Parallel MATLAB

Objectives

- Parallel computing in MATLAB
- Functional parallelism; data parallelism
- Inter-worker communication

Overview of MATLAB

MATLAB stands for "MATrix LABoratory".

It is a *problem-solving (software) environment* (PSE) for numerical computing.

It was developed in the late 1970s by Cleve Moler while teaching at Stanford University with the goal of enabling students to perform numerical computations without having to learn a "low-level" programming language (Fortran).

The MATLAB PSE comes equipped with a fourth-generation programming language with the same name.

Vector operations in MATLAB

The ordinary (serial) version of MATLAB has some important vectorization capabilities.

MATLAB was designed to perform matrix (or vector[1]) operations with high efficiency.

So, any operation(s) that can be converted to matrix operation(s) stand to see an increase in performance.

The main constructs that are most amenable to matrix operations are loops.

The basic distinction is that vectorization results in *one* function call in total, as opposed to one per iteration, thus greatly reducing the overhead.

Multi-threaded vectorization has been available in MATLAB since version R2008a and is a built-in way to reduce the overhead associated with function calls.

[1] In MATLAB, all numerical variables are treated as matrices of the appropriate size; e.g., scalars are 1×1 matrices, etc.

Vector operations in MATLAB

So, for example, the statement

```
C = A*B;
```

is much more efficient than

```
for i=1:m,
  for j=1:n,
    C(i,j) = C(i,j) + A(i,j)*B(i,j);
  end
end
```

For this example, of course `C` would have to be initialized to an appropriately sized matrix of zeros.

There are other functions (often within other toolboxes) that have been written to take advantage of parallelization "automatically" (as options); see

`http://www.mathworks.com/builtin-parallel-support.html`

Before parallelization

Often, a good starting point for developing a good parallel program is from a good serial program.

In some cases, writing a good serial code may be sufficient for your short-term needs.

In general, pre-allocation of arrays (rather than growing them dynamically) is also an important part of writing efficient MATLAB code.

Pre-allocating arrays is an example of *best coding practices*, which you are strongly encouraged to use regardless of language, platform, etc.

Profiling the code to know whether parallelization may help is another good coding practice.

See also

`http://www.mathworks.com/videos/speeding-up-matlab-applications-81729.html`

Multiple processors, one computer

In order to take advantage of multiple processors (or cores) on one computer, in addition to the standard version of MATLAB, access is required to the *Parallel Computing Toolbox* (PCT).

The PCT supports usage of up to 8 processes/threads (called *workers* in MATLAB parlance) on `socrates` (2 quad-core processors per node).

The availability of this toolbox can be determined by typing `ver` at the MATLAB prompt.

If available, the output to this command should have a line that looks something like

```
Parallel Computing Toolbox Version 6.2 (R2013a)
```

You could also specifically search for the distributed computing license by typing

```
ver distcomp
```

GPU

MATLAB has good support for GPU computations built in to the PCT.

Over 200 MATLAB functions have been GPU-enabled.

Many toolboxes (e.g., Optimization, Signal Processing) also have GPU-enabled functions.

Specific support for CUDA is available.

The programming paradigm is a little different in that one issues commands to the GPU (as opposed to starting workers).

Computing with clusters of GPUs requires the Distributed Computing Server.

`zeno` has MATLAB installed.

More information is available at

`http://www.mathworks.com/gpu`

Multiple processors, multiple computers

In order to take advantage of multiple computers, access is *also* required to the MATLAB *Distributed Computing Server* (DCS).

In MATLAB parlance, computers that request jobs are called *clients*.

The DCS will have a number of licenses associated with the installation being accessed.

One available license per requested worker is required!

The licenses themselves are checked out dynamically.

A *scheduler* distributes tasks to workers and ensures each has any licenses required; this requires sufficient available licenses *on the client* for each additional toolbox (i.e., not the PCT) used.

The scheduler also aggregates the results from the workers into a single result on the client.

Multiple processors, multiple computers

The default profile is called `local` and comes with the PCT.

It starts up additional MATLAB sessions on the local machine and is typically used for prototyping or for only mutli-core parallelization.

To utilize a cluster, the MATLAB DCS is required, and a profile for this must be created separately.

Perhaps the easiest way to do this is done by selecting `Parallel` then `Discover Clusters ...` from the menu bar.

Further information is available at

`http://www.mathworks.ca/distconfig`

It is also possible to use third-party schedulers, but we will not delve into the details of this.

Validating a profile

To ensure a profile has been properly initialized, we "validate" it.

Click on `Parallel` in the menu bar, then select `Manage Cluster Profiles ...`.

For example, to test the `local` configuration, select `local`, then click `Validate`.

All of the tests should return as `Passed`.

This configuration should work with up to 8 workers on (one node of) `socrates`.

The properties of the profile can be viewed and edited by clicking on the `Properties` tab.

Note: If the validation fails, it may be because there are insufficient licenses.

You may wish to reduce the number of workers and try again, but the problem is not necessarily with the configuration file per se.

Creating a new profile

To take advantage of more than one node, we need to create another profile.

We can `Discover Clusters` either from the `Parallel` menu or from the `Manage Cluster Profiles ...` window.

A scheduler called `Torque.mat` will be provided.

After typing `load('Torque.mat')`, it is possible to discover the cluster on `socrates` using the built-in `Discover Clusters` tool.

There is only one discoverable cluster on `socrates`; it is called `MJSProfile1`[2].

After selecting it, you can `Verify` it.

The situation on `moneta` is similar for the `local` profile, but it is not a distributed cluster, so there is no scope for discoverable clusters.

[2] MJS stands for *MathWorks Job Scheduler*.

Checking license availability

To check for the availability of licenses, the following commands can be used to query the license manager and check for the relevant information.

On `socrates`, we use

```
/export/apps/MATLAB/R2013a/etc/glnxa64/lmutil lmstat -a
-c /export/apps/MATLAB/R2013a/licenses/network.lic | grep Dist
```

The output from this command would ideally look something like

```
Users of Distrib_Computing_Toolbox:
(Total of 6 licenses issued;  Total of 3 licenses in use)
  "Distrib_Computing_Toolbox" v29, vendor: MLM
Users of MATLAB_Distrib_Comp_Engine:
(Total of 32 licenses issued;  Total of 0 licenses in use)
```

On `moneta`, we use

```
/opt/matlab.R2013a/etc/glnxa64/lmutil lmstat -a
-c /opt/matlab.R2013a/licenses/network.lic | grep Dist
```

Built-in support for parallelization

The PCT was designed so that all of MATLAB's functionality could be seamlessly parallelized.

So there are a lot of useful functions whose parallelization is supported; perhaps the most notable ones for us are the toolboxes for Optimization, Global Optimization, Image Processing, and Statistics.

The full list of supported packages can be found at

http://www.mathworks.com/products/parallel-computing/builtin-parallel-support.html.

The list of MATLAB functions that are not parallelizable is relatively short; perhaps the most notable one for us is the MATLAB compiler.

The full list of unsupported packages can be found at

`http://www.mathworks.com/products/ineligible_programs/`.

MATLAB *parallel constructs*

Before doing any parallel computation in MATLAB, a pool of workers must be *opened* (reserved); e.g.,

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

opens `N` MATLAB workers according to the profile `local`.

MATLAB will open the worker pool automatically when certain commands are encountered, e.g., `parfor`, but this usage is not recommended.

The size of the available worker pool can be determined using

```
matlabpool('size')
```

After the program is completed, the worker pool should be closed (to free up the workers/licenses) via

```
matlabpool('close')
```

The `matlabpool size` command returns 0 if the pool is closed.

MATLAB *parallel constructs*

There are a few basic constructs used by MATLAB for parallel computing:

- `parfor` loops
- `batch`
- `labSend`
- `createJob` / `createTask`
- `spmd`
- `createCommunicatingJob` / `createTask`

The most appropriate option depends on the specific situation, e.g., assigning tasks to workers, distributing data between workers, etc.

parfor

The `parfor` construct can only be used for parallel execution of independent iterations of a for-loop.

The good news is that it is optimized for such computations and thus performs better than the equivalent `createJob` / `createTask` construct, which we will see later.

Also, code conversion from serial is trivial.

However, this also means its use is restricted.

For example, one cannot execute calls to different functions in parallel nor retrieve intermediate results.

Furthermore, the failure of one iteration will cause the entire loop to fail.

The principle to remember is that iterations of a `parfor` loop are executed autonomously.

`parfor`

Things to note about the use of `parfor`:

- Scalar variables defined outside a `parfor` loop but used inside are *broadcasted* to all workers.
- Only the relevant parts of array variables defined outside a `parfor` loop but used inside are *broadcasted* to all workers.
- The `parfor` loop index can be used inside the `parfor` loop.
- The `parfor` loop index must take on integer values.
- `parfor` loops cannot be nested.
- All iteration-dependent behaviour is prohibited.
- Load-balancing is not accounted for.

Some of these issues can be obviated, but not others.

parfor

For example, in order to parallelize the following serial MATLAB code to loop over a non-integer counter,

```
for x = 0:0.1:1,
  % iterations depending on x
end
```

we can write

```
X = 0:0.1:1;
parfor i = 1:length(X),
  x = X(i);
  % iterations depending on x
end
```

parfor

For example, in order to parallelize the following serial MATLAB code with a double-nested loop that uses rows of a matrix A

```
m = 10;
A = rand(m);
for i = 1:m,
  for j = 1:m,
    % iterations depending on A(i,j)
  end
end
```

we can write

```
m = 10;
A = rand(m);
parfor i = 1:m,
  X = A(i,:); % extract row
  for j = 1:m,
    % iterations depending on X(j)
  end
end
```

parfor

Although strictly speaking, `parfor` loops cannot be nested . . .

```
% the following code will not work
parfor i = 1:m,
  parfor j = 1:n,
    ...
  end
end
```

. . . another program can be called, and that program can have a `parfor` loop in it:

```
parfor i = 1:m,
  parfor_function(i)
end

function parfor_function(i)
parfor j = 1:n,
  ...
end
```

`createJob` / `createTask`

The `createJob` / `createTask` construct is much more flexible than `parfor` for creating independent (non-communicating) jobs.

It does not suffer from the aforementioned drawbacks of the `parfor` construct.

You can generally parallelize non-for-loop constructs.

Of course, this comes at the cost of making the code conversion less trivial as well as a performance hit associated with going through the scheduler, i.e., starting up the workers, etc.

The (obvious) rule of thumb is that one will not see the performance advantage of parallelism in practice unless the computation takes (much) longer than it does to start up the workers, etc.

`createJob` / `createTask`

The procedure for using the `createJob` / `createTask` construct is a bit more involved than for `parfor`.

An overview of the steps are as follows:

- Connect to a scheduler.
- Create a job.
- Create tasks.
- Submit the job.
- When the job completes, collect the output data.
- Destroy the job.

createJob / createTask

An independent job on a cluster with a local profile can be created via

```
job = createJob('local')
```

or we can first create a cluster object

```
c = parcluster('local')
job = createJob(c)
```

Tasks are then added to the job via

```
task = createTask(job, functionHandle, numOutputs, inputCell)
```

For example,

```
task = createTask(job, @rand, 1, {3,2})
```

creates a 3×2 random matrix.

Tasks may be created in a for-loop or vectorized.

`createJob` / `createTask`

The job is sent from the client to the scheduler via

`submit(job)`

The job is run immediately if sufficient workers are available for it.

If not, it will be queued until there are.

If you do not plan to shut down the client computer, you can use

`wait(job, 'finished')`

Your MATLAB session is blocked until the job finishes.

Other options include `'queued'` and `'running'`.

The default (no options) is `'finished'`.

(Pressing Ctrl-C will return control to the command window but will not kill the job!)

createJob / createTask

Alternatively, the state of a job can be checked manually using, e.g.,

```
[pending queued running completed] = findJob(c);
```

returns an array of all job objects in cluster object c according to their states.

The output is ordered in a meaningful way: according to creation or position in queue.

The list of `completed` jobs includes those that failed but does not include those that were deleted (or whose status was otherwise unavailable).

createJob / createTask

When the job has finished, output arguments can be collected via

```
results = fetchOutputs(job);
```

`results` is a cell array that will be empty if the job has not finished.

Tasks that finished with an error return [].

Now all that remains is to destroy the job to free up the resources:

```
delete(job)
```

This command will terminate `job` whether it has completed or not!

Note that `clear`ing a job does not destroy the job.

Example

We determine the largest eigenvalue of a number of random matrices.

Here is what a serial code might look like:

```
function a = example1_serial()

% find the largest eigenvalue of N random matrices of size m

N = 2000;       % number of trials
m = 50;         % size of random matrices
a = zeros(N,1); % pre-allocate vector of zeros to store results
seeds = (1:N);

tic; % serial for-loop
for I = 1:N
    a(I) = largestEigenvalue(m,seeds(I));
end
t = toc;

disp(['Time for serial calculation: ', num2str(t), ' seconds.'])
```

Example

The `largestEigenvalue` function is straightforward:

```
function lambdaMax = largestEigenvalue(m, seed)
RandStream.setDefaultStream(RandStream('mt19937ar','seed',seed));
lambdaMax = max(eig(rand(m)));
```

The function selects and seeds the random number generator, then creates a matrix with random elements between 0 and 1, finds its eigenvalues, and takes the maximum of them (as measured by their modulus for complex numbers).

The output from this code run on `socrates` is

```
Time for serial calculation: 3.9776 seconds.
```

Example

A quick analysis of this code according to Foster's methodology yields the following results:

1. *Partitioning.* The essential tasks are to generate the seeds, generate the random matrices, compute the eigenvalues, and take the maximum.

2. *Communication.* Send seeds; receive result.

3. *Aggregation.* It makes sense to aggregate the tasks of generating the random matrices, computing the eigenvalues, and taking the maximum of them and perform them locally.

4. *Mapping.* Each iteration to a worker.

This is an ideal example for using the `parfor` construct.

Memory and performance profiling

To verify our analysis of the parallelization process, we determine where the serial code spends its time.

In order to do this, we *profile* the code; i.e., we use a tool that gives us the times spent in the different function calls within the code.

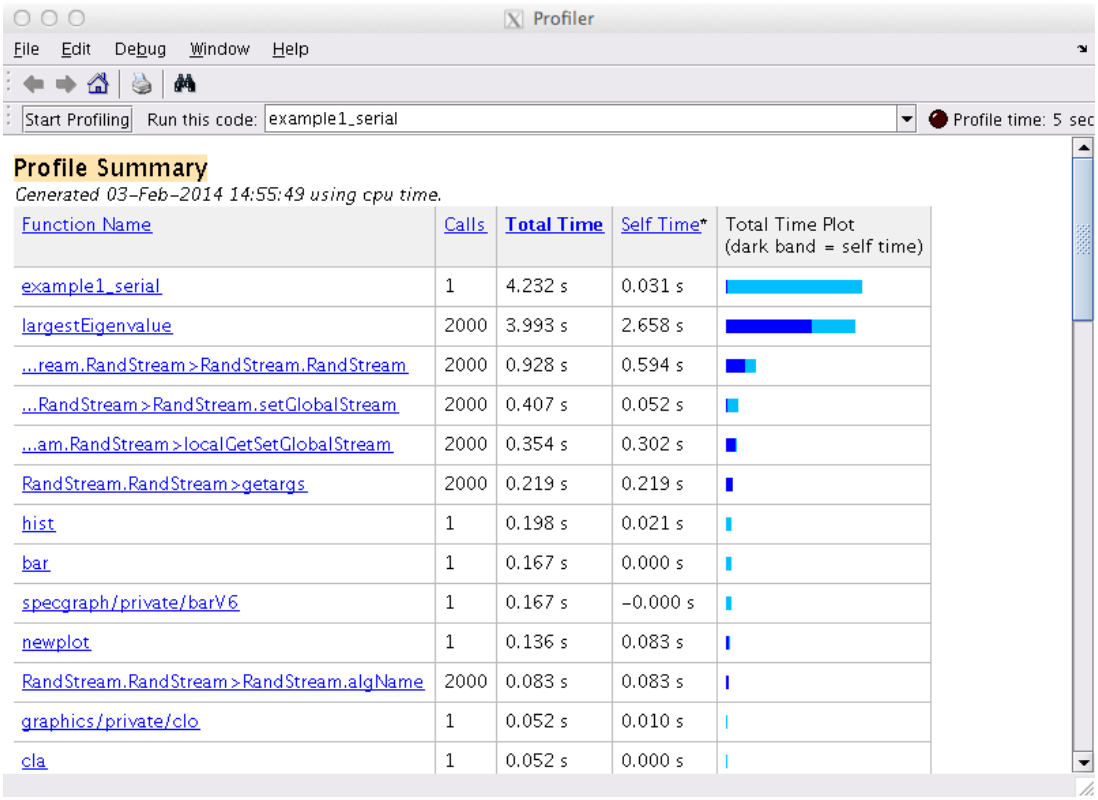
To profile the code, we type `profile viewer` at the MATLAB prompt.

In the profiler window, type `example1_serial` in the field after "Run this code:".

The profiler displays a window listing the times taken by the functions called in executing `example1_serial.m`, including `example1_serial.m` itself.

The output is given in the following figure.

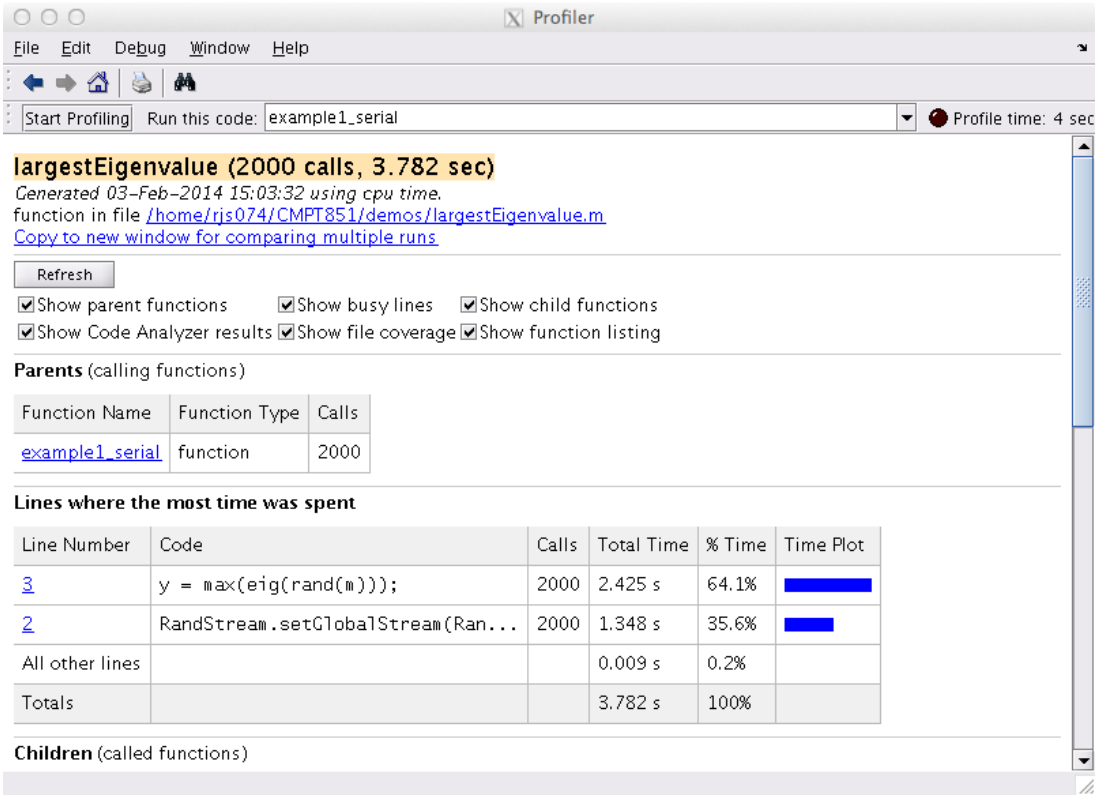
Memory and performance profiling



This shows that the majority of the work is done by `largestEigenvalue` and not some external function.

Memory and performance profiling

After clicking on `largestEigenvalue`, the profiling of the code itself is performed.



This tells us which line dominates the CPU requirements, i.e., line 3, `max(eig(rand(m)))`.

Memory and performance profiling

To profile the memory, we can type

```
whos
```

at the MATLAB command prompt.

This gives output

```
  Name        Size            Bytes  Class  Attributes

 ans         2000x1          16000  double
```

There is not much to this because only the output array is returned.

All other temporary variables have been recycled.

Memory and performance profiling

A more detailed study of memory allocation could be achieved by declaring `persistent` variables.

`persistent` variables are variables that are known only in the function in which they are declared, but their values are stored in memory between function calls.

This is similar to but a more elegant solution than declaring `global` variables, which would be known to (and hence could be changed by) all other functions.

However, permanent storage of the variables occurs in either case.

Back to the example

If we recall the serial code for this example, we quickly see that only 3 lines[3] need to be changed:

```
function a = example1_parallel()
% find the largest eigenvalue of N random matrices of size m

N = 2000;        % number of trials
m = 50;          % size of random matrices
p = 2;           % number of workers
a = zeros(N,1); % pre-allocate vector of zeros to store results
seeds = (1:N);

matlabpool('open', 'local', p)
tic; % parfor-loop
parfor I = 1:N
    a(I) = largestEigenvalue(m,seeds(I));
end
t = toc;
matlabpool('close')

disp(['Time for serial calculation: ', num2str(t), ' seconds.'])
```

[3] In fact it is only 1 line if you open and close the MATLAB pool from the command line. But this is not recommended!

Back to the example

Although there should be little doubt in this case, it is usually a good idea to verify that the results produced by the serial and parallel codes are "equivalent".

We can execute a simple script to do this:

```
a1 = example1_serial();
a2 = example1_parallel();

% Compare results
if isequal(a1, a2)
    disp('The results are the same.')
end
```

Output from this script run on `socrates` is

```
>> example1_compare
Time for serial calculation: 3.5154 seconds.
Starting matlabpool using the 'local' profile ... connected to 2 workers.
Sending a stop signal to all the workers ... stopped.
Time for parallel calculation: 2.3152 seconds.
The results are the same.
```

Batch jobs in MATLAB

One drawback of the interactive mode of parallel computing in MATLAB is the requirement of having an active session while the computation runs.

The technical reason for this is the requirement of having an open `matlabpool`.

As with classical HPC environments, it is possible to submit a script to be run via a *batch* command.

If the following script is called `example1_batch.m`,

```
N = 2000;
m = 50;
a = zeros(N,1);
seeds = (1:N);

parfor I = 1:N
    a(I) = largestEigenvalue(m,seeds(I));
end
```

the it can be batched using the command

```
job = batch('example1_batch');
```

Batch jobs in MATLAB

The job will then (eventually) be run in the "background" using the default scheduler, but control is returned to the command window in the meantime.

The *state* of the job can be monitored, e.g., to see when it completes.

```
done = findJob(job,'State','completed')
```

There is also a Job Monitor GUI, which can be started from the MATLAB desktop by clicking `Parallel` then `Monitor Jobs`.

When a job is completed, the variables can be extracted and used in post-processing.

```
vars = load(job, 'a');
hist(vars.a) % plot histogram of results
```

A note on paths

The workers must have access to all the files necessary to perform their tasks; this includes the driver code.

Files (or entire directories containing the files) that must be sent from the client to the workers can be specified via

```
batch(job, 'AttachedFiles', {'file1.m', 'file2.m'})
```

Adding directories to the path for workers can be specified via, e.g.,

```
batch(job, 'AdditionalPaths', '/home/nsid/')
```

MATLAB *parallel constructs:* `spmd`

We have seen the MATLAB parallel constructs `parfor`, `CreateJob`, and `CreateTask`.

The remaining MATLAB parallel constructs are `spmd`, `CreateCommunicatingJob`, and `CreateTask`.

Not surprisingly, `spmd` in this context also stands for *single program multiple data*.

The default way to use `spmd` is via

```
spmd
   <statements>
end
```

Perhaps more accurately, `spmd` defines a *block* of statements to be executed in SPMD style.

Naturally, in order for this to work, a `matlabpool` must be open before entering the `spmd` block and remain open until exiting it.

This form of the command executes the statements in parallel on all *workers* in the `matlabpool`.

spmd

A precise number of workers to be used is specified via

```
spmd (n)
   <statements>
end
```

where `n` cannot exceed the number of open workers in the `matlabpool`.

If the pool is large enough but there are insufficient available workers, the statements will not execute until sufficiently many are available.

It is possible to set `n = 0`, in which case the statements execute on the local client (in serial), i.e., as if there was no open `matlabpool`.

A range of workers on which to execute a block of statements can be specified via

```
spmd (m,n)
   <statements>
end
```

where `m` is the minimum number of workers required and `n` is the maximum number of workers to be used.

spmd

Each worker used in an `spmd` block has a unique identifier called its *labindex*, equivalent to the concept of *rank*.

So, standard SPMD techniques can be used to customize program execution, e.g., to create differently sized random matrices on different workers, we can use

```
spmd (3)
  if (labindex == 1),
    R = rand(1);
  else
    R = rand(2);
  end
end
```

spmd

with output (something like)

```
Lab 1:

  R =

       0.9173

Lab 2:

  R =

       0.2951    0.3277
       0.0990    0.6902

Lab 3:

  R =

       0.3527    0.3007
       0.9411    0.4783
```

spmd

The workers that are executing an `spmd` block operate simultaneously and are aware of each other.

As on a normal distributed-memory system, the programmer can directly control communications between the workers, transfer data between them, and use *co-distributed arrays* among them.

Within an `spmd` block, all command-line (textual) worker output displays in the client Command Window.

The workers are MATLAB sessions initiated without graphical display; hence graphical output from workers is not displayed.

MATLAB *interlab communication*

We have seen `labindex` with the `spmd` construct.

Other constructs for interlab communication include

- `numlabs`. Number of labs working on current job.
- `psave`. Save data from each lab workspace to file.
- `pload`. Load data from a previous `psave`.
- `labBarrier`. Pause until all labs reach this call.
- `labBroadcast`. Send/receive data from all labs.
- `labSend`. Send data to another lab.
- `labReceive`. Receive data from another lab.
- `labSendReceive`. Simultaneous data exchange.
- `labProbe`. Test if other lab is ready to receive.

MATLAB *interlab communication*

There is also a series of global operators:

- `gplus`.

  `S = gplus(X)`

  adds up the local values of the array `X`, returns the result in `S`, and then broadcasts `S` to all labs.

  The variant `S = gplus(X,targetLab)` stores `S` only on `targetLab`; other labs receive [].

  For example, with four labs, `S = gplus(labindex)` returns `S=10` on all workers.

MATLAB *interlab communication*

- `gcat`.

  `XX = gcat(X)`

  concatenates the local values of the array `X` (by default along the second dimension ("columns")), returns the result in the array `XX`, and then broadcasts `XX` to all workers.

  The variants `XX = gcat(X, dim)` performs the concatenation over dimension `dim` and `XX = gcat(X, dim, targetLab)` stores `XX` only on `targetLab`; other workers receive [].

  For example, with four workers, `XX = gcat(labindex)` returns `XX=[1 2 3 4]` on all workers.

MATLAB *interlab communication*

- gop. This is the general command for distributing a global operation across workers:

  ```
  result = gop(@f, x)
  ```

  reduces the local value `x` on each lab via the function `f`, returns the result in `result`, and then broadcasts `result` to all workers.

  The function `f` must accept exactly two arguments and return one argument, with all three arguments being of the same type.

  This is because the function is used iteratively as

  ```
  f(f(x1,x2),f(x3,x4))
  ```

  The function `f` must also be *associative*, i.e.,

  ```
  f(f(x1,x2),x3) = f(x1,f(x2,fx3))
  ```

The variant `result = gop(@f,x,targetLab)` stores `result` only on `targetLab`; other workers receive [].

For example, with four workers, the command

```
result = gop(@plus,x)
```

sums all lab values of `x` and the command

```
result = gop(@max,x)
```

returns the maximum value of `x` from all the workers.

The command

```
result = gop(@(a1,a2)norm([a1 a2]),x)
```

returns the 2-norm of `x` from all the workers.

Summary

- Parallel computing in MATLAB
- Functional parallelism; data parallelism
- Inter-worker communication