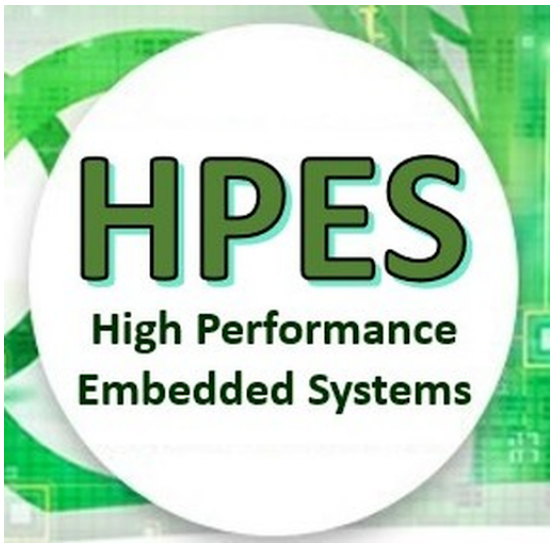


**EEE4120F**

## High Performance Embedded Systems

# REFERENCE GUIDE

MATLAB Parallel Computing Toolkit(PCT)

### Course Team

**Course Convener:** Associate Professor Simon Winberg

**Teaching Assistant:** Travimadox Webb

University of Cape Town 2026

# MATLAB Parallel Computing Toolkit

A Practical Guide for Practical 2

## 1 Introduction

The MATLAB Parallel Computing Toolbox (PCT) enables you to leverage multicore processors, GPUs, and computer clusters to accelerate computations and handle large datasets efficiently. Parallel computing involves breaking down problems into smaller parts that can be solved simultaneously. When this decomposition is straightforward, we call it **"embarrassingly parallel"**.

**Tip**

**Installation:** Ensure you have MATLAB installed with the Parallel Computing Toolbox add-on. This usually requires a separate installation via the "Add-Ons" menu.

### 1.1 Parallel Workers

**Parallel Workers** are computational engines (or threads) that execute code in parallel. They are the fundamental processing units in PCT and are essentially headless MATLAB instances running in the background.

## 2 Converting Serial Code to Parallel Code

The simplest way to parallelise MATLAB code is replacing `for` loops with `parfor` (parallel for) loops. MATLAB intelligently analyses the loop body, checking data dependencies to ensure correct parallel execution without breaking dependencies or overwriting data mid-process.

```
% Serial code
for i = 1:1000
    A(i) = computeIntensiveTask(i);
end

% Parallel code - automatically parallelised
parfor i = 1:1000
    A(i) = computeIntensiveTask(i);
end
```

## 3 Parallel Performance

Parallel computing doesn't guarantee linear speedup. Several factors affect performance:

- **Overhead:** Starting workers and transferring data takes time
- **Amdahl's Law:** Speedup limited by serial portions of code
- **Load Balancing:** Unequal iteration times reduce efficiency

**Tip**

**Performance Tip:** Use parallel computing when the benfits outweigh the overhead costs. *Do not include pool startup time in benchmarking!*

# 4 Parallel Pools and MATLAB Workers

A parallel pool is a collection of MATLAB workers on your local machine or cluster. See example below:

```
% Start a parallel pool with N workers on your local machine
parpool('local', N)

% Check if a pool exists (without creating one)
poolobj = gcp('nocreate');

% Create a pool only if one doesn't exist
if isempty(poolobj)
    parpool('local', N);
end

% Alternative: Using parcluster for more control
c = parcluster;
p = c.parpool(4);  % Creates pool with 4 workers

% Delete the current pool
delete(gcp('nocreate'))
% or
p.delete  % If you created pool with parcluster
```

**Tip**

**Best Practice:** MATLAB automatically creates a pool when you first use `parfor`. Use `gcp('nocreate')` to work with existing pools and avoid unnecessary overhead.

# 5 Understanding the Rules of parfor

## 5.1 Parallel Execution Order

Unlike regular for-loops, **parfor iterations execute in random order**. Each iteration must be independent. Note that you cannot query which worker thread is executing a particular iteration within `parfor`.

## 5.2 SPMD Blocks: Single Program Multiple Data

**SPMD** stands for "Single Program Multiple Data" i.e the same code runs on different workers but operates on different data. This is similar to how GPU cores operate in lockstep.

```
% SPMD example - all workers run the same code
spmd
    fprintf('Worker %d says Hello World!\n', labindex);
    % labindex tells you which worker number (1 = master thread)
    % Note: spmdIndex is preferred in MATLAB 2022+
end
```

Within `spmd` blocks, use `labindex` (or `spmdIndex` in MATLAB 2022+) to identify which worker is executing. If `labindex` returns 1, you're in the master thread.

### 5.3 Non-Parallelisable Code

Code that cannot be parallelised with `parfor` includes:

- Iterations that depend on previous iterations
- Code that modifies the same variable in unpredictable ways
- Nested `parfor` loops (outer loop can be `parfor`, inner must be `for`)

**Warning**

**Invalid Example:** Each iteration depends on the previous one

```
parfor i = 2:n
    A(i) = A(i-1) + B(i);  % Error!
end
```

## 6 Types of Variables in parfor-Loops

### 6.1 Loop Index Variables

The loop index (e.g., `i` in `parfor i = 1:n`) is automatically managed and each worker gets its assigned values.

### 6.2 Reduction Operations

Reduction variables accumulate values across iterations using associative operations (+, *, min, max, etc.).

```
total = 0;
parfor i = 1:1000
    total = total + computeValue(i);  % Valid reduction
end
```

### 6.3 Sliced and Broadcast Variables

**Sliced variables:** Arrays where each iteration accesses different elements in a predictable pattern.

```
A = zeros(100,1);
parfor i = 1:100
    A(i) = i^2;  % A is sliced (each worker gets different indices)
end
```

**Broadcast variables:** Read-only variables sent to all workers. Large broadcast variables reduce performance.

### 6.4 Distributed Arrays

Arrays can be automatically distributed across multiple workers for processing and then gathered back to the master thread.

```
% Distribute array A across workers
D = distributed(A);  % D is distributed version accessible by workers

% Example with SPMD and distributed data
p = parpool('Processes', 4);
spmd
    C = rand(3, labindex-1);  % Each worker creates different size array
end

% View how data is distributed (each lab has different portion)
C  % Shows: Lab 1: [3x0], Lab 2: [3x1], Lab 3: [3x2], Lab 4: [3x3]

% Gather distributed data back to master thread
finalResult = gather(C);  % Consolidates all data into single array

delete(p);
```

**Tip**

**Key Functions:** Use `distributed()` to split data across workers, and `gather()` (or `gplus`/`spmdPlus`) to collect results back to the master thread.

### 6.5 Limit the Size of Broadcast Variables

**Tip**

**Optimisation:** Extract only needed data before the `parfor` loop to minimize broadcast size.

```
% Instead of broadcasting entire structure:
params = largeStruct.neededParams;
parfor i = 1:n
    result(i) = compute(params, i);
end
```

### 6.6 Improve Efficiency with Sliced and Temporary Variables

Use temporary variables inside the loop to reduce classification complexity and improve performance.

```
parfor i = 1:n
    temp = expensiveComputation(data(i));
    result(i) = processTemp(temp);
end
```

## 7 Nested Loops with Shared Memory

Only the outermost loop can be a `parfor`. Inner loops must remain serial.

```
% Parallelize outer loop
parfor i = 1:m
```

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

**Tip**

**Performance Strategy:** Parallelise the loop with the most iterations or longest execution time. Test both configurations to find the optimal approach.

# 8 GPU Computing

MATLAB can utilise GPUs for certain computations and may do so automatically for various functions, particularly when data size and processing type merit GPU use.

## 8.1 Checking GPU Availability

To verify if your GPU is accessible to MATLAB, call `gpuArray` without arguments:

```
>> gpuArray
ans = [] % Empty array indicates GPU is working
```

If you see an error indicating no GPU available or drivers not found, and you have a compatible GPU (e.g., not too ancient NVIDIA card), you may need to update your graphics drivers, reboot, and restart MATLAB.

You can also check the number of available GPUs:

```
>> gpuDeviceCount
ans = 1
```

## 8.2 Basic GPU Workflow

The typical approach involves creating/loading data on the CPU, transferring to GPU, performing computations, then gathering results back.

```
% Create an array on CPU
X = [1, 2, 3];

% Transfer data to GPU
G = gpuArray(X);

% Verify data is on GPU
isgpuarray(G)  % Returns logical 1 (true)

% Perform element-wise computation (SPMD operation on GPU)
GSq = G.^2;    % GPU computes what needs to be done

% Transfer result back to CPU
XSq = gather(GSq)  % Gathers distributed data blocks
% Result: XSq = [1, 4, 9]

% Verify data is back on CPU
isgpuarray(XSq)  % Returns logical 0 (false)
```

**Tip**

**Key Point:** You do NOT need to put GPU code inside an `spmd` block. The `gpuArray` function handles data transfer, and operations on GPU arrays automatically execute on the GPU. The GPU inherently operates in SPMD fashion across its many cores.

**Tip**

**When to use GPU:** Matrix operations, element-wise operations, and algorithms with high arithmetic intensity. GPUs excel at operations on large arrays but have overhead for small data.

**Warning**

**Note:** GPU computing requires a compatible NVIDIA GPU and the Parallel Computing Toolbox.

## 9 Practice & Next Steps

For more information, see: MATLAB Quick Start Tutorial.