Multiprocessing package - torch.multiprocessing — PyTorch 1.9.0 documentation

https://pytorch.org/docs/stable/multiprocessing.html#module-torch.multiprocessing

None Wed Jul, 28 17:00

torch.multiprocessing is a wrapper around the native `multiprocessing` module. It registers custom reducers, that use shared memory to provide shared views on the same data in different processes. Once the tensor/storage is moved to shared_memory (see `share_memory_()`), it will be possible to send it to other processes without making any copies.

The API is 100% compatible with the original module - it's enough to change `import multiprocessing` to `import torch.multiprocessing` to have all the tensors sent through the queues or shared via other mechanisms, moved to shared memory.

Because of the similarity of APIs we do not document most of this package contents, and we recommend referring to very good docs of the original module.

Warning

If the main process exits abruptly (e.g. because of an incoming signal), Python's `multiprocessing` sometimes fails to clean up its children. It's a known caveat, so if you're seeing any resource leaks after interrupting the interpreter, it probably means that this has just happened to you.

Strategy management¶

`torch.multiprocessing.get_all_sharing_strategies`()[source]¶
Returns a set of sharing strategies supported on a current system.

`torch.multiprocessing.get_sharing_strategy`()[source]¶
Returns the current strategy for sharing CPU tensors.

`torch.multiprocessing.set_sharing_strategy`(*new_strategy*)[source]¶
Sets the strategy for sharing CPU tensors.

Parameters
new_strategy (*str*) – Name of the selected strategy. Should be one of the values returned by `get_all_sharing_strategies()`.

Sharing CUDA tensors¶

Sharing CUDA tensors between processes is supported only in Python 3, using a `spawn` or `forkserver` start methods.

Unlike CPU tensors, the sending process is required to keep the original tensor as long as the receiving process retains a copy of the tensor. The refcounting is implemented under the hood but requires users to follow the next best practices.

Warning

If the consumer process dies abnormally to a fatal signal, the shared tensor could be forever kept in memory as long as the sending process is running.

1. Release memory ASAP in the consumer.

```
## Good
x = queue.get()
# do somethings with x
del x
```

```
## Bad
x = queue.get()
# do somethings with x
# do everything else (producer have to keep x in memory)
```

2. Keep producer process running until all consumers exits. This will prevent the situation when the producer process releasing memory which is still in use by the consumer.

```
## producer
# send tensors, do something
event.wait()
```

```
## consumer
# receive tensors and use them
event.set()
```

1. Don't pass received tensors.

```
# not going to work
x = queue.get()
queue_2.put(x)
```

```
# you need to create a process-local copy
x = queue.get()
x_clone = x.clone()
queue_2.put(x_clone)
```

```
# putting and getting from the same queue in the same process will likely end up with segfault
queue.put(tensor)
x = queue.get()
```

Sharing strategies¶

This section provides a brief overview into how different sharing strategies work. Note that it applies only to CPU tensor - CUDA tensors will always use the CUDA API, as that's the only way they can be shared.

File descriptor - `file_descriptor` ¶

Note

This is the default strategy (except for macOS and OS X where it's not supported).

This strategy will use file descriptors as shared memory handles. Whenever a storage is moved to shared memory, a file descriptor obtained from `shm_open` is cached with the object, and when it's going to be sent to other processes, the file descriptor will be transferred (e.g. via UNIX sockets) to it. The receiver will also cache the file descriptor and `mmap` it, to obtain a shared view onto the storage data.

Note that if there will be a lot of tensors shared, this strategy will keep a large number of file descriptors open most of the time. If your system has low limits for the number of open file descriptors, and you can't raise them, you should use the `file_system` strategy.

File system - `file_system` ¶

This strategy will use file names given to `shm_open` to identify the shared memory regions. This has a benefit of not requiring the implementation to cache the file descriptors obtained from it, but at the same time is prone to shared memory leaks. The file can't be deleted right after its creation, because other processes need to access it to open their views. If the processes fatally crash, or are killed, and don't call the storage destructors, the files will remain in the system. This is very serious, because they keep using up the memory until the system is restarted, or they're freed manually.

To counter the problem of shared memory file leaks, `torch.multiprocessing` will spawn a daemon named `torch_shm_manager` that will isolate itself from the current process group, and will keep track of all shared memory allocations. Once all processes connected to it exit, it will wait a moment

to ensure there will be no new connections, and will iterate over all shared memory files allocated by the group. If it finds that any of them still exist, they will be deallocated. We've tested this method and it proved to be robust to various failures. Still, if your system has high enough limits, and `file_descriptor` is a supported strategy, we do not recommend switching to this one.

Spawning subprocesses¶

Note

Available for Python >= 3.4.

This depends on the `spawn` start method in Python's `multiprocessing` package.

Spawning a number of subprocesses to perform some function can be done by creating `Process` instances and calling `join` to wait for their completion. This approach works fine when dealing with a single subprocess but presents potential issues when dealing with multiple processes.

Namely, joining processes sequentially implies they will terminate sequentially. If they don't, and the first process does not terminate, the process termination will go unnoticed. Also, there are no native facilities for error propagation.

The `spawn` function below addresses these concerns and takes care of error propagation, out of order termination, and will actively terminate processes upon detecting an error in one of them.

`torch.multiprocessing.spawn`(*fn, args=(), nprocs=1, join=True, daemon=False, start_method='spawn'*)[source]¶
Spawns `nprocs` processes that run `fn` with `args`.

If one of the processes exits with a non-zero exit status, the remaining processes are killed and an exception is raised with the cause of termination. In the case an exception was caught in the child process, it is forwarded and its traceback is included in the exception raised in the parent process.

Parameters

- **fn** (*function*) –

  Function is called as the entrypoint of the spawned process. This function must be defined at the top level of a module so it can be pickled and spawned. This is a requirement imposed by multiprocessing.

  The function is called as `fn(i, *args)`, where `i` is the process index and `args` is the passed through tuple of arguments.

- **args** (*tuple*) – Arguments passed to `fn`.
- **nprocs** (*int*) – Number of processes to spawn.
- **join** (*bool*) – Perform a blocking join on all processes.
- **daemon** (*bool*) – The spawned processes' daemon flag. If set to True, daemonic processes will be created.
- **start_method** (*string*) – (deprecated) this method will always use `spawn` as the start method. To use a different start method use `start_processes()`.

Returns

None if `join` is `True`, `ProcessContext` if `join` is `False`

class `torch.multiprocessing.SpawnContext` [source]¶

Returned by `spawn()` when called with `join=False`.

`join`(*timeout=None*)¶

Tries to join one or more processes in this spawn context. If one of them exited with a non-zero exit status, this function kills the remaining processes and raises an exception with the cause of the first process exiting.

Returns `True` if all processes have been joined successfully, `False` if there are more processes that need to be joined.

Parameters

timeout (*float*) – Wait this long before giving up on waiting.