redefined and then used within each iteration of a loop.

If MATLAB can't classify a variable otherwise, it **assumes** it is temporary. It then checks to ensure that, on every iteration, the variable is assigned and then used. If not, it generates an error.

So if you have mishandled a sliced or reduction variable, MATLAB will try to classify it as a temporary. You may get a confusing message about an uninitialized temporary variable.

# Behind the Magic: Temporary Variables

In this loop, **angle**, **nm1**, **c**, **s**, and **ui** are temporary variables.

MATLAB can parallelize this as soon as **for** is replaced by **parfor**:

```
u = rand ( 1, n );  
v = rand ( 1, n );  
for i = 1 : n  
    nm1 = n - 1;  
    angle = ( i - 1 ) * pi / nm1;  
    c = cos ( angle );  
    s = sin ( angle );  
    ui = u(i);  
    u(i) = c * u(i) + s * v(i);  
    v(i) = - s * ui + c * v(i);  
end
```





# MATLAB Parallel Computing

- Introduction
- Local Parallel Computing
- Parallelism with PARFOR
- The PRIME\_NUMBER Example
- Behind the Magic
- **Bumps in the Road**
- The MD Example
- Parallelism with SPMD
- Parallel Computing with `fmincon`
- Codistributed Arrays
- A 2D Heat Equation
- Parallel MATLAB at Virginia Tech
- Conclusion



VirginiaTech

# Bumps in the Road: And How to Avoid Some of Them



Are there hazards awaiting the new user?

## Bumps in the Road:

A big concern when using parallel programming is that the parallelized code can actually run **more slowly** than the original. (This is a *semantic* question, about programming performance.)

But right now we ask a more basic question - What programming causes MATLAB to refuse to parallelize your **parfor** loop. (This is a *syntactic* question, about programming language.)

If you enable *mlint* in MATLAB's editor, you will get warnings right there, with the offending loop highlighted, and an error message.

Otherwise, you will get error messages at run time, which are sometimes less specific.

# Bumps in the Road: WrapAround Variables

A 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.

## Bumps in the Road: 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.

# Bumps in the Road: 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!

# Bumps in the Road: 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
```



# Bumps in the Road: 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.



# Bumps in the Road: Arrays that Won't Slice

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 or to argue with it!

# Bumps in the Road: Arrays that Won't Slice

```
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
```



# Bumps in the Road: Arrays that Won't Slice

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

Q should “obviously” be a sliced variable. It’s an indexed variable whose entries can be divided up among different processors in a simple way.

However, MATLAB’s decisions are “final”, so if we really want the code to run in parallel, we have to reword it so that MATLAB is happy with it. One way is to use a vector in the loop, and copy its entries into Q.

# Bumps in the Road: Arrays that Won't Slice

```
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
```



## Bumps in the Road: Arrays that Won't Slice

Here is another example of a calculation that is completely parallel, in X, in Y, (and even in time T). Again, MATLAB refuses to parallelize the loop over X and Y.

Q should “obviously” be a sliced variable. It’s an indexed variable whose entries can be divided up among different processors in a simple way.

However, MATLAB’s decisions are “final”, so if we really want the code to run in parallel, we have to reword it so that MATLAB is happy with it. One way is to use a vector in the loop, and copy its entries into Q.

# Bumps in the Road: Arrays that Won't Slice

```
for t=1:Nt
    tm=dt*(t-1);
    a=eps*sin(w*tm);
    b=1-2*eps*sin(w*tm);
    for i=1:Nx+1
        for j=1:Ny+1
            f=a*x(i,j)*x(i,j)+b*x(i,j);
            dfx=2*a*x(i,j)+b;
            u(i,j,t)=-pi*A*sin(pi*f)*cos(pi*y(i,j));
            v(i,j,t)= pi*A*cos(pi*f)*sin(pi*y(i,j))*dfx;
        end
    end
end
```



## Bumps in the Road: Arrays that Won't Slice

If we turn the doubly-dimensioned array into a vector, MATLAB allows us to parallelize over the vector index  $K$ .

We have to use the relationships

$$k = i + (j - 1) * (N_x + 1);$$

$$i = \text{mod}(k, N_x + 1);$$

$$j = \text{floor}(k / (N_x + 1)) + 1;$$

or we could use the builtin MATLAB commands **ind2sub** and **sub2ind**.



## Bumps in the Road: Arrays that Won't Slice

```
for t=1:Nt
    tm=dt*(t-1);
    a=eps*sin(w*tm);
    b=1-2*eps*sin(w*tm);
    parfor k = 1 : kmax
%       i = mod ( k - 1, Nx + 1 ) + 1;
%       j = floor ( k / ( Nx + 1 ) ) + 1;
        f=a*x(k)*x(k)+b*x(k);
        dfx=2*a*x(k)+b;
        u(k,t)=-pi*A*sin(pi*f)*cos(pi*y(k));
        v(k,t)= pi*A*cos(pi*f)*sin(pi*y(k))*dfx;
    end
end
```





# MATLAB Parallel Computing

- Introduction
- Local Parallel Computing
- Parallelism with PARFOR
- The PRIME\_NUMBER Example
- Behind the Magic
- Bumps in the Road
- **The MD Example**
- Parallelism with SPMD
- Parallel Computing with `fmincon`
- Codistributed Arrays
- A 2D Heat Equation
- Parallel MATLAB at Virginia Tech
- Conclusion



VirginiaTech

# 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 Energy	Kinetic Energy	(P+K-E0)/E0 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)
<a href="#">md</a>	1	415.847 s	0.096 s	
<a href="#">compute</a>	11	415.459 s	410.703 s	
<a href="#">repmat</a>	11000	4.755 s	4.755 s	
<a href="#">timestamp</a>	2	0.267 s	0.108 s	
<a href="#">datestr</a>	2	0.130 s	0.040 s	
<a href="#">timefun/private/formatdate</a>	2	0.084 s	0.084 s	
<a href="#">update</a>	10	0.019 s	0.019 s	
<a href="#">datevec</a>	2	0.017 s	0.017 s	
<a href="#">now</a>	2	0.013 s	0.001 s	
<a href="#">datenum</a>	4	0.012 s	0.012 s	
<a href="#">datestr&gt;getdateform</a>	2	0.005 s	0.005 s	
<a href="#">initialize</a>	1	0.005 s	0.005 s	
<a href="#">etime</a>	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 from 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;

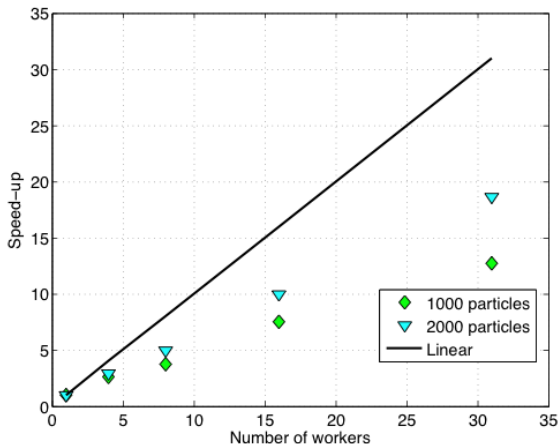
for i = 1 : np
    Ri = pos - repmat ( pos( :, i ), 1, np );    % array of vectors to 'i'
    D = sqrt ( sum ( Ri.^2 ) );                  % array of distances
    Ri = Ri( :, ( D > 0.0 ) );
    D = D( D > 0.0 );                             % save only pos values
    D2 = D .* ( D <= pi2 ) + pi2 * ( 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

- Introduction
- Local Parallel Computing
- Parallelism with PARFOR
- The PRIME\_NUMBER Example
- Behind the Magic
- Bumps in the Road
- The MD Example
- **Parallelism with SPMD**
- Parallel Computing with `fmincon`
- Codistributed Arrays
- A 2D Heat Equation
- Parallel MATLAB at Virginia Tech
- Conclusion



VirginiaTech

# The SPMD Command

**SPMD**: for **S**ingle **P**rogram, **M**ultiple **D**ata:

The **parfor** command 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 lab runs which iteration, we can't examine the work of any individual lab.

The **SPMD** command allows a programmer more control over how parallelism is done. Using **SPMD** is like working with a very simplified version of **MPI**.



# The SPMD Command

Let's assume we've issued a **matlabpool** command, and have a *client* (that is, the “main” copy of MATLAB) and a number of workers or labs.

The first thing to notice about a program using **SPMD** is that certain blocks of code are delimited:

```
fprintf ( 1, ' Set up the integration limits:\n' );  
spmd  
    a = ( labindex - 1 ) / numlabs;  
    b =   labindex       / numlabs;  
end
```

# The SPMD Command

The **spmd** delimiter marks a section of code which is to be carried out by each lab, 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**!

# The SPMD Command

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 lab 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
```

# The SPMD Command

Each worker will compute different values of **a** and **b**. These values are stored locally on that worker's memory.

The client can access the values of these variables, but it must specify the particular lab from whom it wants to check the value, using “curly brackets”: **a{i}**.

The variables stored on the workers are called *composite variables*; they are somewhat similar to MATLAB's cell arrays.

It's important to respect the rules for composite variable names. In particular, if **a** is used on the workers, then the name **a** is also “reserved” on the client program (although there it's an indexed variable). The client should not try to use the name **a** for other variables!

# The SPMD Command

So we could print all the values of **a** and **b** in two ways:

```
spmd
    a = ( labindex - 1 ) / numlabs;
    b =   labindex       / numlabs;
    fprintf ( 1, '  A = %f, B = %f\n', a, b );
end
```

or

```
spmd
    a = ( labindex - 1 ) / numlabs;
    b =   labindex       / numlabs;
end
for i = 1 : numlabs
    fprintf ( 1, '  A = %f, B = %f\n', a{i}, b{i} );
end
```



VirginiaTech

# The SPMD Command

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
```

# The SPMD Command

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:

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

with result:

```
Approximation 3.14159265
```

# QUAD\_SPMD Source Code

```
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
        if ( n == 1 )
            quad_part = ( b - a ) * f ( ( a + b ) / 2 );
        else
            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 );
        end
        fprintf ( 1, 'Approx%f\n', quad_part );
    end

    fprintf ( 1, 'Use_GPLUS_to_sum_the_parts.\n' );
    spmd
        quad = gplus ( quad_part );
        if ( labindex == 1 )
            fprintf ( 1, 'Approximation=%f\n', quad )
        end
    end

    return
end
```



# The SPMD Command

MATLAB also provides commands to combine values directly on the labs. The command we need is called **gplus()**; it computes the sum across all the labs of the given variable, and returns the value of that sum to each lab:

```
spmd
    x = linspace ( a, b, n );
    fx = f ( x );
    quad_part = ( fx(1) + 2 * sum(fx(2:n-1)) + fx(n) )
                /2 /(n-1);
    quad = gplus(quad_part);
    if ( labindex == 1 )
        fprintf ( 1, ' Approximation %f\n', quad );
    end
end
```



# The SPMD Command: Reduction Operators

**gplus()** is implemented by the **gop()** command, which carries out an operation across all the labs.

**gplus(a)** is really shorthand for **gop ( @plus, a )**, where **plus** is the name of MATLAB's function that actually adds numbers.

Other reduction operations include:

- **gop(@max,a)**, maximum of **a**;
- **gop(@min,a)**, minimum of **a**;
- **gop(@and.a)**, AND of **a**;
- **gop(@or.a)**, OR of **a**;
- **gop(@xor.a)**, XOR of **a**;
- **gop(@bitand.a)**, bitwise AND of **a**;
- **gop(@bitor.a)**, bitwise OR of **a**;
- **gop(@bitxor.a)**, bitwise XOR of **a**.



# The SPMD Command: MPI-Style Messages

SPMD supports some commands that allow the programmer to do message passing, in the MPI style:

- **labSend**, send data to another lab;
- **labReceive**, receive data from another lab;
- **labSendReceive**, interchange data with another lab.

For details, you will need to see the MATLAB HELP facility!

# MATLAB Parallel Computing

- Introduction
- Local Parallel Computing
- Parallelism with PARFOR
- The PRIME\_NUMBER Example
- Behind the Magic
- Bumps in the Road
- The MD Example
- Parallelism with SPMD
- **Parallel Computing with `fmincon`**
- Codistributed Arrays
- A 2D Heat Equation
- Parallel MATLAB at Virginia Tech
- Conclusion



VirginiaTech

# fmincon and UseParallel

In most cases, making use of parallelism requires some re-coding, perhaps even serious restructuring of your approach. Beginning with Version 4.0 (R2008a) of the Optimization Toolbox we can easily take advantage of parallelism in constructing finite-difference estimates of the gradient of the cost functional and the Jacobian of any nonlinear constraint functions.

Using the `optimset` command we simply set the flag `UseParallel` to (the string) `always`.

In the `run_opt` example we seek an optimal steering history for a boat moving in a spatially varying current. The control history is approximated as piecewise constant on a given time-grid. The optimization parameter is the vector of the values of the steering angle on the intervals. The cost functional and constraints depend on the final position of the boat in the plane.



The main work in evaluating these functions is the (numerical) integration of the dynamics with a prescribed steering history.

The dynamics are given by

$$\begin{aligned}\dot{x}(t) &= -\kappa y(t) + \cos(\theta(t)) \\ \dot{y}(t) &= \sin(\theta(t))\end{aligned}$$

with initial condition  $x(0) = y(0) = 0$ .

The problem is to maximize  $x(t_f)$  with the constraint  $y(t_f) > y_f$  ( $t_f, y_f$ , and  $\kappa$  are given).

# The RUN\_OPT Example

```
function z_star = run_opt(f_name, n)
% Function to run a finite dimensional optimization problem
% based on a discretization of a Mayer problem in optimal control.

% f_name points to a user-supplied function with a single input argument
% n is a discretization parameter. The finite-dimensional problem arises
% by treating the (scalar) control as piecewise constant
% The function referenced by f_name must define the elements of
% the underlying optimal control problem. See 'zermelo' as an example.

%% Problem data

    PAR = feval(str2func(f_name), n);

% some lines omitted

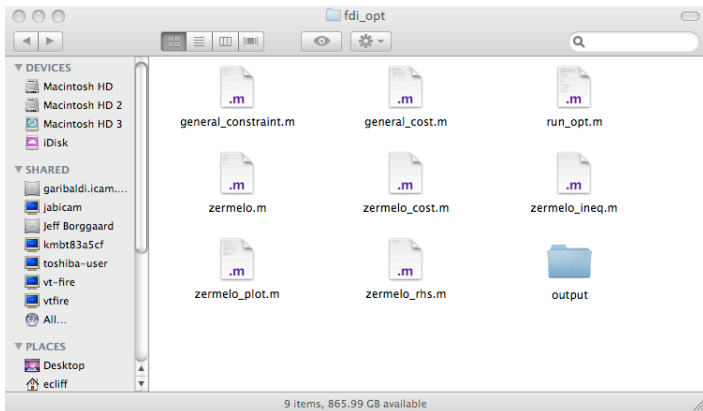
%% Algorithm set up
    OPT = optimset('fmincon'), ...
        'LargeScale','off', ...
        'Algorithm','active-set', ...
        'Display','iter', ...
        'UseParallel','Always');
    h_cost = @(z) general_cost( z, PAR);
    h_cnst = @(z) general_constraint( z, PAR);

%% Run the algorithm
    [z_star, f_star, exit] = ...
        fmincon(h_cost, z0, [], [], [], [], LB, UB, h_cnst, OPT);
    if exit >= 0 && isfield(PAR, 'plot')
        feval(PAR.plot, z_star, PAR)
    end
```



# The RUN\_OPT Example: source material

A folder with the software and example output is in the `parallel_matlab` folder on your desktop. The folder looks like:





# Cell Arrays

cell arrays are rectangular arrays, whose content can be any MATLAB variable, including a cell array

```
>> A = eye(2); B = ones(2); C = rand(3,4); D = 'a string';  
>> G = { A B ; C D};  
>> G
```

```
G =      [2x2 double]      [2x2 double]  
        [3x4 double]      'a string'
```

```
>> isa(G, 'cell')
```

```
ans =      1
```

# Cell Arrays: Two ways of indexing

A cell array may be indexed in two ways:

- 1  $G(1)$  - the result of cell indexing is a cell array
- 2  $G\{1\}$  - the result of content indexing is the contents of the cell(s)

```
>> F1 = G(1, 1:2)
```

```
F1 =      [2x2 double]      [2x2 double]
```

```
>> isa(F1, 'cell')
```

```
ans = 1
```

## Cell Arrays: Two ways of indexing

$G\{1\}$  - the result of content indexing is the cell's contents

```
>> F2 = G{1, 2}
```

```
F2=      1      1  
      1      1
```

```
>> whos
```

Name	Size	Bytes	Class	Attributes
A	2x2	32	double	
B	2x2	32	double	
C	3x4	96	double	
D	1x8	16	char	
F1	1x2	184	cell	
F2	2x2	32	double	
G	2x2	416	cell	

## SPMD mode: composite variables

SPMD mode creates a composite object on the client  
composite objects are indexed in the same ways as cell arrays

```
>> spmd
V = eye(2) + (labindex -1);
end
>> V{1}
ans = 1      0
      0      1
>> V{2}
ans = 2      1
      1      2
>> whos
```

Name	Size	Bytes	Class	Attributes
V	1x2	373	Composite	



VirginiaTech

# MATLAB Parallel Computing

- Introduction
- Local Parallel Computing
- Parallelism with PARFOR
- The PRIME\_NUMBER Example
- Behind the Magic
- Bumps in the Road
- The MD Example
- Parallelism with SPMD
- Parallel Computing with `fmincon`
- **Codistributed Arrays**
- A 2D Heat Equation
- Parallel MATLAB at Virginia Tech
- Conclusion



VirginiaTech

# Codistributed Arrays

Codistributed arrays allow the user to build ( $m \times n$ ) matrices so that, for example, each 'lab' stores/operates on a contiguous block of columns. More general (rectangular) constructs are possible but are not covered here.

We shall demonstrate these ideas in `pmode`

```
>> pmode start 4
```

```
Starting pmode using the parallel configuration 'local'.  
Waiting for parallel job to start...  
Connected to a pmode session with 4 labs.
```

Many of the builtin Matlab matrix constructors can be assigned the class 'codistributed'. For example:

```
M = speye(1000, codistributor());
```



VirginiaTech

## Codistributed arrays (cont'd)

'codistributor' is the constructor and specifies which dimension is used to distribute the array. With no argument, we take the default, which is '1d' or one-dimensional. By default, two dimensional arrays are distributed by columns.

**codistributor(M)** returns information about the distributed structure of the array **M**.

If the number of columns is an integer multiple of the number of 'labs', then the (default) distribution of columns among the labs is obvious. Else we invoke `codistributor` (or other MATLAB supplied procedure).

**localPart(M)** returns the part of the codistributed array on this lab.

## Codistributed arrays (cont'd)

```
%%% run these in Matlab
pmode start 4
M = speye(1000, codistributor() )
M = ones(1000, codistributor() )
codistributor(M)

M = ones(1000, 1, codistributor() )
codistributor(M)
%%%
```



## Codistributed arrays (cont'd)

One can construct local arrays on the labs and assemble them into a codistributed array:

```
%%% run these in Matlab
M = rand(100, 25) + labindex;
Mc = codistributed(M);
max(max(abs(M - localPart(Mc))))
Mc(12,13)
%%%
```

Of course, in applications the construction on each lab will involve user-defined code. We will now demonstrate this idea for an unsteady heat equation in two space dimensions.

# MATLAB Parallel Computing

- Introduction
- Local Parallel Computing
- Parallelism with PARFOR
- The PRIME\_NUMBER Example
- Behind the Magic
- Bumps in the Road
- The MD Example
- Parallelism with SPMD
- Parallel Computing with `fmincon`
- Codistributed arrays
- **A 2D Heat Equation**
- Parallel MATLAB at Virginia Tech
- Conclusion



VirginiaTech

# 2D Heat Equation

An example: 2D unsteady heat equation

$$\sigma C_p \frac{\partial T}{\partial t} = \frac{\partial}{\partial x} \left( k_x \frac{\partial T}{\partial x} \right) + \frac{\partial}{\partial y} \left( k_y \frac{\partial T}{\partial y} \right) + F(x, y, t)$$
$$(x, y) \in \{(x, y) \mid 0 \leq x \leq L, \quad 0 \leq y \leq w\} \subset \mathbb{R}^2, \quad t > 0,$$

where:

- $F(x, y, t)$  is a specified source term,
- $\sigma > 0$  is the areal density of the material,
- $C_p > 0$  is the thermal capacitance of the material, and
- $k_x > 0$  ( $k_y > 0$ ) is the conductivity in the  $x$  direction (the  $y$ -direction).

## 2D Heat Equation (cont'd)

Boundary conditions for our problem are:

$$\frac{\partial T(x, 0)}{\partial y} = \frac{\partial T(x, w)}{\partial y} = 0 ,$$

$$k_x \frac{\partial T(L, y)}{\partial x} = f(y) ,$$

$$k_x \frac{\partial T(0, y)}{\partial x} = \alpha(y) (T(0, y) - \beta(y)) .$$

## 2D Heat Equation (cont'd)

We use backward Euler in time and finite-elements in space to arrive at

$$\int_{\Omega} \left( T^{n+1} - T^n - \frac{\Delta t}{\sigma C_p} F(x, y, t_{n+1}) \right) \Psi \, d\omega \\ + \frac{\Delta t}{\sigma C_p} \left[ \int_{\Omega} (k \nabla T^{n+1} \cdot \nabla \Psi) \, d\omega + \int_{\partial\Omega} (k \nabla T^{n+1} \cdot \hat{n}) \, \Psi \, d\sigma \right] = 0 ,$$

where  $T^n(x, y) \triangleq T(n \Delta t, x, y)$ , and  $\Psi \in H^1(\Omega)$  is a test function.

## 2D Heat Equation (cont'd)

Imposing the specified boundary conditions, the boundary term evaluates to

$$\begin{aligned} \int_{\partial\Omega} (k \nabla T^{n+1} \cdot \hat{n}) \Psi \, d\sigma &= \int_0^w f(y) \Psi(L, y) \, dy \\ &\quad - \int_w^0 \alpha(y) [T^{n+1}(0, y) - \beta(y)] \Psi(0, y) \, dy . \end{aligned}$$

Details are described in the `2D_heat_ex.pdf` file in the distribution material.

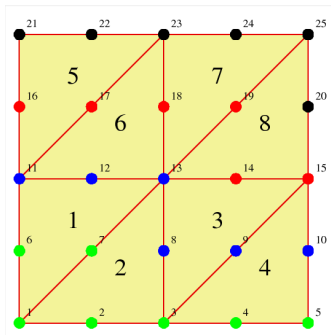
## 2D Heat Equation (cont'd)

We use quadratic functions on triangular elements

Impose a regular  $n_x \times n_y = ((2\ell + 1) \times (2m + 1))$  grid.

Using the odd-labeled points we generate  $\ell$   $m$  rectangles; diagonals divide these into  $2 \ell m$  triangles.

Here's the case  $n_x = n_y = 5$  (8 elements, 25 grid points):



## 2D Heat Equation (cont'd)

Seek an approximate solution:  $T_N^n(x, y) = \sum_{j=1}^N z_j^n \Phi_j(x, y)$ .

$$\begin{aligned} & \sum_j \left[ \int_{\Omega} \Phi_j(x, y) \Phi_i(x, y) d\omega \right. \\ & + \frac{\Delta t}{\sigma C_p} \left( \int_{\Omega} (k \nabla \Phi_j \cdot \nabla \Phi_i) d\omega + \int_w^0 \alpha(y) \Phi_j(0, y) \Phi_i(0, y) dy \right) \Big] z_j^{n+1} \\ & - \sum_j \left[ \int_{\Omega} \Phi_j(x, y) \Phi_i(x, y) d\omega \right] z_j^n - \left[ \frac{\Delta t}{\sigma C_p} \int_{\Omega} F(x, y, t_{n+1}) \Phi_i d\omega \right] \\ & - \frac{\Delta t}{\sigma C_p} \left[ \int_0^w f(y) \Phi_i(L, y) dy + \int_w^0 \alpha(y) \beta(y) \Phi_i(0, y) dy \right] = 0 \end{aligned}$$

In matrix terminology

$$(\mathbf{M}_1 + \mathbf{M}_2) \mathbf{z}^{n+1} - \mathbf{M}_1 \mathbf{z}^n + \mathbf{F}(t_{n+1}) + \mathbf{b} \quad \text{VirginiaTech}$$



# Modifying a Code

We began with a serial code for building  $\mathbf{M}_1$ ,  $\mathbf{M}_2$ ,  $\mathbf{F}$  and  $\mathbf{b}$ .

Here, we briefly note the changes to build codistributed versions of these.

# ASSEMB\_CO Source Code (begin)

```
function [M1, M2, F, b, x, e_conn] = assemb_co(param)
% The FEM equation for the temp. dist at time t_{n+1} satisfies
% (M_1 + M_2) z^{n+1} - M_1 z^n + F + b = 0

%% Initialization & geometry
%——lines omitted
%% Set up codistributed structure

% column pointers and such for codistributed arrays
Vc = codcolon(1, n_equations);
IP = localPart(Vc); IP_1 = IP(1); IP_end = IP(end);
dPM = distributionPartition(codistributor(Vc));
col_shft = [0 cumsum(dPM(1:end-1))];

% local sparse arrays
M1_lab = sparse(n_equations, dPM(labindex)); M2_lab = M1_lab;
b_lab = sparse(dPM(labindex), 1); F_lab = b_lab;

%% Build the finite element matrices — Begin loop over elements
for n_el=1:n_elements
    nodes_local = e_conn(n_el,:); % which nodes are in this element
    % subset of nodes/columns on this lab
    lab_nodes_local = my_extract(nodes_local, IP_1, IP_end);
    if ~isempty(lab_nodes_local) % continue the calculation for this elmnt
%—— calculate local arrays — lines omitted
    end
end
```

# ASSEMB\_CO Source Code (end)

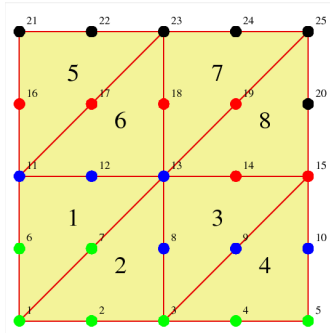
```
%% Assemble contributions into the global system matrices (on this lab)
%-----
%
for n_t = 1:nel_dof           % local DOF - test fcn
    t_glb = nodes_local(n_t); % global DOF - test fcn
    for n_u = 1:size(lab_nodes_local, 1)
        n_locj = lab_nodes_local(n_u, 1); % local DOF in current n-el
        n_glbj = lab_nodes_local(n_u, 2) ...
            - col_shft(labindex); % global DOF
        M1_lab(t_glb, n_glbj) = M1_lab(t_glb, n_glbj) ...
            + M1_loc(n_t, n_locj);
        M2_lab(t_glb, n_glbj) = M2_lab(t_glb, n_glbj) ...
            + param.dt*M2_loc(n_t, n_locj);
    end
end

%
if t_glb >= IP_1 && t_glb <= IP_end % is node on this lab ?
    t_loc = t_glb - col_shft(labindex);
    b_lab(t_loc, 1) = b_lab(t_loc, 1) - param.dt*b_loc(n_t, 1);
    F_lab(t_loc, 1) = F_lab(t_loc, 1) - param.dt*F_loc(n_t, 1);
end
end % for n_t
end % if not empty
end % n-el


%
% Assemble the lab contributions in a codistributed format
M1 = codistributed(M1_lab, codistributor('ld', 2));
M2 = codistributed(M2_lab, codistributor('ld', 2));
b = codistributed(b_lab, codistributor('ld', 1));
F = codistributed(F_lab, codistributor('ld', 1));
```

## Example: $5 \times 5$ grid on 4 labs

There are 8 triangular elements, and 25 nodes.  
The nodes are color-coded for the four labs.



Note that lab 1 (green) requires evaluation on 4 of 8 elements, while lab 2 (blue) requires 7 of 8.

Clearly, our naive nodal assignment to labs leaves the  VirginiaTech computational load badly balanced.

## Demo: $5 \times 5$ grid on 4 labs

```
%%% run these in Matlab
pmode start 4
Vc = codcolon(1, 25)
dPM = distributionPartition(codistributor(Vc))
col_shft = [ 0 cumsum(dPM(1:end-1))]
whos
%%%
```

# RUN\_ASSEMB\_CO Source Code

```
% Script to assemble matrices for a 2D diffusion problem

%% set path
addpath './subs_source/oned'; addpath './subs_source/twod'

%% set parameter values and assemble arrays
param = p_data();
[M1, M2, F, b, x, e_conn] = assemb_co(param);

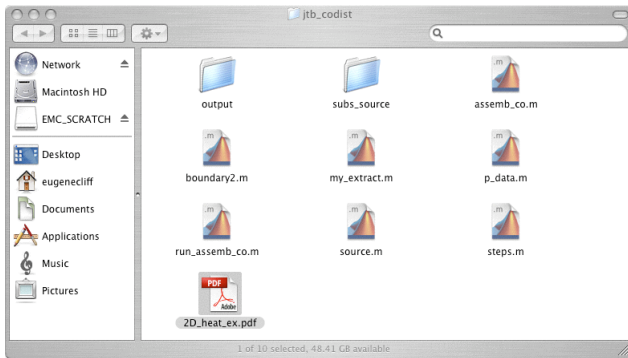
%% clean-up path
rmpath './subs_source/oned'; rmpath './subs_source/twod'

%% Steady state solutions
z_tmp = -full(M2)\full(F+b); % Temperature distribution
z_ss = gather(z_tmp, 1);

%% Plot and save a surface plot
if labindex == 1
    xx = x(1:param.nodesx, 1);
    yy = x(1:param.nodesx:param.nodesx*param.nodesy, 2);
    figure
    surf(xx, yy, reshape(z_ss, param.nodesx, param.nodesy)');
    xlabel('\bf_x'); ylabel('\bf_y'); zlabel('\bf_T')
    t_axis = axis;
    print -dpng fig_ss.png
    close all
end
```

# The RUN\_ASSEMB\_CO Example: source material

A folder with the software, example output and descriptive material is in the `parallel_matlab` folder on your desktop. The folder should look like:



# MATLAB Parallel Computing

- Introduction
- Local Parallel Computing
- Parallelism with PARFOR
- The PRIME\_NUMBER Example
- Behind the Magic
- Bumps in the Road
- The MD Example
- Parallelism with SPMD
- Parallel Computing with `fmincon`
- Codistributed Arrays
- A 2D Heat Equation
- **Parallel MATLAB at Virginia Tech**
- Conclusion



VirginiaTech



# VT Parallel MATLAB: The Past

Until recently, Advanced Research Computing at Virginia Tech had 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 New Ithaca System

In June 2009, ARC received a new IBM iDataPlex cluster known as **Ithaca**.

Ithaca will be opened to the general computing community in October 2009. In the meantime, a limited number of “friendly users” will be able to work on the machine, trying it out, porting codes, and reporting problems.

Ithaca is an *expandable* system; as it takes over the load from System X and other research clusters, more racks will be added.

# VT Resources: The New Ithaca System



The current system has

- 84 **nodes**, each with 24 GB of memory (some have 48 GB)
- 168 **processors**, each node has two Intel Nehalem processors
- 672 **cores**, each processor has 4 cores

# VT Resources: The File System

Ithaca shares the **sysx** file system already used by System X and the other ARC computing clusters. Any files you stored or created on System X can be seen and used by Ithaca.

To copy files between Ithaca and your local system, use the **sftp** program:

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

```
cd ithaca/my_project
```

```
put my_m_file.m
```

```
get my_output.txt
```

```
quit
```

# VT Resources: Logging in to Ithaca

For interactive access to Ithaca, log in using the **ssh** command.

```
ssh my_name@ithaca.arc.vt.edu
```

Especially if you are going to run MATLAB interactively, you will want to use the **-X** switch, to enable X-window graphics!

```
ssh -X my_name@ithaca.arc.vt.edu
```

If you don't do this, and start MATLAB, you'll get the warning:

```
Warning: No display specified.
```

```
You will not be able to display graphics on the screen.
```

# VT Resources: Running MATLAB Interactively

MATLAB can be run interactively on Ithaca.

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 Interactively

If you've enabled X Window graphics, MATLAB will open a separate command window. In that case, it's really convenient to be able to move to the Ithaca command window and issue commands there as well.

But Ithaca won't let you do this unless you use the `&` ("ampersand") switch at the end of the command that starts MATLAB:

```
matlab &
```

This means start up MATLAB, but also give me UNIX commandline access immediately. The MATLAB command window will be available by selecting it.

# VT Resources: Running parallel MATLAB Locally

Even though Ithaca is a cluster, when you log in, you see only one node, which has 8 cores.

You can easily run parallel MATLAB locally, with up to 8 workers.

```
matlabpool open local 8
```

```
samson
```

```
matlabpool close
```

The "**local**" keyword refers to the local *configuration*. A configuration tells MATLAB how to get the workers you ask for.



# VT Resources: Configurations

When computing in parallel, it's the **cores** that do the work.

When you log into Ithaca and start MATLAB, you have local access to one node, which has 8 cores, and that means you can get at most 8 MATLAB workers.

If more workers are desired, you have to tell MATLAB to find some more nodes and coordinate them.

MATLAB only knows how to do this if, in the **matlabpool** command, you replace the **local** configuration by the **ithaca** configuration which explains where all the nodes are and how to get them.

# VT Resources: Setting Up the Ithaca Configuration

- Start MATLAB;
- Select the **Parallel** menu, then **Manage Configurations**;
- Select the **File** menu, then **Import**;
- Select the file `/apps/share/ithacaq.mat` and click **Import**;
- Your configuration list should now include **local** and **ithaca**;
- Double click on **ithaca**, which should open a dialogue box;
- In **DataLocation** insert your PID in `/home/YOUR_USERNAME`;
- In **SubmitArguments** replace `-lwalltime=HH:MM:SS`, by `-lwalltime=00:10:00`;
- Click **OK** to save the configuration;
- Click **Start Validation**;
- Once validation has completed, exit MATLAB.



## VT Resources: Validate the Ithaca Configuration

The time limit you specify for this configuration will apply to all jobs you submit. You can go back into the **Parallel** menu and change this value if you need to.

If the validation process fails, then there is something wrong with the configuration file, the other parameters you typed, file permissions, or the setup of MATLAB. Check your steps one more time, and then report the problem to ARC.

From now on, you have *two* configurations defined, **local** and **ithaca**. You choose the appropriate configuration on the **matlabpool** command.

## VT Resources: MATLAB On Multiple Ithaca Nodes

If you want to run MATLAB on more than 8 cores, you need to use more than 1 node. The **ithaca** configuration takes care of getting the necessary nodes and starting up MATLAB workers on them. You specify the configuration in your **matlabpool** command:

```
matlabpool open ithaca 16
```

```
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 command input will 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: Running MATLAB as a Queue Job

Suppose you want to run many jobs, or jobs that take a long time to start or run, and you don't want to remain logged in to Ithaca, running MATLAB and waiting.

You can submit jobs to the ARC queueing system. The system will take care of running all the jobs on the available resources, and preserving the printed output in files. You can submit your jobs to the queue and log out.

The queueing system on Ithaca uses the same set of **PBS** commands to set time limits, number of nodes and so on.

What follows is an example of a job that runs a parallel MATLAB program under the queueing system.

# VT Resources: Running MATLAB as a Queue Job

```
#!/bin/bash
#PBS -lwalltime=00:05:00
#PBS -lnodes=2:ppn=8
#PBS -W group_list=matlab
#PBS -A matlab0001
#PBS -q ithaca_q@admin01
#PBS -lpartition=ITHACA

cd $PBS_O_WORKDIR
export PATH=/nfs/software/bin:$PATH

matlab -nodisplay < example_run.m > example_output.txt
```

# VT Resources: User Accounts

You have been assigned an Ithaca account temporarily. These accounts are intended to let you carry out the class assignments, and familiarize yourself with the system. They will expire within a week.

Ithaca will be open to general users in October.

If you are interested in parallel MATLAB on Ithaca, and want to get an early start, you can participate in our **Friendly User** program, which will give you an account to develop, test and benchmark your programs.



# MATLAB Parallel Computing

- Introduction
- Local Parallel Computing
- Parallelism with PARFOR
- The PRIME\_NUMBER Example
- Behind the Magic
- Bumps in the Road
- The MD Example
- Parallelism with SPMD
- Parallel Computing with `fmincon`
- Codistributed Arrays
- A 2D Heat Equation
- Parallel MATLAB at Virginia Tech
- **Conclusion**



VirginiaTech

## 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

Users will find it convenient to work with MATLAB interactively for program development and debugging.

However, jobs that use many nodes or run for a long time are better handled by the queueing system.

The commands in your job file might move to the appropriate directory, start up MATLAB with an input command file. The output will be preserved in the queue log, or can be written to a separate text file.

## Conclusion: MathWorks Training

This class is intended as an overview to parallel MATLAB and Ithaca.

The Mathworks has official training classes on parallel MATLAB.

If we can get 20 people to register, we can have the Mathworks give a half-day presentation on parallel MATLAB.

We hope that this class has given you enough information so that you can experiment with parallel MATLAB, and come to the MathWorks training session with some hard questions!

# MATLAB Parallel Computing: Reminder



Please don't forget!

# MATLAB Parallel Computing: Reminder

**ITHACA** is an IBM iDataPlex cluster recently installed by Virginia Tech's Advanced Research Computing facility.

It is intended to gradually take over the high performance computing load from System X.

**ITHACA** supports OpenMP, MPI and Parallel MATLAB programming.

- **Right Now:** Friendly users accepted for Ithaca (*talk to John Burkardt if you are interested,*);
- **Mid October:** Ithaca opened to general users (*accounts given out through online application.*).

# MATLAB Parallel Computing: Reminder

**MATLAB Training** is available this fall.

These classes will be presented by the MathWorks. Some of these classes are tentative. Check the FDI website for details.

- **2 October:** Simulink (daylong);
- **3 October:** SimMechanics (daylong);
- **8 October:** MATLAB Programming Techniques.
- **8 October:** Parallel Computing with MATLAB.
- **29 October:** Parallel Computing with MATLAB (daylong).
- **19 November:** Real-time Data Acquisition and Control.
- **19 November:** Statistical Methods in MATLAB.



VirginiaTech