MATLAB Parallel Computing

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

....

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

....

Introduction to Parallel MATLAB at Virginia Tech
https://people.sc.fsu.edu/∼jburkardt/presentations/
matlab_parallel_2010_vt.pdf

08 February 2010

MATLAB Parallel Computing

- **Introduction**
- Local Parallel Computing
- The MD Example
- PRIME_NUMBER Example
- Remote Computing
- KNAPSACK Example
- SPMD Parallelism
- `fmincon` Example
- Codistributed Arrays
- A 2D Heat Equation
- Conclusion

INTRO: 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.

INTRO: Local Parallelism

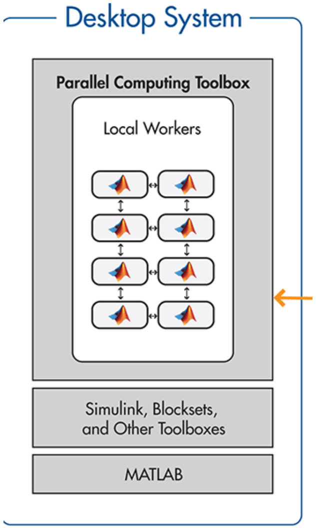
MATLAB has developed a *Parallel Computing Toolbox* which is required for all parallel applications.

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.

INTRO: Local and Remote MATLAB Workers


Desktop System
Parallel Computing Toolbox
Local Workers
Simulink, Blocksets, and Other Toolboxes
MATLAB

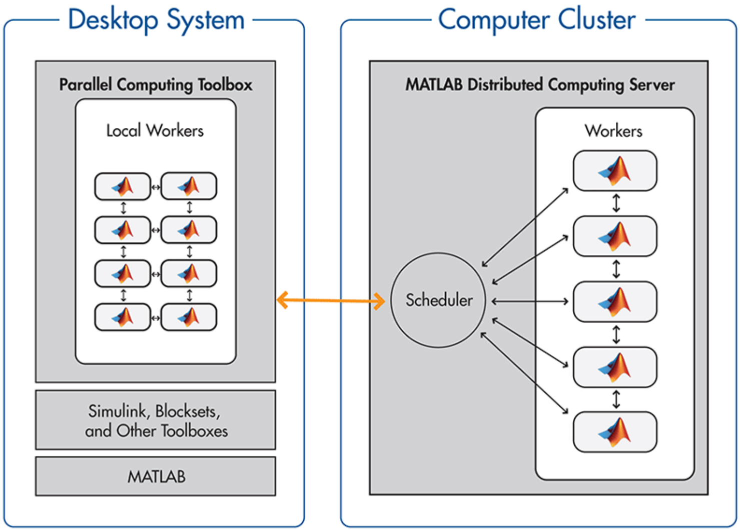
INTRO: Remote Parallelism

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.

INTRO: Local and Remote MATLAB Workers


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

INTRO: 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 ("Single Program, Multiple Data") 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.

INTRO: 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 only in batch mode. Currently, Virginia Tech permits interactive access to the cluster as well, but may soon also go to batch-only access.

INTRO: 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.

INTRO: ITHACA

Virginia Tech has installed the ITHACA cluster of 84 nodes. Each node is a separate computer with 2 quadcore processors.

This means each node can run 8 MATLAB workers.

At Virginia Tech, 8 nodes with 8 cores are dedicated to the parallel MATLAB cluster, so theoretically you can run a job with 64 workers.

You should not routinely ask for all 64 workers. Currently, one node is down, so there are only 56. Moreover, if one job ties up all the workers, no one else can run. So we encourage the use of 24 or 32 workers at a time instead.

MATLAB Parallel Computing

- Introduction
- **Local Parallel Computing**
- The MD Example
- PRIME_NUMBER Example
- Remote Computing
- KNAPSACK Example
- SPMD Parallelism
- `fmincon` Example
- Codistributed Arrays
- A 2D Heat Equation
- Conclusion

LOCAL: 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. (although some MATLAB linear algebra routines already use multithreading to involve more cores.).

The Parallel Computing Toolbox runs up to 8 cooperating copies of MATLAB, using the extra cores on your machine.

You'll need:

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

LOCAL: What Do You Need?

1. Your machine must have multiple cores:
   - On a PC: Go to **Start**, choose **Settings**, then **Control Panel**, then **System**.
   - On a Mac: From the Apple Menu, choose **About this Mac**, then **More Info...**.
2. Your MATLAB must be **version 2008a** or later:
   - To check MATLAB's version, go to the **HELP** menu, and choose **About Matlab**.
3. Your MATLAB needs the **Parallel Computing Toolbox**:
   - To list *all* your toolboxes, type the MATLAB command **ver**.

LOCAL: 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: Running A Program Interactively

To run an M file called, say, *md_parallel.m* in parallel on your machine, type:

```
matlabpool open local 4
md_parallel
matlabpool close
```

The word **local** is choosing the local configuration, 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 a local machine. It does not have to match the number of cores you have.

LOCAL: 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: Timing A Program

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

```
matlabpool open local 4

tic
md_parallel
toc

matlabpool close
```

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

LOCAL: 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
    md_parallel
    toc
    if ( 0 < labs ) matlabpool ( 'close' )
end
```

Because **labs** is a variable, we must use the “function” form of **matlabpool()** with parentheses and quoted strings.

LOCAL: Parallel Starts at 2 Labs

To run a parallel job with 0 labs means to run it sequentially.

To run a parallel job with 1 lab means the client sends all the data to the single lab and waits while the 1 lab does the job.

Only when we get to 2 labs do we have any hope of a speedup. Since it takes some time to set up the parallel execution and transfer data, we still won't see a speedup if the job is too small.

Since the machines in this classroom only have 2 processors, this means our demonstrations won't get much speedup today!

(For **local** parallel computing, it is possible to run 1 client and 2 workers on 2 cores. For **remote** computing, we would need 3 cores, with one dedicated to the client.)

MATLAB Parallel Computing

- Introduction
- Local Parallel Computing
- **The MD Example**
- PRIME_NUMBER Example
- Remote Computing
- KNAPSACK Example
- SPMD Parallelism
- `fmincon` Example
- Codistributed Arrays
- A 2D Heat Equation
- Conclusion

MD: The Molecular Dynamics 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.

MD: Run MATLAB's Profiler

```
>> profile on
>> md
>> profile viewer
```

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

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

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

MD: Profile Results

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

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

MD: 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
```

MD: Speedup

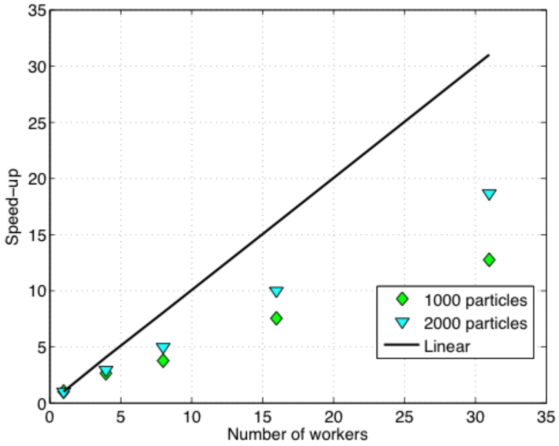
In this compute function, the important quantity is the force vector **f**. For each particle **i**, the force is computed by determining the distance to all other particles, squaring, truncating, and taking the sine.

The important thing to notice is that the computation for each particle can be done independently. That means we could compute each value on a separate worker, at the same time.

The MATLAB command **parfor** can replace **for** in this situation. It will distribute the iterations of the loop across the available workers.

MD: Speedup

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



MD: Speedup

This simple example demonstrates a case in which parallel execution of a MATLAB program gives a huge improvement in performance.

There is some overhead in starting up the parallel process, and in transferring data to and from the workers each time a **parfor** loop is encountered. So we should not simply try to replace every **for** loop with **parfor**.

That's why, in this example, we first searched for the function that was using most of the execution time.

The **parfor** command is the simplest way to make a parallel program, but we will see some alternatives as well.

MATLAB Parallel Computing

- Introduction
- Local Parallel Computing
- The MD Example
- **PRIME_NUMBER Example**
- Remote Computing
- KNAPSACK Example
- SPMD Parallelism
- `fmincon` Example
- Codistributed Arrays
- A 2D Heat Equation
- Conclusion

PRIME: The Prime Number Example

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** makes the run time increase by a factor of 4.

PRIME: Program Text

```
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 : sqrt ( i )
      if ( mod ( i, j ) == 0 )
        prime = 0;
        break
      end
    end

    total = total + prime;

  end

  return
end
```

PRIME: We can run this 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!

PRIME: Execution Commands

```
matlabpool ( 'open', 'local', 4 )

n = 50;

while ( n <= 500000 )
  primes = prime_number_parallel ( n );
  fprintf ( 1, '  %8d  %8d\n', n, primes );
  n = n * 10;
end

matlabpool ( 'close' )
```

PRIME: 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
```

PRIME: 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.

PRIME: Using the BATCH Command

Instead of running a program from the MATLAB command line, we can use the **batch** command, and have it execute "elsewhere".

Elsewhere might simply be on other workers; later we will see that we could also run the job on a remote cluster, such as ITHACA.

We have to run a script, not a function (we can't give it input!). So we might run our **prime_number_parallel** function with **n** fixed at 500,000.

The **matlabpool** command now needs 1 extra worker to be the client. On our desktop PC's, we only have 2 cores, so we won't gain anything in speed.

PRIME: A parallel script version

```
  n = 500000;

%function total = prime_number ( n )

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

  total = 0;

  parfor i = 2 : n

    prime = 1;

    parfor j = 2 : sqrt ( i )
      if ( mod ( i , j ) == 0 )
        prime = 0;
        break
      end
    end

    total = total + prime;

  end

%end
```

PRIME: Using the BATCH Command

```
job = batch ( 'prime_number_script', ...
  'configuration', 'local', ...    <-- Run it locally.
  'matlabpool', 2 )                <-- Two workers.

wait ( job );  <-- One way to find out when job is done.

load ( job );  <-- Load the output variables from
                   the job into the MATLAB workspace.

total          <-- We can examine the value of TOTAL.

destroy ( job );  <-- Clean up
```

PRIME: Using the BATCH Command

Using the **wait** command is easy, but it locks up your MATLAB session.

Using **batch**, you can submit multiple jobs:

```
job1 = batch ( ... )
job2 = batch ( ... )
```

Using **get**, you can check on any job's status:

```
get ( job1, 'state' )
```

Using **load**, you can examine just a single output variable from a finished job if you list its name:

```
total = load ( job2, 'total' )
```

PRIME: Remote Usage

The BATCH command can run your job elsewhere!

```
job = batch ( 'prime_number_script', ...
  'configuration', 'ithaca_2009b', ... <-- Run remotely
  'matlabpool', 32 )                 <-- Use 32 workers.

get ( job, 'State' );  <-- 'finished' if job is done,
     (and doesn't lock your session like wait() does).

load ( job );  <-- Loads all output from the job.

total          <-- We can examine the value of TOTAL.

destroy ( job );  <-- Clean up
```

MATLAB Parallel Computing

- Introduction
- Local Parallel Computing
- The MD Example
- PRIME_NUMBER Example
- **Remote Computing**
- KNAPSACK Example
- SPMD Parallelism
- `fmincon` Example
- Codistributed Arrays
- A 2D Heat Equation
- Conclusion

REMOTE: Enabling Remote Computing

MATLAB can run your programs on a remote machine.
From your desktop, you can submit jobs and review results.

Setting this up takes some work:

1. Update your MATLAB to R2009(a/b) with PCT;
2. Copy some functions into your toolbox/local directory;
3. Copy and customize a new configuration file;
4. Create a MATLAB work directory on your PC;
5. Get an account on Ithaca;
6. Create a MATLAB work directory on Ithaca;
7. Enable passwordfree logins.

The file **matlab_remote_submission.pdf** discusses these steps.

REMOTE: Communication

It's important to have some idea of how this communication works.

Your desktop **batch** command prompts MATLAB to copy the script file (and any file dependencies) and send them up to Ithaca's MATLAB work directory.

On Ithaca, a command is sent to the queueing system, requesting access to the appropriate number of nodes, to run your script with Ithaca's copy of MATLAB.

Output from the script is copied into Ithaca's MATLAB work directory and then copied back to your PC work directory.

You don't actually have to be running MATLAB on your PC while the job is running on Ithaca.

PRIME: The BATCH Command

```
job_id = batch (
  'script_to_run', ...
  'configuration', 'local' or 'ithaca_2009b', ...
  'FileDependencies', 'file' or {'file1','file2'}, ...
  'PathDependencies', 'path' or {'path1','path2'}, ...
  'matlabpool', number of workers (can be zero!) )
```

Note that you do not include the file extension when naming the script to run, or the files in the FileDependencies.

See page 13-2 of the PCT User's Guide for more information.
This slide is NOT in your handouts.

REMOTE: The Job State

When you submit a job to run remotely, a file is created in the local MATLAB work directory, containing a string that is the job's current state. From your local MATLAB, the command

```
  get ( job, 'State' )
```

will print out the current value (by simply printing this file's contents).

Thus, instead of using the **wait(job)** command, you can simply check the job's state from time to time to see if it is 'finished', and otherwise go on talking to MATLAB.

REMOTE: The Job State

Typical values of the job state variable include:

- **'pending'**: not yet submitted to the queue
- **'queued'**: submitted to the queue
- **'running'**: running
- **'finished'**: ran successfully
- **'failed'**: did not run or failed during run
- **'destroyed'**: you discarded this information

REMOTE: The Configuration File

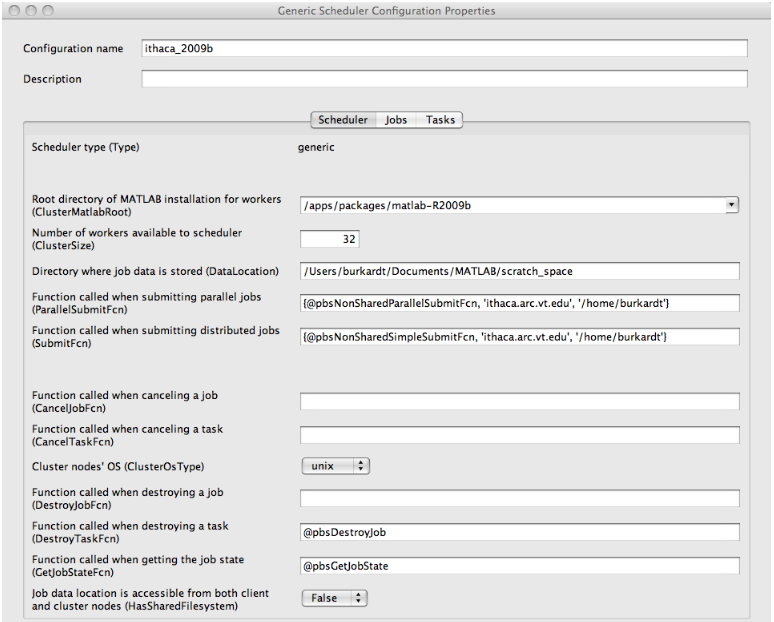
The configuration file tells MATLAB the names of work directories, the number of workers on the remote system, and so on.

If you want to set up access to Ithaca, you will get a partially filled out configuration file. You complete it by filling in names of directories for job data on:

- your PC: C:\matlab_jobdata
- or your Mac: /Users/burkardt/matlab_jobdata
- Ithaca: /home/burkardt/matlab_jobdata

These names are arbitrary, but the named directories must exist or be created before MATLAB can use them.

REMOTE: Example Configuration File



Configuration name	ithaca_2009b
Description	
Scheduler type (Type)	generic
Root directory of MATLAB installation for workers (ClusterMatlabRoot)	/apps/packages/matlab-R2009b
Number of workers available to scheduler (ClusterSize)	32
Directory where job data is stored (DataLocation)	/Users/burkardt/Documents/MATLAB/scratch_space
Function called when submitting parallel jobs (ParallelSubmitFcn)	{@pbsNonSharedParallelSubmitFcn, 'ithaca.arc.vt.edu', '/home/burkardt'}
Function called when submitting distributed jobs (SubmitFcn)	{@pbsNonSharedSimpleSubmitFcn, 'ithaca.arc.vt.edu', '/home/burkardt'}
Function called when canceling a job (CancelJobFcn)	
Function called when canceling a task (CancelTaskFcn)	
Cluster nodes' OS (ClusterOsType)	unix
Function called when destroying a job (DestroyJobFcn)	
Function called when destroying a task (DestroyTaskFcn)	@pbsDestroyJob
Function called when getting the job state (GetJobStateFcn)	@pbsGetJobState
Job data location is accessible from both client and cluster nodes (HasSharedFilesystem)	False

MATLAB Parallel Computing

- Introduction
- Local Parallel Computing
- The MD Example
- PRIME_NUMBER Example
- Remote Computing
- **KNAPSACK Example**
- SPMD Parallelism
- `fmincon` Example
- Codistributed Arrays
- A 2D Heat Equation
- Conclusion

KNAPSACK: Distributed Computing

In the examples of parallel programming that we have seen, the parallelism really does mean "parallel", that is, *at the same time*. In a **parfor** computation, the master process starts with all the data and the program, the workers always start and end together, and the results from each worker are collected by the master program.

The term *distributed computing* describes a "looser" computation which may be broken into completely independent parts which don't communicate, don't run at the same time, and don't run in any particular order.

We will look at an example, called the **knapsack** problem, which demonstrates this method of computing.

KNAPSACK: Problem Definition

Suppose we have a knapsack with a limited capacity, and a number of objects of varying weights. We want to find a subset of the objects which exactly meets the capacity of the knapsack.

(This is sometimes called the *greedy burglar's problem!*)

Symbolically, we are given a *target value* **t**, and **n** *weights* **w**. We seek **k** indices **s**, a subset of the weights, so that

$$t = \sum_{i=1}^{k} w(s(i))$$

We don't know if a given problem has 0, 1, or many solutions.

KNAPSACK: Encoding

A solution of the problem is a subset of **w**. A subset of a set of **n** elements can be represented by a binary string of length **n**. Therefore every binary string from 0 to $2^n - 1$ is also a code for a subset that is a possible solution.

For weights **w**={15,11,10,8,3}, target **t**=24, we have:

Code	Binary Code	Subset	Weight
0	00000	{}	0
1	00001	{1}	3
2	00010	{2}	8
3	00011	{2,1}	11
4	00100	{3}	10
5	00101	{3,1}	13
6	00110	{3,2}	18
...	...	...	...
31	11111	{5,4,3,2,1}	47

KNAPSACK: Algorithm

Although more sophisticated methods are available, a simple search scheme can be used. We simply examine **code** over the range 0 to $2^n - 1$, compute the corresponding subset, add up the selected weights, and compare to **t**.

For instance, the code of 22 = binary 10110 = subset {5,3,2} and hence a weight of 15+10+8=33, which is too high.

Notice that the process of checking one possibility is completely independent of checking any other.

One program could check them all, or we could subdivide the range, and check the subranges in any order and at any time.

KNAPSACK: MATLAB Program

```
function [ code, subset ] = knapsack ( w, t )

  n = length ( w );
  for code = 0 : 2^n-1
%  Convert CODE into vector of indices in W.
    subset = find ( bitget ( code, 1:n ) );
%  Did we match the target sum?
    if ( sum ( w(subset) ) == t )
      return
    end
  end

  return
end
```

KNAPSACK: Distributed Version

Suppose we break the problem into distinct subranges to check.

MATLAB's distributed computing option calls the original problem the **job**. Checking a subrange is one **task** of the job. Each task calls the same MATLAB function with different arguments.

MATLAB lets us "submit" the job; a task is assigned to a worker that is available. These tasks can run locally or remotely, simultaneously or sequentially or at substantially different times. Because each tasks runs when it can, and they don't communicate, overhead and scheduling delays are avoided.

The job completes when all tasks are run.

KNAPSACK: Distributed MATLAB Program

```
function [ code, subset ] = knapdist ( w, t, range )

  n = length ( w );
  for code = range(1) : range(2)
%  Convert CODE into vector of indices in W.
    subset = find ( bitget ( code, 1:n ) );
%  Did we match the target sum?
    if ( sum ( w(subset) ) == t )
      return
    end
  end

  return
end
```

KNAPSACK: Distributed MATLAB Program

The program can work on the whole problem or a given subrange, depending on the values in **range**.

The MATLAB function **bitget** returns a vector of 0's and 1's for positions 1 to **n** in **code**.

The function **find** returns the locations of the 1's, which is how we get our list of weights to try for this subset.

KNAPSACK: Define the Job and its Tasks

```
job = createJob ( 'configuration', 'local', ...
  'FileDependencies', ...
  { 'knapdist.m', 'knapweights.mat' } );

i2 = -1;
for task = 1 : 4
  i1 = i2 + 1;
  i2 = floor ( ( 2^n - 1 ) * task / 4 );
  createTask ( job, @knapdist, 2, { w, t, [ i1, i2 ] } );
end
```

KNAPSACK: Define the Job and its Tasks

The **createJob** command is like the **batch** command, except it doesn't say what we're going to do, or request a specific number of workers.

```
job_id = createJob (
  'configuration', 'local' or 'ithaca_2009b', ...
  'FileDependencies', 'file' or {'file1','file2'}, ...
  'PathDependencies', 'path' or {'path1','path2'} )
```

See page 13-52 of the PCT User's Guide.
This slide is NOT in your handouts.

KNAPSACK: Define the Job and its Tasks

The **createTask** command defines the tasks that make up the job. In particular, it names the MATLAB function that will be called, the number of output arguments it has, and the values of the input arguments.

```
task_id = createTask (
  job_id, ...  <-- ID of the job
  @function, ... <-- MATLAB function to be called
  numarg, ...    <-- Number of output arguments
  { arg1,arg2,...} ) <-- Input arguments
```

See page 13-59 of the PCT User's Guide.
This slide is NOT in your handouts.

KNAPSACK: Submit the Job

With the following commands, we submit the job, and then pause our interactive MATLAB session until the job is finished.

We then retrieve the output arguments *from each task*, in a cell array we call **results**.

```
submit ( job );   <-- Sends the job
wait ( job );     <-- Waits for completion.
results = getAllOutputArguments ( job );
destroy ( job );  <-- Clean up
```

KNAPSACK: BATCH versus CREATEJOB

The **batch** command can be thought of as a simplified version of **createJob** + **createTask** + **submit**.

It assumes you only have 1 task and that the task is defined by a script with no input arguments, and it's ready to submit.

Otherwise, both commands are doing the same thing, finding out what you want to and where it should be executed, assigning it a logical identifier, and sending the work to the right machine.

For the distributed **KNAPSACK** job, of course, we needed the extra flexibility of the **createJob** command.

KNAPSACK: Examine the results

Because your job involved multiple tasks, the output must be returned to you in a *cell array*. To see output result 2 from task 3, you refer to **results**{**3,2**}.

```
for task = 1 : 4
  if ( isempty ( results{task,1} ) )
    fprintf ( 1, 'Task %d found no solutions.\n', task )
  else
    disp ( 'Weights:' );
    disp ( results{task,1} );
    disp ( 'Weight Values:' );
    disp ( results{task,2} );
  end
end
```

KNAPSACK: Running Jobs Remotely

If you have set up your machine so that the local copy of MATLAB can talk to remote copies, then this same distributed job can be run on a remote machine, such as the Virginia Tech **ithaca** cluster.

All you have to do is change the configuration argument when you define the job:

```
job = createJob ( 'configuration', 'ithaca_2009b', ...
  'FileDependencies', ...
  { 'knapdist.m', 'knapweights.mat' } );
```

MATLAB Parallel Computing

- Introduction
- Local Parallel Computing
- The MD Example
- PRIME_NUMBER Example
- Remote Computing
- KNAPSACK Example
- **SPMD Parallelism**
- `fmincon` Example
- Codistributed Arrays
- A 2D Heat Equation
- Conclusion

SPMD: Single Program, Multiple Data

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.

Distributed programming allows us to run different programs, or the same program with different inputs, but they can't talk to each other, that is, communicate or share results.

The **SPMD** command is like working with a very simplified version of **MPI**. There is a client process and workers, but now the workers are given identifiers. Each worker decides what to do based on its ID. Each worker can communicate with the client. Any two workers can communicate through the client.

SPMD: 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
```

SPMD: Each worker has an ID

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**!

SPMD: Use the ID to assign work

Now let's go back to our program fragment. But first we must explain that we are trying to approximate an integral over the interval [0,1]. Using **SPMD**, we are going to have each 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
```

SPMD: One Name Must Reference Several Values

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!

SPMD: Dealing with Composite Variables

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 : 4   <-- "numlabs" wouldn't work here!
  fprintf ( 1, '  A = %f, B = %f\n', a{i}, b{i} );
end
```

SPMD: The Solution in 4 Parts

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

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

with result:

```
2   Partial approx 0.874676
4   Partial approx 0.567588
1   Partial approx 0.979915
3   Partial approx 0.719414
```

SPMD: Combining Partial Results

We really want one answer, the sum of all these approximations.

One way to do this is to gather the answers back on the client:

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

with result:

```
Approximation 3.14159265
```

SPMD: Source Code for QUAD_SPMD

```
function value = quad_spmd ( n  )

  fprintf ( 1, 'Compute limits\n' );
  spmd
    a = ( labindex - 1 ) / numlabs;
    b =   labindex       / numlabs;
    fprintf ( 1, '  Lab %d works on [%f,%f].\n', labindex, a, b );
  end

  fprintf ( 1, 'Each lab estimates part of the integral.\n' );
  spmd
    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
```

SPMD: Combining Values Directly

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
```

SPMD: 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**.

SPMD: MPI-Style Messages

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

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

For details on how these commands work, start with the MATLAB HELP facility!

For more information, refer to the documentation for the Parallel Computing Toolbox.

MATLAB Parallel Computing

- Introduction
- Local Parallel Computing
- The MD Example
- PRIME_NUMBER Example
- Remote Computing
- KNAPSACK Example
- SPMD Parallelism
- **`fmincon` Example**
- Codistributed Arrays
- A 2D Heat Equation
- Conclusion

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.

fmincon and UseParallel

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

The dynamics are given by

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

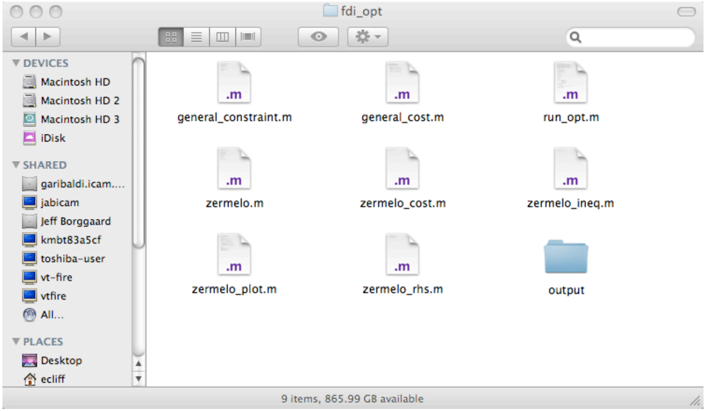
with initial condition $x(0) = y(0) = 0$.
The problem is to `maximize` $x(t_f)$ with the constraint $y(t_f) > y_f$ ($t_f, y_f,$ and κ are given).

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(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:



general_constraint.m
general_cost.m
run_opt.m
zermelo.m
zermelo_cost.m
zermelo_ineq.m
zermelo_plot.m
zermelo_rhs.m
output

9 items, 865.99 GB available

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
```

MATLAB Parallel Computing

- Introduction
- Local Parallel Computing
- The MD Example
- PRIME_NUMBER Example
- Remote Computing
- KNAPSACK Example
- SPMD Parallelism
- `fmincon` Example
- **Codistributed Arrays**
- A 2D Heat Equation
- Conclusion

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());
```

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).

getLocalPart(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() )
  >> 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 - getLocalPart(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
- The MD Example
- PRIME_NUMBER Example
- Remote Computing
- KNAPSACK Example
- SPMD Parallelism
- `fmincon` Example
- Codistributed Arrays
- **A 2D Heat Equation**
- Conclusion

2DHEAT: State Equations

An example: 2D unsteady heat equation:

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

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).

2DHEAT: Boundary Conditions

Boundary conditions for our problem are:

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

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

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

2DHEAT: Time-Discretized State Equations

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

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

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

2DHEAT: Time-Discretized Boundary Conditions

Imposing the specified boundary conditions, the boundary term evaluates to

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

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

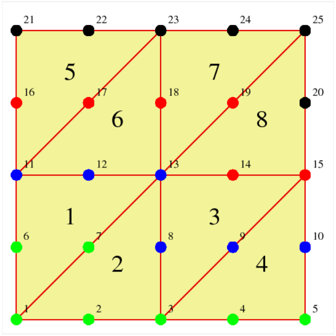
2DHEAT: Finite Element Grid

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 ℓ m rectangles; diagonals divide these into 2 ℓ m triangles.

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



2DHEAT: Finite Element Equations

Seek an approximate solution: $T_N^n(x,y) = \sum_{\jmath=1}^{N} z_\jmath^n \, \Phi_\jmath(x,y)$.

$$
\begin{aligned}
&\sum_{\jmath} \Bigg[\int_\Omega \Phi_\jmath(x,y)\, \Phi_\imath(x,y)\, \mathrm{d}\omega \\
&+ \frac{\Delta t}{\sigma\, C_p} \left(\int_\Omega (k \nabla \Phi_\jmath \cdot \nabla \Phi_\imath)\ \mathrm{d}\omega + \int_w^0 \alpha(y)\, \Phi_\jmath(0,y)\, \Phi_\imath(0,y)\, \mathrm{d}y \right) \Bigg] z_\jmath^{n+1} \\
&- \sum_{\jmath} \left[\int_\Omega \Phi_\jmath(x,y)\, \Phi_\imath(x,y)\, \mathrm{d}\omega \right] z_\jmath^n - \left[\frac{\Delta t}{\sigma\, C_p} \int_\Omega F(x,y,t_{n+1}) \Phi_\imath\, \mathrm{d}\omega \right] \\
&- \frac{\Delta t}{\sigma\, C_p} \left[\int_0^w f(y) \Phi_\imath(L,y)\, \mathrm{d}y + \int_w^0 \alpha(y)\beta(y)\Phi_\imath(0,y)\, \mathrm{d}y \right] = 0
\end{aligned}
$$

We can rewrite this in matrix terminology:

$$
(\mathrm{M}_1 + \mathrm{M}_2)\, z^{n+1} \ - \ \mathrm{M}_1\, z^n \ + \ \mathrm{F}(t_{n+1}) \ + \ \mathrm{b} = 0 \ .
$$

2DHEAT: Modifying a Serial Code

So our computation requires repeatedly forming and solving systems of the form:

$$(\mathrm{M}_1 + \mathrm{M}_2)\, z^{n+1} \;-\; \mathrm{M}_1\, z^n \;+\; \mathrm{F}(t_{n+1}) \;+\; \mathrm{b} = 0\ .$$

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

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

2DHEAT: ASSEMB_CO Source Code (begin)

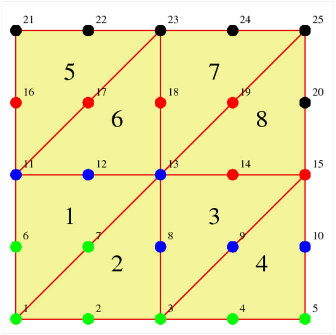
```
  function [M1, M2, F, b, x, e_conn] = assemb_co(param)
  % The FEM equation for the temp. dist at time t_{n+1} satisfies
  %   (M_1 + M_2) z^{n+1} - M_1 z^n + F + b = 0

%% 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
```

2DHEAT: 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

%
         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('1d', 2));
  M2 = codistributed(M2_lab, codistributor('1d', 2));
  b  = codistributed( b_lab, codistributor('1d', 1));
  F  = codistributed( F_lab , codistributor('1d', 1));
```

2DHEAT: 5×5 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 computational load badly balanced.

2DHEAT: 5×5 grid on 4 labs

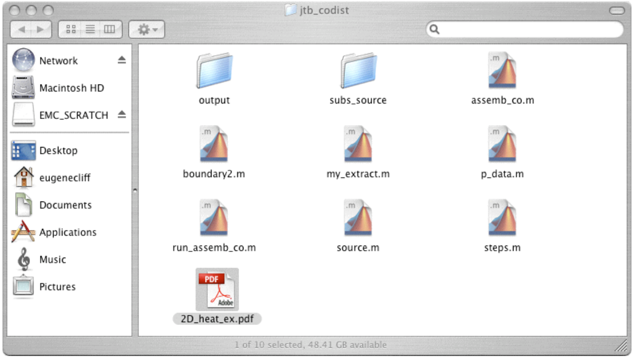
```
%%%% run these in Matlab
  >> pmode start 4
  >> Vc = codcolon(1, 25)
  >> dPM = distributionPartition(codistributor(Vc))
  >> col_shft = [ 0 cumsum(dPM(1:end-1))]
  >> whos
%%%%
```

2DHEAT: 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
```

2DHEAT: RUN_ASSEMB_CO 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
- The MD Example
- PRIME_NUMBER Example
- Remote Computing
- KNAPSACK Example
- SPMD Parallelism
- `fmincon` Example
- Codistributed Arrays
- A 2D Heat Equation
- **Conclusion**

Conclusion: Desktop Experiments

Virginia Tech has a limited number of concurrent MATLAB licenses, which include the Parallel Computing Toolbox.

This is one way you can test parallel MATLAB on your desktop machine.

If you don't have a multicore machine, you won't see any speedup, but you may still be able to run some "parallel" programs.

Conclusion: Cluster Experiments

If you want to work with parallel MATLAB on Ithaca, there should soon be a way to apply for accounts online at the ARC website:

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

Until then, you can get a “friendly user” account by sending mail to John Burkardt **burkardt@vt.edu**.

Conclusion: PC-to-Cluster Submission

If you want to use parallel MATLAB regularly, you may want to set up a way to submit jobs from your PC to Ithaca, without logging in directly.

This requires defining a configuration file on your PC, adding some scripts to your MATLAB directory, and setting up a secure connection between your PC and Ithaca. The steps for doing this are described in the document:

```
http://people.sc.fsu.edu/~burkardt/pdf/...
  matlab_remote_submission.pdf
```

We will be available to help you with this process.

Conclusion: Documentation

On the MathWorks directory, there is a directory for the Parallel Computing Toolbox:

`http://www.mathworks.com/products/parallel-computing/`

Look under "Documentation". This is a reference manual, not a guide. It will tell you the details about how to use a particular command, but not so much which commands you should use.

Under "Demos and Webinars" you may find help about how to design the right kind of parallel program for your problem.