Parallel Processing Tool-box

- Start up MATLAB in the regular way. This copy of MATLAB that you start with is called the "client" copy; the copies of MATLAB that will be created to assist in the computation are known as "workers".

- The process of running your program in parallel now requires three steps:
  1. Request a number of workers;
  2. Issue the normal command to run the program. The client program will call on the workers as needed;
  3. Release the workers;

- For e.g. Supposing that your compute node has 8 cores, and that your M file is named "mainprogram.m", the command you might actually issue could look like this:

  ```matlab
  >> matlabpool open local 4
  Starting matlabpool using the parallel configuration 'local'.
  Waiting for parallel job to start...
  Connected to a matlabpool session with 4 labs.
  >> mainprogram.m
  >> matlabpool close
  Sending a stop signal to all the labs...
  Waiting for parallel job to finish...
  Performing parallel job cleanup...
  Done.
  ```

- The number of workers you request can be any value from 0 up to 4
**matlabpool**

- Open or close pool of MATLAB sessions for parallel computation, enables the parallel language features in the MATLAB language (e.g., `parfor`) by starting a parallel job that connects this MATLAB client with a number of labs.

**Ex:** Start a pool of 4 labs using a configuration called `myConf`:

```
matlabpool open myConf 4
```

**pctRunOnAll**

- Run command on client and all workers in `matlabpool`

**Ex:**

- Clear all loaded functions on all labs:
  
  ```matlab
  pctRunOnAll clear functions
  ```

- Change the directory on all workers to the project directory:
  
  ```matlab
  pctRunOnAll cd /opt/projects/new_scheme
  ```

- Add some directories to the paths of all the labs:
  
  ```matlab
  pctRunOnAll ('addpath /usr/share/path1');
  ```
The simplest way of parallelizing a MATLAB program focuses on the for loops in the program.

Q: Can the iterations of the loop be performed in any order without affecting the results? If the answer is "yes", then generally the loop can be parallelized.

If you have nested for loops, then generally it is not useful to replace these by nested parfor loops.
- If the outer loop can be parallelized, then that is the one that should be controlled by a parfor.
- If the outer loop cannot be parallelized, then you are free to try to parallelize some of the inner for loops.

The safest assumption about a parfor-loop is that each iteration of the loop is evaluated by a different MATLAB worker. If you have a for-loop in which all iterations are completely independent of each other, this loop is a good candidate for a parfor-loop.
The next example, which attempts to compute Fibonacci numbers, is not a valid `parfor`-loop because the value of an element of \( f \) in one iteration depends on the values of other elements of \( f \) calculated in other iterations.

```matlab
>> matlabpool open local 4
Destroying 1 pre-existing parallel job(s) created by matlabpool that were in the finished or failed state.

Starting matlabpool using the parallel configuration 'local'.
Waiting for parallel job to start...
Connected to a matlabpool session with 4 labs.
>> f=zeros(1,50);
>> f(1)=1;f(2)=2;
>> parfor n=3:50
   f(n) = f(n-1) + f(n-2);
   end
??? Error: The variable f in a parfor cannot be classified.
See Parallel for Loops in MATLAB, "Overview".
```

The body of a `parfor`-loop must be transparent, meaning that all references to variables must be "visible" (i.e., they occur in the text of the program).

In the following example, because \( X \) is not visible as an input variable in the `parfor` body (only the string 'X' is passed to `eval`), it does not get transferred to the workers. As a result, MATLAB issues an error at run time:
‘parfor’ Limitations

Nested spmd Statements

- The body of a parfor-loop cannot contain an spmd statement, and an spmd statement cannot contain a parfor-loop.

Break and Return Statements

- The body of a parfor-loop cannot contain `break` or `return` statements.

Global and Persistent Variables

- The body of a parfor-loop cannot contain `global` or `persistent` variable declarations.

Handle Classes

- Changes made to handle classes on the workers during loop iterations are not automatically propagated to the client.

P-Code Scripts

- You can call P-code script files from within a parfor-loop, but P-code script cannot contain a parfor-loop.

- For more details about parfor limitation please refer:
  
  http://www.mathworks.com/help/toolbox/distcomp/bq9u0a2.html
Sample `parfor` Program in MATLAB

```matlab
matlabpool open local 4;
ms.UseParallel='always';
pctRunOnAll ('addpath /home/Proposed');
N = 100; % number of iterations

parfor i=1:1:N
    fprintf('-----------------------------------------------------
');
    fprintf('-----------------------------------------------------
');
    fprintf('Iteration Order no = %2d\n',i);
    fprintf('-----------------------------------------------------
');
    fprintf('-----------------------------------------------------
');
end
matlabpool close
```
The `SPMD` command allows a programmer to set up parallel computations that require more user control than the simple `parfor` command.

MATLAB executes the `spmd` body denoted by statements on several MATLAB workers simultaneously.

Inside the body of the `spmd` statement, each MATLAB worker has a unique value of `labindex`, while `numlabs` denotes the total number of workers executing the block in parallel.

Within the body of the `spmd` statement, communication functions for parallel jobs (such as `labSend` and `labReceive`) can transfer data between the workers.

Values returning from the body of an `spmd` statement are converted to `Composite` objects on the MATLAB client.

A Composite object contains references to the values stored on the remote MATLAB workers, and those values can be retrieved using cell-array indexing. The actual data on the workers remains available on the workers for subsequent `spmd` execution, so long as the Composite exists on the client and the MATLAB pool remains open.
Using ‘\texttt{spmd}’ for Parallel Programs

- Parallel sections of the code begin with the \texttt{spmd} statement, and end with an \texttt{end} statement. The computations in these blocks occur on the MATLAB workers. The client sits idly and "watches".

- Each worker has access to the variable \texttt{numlabs}, which contains the number of workers. Each worker has a unique value of the variable \texttt{labindex}, between 1 and \texttt{numlabs}.

- Any variable defined by the client is "visible" to the workers and can be used on the RHS of eqns within the \texttt{spmd} blocks.

- Any variable defined by the workers is a "\textit{composite}" variable. If a variable called \texttt{X} is defined by the workers, then each worker has its own value, and the set of values is accessible by the client, using the worker's index. Thus \texttt{X\{1\}} is the value of \texttt{X} computed by worker 1.

- A program can have several \texttt{spmd} blocks. If the program completes an \texttt{spmd} block, carries out some commands in the client program, and then enters another \texttt{spmd} block, then all the variables defined during the previous \texttt{spmd} block still exist.
Using ‘spmd’ for Parallel Programs

- Workers cannot directly see each other's variables. Communication from one worker to another can be done through the client.

- However, a limited number of special operators are available, that can be used within spmd blocks, which combine variables. In particular, the command `gplus` sums the values of a variable that exists on all the workers, and returns to each worker the value of that sum.

**When to Use spmd**

- The "single program" aspect of spmd means that the identical code runs on multiple labs. When the spmd block is complete, your program continues running in the client.

- The "multiple data" aspect means that even though the spmd statement runs identical code on all labs, each lab can have different, unique data for that code. So multiple data sets can be accommodated by multiple labs.

- Typical applications appropriate for spmd are those that require running simultaneous execution of a program on multiple data sets, when communication or synchronization is required between the labs. Some common cases are:
  - **Programs that take a long time to execute** — spmd lets several labs compute solutions simultaneously.
  - **Programs operating on large data sets** — spmd lets the data be distributed to multiple labs.
Using ‘spmd’ for Parallel Programs

Displaying Output

- When running an spmd statement on a MATLAB pool, all command-line output from the workers displays in the client Command Window. Because the workers are MATLAB sessions without displays, any graphical output (for example, figure windows) from the pool does not display at all.

Creating Composites Outside spmd Statements

- The Composite function creates Composite objects without using an spmd statement. This might be useful to prepopulate values of variables on labs before an spmd statement begins executing on those labs. Assume a MATLAB pool is already open:

  \[ PP = \text{Composite}( ) ; \]

- By default, this creates a Composite with an element for each lab in the MATLAB pool. You can also create Composites on only a subset of the labs in the pool. The elements of the Composite can now be set as usual on the client, or as variables inside an spmd statement.

- For details about accessing data with composites for spmd please see:
  

- For details about Distributing arrays, co-distributed arrays and distributed arrays please refer:
  
  http://www.mathworks.com/help/toolbox/distcomp/br9_n7w-1.html
Composite

- Creates Composite object

- **Syntax:**
  
  \[
  C = \text{Composite}()
  \]
  \[
  C = \text{Composite}(\text{nlabs})
  \]

  - \( C = \text{Composite}() \) creates a Composite object on the client using labs from the MATLAB pool. Generally, you should construct Composite objects outside any `spmd` statement.

  - \( C = \text{Composite}(\text{nlabs}) \) creates a Composite object on the parallel resource set that matches the specified constraint. \( \text{nlabs} \) must be a vector of length 1 or 2, containing integers or Inf.

- A Composite object has one entry for each lab; initially each entry contains no data. Use either indexing or an `spmd` block to define values for the entries.

**Examples**

- Create a Composite object with no defined entries, then assign its values:

  ```matlab
  c = Composite(); % One element per lab in the pool
  for ii = 1:length(c) % Set the entry for each lab to zero
    c{ii} = 0;
  end % Value stored on each lab
  ```
Create distributed array from data in client workspace

Syntax: \( D = \text{distributed}(X) \)

\( D = \text{distributed}(X) \) creates a distributed array from \( X \). \( X \) is an array stored on the MATLAB client, and \( D \) is a distributed array stored in parts on the workers of the open MATLAB pool.

Examples

Create a small array and distribute it:

\[
\begin{align*}
\text{Nsmall} &= 50; \\
\text{D1} &= \text{distributed}(	ext{magic(Nsmall)});
\end{align*}
\]

Create a large distributed array using a static build method:

\[
\begin{align*}
\text{Nlarge} &= 1000; \\
\text{D2} &= \text{distributed}.\text{rand}(\text{Nlarge});
\end{align*}
\]
USING ‘spmd’ FOR PARALLEL PROGRAMS

```matlab
matlabpool open local 4;
ms.UseParallel='always';
pctRunOnAll('addpath /home/aliakber');

if labindex==1
    R=rand(3,3);
else
    R=rand(4,4);
end

matlabpool close
```

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

1
R =
0.8147 0.9134 0.2785
0.9058 0.6324 0.5469
0.1270 0.0975 0.9575

2
R =
0.9133 0.4624 0.2155 0.4621
0.6823 0.1562 0.4978 0.9846
0.8661 0.4526 0.2934 0.9587
0.4809 0.9009 0.5071 0.5795

3
R =
0.2951 0.7010 0.9143 0.7375
0.9909 0.3321 0.2740 0.5407
0.3577 0.9622 0.6484 0.6348
0.6902 0.7780 0.2781 0.0948

4
R =
0.3527 0.5647 0.0360 0.8565
0.9411 0.8864 0.4363 0.6978
0.3007 0.9689 0.7699 0.7676
0.4783 0.4288 0.3194 0.3136

Sending a stop signal to all the labs...
Waiting for parallel job to finish...
Performing parallel job cleanup...
Done.
```
How to Measure and Report Elapsed Time

- You can use the `tic` and `toc` functions to begin and end timing.

- The call to `toc` returns the number of seconds elapsed since `tic` was called.

- Here is an example of the use of both `tic` and `toc` when measuring performance of a parallel computation.

```matlab
>> tic;
>> parfor n=3:50
   f(n) = f(n) + 2;
end
>> toc
Elapsed time is 16.239578 seconds.
```
One MATLABPOOL, Many Uses

- `spmd ... end`
  - A MATLAB block
  - Data Parallel (Worker communication)

- `parfor ... end`
  - A MATLAB block
  - Task Parallel (No worker communication)

- Built-in parallelism in some toolboxes

- MATLAB manages data transfer between workers and desktop

- Use the same pool of MATLAB workers

- Can exist along with serial code in the same program
  - Execute interactively from MATLAB command line
  - Execute off-line using `batch` (scripts) or `createMATLABpooljob` (functions)

- All run serially if no workers are available
Following are the types of functions available in parallel tool box

| 1. | Parallel Code Execution | Constructs for automatically running code in parallel |
| 2. | Distributed and Codistributed Arrays | Data partitioned across multiple MATLAB sessions |
| 3. | Jobs and Tasks | Parallel computation through individual tasks |
| 4. | Interlab Communication Within a Parallel Job | Communications between labs during job execution |
| 5. | Graphics Processing Unit | Transferring data and running code on the GPU |
| 6. | Utilities | Utilities for using Parallel Computing Toolbox |

For more details:

Parallel Computing Toolbox 4.0

- Perform parallel computations on multicore computers and computer clusters

Parallel Computing Toolbox lets you solve computationally and data-intensive problems using MATLAB® and Simulink® on multicore and multiprocessor computers. Parallel processing constructs such as parallel for-loops and code blocks, distributed arrays, parallel numerical algorithms, and message-passing functions let you implement task- and data-parallel algorithms in MATLAB at a high level without programming for specific hardware and network architectures.

As a result, converting serial MATLAB applications to parallel MATLAB applications requires few code modifications and no programming in a low-level language. You can run your applications interactively or offline, in batch environments.

You can use the toolbox to execute applications on a single multicore or multiprocessor desktop. Without changing the code, you can run the same application on a computer cluster (using MATLAB Distributed Computing Server™). Parallel MATLAB applications can be distributed as executables or shared libraries (built using MATLAB Compiler™) that can access MATLAB Distributed Computing Server.

- View data sheet (1883k)
- FREE Product Technical Kit
FOR FURTHER ASSISTANCE:

Please Contact:

hpc@kfupm.edu.sa,

Or visit:

http://hpc.kfupm.edu.sa