First Access Your Account

- Log into your accounts
  - Username or login = hpc_user**X**
  - Where X = serial number 1 – 47 from the sign-in tokens
  - Password = **cacds2014**

Accessing Tutorial Materials

First login into the cluster:

TYPE AND EXECUTE COMMANDS IN Green!!!

```
cd
cp /share/apps/tutorials/spring2015/matlab/intro2matlab_part2.pdf  ~
cp /share/apps/tutorials/spring2015/matlab/intro2matlab_part2.zip  ~
unzip intro2matlab_part2.zip
cd intro2matlab_part2
module add matlab
matlab
```

UNIVERSITY of HOUSTON

CENTER FOR ADVANCED COMPUTING & DATA SYSTEMS

Introduction to MATLAB Part II

Amit Amritkar

Center for Advanced Computation and Data Systems (CACDS)

http://cacds.uh.edu

http://support.cacds.uh.edu

University of Houston

Houston, TX

About CACDS

Mission Statement

- **CACDS provides High Performance Computing resources and services to advance Tier One research and education goals at the University of Houston**

Overview

- Review
- Introduction to parallel MATLAB
  - Parallel computing
    - Execution models
    - parfor
    - spmd
- MATLAB compiler and coder
- Conclusion

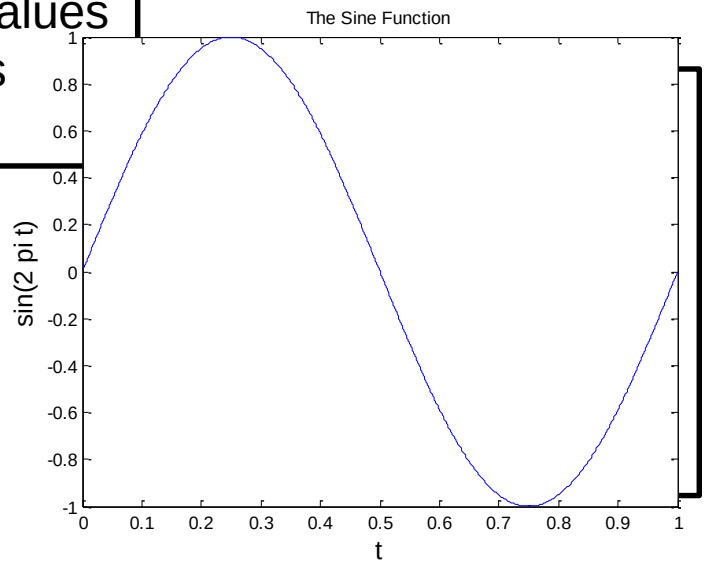
Review

- Introduction to MATLAB part I
  - Variables, arrays, matrices, etc
    - Arrays are the fundamental units of data
  - Operators
  - Flow control
    - if, while, for
  - Using M files
    - Scripts and functions
  - Plots

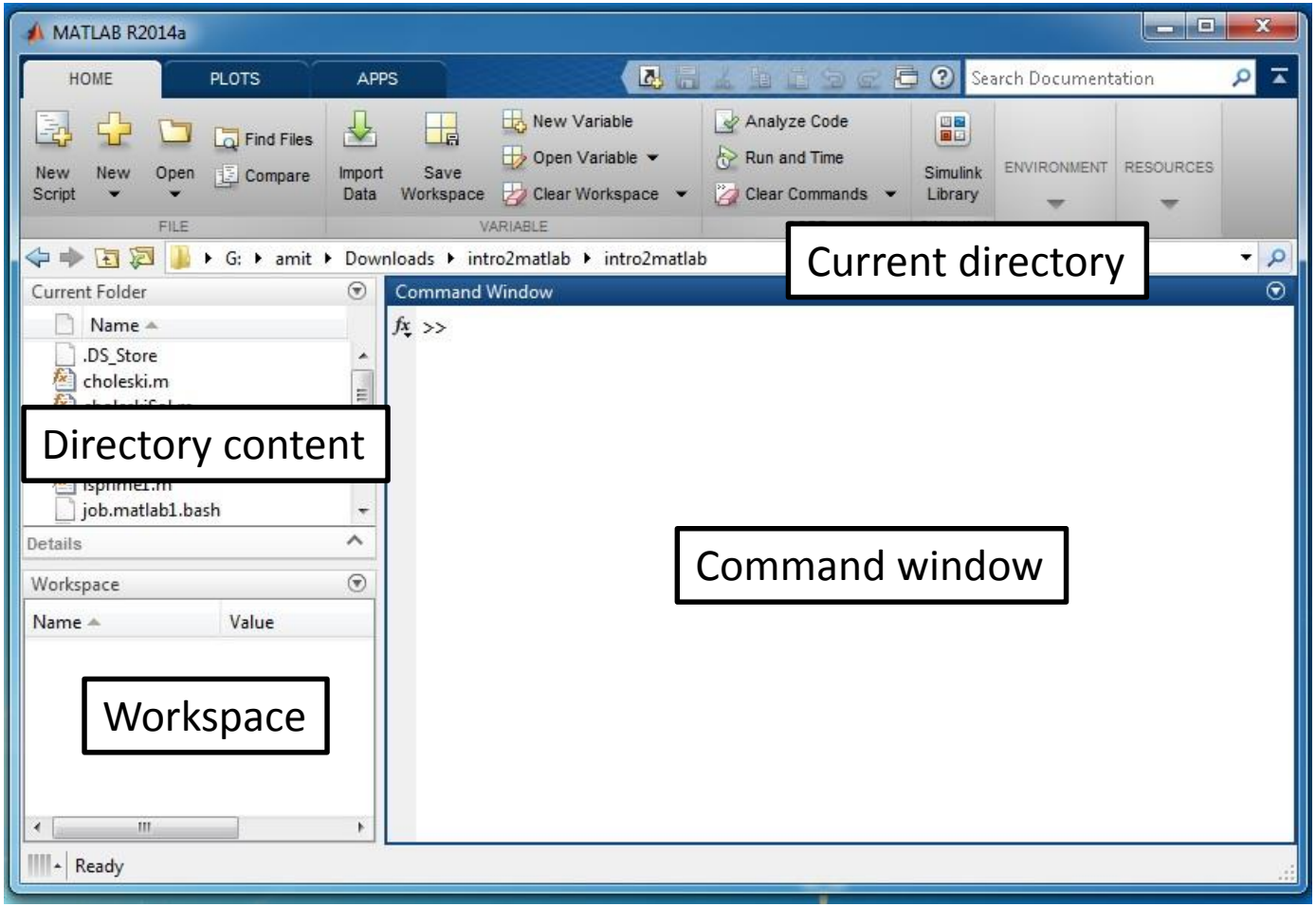
```
>>sum(M)
```

```
>>a= [1 2 3; 4 5 6; 7 8 9; 10 11 12];
     Column major
```

```
for index = values
   statements
end
```

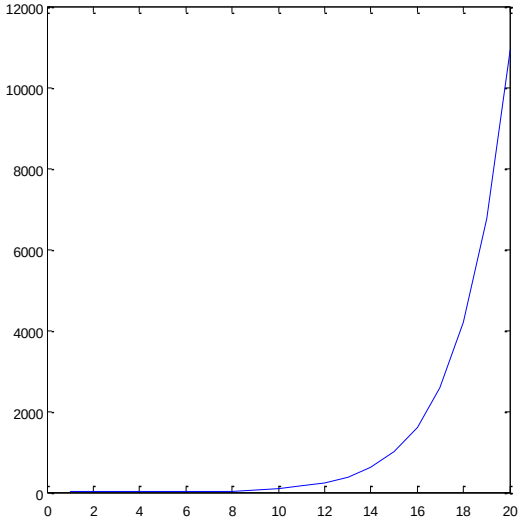


Review – MATLAB GUI



Review exercise

- Write a program to plot the Fibonacci series – first 20 numbers in the series
- Hint: use the fibonacci.m function file
- >>plot(fibonacci(20));



INTRO: Parallel MATLAB

- Parallel MATLAB is an extension of MATLAB that takes advantage of multicore desktop machines and clusters.
- The Parallel Computing Toolbox or PCT runs on a desktop, and can take advantage of cores (R2014a has no limit, R2013b limit is 12, ...). Parallel programs can be run interactively or in batch.
- The MATLAB Distributed Computing Server (MDCS) controls parallel execution of MATLAB on a cluster with tens or hundreds of cores.

INTRO: What Do You Need?

- Your machine should have multiple processors or cores:
  - On your Linux desktop :: terminal : type command - lscpu
  - On a PC: Start :: Settings :: Control Panel :: System
  - On a Mac: Apple Menu :: About this Mac :: More Info...
- Your MATLAB must be version 2012a or later:
  - Go to the HELP menu, and choose About MATLAB.
- You must have the Parallel Computing Toolbox (PCT):
  - At UH, the concurrent (& student) license includes the PCT.
- The standalone license does not include the PCT.
- To list all your toolboxes, type the MATLAB command ver.
- When using an MDCS (server) be sure to use the same version of MATLAB on your client machine.

PROGRAMMING: Obtaining Parallelism

- Three ways to write a CPU parallel MATLAB program:
  - suitable for loops can be made into parfor loops;
  - the spmd statement can define cooperating synchronized processing;
  - the task feature creates multiple independent programs.
- GPU Computing

- The parfor approach is a limited but simple way to get started.
- spmd is powerful, but may require rethinking the program/data.
- The task approach is simple, but suitable only for computations that need almost no communication.

PARFOR: Parallel FOR Loops

- The simplest path to parallelism is the parfor statement, which indicates that a given for loop can be executed in parallel.
- When the “client” MATLAB reaches such a loop, the iterations of the loop are automatically divided up among the workers, and the results gathered back onto the client.
- Using parfor requires that the iterations are completely independent; there are also some restrictions on array-data access.
- parfor Syntax:

  parfor loopvar = initval:endval; statements; end

  parfor (loopvar = initval:endval, M); statements; end
- OpenMP implements a directive for ’parallel for loops’

‘SPMD’ Single Program Multiple Data

- MATLAB can also work in a simplified kind of MPI model.
- There is always a special “client” process.
- Each worker process has its own memory and separate ID.
- There is a single program, but it is divided into client and worker sections; the latter marked by special spmd/end statements.
- Workers can “see” the client’s data; the client can access and change worker data.
- The workers can also send messages to other workers.
- Spmd syntax:

  spmd, statements, end

  spmd(n,m), statements, end
- OpenMP includes constructs similar to spmd.

‘SPMD’ Distributed Arrays

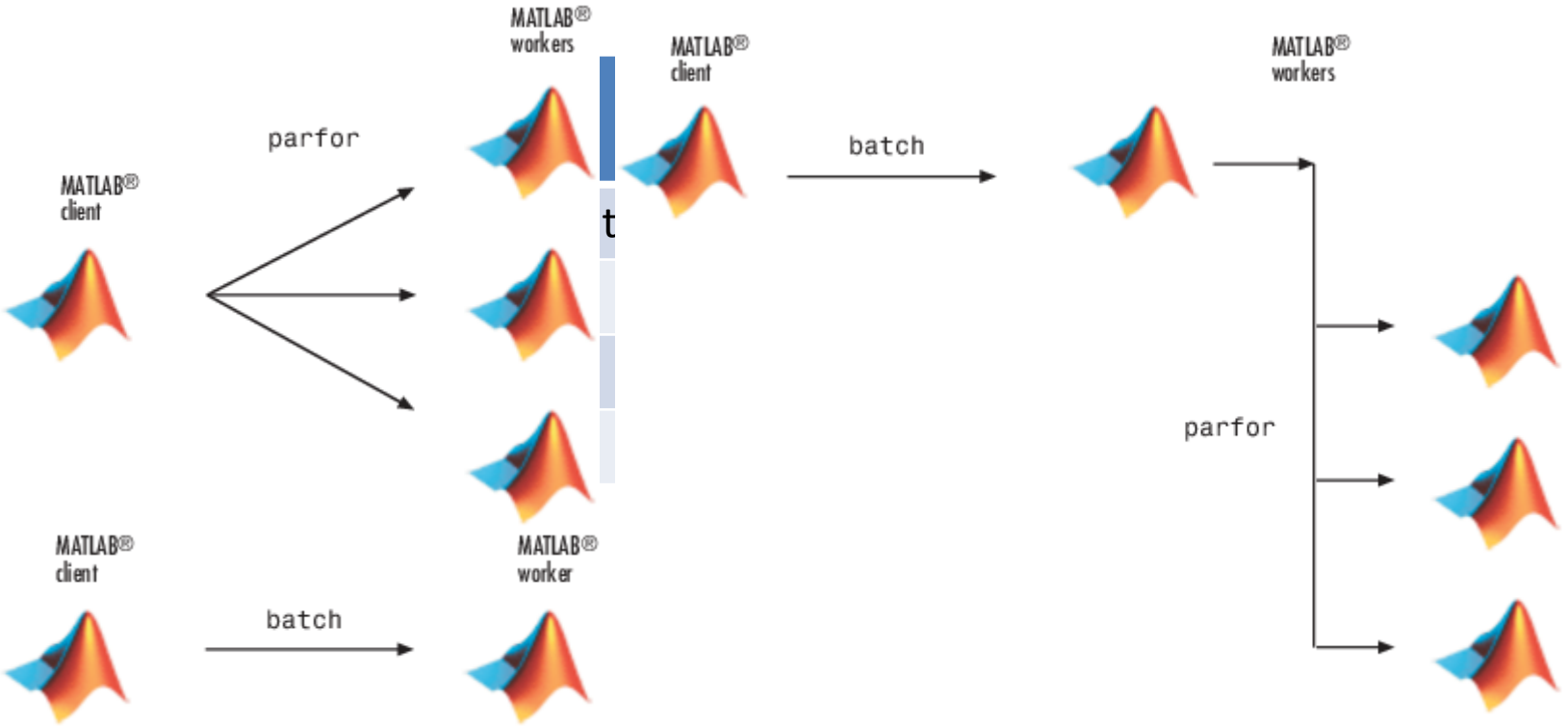
- SPMD programming includes distributed arrays.
- A distributed array is logically one array, and a large set of MATLAB commands can treat it that way (e.g. ‘backslash’).
- However, portions of the array are scattered across multiple processors. This means such an array can be really large.
- The local part of a distributed array can be operated on by that processor very quickly.
- A distributed array can be operated on by explicit commands to the SPMD workers that “own” pieces of the array, or implicitly by commands at the global or client level.

GPU Computing

- Identify and Select a GPU Device (compute capability 2.0 and above)
  - gpuDevice and gpuDeviceCount
- Transfer or Create arrays on GPU
  - Transfer Arrays Between Workspace and GPU
    - X = rand(100); G = gpuArray(X);
  - Create GPU Arrays Directly
    - G = rand(100,'gpuArray');
- Run built-in functions on GPU
  - discrete Fourier transform (fft), matrix multiplication (mtimes), left matrix division (mldivide)
  - If any of the input arguments is a gpuArray, the function executes on the GPU and returns a gpuArray
- Run Element-wise MATLAB Code on GPU
  - You can run your own MATLAB function of element-wise operations on a GPU with just the addition of defining arrays on GPU
- Limited availability on UH cluster

EXECUTION: Models

- There are several ways to execute a parallel MATLAB program:



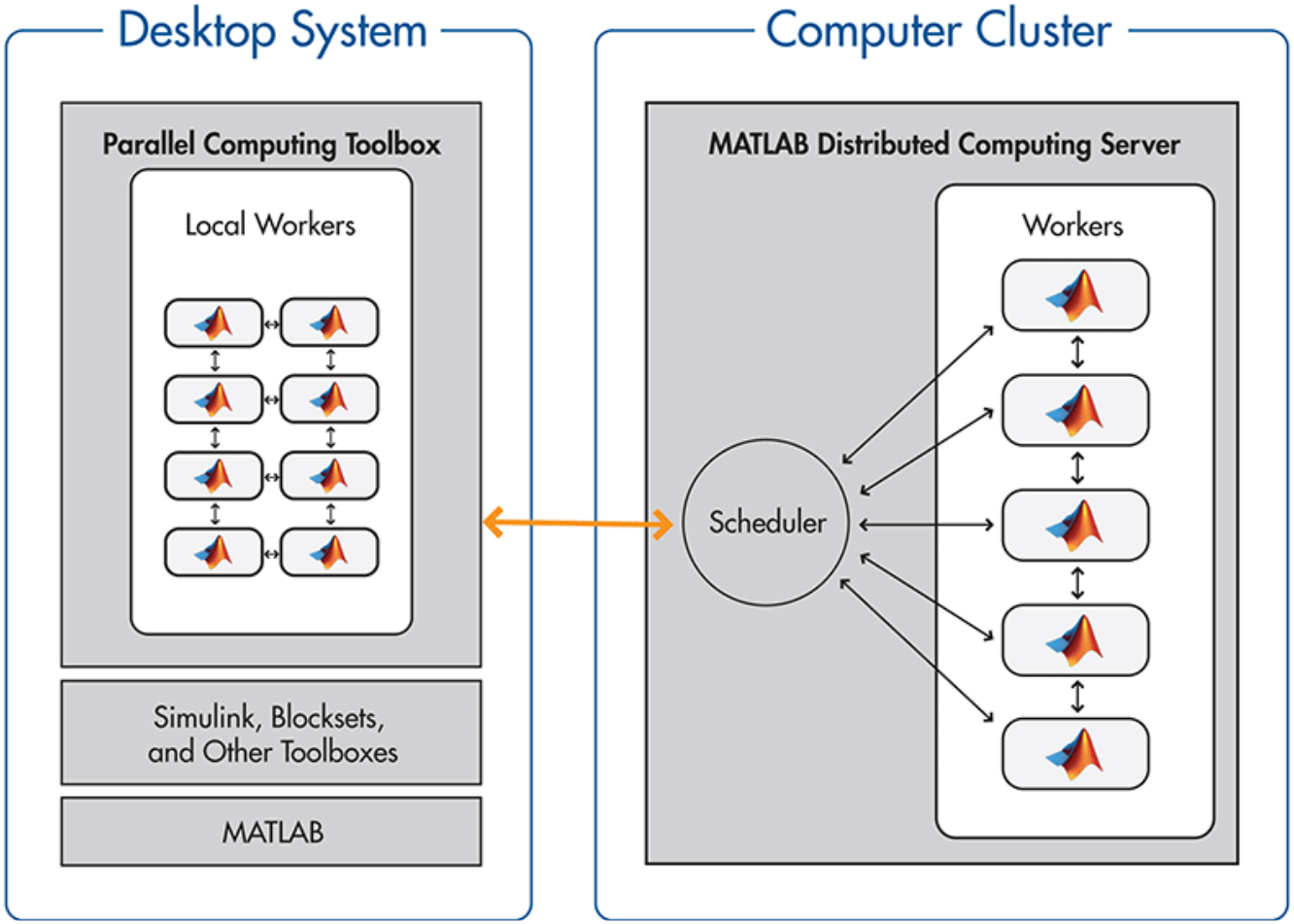
EXECUTION: Direct using matlabpool / parpool

- Parallel MATLAB jobs can be run directly, that is, interactively.
- The matlabpool command is used to reserve a given number of workers on the local (or perhaps remote) machine.
- matlabpool was replaced by parpool in R2013b, and in R2014a issues a warning.
- Once these workers are available, the user can type commands, run scripts, or evaluate functions, which contain parfor statements.
- The workers will cooperate in producing results.
- Interactive parallel execution is great for desktop debugging of short jobs.
- Note: Starting in R2013b, if you try to execute a parallel program and a pool of workers is not already open, MATLAB will open it for you. The pool of workers will then remain open for a time that can be specified under Parallel → Parallel Preferences (default = 30 minutes).

EXECUTION: Indirect Local using batch

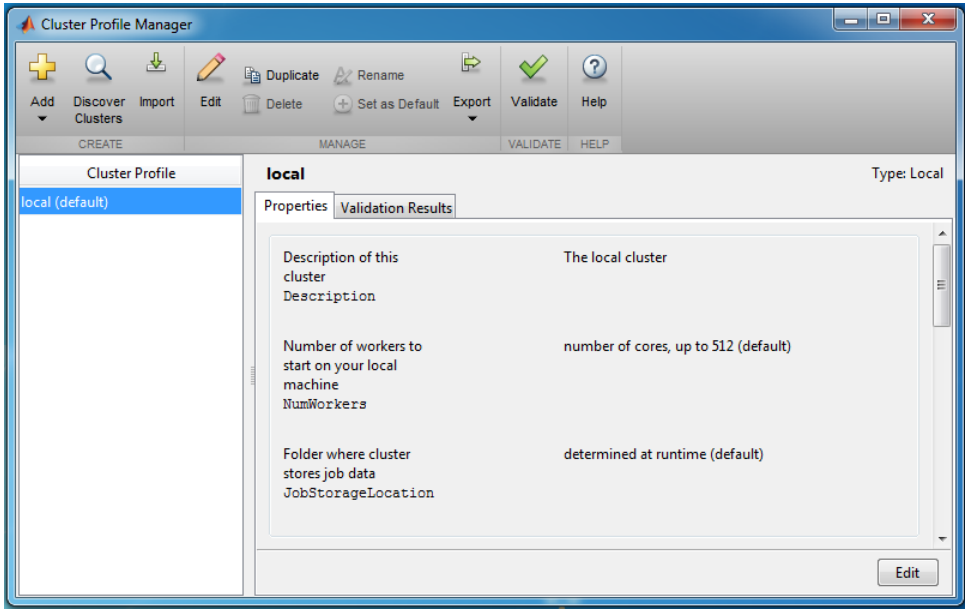
- Parallel MATLAB jobs can be run indirectly.
- The batch command is used to specify a MATLAB code to be executed, to indicate any files that will be needed, and how many workers are requested.
- The batch command starts the computation in the background. The user can work on other things, and collect the results when the job is completed.
- The batch command works on the desktop, and *can be set up* to access the maxwell cluster.

EXECUTION: Local and Remote MATLAB Workers



EXECUTION: Managing Cluster Profiles

- MATLAB uses Cluster Profiles (previously called \configurations") to set the location of a job. `local' is the default. Others can be added to send jobs to other clusters (e.g. maxwell).



EXECUTION: Ways to Run

- Interactively, we call parpool (or matlabpool) and then our function:

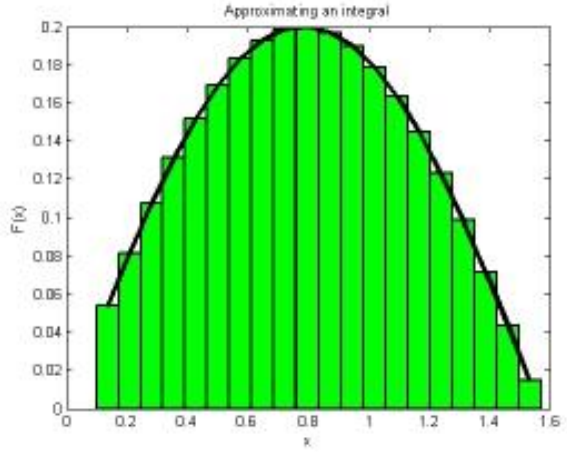
```
mypool = parpool ( 'local', 4 )
                (or)
matlabpool ( 'open', 'local', 4 ) %deprecated
q = quad_fun ( n, a, b );
delete('mypool')
                (or)
matlabpool('close') %deprecated
```

- 'local' is a default Cluster Profile defined as part of the PCT. The batch command runs a script, with a matlabpool argument:

```
job = batch ( 'quad_script', 'matlabpool', 4 )
                                (or)
job = batch ( 'Profile','local', 'quad_script', ...
                                    'matlabpool', 4 )
```

QUAD : Estimating integral

```
function q = quad_fun (n,a,b)
    q=0.0;
    w=(b-a)/n;
    for i =1:n
        x = ((n-i )*a+(i-1)*b)/(n-1);
        fx= 4./(1+x .^2);
        q = q+w*fx ;
    end
    return
end
```



QUAD : comments

- The function quad_fun estimates the integral of a particular function over the interval [a,b].
- It does this by evaluating the function at n evenly spaced points, multiplying each value by the weight (b − a)/n.
- These quantities can be regarded as the areas of little rectangles that lie under the curve, and their sum is an estimate for the total area under the curve from a to b.
- We could compute these subareas in any order we want.
- We could even compute the subareas at the same time, assuming there is some method to save the partial results and add them together in an organized way.

QUAD: The Parallel QUAD Function

```
function q = quad_fun (n,a,b)
    q=0.0;
    w=(b-a)/n;
    parfor i =1:n
        x = ((n-i )*a+(i-1)*b)/(n-1);
        fx= 4./(1+x .^2);
        q = q+w*fx ;
    end
    return
end
```

Parallel QUAD : comments

- The parallel version of quad_fun does the same calculations.
- The parfor statement changes how this program does the calculations. It asserts that all the iterations of the loop are independent, and can be done in any order, or in parallel.
- Execution begins with a single processor, the client. When a parfor loop is encountered, the client is helped by a “pool” of workers.
- Each worker is assigned some iterations of the loop. Once the loop is completed, the client resumes control of the execution.
- MATLAB ensures that the results are the same (with exceptions) whether the program is executed sequentially, or with the help of workers.
- The user can wait until execution time to specify how many workers are actually available.

QUAD: Interactive PARPOOL

- To run quad_fun.m in parallel on your desktop, type:

```
n = 1000000; a = 0.5; b = 1;
parpool ('local',4);
q = quad_fun ( n, a, b );
delete(gcp)
```

- The word local is choosing the local profile, that is, the cores assigned to be workers will be on the local machine.

- The value ”4” is the number of workers you are asking for. It can be up to 8 on the current local machine. It does not have to match the number of cores you have.

QUAD: Indirect Local BATCH

- The batch command, for indirect execution, accepts scripts (and since R2010b, functions). We can make a suitable script called quad_script.m:

```
n = 1000000; a = 0.5; b = 1;
q = quad_fun ( n, a, b )
```

- Now we assemble the job information needed to run the script and submit the job:

```
job = batch ( 'quad_script', 'matlabpool', 4, ...
    'Profile', 'local', ...
    'AttachedFiles', { 'quad_fun' } )
```

QUAD: Indirect Local BATCH

- After issuing batch(), the following commands wait for the job to finish, gather the results, and clear out the job information:

```
wait ( job );   % no prompt until the job is finished
load ( job );   % load data from the job's Workspace
delete ( job ); % clean up (destroy prior to R2012a)
```

QUAD: Indirect Remote BATCH

- The batch command can send your job anywhere, and get the results back, as long as you have set up an account on the remote machine, and you have defined a Cluster Profile on your desktop that tells it how to access the remote machine.
- The job is submitted. You may wait for it, load it and destroy/delete it, all in the same way as for a local batch job.

PBS Script for Executing MATLAB script Using Proprietary MATLAB runtime (sample I)

```
#!/bin/bash
#PBS -N matlabjob
#PBS -l nodes=1:ppn=1,pmem=1gb
#PBS -S /bin/bash
#PBS -l walltime=00:05:00
cd
$PBS_O_WORKDIR
##set up your environment
module add matlab
matlab <<EOF
a=10; b=20; c=30;
d=sqrt((a+b+c)/pi)
exit
EOF
```

See job filename: job.matlab1.bash

PBS Script for Executing MATLAB script Using Proprietary MATLAB runtime (sample II)

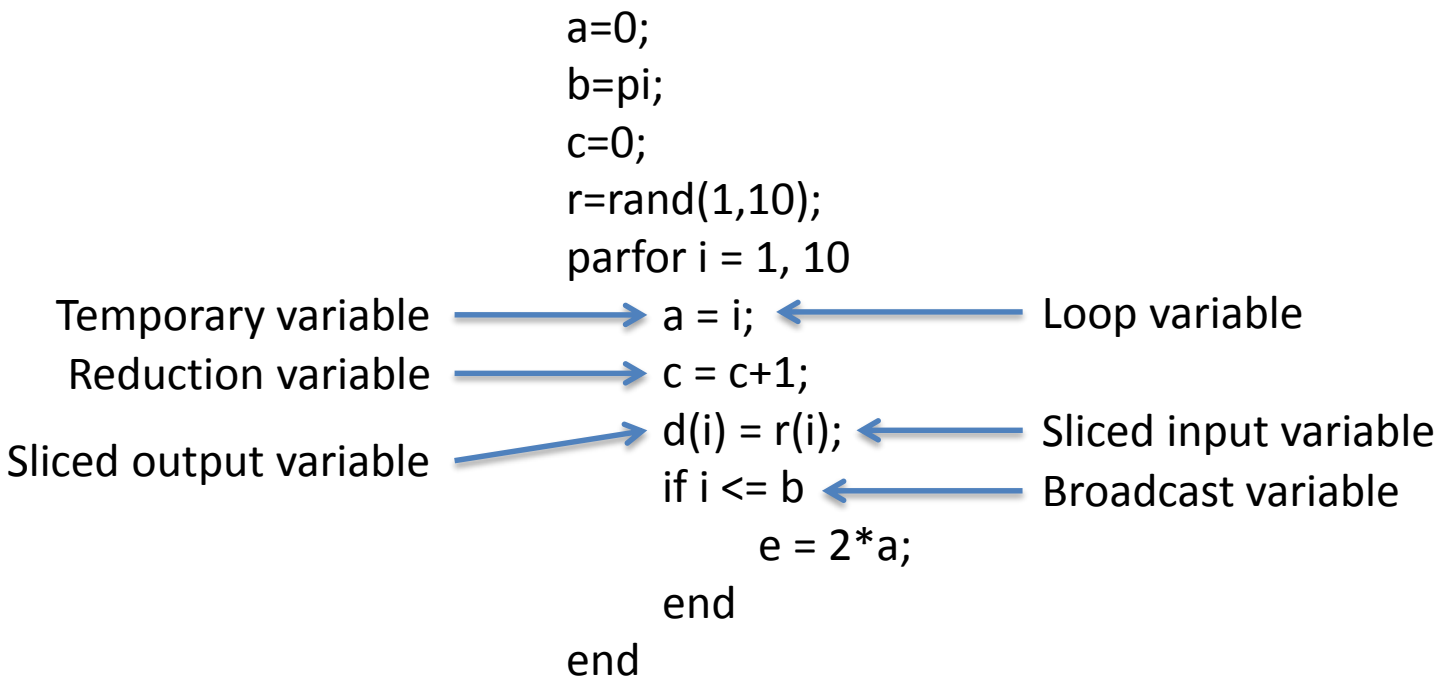
```
#!/bin/bash
#PBS -N matlabjob
#PBS -l nodes=1:ppn=1,pmem=1gb
#PBS -S /bin/bash
#PBS -l walltime=00:05:00
cd
$PBS_O_WORKDIR
##set up your environment
module add matlab
## run matlab program compute pseudo inverse for 100 matrices of size 400x400
time matlab <matrix_inversion100_matlab.m
```

See job filename: job.matlab1.bash

CLASSIFICATION: variable types in PARFOR loops

Classification	Description
Loop	Serves as a loop index for arrays
Sliced	An array whose segments are operated on by different iterations of the loop
Broadcast	A variable defined before the loop whose value is used inside the loop, but never assigned inside the loop
Reduction	Accumulates a value across iterations of the loop, regardless of iteration order
Temporary	Variable created inside the loop, but unlike sliced or reduction variables, not available outside the loop

CLASSIFICATION: an example

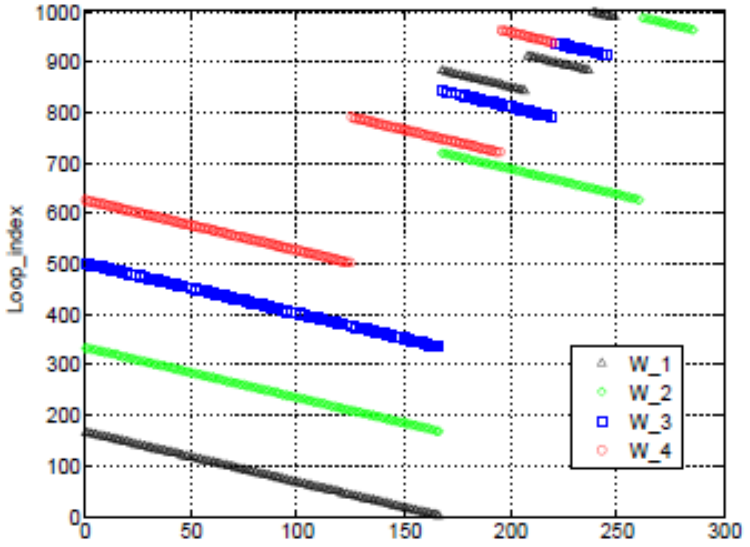


- The range for parfor statement should be increasing consecutive integers

Quiz: what would be values of a, i and e when the loop ends

parfor : comments

- How are the loop indices distributed among the workers?
- Run ii=1:1000 on 4 workers.
  - 63% indices allocated in the first 4 chunks.
- Indices then assigned in smaller chunks as a worker finishes
- Similar to OpenMP's Dynamic assignment



PCT vs Multithread:

- At least since R2007a basic MATLAB has incorporated multithreading
- Many linear algebra functions (e.g. LU, QR, SVD) use multithreads
- In the PCT each worker/lab runs a single thread
- For some codes using parfor will *increase* the execution time

Exercise (1)

Parallel For Loop or parfor

Perform three large eigenvalue computations (hint: c(:,i)=eig(rand(1000))) using three workers or cores with Parallel Computing Toolbox software. Time your computation (hint: tic/toc).

```
tic
parpool(3)
parfor i=1:3;
        c(:,i)=eig(rand(1000));
end;
size(c)
toc
delete(gcp)
```

See job filename: job.matlab4.bash

parfor: number of prime numbers

- For our next 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 multiplies the run time roughly by 4.

parfor: MATLAB code

```
function total = prime_fun ( n )
%% PRIME 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 ;
                  end
            end
            total = total + prime ;
      end
      return
end
```

Running in parallel

- We can parallelize the loop whose index is i, replacing for by parfor. The computations for different values of i are 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 summation in parallel.
- To make the program parallel, we replace for by parfor. That's all!
- Beware - reduction assignments require care

Execution with parpool

```
poolobj = parpool ( 'local' , 4 ) ;

tic
n=10000;
while ( n <= 1000000 )
      primes = prime_fun ( n ) ;
      fprintf ( 1 , ' %8d %8dnn ' , n , primes ) ;
      n = n * 10 ;
end
toc

delete (poolobj) ;
```

Exercise (2)

- Execute prime_fun with 4 workers and vary the maximum value of n as below
- Fill out the timings in the following table
- Use MATLAB profiler (run and time)

n	1+0	1+4
10,000	0.043	0.140
100,000	0.985	0.503
1,000,000	21.89	3.413

SPMD

- There is one client process, supervising workers who cooperate on a single program. Each worker (sometimes also called a "lab") has an identifier, knows how many total workers there are, and can determine its behavior based on that identifier.
- each worker runs on a separate core (ideally);
- each worker uses a separate workspace;
- a common program is used;
- workers meet at synchronization points;
- the client program can examine or modify data on any worker;
- any two workers can communicate directly via messages.

SPMD – Getting workers

- An spmd program needs workers to cooperate on the program. So on a desktop, we issue an interactive parpool request:

```
pool_obj = parpool('local', 4 );
results = myfunc ( args );
```

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

```
job = batch ( 'myscript', 'Profile', 'local', ...
                          'pool', 4 )
```

- or send the batch command to the Remote cluster:

```
job = batch ( 'myscript', 'Profile', ...
                          ‘remote_r2015a', 'pool', 7 )
```

The SPMD Environment

- MATLAB sets up one special agent called the client.
- MATLAB sets up the requested number of workers, each with a copy of the program. Each worker " knows" it's a worker, and has access to two special functions:

  numlabs(), the number of workers;

  labindex(), a unique identifier between 1 and numlabs()
- The empty parentheses are usually dropped, but remember, these are functions, not variables!
- If the client calls these functions, they both return the value 1! That's because when the client is running, the workers are not. The client could determine the number of workers available by

  n = parpool ('size')

The SPMD Command

- The client and the workers share a single program in which some commands are delimited within blocks opening with spmd and closing with end.
- The client executes commands up to the first spmd block, when it pauses. The workers execute the code in the block. Once they finish, the client resumes execution.
- The client and each worker have separate workspaces, but it is possible for them to communicate and trade information.
- The value of variables defined in the " client program" can be referenced by the workers, but not changed.
- Variables defined by the workers can be referenced or changed by the client, but a special syntax is used to do this.

How SPMD Workspaces Are Handled

Code	Client			Worker 1			Worker 2		
	a	b	e	c	d	f	c	d	f
a=3;	3								
b=4;	3	4							
spmd c=labindex();	3	4		1			2		
d=c+a; end	3	4		1	4		2	5	
e=a+d{1};	3	4	7	1	4		2	5	
c{2}=5;	3	4	7	1	4		5	5	
spmd f=c*b end	3	4	7	1	4	4	5	5	20

When is Workspace Preserved?

A program can contain several spmd blocks. When execution of one block is completed, the workers pause, but they do not disappear and their workspace remains intact. A variable set in one spmd block will still have that value if another spmd block is encountered. Unless the client has changed it, as in our example. You can imagine the client and workers simply alternate execution. In MATLAB, variables defined in a function " disappear" once the function is exited. The same thing is true, in the same way, for a MATLAB program that calls a function containing spmd blocks. While inside the function, worker data is preserved from one block to another, but when the function is completed, the worker data defined there disappears, just as the regular MATLAB data does. It's not legal to nest an spmd block within another spmd block or within a parfor loop. Some additional limitations are discussed at http://www.mathworks.com/help/distcomp/programming-tips_brukbnp-9.html?searchHighlight=nested+spmd

SPMD – magic squares example

```
parpool(3)
spmd
  % build magic squares in parallel
  q = magic(labindex + 2);
end
for ii=1:length(q)
  % plot each magic square
  figure, imagesc(q{ii});
end
delete(gcp)
```

Distributed arrays

If the client process has a 300x400 array called a, and there are 4 SPMD workers, then the simple command

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

distributes the elements of **a** by columns. Each worker can make a local variable containing its part:

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

- Notice that local and global column indices will differ!
- By default, the last dimension is used for distribution.

The Client Can Collect Results

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

  worker3_array = al{3};

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

  a2 = [al{:}]

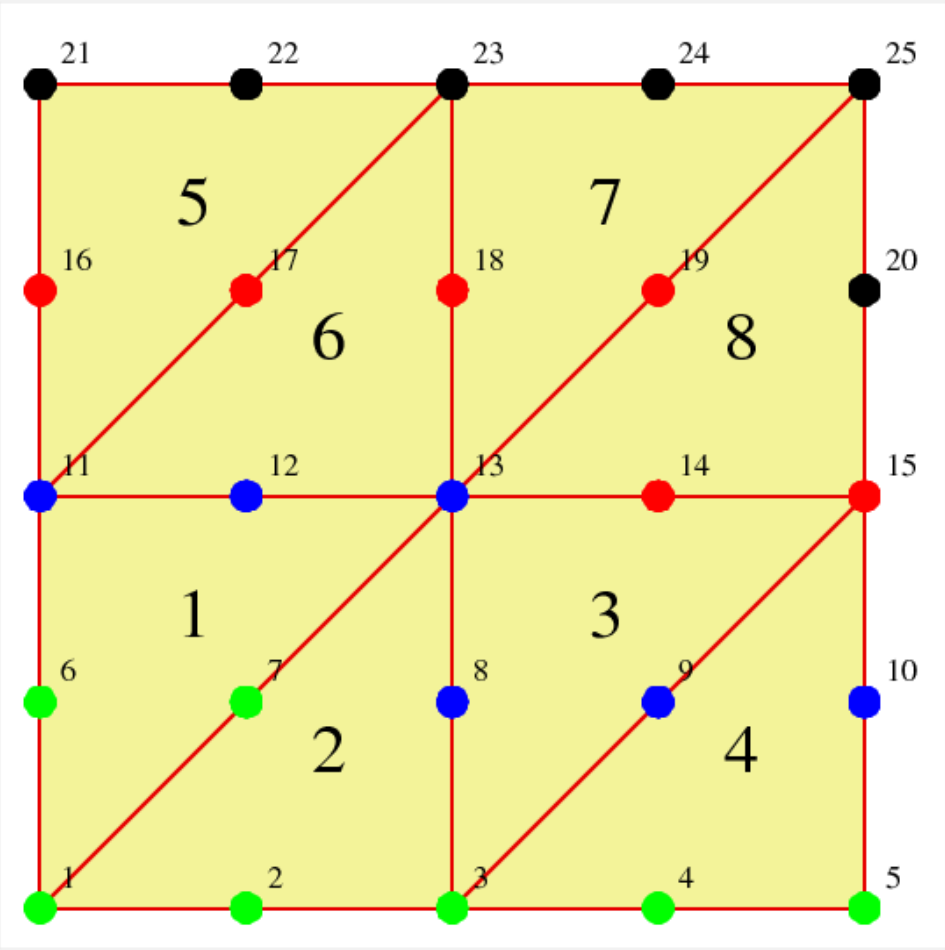
Distributed arrays - comment

- We can remove the spmd block and simply invoke distributed(); the operational commands don't change.
- The communication overhead can be severely increased
- Not all MATLAB operators have been extended to work with distributed memory. In particular, (the last time we asked), the backslash or "linear solve" operator x=A\b can't be used yet for sparse distributed arrays.
- Getting "real" data (as opposed to matrices full of random numbers) properly distributed across multiple processors involves more choices and more thought than is suggested by the examples!

2D Finite Element Heat Model

- Next, we consider an example that combines SPMD and distributed data to solve a steady state heat equations in 2D, using the finite element method. Here we demonstrate a different strategy for assembling the required arrays.
- Each worker is assigned a subset of the finite element nodes. That worker is then responsible for constructing the columns of the (sparse) finite element matrix associated with those nodes.
- Although the matrix is assembled in a distributed fashion, it has to be gathered back into a standard array before the linear system can be solved, because sparse linear systems can't be solved as a distributed array (yet).
- This example is available as in the fem 2D folder.

The Grid & Node Coloring for 4 labs



Finite Element System matrix

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

$$K z = F + b$$

where K is the stiffness matrix, z is the unknown finite element coefficients, F contains source terms and b accounts for boundary conditions.

In the parallel implementation, the system matrix K and the vectors F and b are distributed arrays. The default distribution of K by columns essentially associates each SPMD worker with a group of finite element nodes.

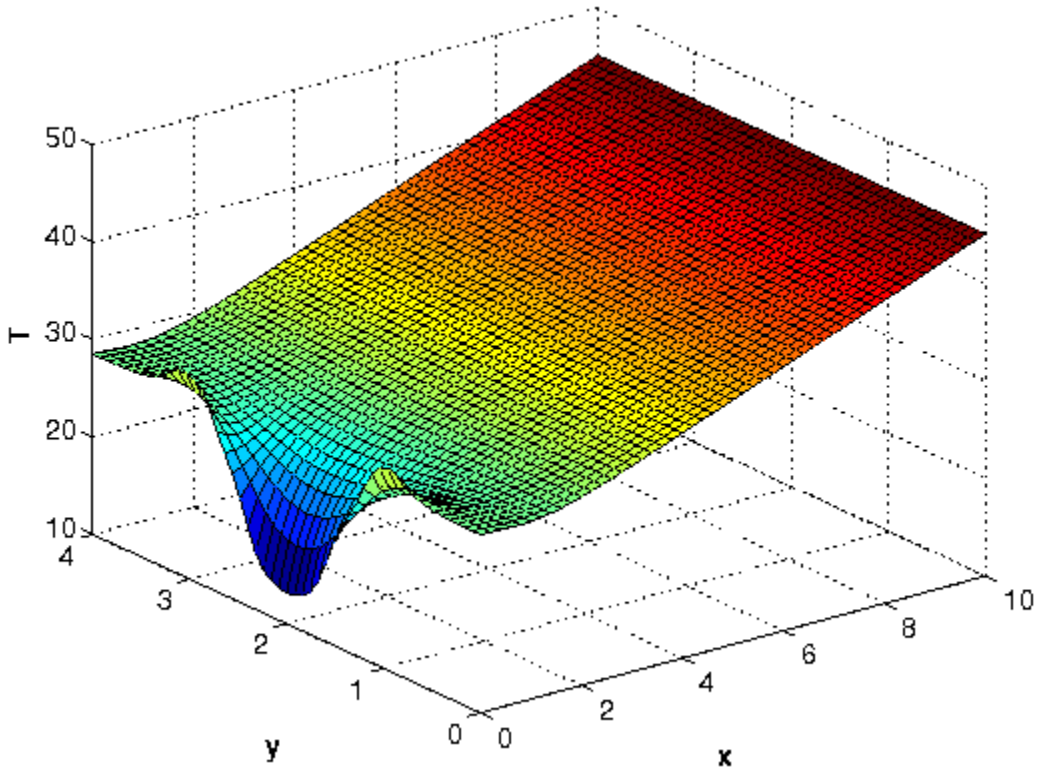
Finite Element System matrix

- To assemble the matrix, each worker loops over all elements. If element E contains any node associated with the worker, the worker computes the entire local stiffness matrix K. Columns of K associated with worker nodes are added to the local part of K. The rest are discarded (which is OK, because they will also be computed and saved by the worker responsible for those nodes).
- When element 5 is handled, the " blue", " red" and " black“ processors each compute K. But blue only updates column 11 of K, red columns 16 and 17, and black columns 21, 22, and 23.
- At the cost of some redundant computation, we avoid a lot of communication.

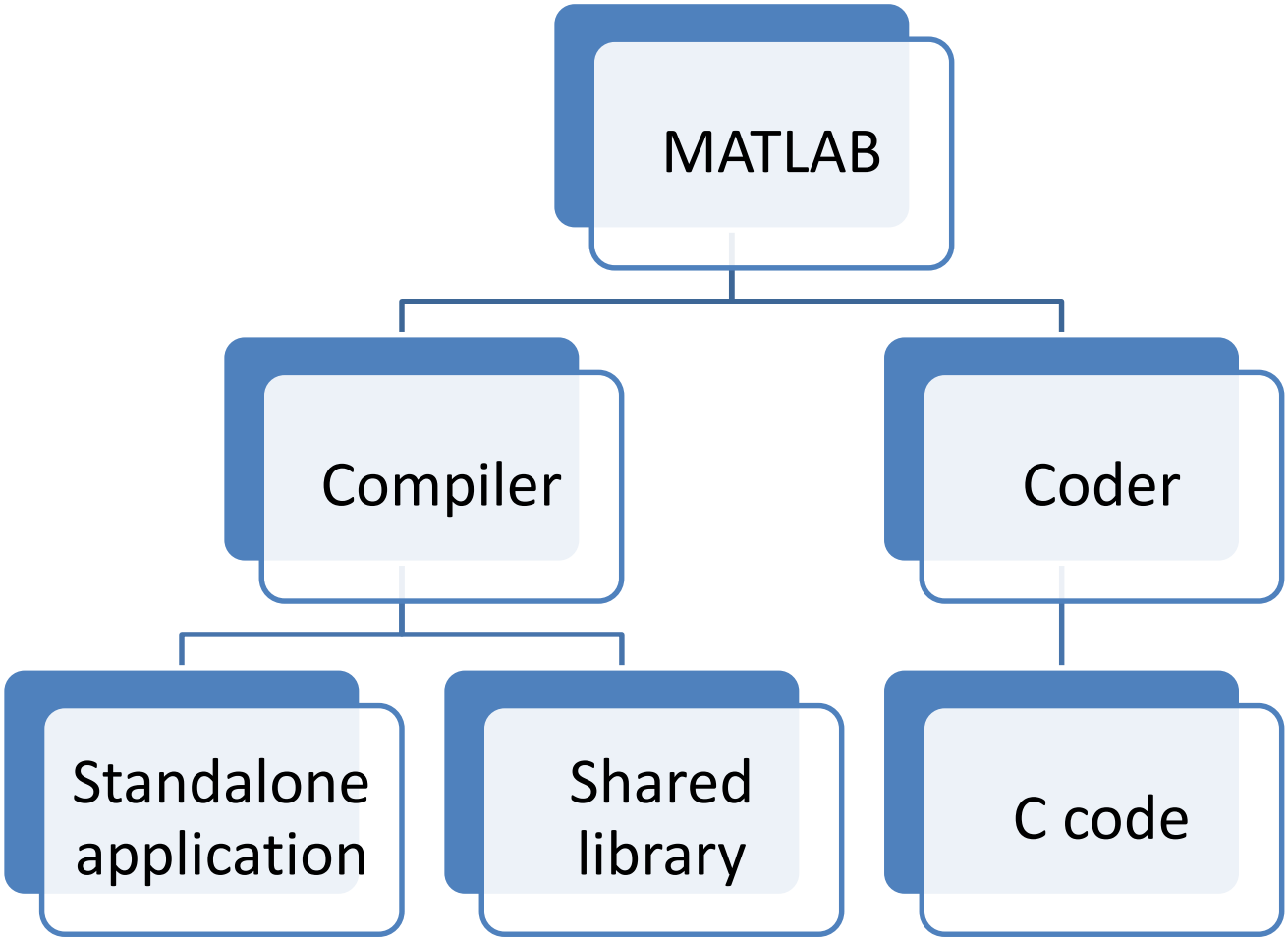
Assemble Codistributed Arrays - code fragment

```
        spmd
%
% Set up c o d i s t r i b u t e d s t r u c t u r e
%
% Column p o i n t e r s and such f o r c o d i s t r i b u t e d a r r a y s .
%
        Vc = c o d i s t r i b u t e d . c o l o n ( 1 , n e q u a t i o n s ) ;
        lP = g e t L o c a l P a r t ( Vc ) ;
        lP 1= lP ( 1 ) ; lP end = lP ( end ) ; %f i r s t and l a s t columns o f K on t h i s l a b
        c o d i s t Vc = g e t C o d i s t r i b u t o r ( Vc ) ; dPM = c o d i s t Vc . P a r t i t i o n ;
. . .
% s p a r s e a r r a y s on each l a b
%
        K lab = s p a r s e ( n e q u a t i o n s , dPM( l a b i n d e x ) ) ;
. . .
% Bu i l d the f i n i t e e l ement ma t r i c e s  Begin l o op o v e r e l eme n t s
%
        f o r n e l =1: n e l eme n t s
                n o d e s l o c a l = e conn ( n e l , : ) ;% which nodes a r e i n t h i s e l ement
        % s u b s e t o f nodes / columns on t h i s l a b
                l a b n o d e s l o c a l = e x t r a c t ( n o d e s l o c a l , lP 1 , lP end ) ;
                . . . i f empty do nothing , e l s e ac cumulat e K lab , e t c end
        end % n e l
%
% As semble the ' lab ' p a r t s i n a c o d i s t r i b u t e d format .
% s y n t a x f o r v e r s i o n R2009b
        c o d i s t m a t r i x = c o d i s t r i b u t o r 1 d ( 2 , dPM, [ n e q u a t i o n s , n e q u a t i o n s ] ) ;
        K = c o d i s t r i b u t e d . b u i l d ( K lab , c o d i s t m a t r i x ) ;
        end % spmd
```

2D Heat Equation - The Results



Code deployment



Compiling MATLAB Code

Compiling MATLAB Code

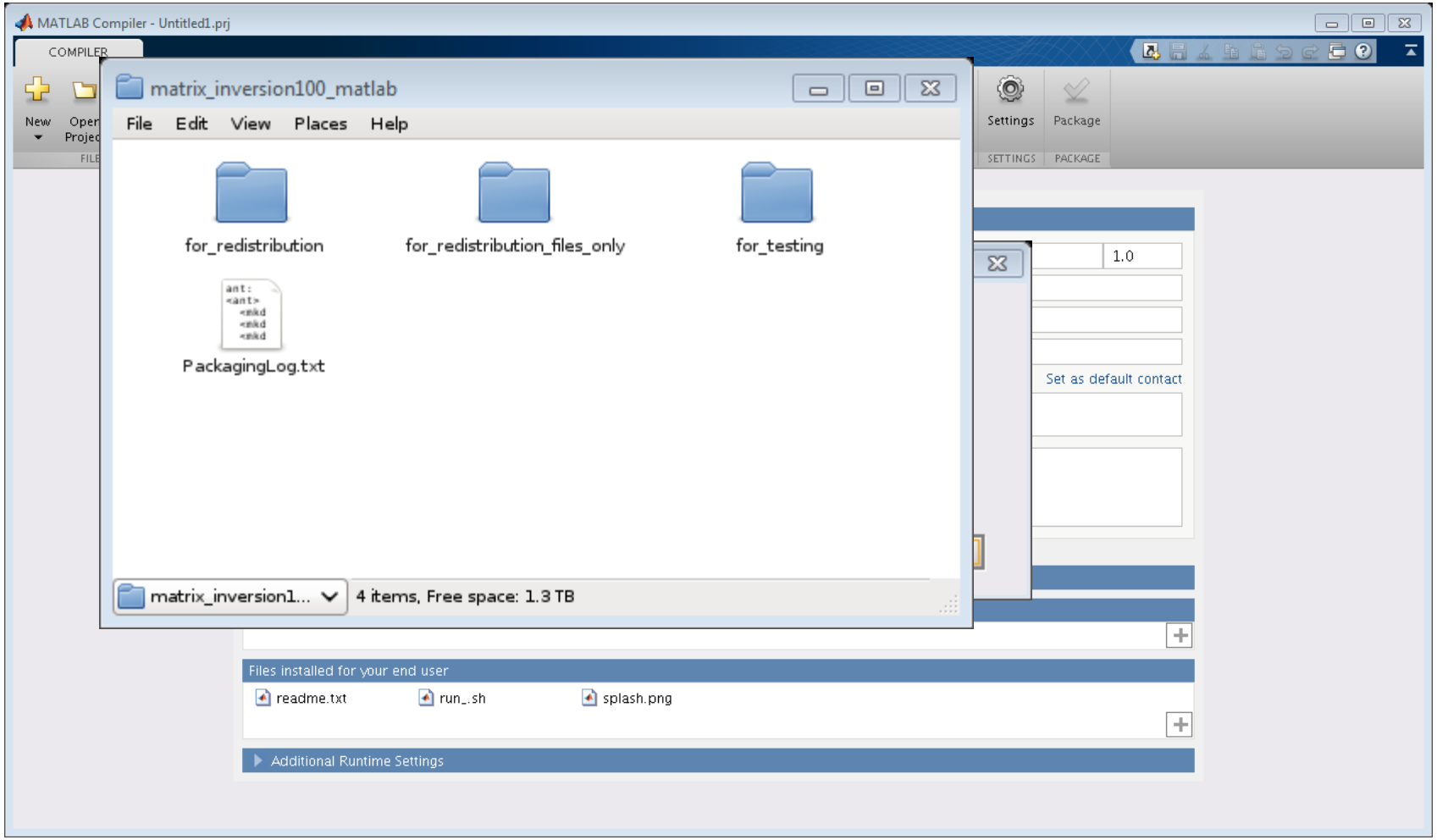
The MATLAB Compiler converts user-written MATLAB scripts into stand-alone applications that do not need to be run within the MATLAB environment. These applications can be run repeatedly and directly on any machine. When you run these applications, they do not require the use of MATLAB licenses. That is very beneficial on an HPC cluster running many concurrent MATLAB jobs with the PBS Pro batch system.

- MATLAB Compiler
  - mcc
  - Example

    scripts: problem.m LUsol.m LUdec.m

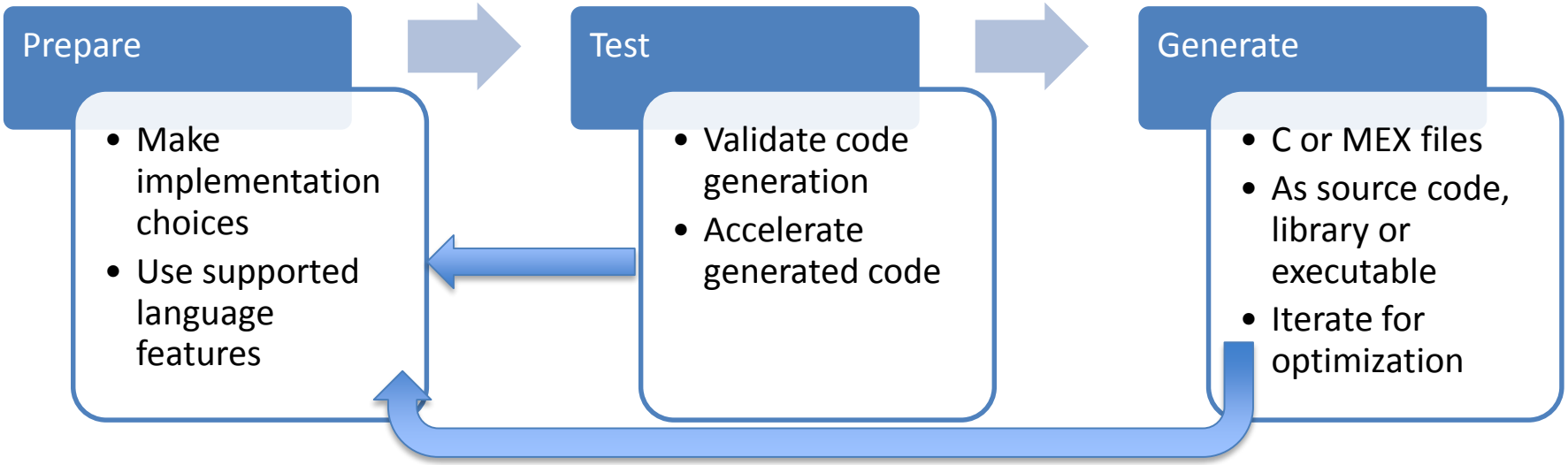
    mcc --e problem Lusol LUdec

Code compiling (GUI mode)



MATLAB to C code

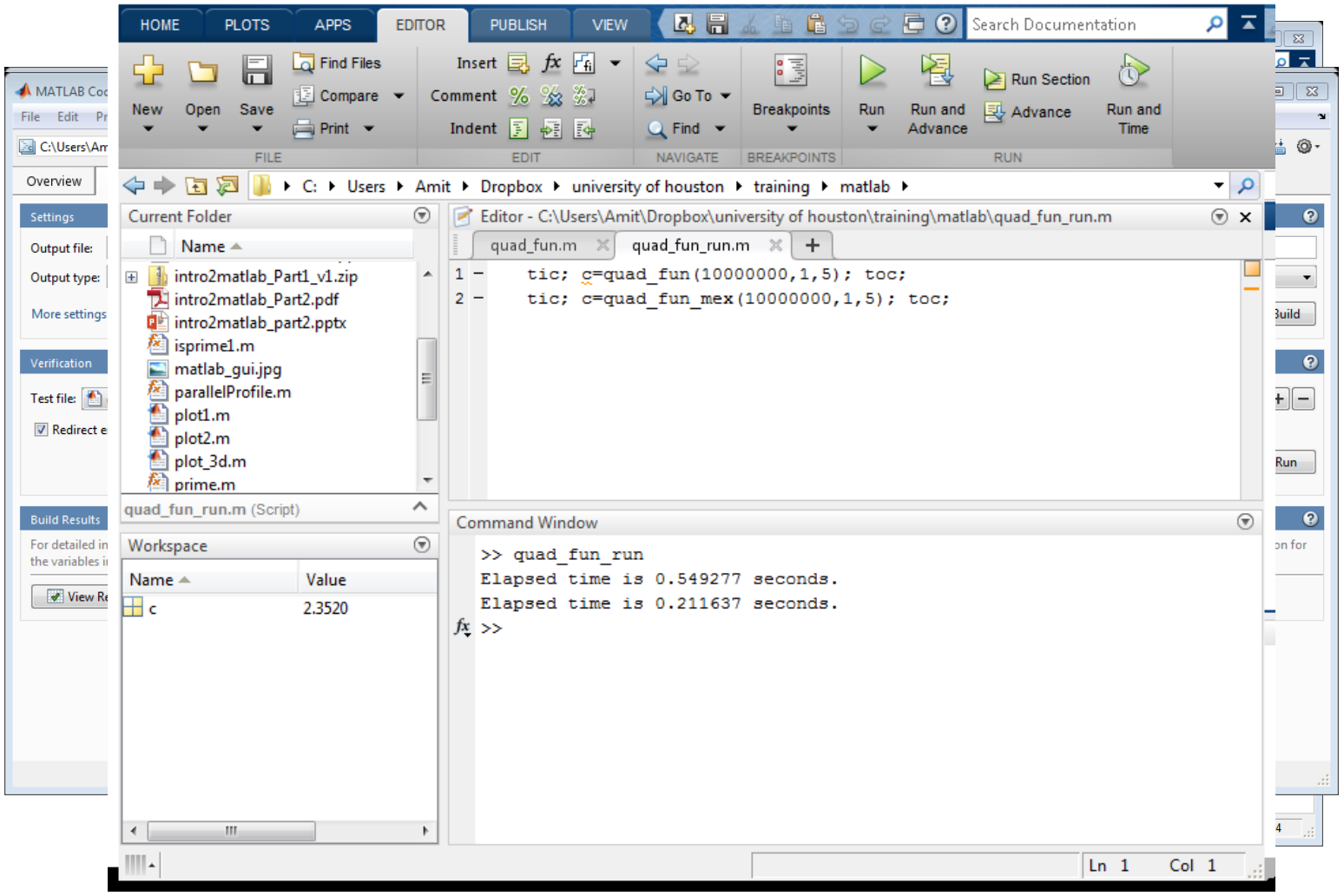
- Automated process
  - Avoids human errors
  - Single code maintenance (prototype to production)
- Suggested workflow



Code generation (command line)

- Generate C code from MATLAB code
  - use quad_fun.m
- Add $#codegen
- >>codegen –args {} *.m
  - Create MEX file
  - Check the MEX (*_mex.m) file for correctness
- >>codegen –args {} –report –config:lib *.m
  - Create C static/dynamic library
  - Verify the code generated
- Compare the performance
  - The MEX file performs faster than original MATLAB code
    - MATLAB is interpreted language
    - MEX files are compiled implementation of the code

Code generation (GUI mode)



Coder vs Compiler Comparison

	Coder	Compiler
Output	Portable and readable C source code	Software components
MATLAB support	Subset of language Some toolboxes	Full language Most toolboxes Graphics
Additional libraries	None	MATLAB Runtime
License model	Royalty-free	Royalty-free
Extensions	Embedded coder	MALTAB production Server

Conclusion

- Introduction: Parallel Computing Toolbox
- Models of parallelism:
  - parfor, spmd, GPU computing and task
- Models of execution:
  - Interactive vs. Indirect
  - Local vs. Remote
- Quadrature example: Parallelizing and Running
- Number of prime numbers: need for parallelism
- 2D FEM: distributed arrays (spmd)
- Code compiler and coder

Slides and exercises

- Please email the slides and exercises to yourself

- My office – PGH 223
- Email – aramritkar@uh.edu
- Office hours – Wednesdays 10 to 12

- Acknowledgement:
  - Jerry Ebalunode (UH), John Burkardt (FSU) and Gene Cliff (VT)

Please complete the evaluation form

Thanks!