

Programming Tips

In this section...

“Program Development Guidelines” on page 6-26
 “Current Working Directory of a MATLAB Worker” on page 6-27
 “Writing to Files from Workers” on page 6-28
 “Saving or Sending Objects” on page 6-28
 “Using clear functions” on page 6-29
 “Running Tasks That Call Simulink Software” on page 6-29
 “Using the pause Function” on page 6-29
 “Transmitting Large Amounts of Data” on page 6-29
 “Interrupting a Job” on page 6-30
 “Speeding Up a Job” on page 6-30

Program Development Guidelines

When writing code for Parallel Computing Toolbox software, you should advance one step at a time in the complexity of your application. Verifying your program at each step prevents your having to debug several potential problems simultaneously. If you run into any problems at any step along the way, back up to the previous step and reverify your code.

The recommended programming practice for distributed or parallel computing applications is

- 1 Run code normally on your local machine.** First verify all your functions so that as you progress, you are not trying to debug the functions and the distribution at the same time. Run your functions in a single instance of MATLAB software on your local computer. For programming suggestions, see “Techniques for Improving Performance” in the MATLAB documentation.
- 2 Decide whether you need an independent or communicating job.** If your application involves large data sets on which you need simultaneous calculations performed, you might benefit from a communicating job

with distributed arrays. If your application involves looped or repetitive calculations that can be performed independently of each other, an independent job might be appropriate.

- 3 Modify your code for division.** Decide how you want your code divided. For an independent job, determine how best to divide it into tasks; for example, each iteration of a for-loop might define one task. For a communicating job, determine how best to take advantage of parallel processing; for example, a large array can be distributed across all your workers.
- 4 Use pmode to develop parallel functionality.** Use pmode with the local scheduler to develop your functions on several workers in parallel. As you progress and use pmode on the remote cluster, that might be all you need to complete your work.
- 5 Run the independent or communicating job with a local scheduler.** Create an independent or communicating job, and run the job using the local scheduler with several local workers. This verifies that your code is correctly set up for batch execution, and in the case of an independent job, that its computations are properly divided into tasks.
- 6 Run the independent job on only one cluster node.** Run your independent job with one task to verify that remote distribution is working between your client and the cluster, and to verify proper transfer of additional files and paths.
- 7 Run the independent or communicating job on multiple cluster nodes.** Scale up your job to include as many tasks as you need for an independent job, or as many workers as you need for a communicating job.

Note The client session of MATLAB must be running the Java® Virtual Machine (JVM™) to use Parallel Computing Toolbox software. Do not start MATLAB with the -nojvm flag.

Current Working Directory of a MATLAB Worker

The current directory of a MATLAB worker at the beginning of its session is

```
CHECKPOINTBASE\HOSTNAME_WORKERNAME_mlworker_log\work
```

where CHECKPOINTBASE is defined in the `mdce_def` file, HOSTNAME is the name of the node on which the worker is running, and WORKERNAME is the name of the MATLAB worker session.

For example, if the worker named `worker22` is running on host `nodeA52`, and its CHECKPOINTBASE value is `C:\TEMP\MDCE\Checkpoint`, the starting current directory for that worker session is

```
C:\TEMP\MDCE\Checkpoint\nodeA52_worker22_mlworker_log\work
```

Writing to Files from Workers

When multiple workers attempt to write to the same file, you might end up with a race condition, clash, or one worker might overwrite the data from another worker. This might be likely to occur when:

- There is more than one worker per machine, and they attempt to write to the same file.
- The workers have a shared file system, and use the same path to identify a file for writing.

In some cases an error can result, but sometimes the overwriting can occur without error. To avoid an issue, be sure that each worker or `parfor` iteration has unique access to any files it writes or saves data to. There is no problem when multiple workers read from the same file.

Saving or Sending Objects

Do not use the `save` or `load` function on Parallel Computing Toolbox objects. Some of the information that these objects require is stored in the MATLAB session persistent memory and would not be saved to a file.

Similarly, you cannot send a parallel computing object between parallel computing processes by means of an object's properties. For example, you cannot pass an MJS, job, task, or worker object to MATLAB workers as part of a job's `JobData` property.

Also, system objects (e.g., Java classes, .NET classes, shared libraries, etc.) that are loaded, imported, or added to the Java search path in the MATLAB client, are not available on the workers unless explicitly loaded, imported, or added on the workers, respectively. Other than in the task function code, typical ways of loading these objects might be in `taskStartup`, `jobStartup`, and in the case of workers in a MATLAB pool, in `poolStartup` and using `pctRunOnAll`.

Using clear functions

Executing

```
clear functions
```

`clears` all Parallel Computing Toolbox objects from the current MATLAB session. They still remain in the MJS. For information on recreating these objects in the client session, see “Recover Objects” on page 7-15.

Running Tasks That Call Simulink Software

The first task that runs on a worker session that uses Simulink software can take a long time to run, as Simulink is not automatically started at the beginning of the worker session. Instead, Simulink starts up when first called. Subsequent tasks on that worker session will run faster, unless the worker is restarted between tasks.

Using the pause Function

On worker sessions running on Macintosh or UNIX operating systems, `pause(inf)` returns immediately, rather than pausing. This is to prevent a worker session from hanging when an interrupt is not possible.

Transmitting Large Amounts of Data

Operations that involve transmitting many objects or large amounts of data over the network can take a long time. For example, getting a job’s `Tasks` property or the results from all of a job’s tasks can take a long time if the job contains many tasks. See also “Object Data Size Limitations” on page 6-48.

Interrupting a Job

Because jobs and tasks are run outside the client session, you cannot use **Ctrl+C** (^C) in the client session to interrupt them. To control or interrupt the execution of jobs and tasks, use such functions as `cancel`, `delete`, `demote`, `promote`, `pause`, and `resume`.

Speeding Up a Job

You might find that your code runs slower on multiple workers than it does on one desktop computer. This can occur when task startup and stop time is significant relative to the task run time. The most common mistake in this regard is to make the tasks too small, i.e., too fine-grained. Another common mistake is to send large amounts of input or output data with each task. In both of these cases, the time it takes to transfer data and initialize a task is far greater than the actual time it takes for the worker to evaluate the task function.