shmem4py

Release 1.0.0

Lisandro Dalcin

May 13, 2024

CONTENTS

1 Introduction **3**
- 1.1 OpenSHMEM . . . 3
- 1.2 shmem4py . . . 4
- 1.3 Symmetric variables . . . 4
- 1.4 Resources . . . 4
- 1.5 Acknowledgements . . . 5

2 Installation **7**
- 2.1 Requirements . . . 7
- 2.2 Containers . . . 7
- 2.3 Recommended versions . . . 7
- 2.4 Installing shmem4py . . . 8
- 2.5 Next steps . . . 8

3 Usage examples **9**
- 3.1 Hello world . . . 9
- 3.2 Get a remote value . . . 9
- 3.3 Broadcast an array from root to all PEs . . . 10
- 3.4 Approximate the value of Pi with reductions . . . 11
- 3.5 Collect the same number of elements from each PE . . . 12
- 3.6 Collect a different number of elements from each PE . . . 13
- 3.7 Atomic conditional swap on a remote data object . . . 13
- 3.8 Test if condition is met . . . 14
- 3.9 All to all communication . . . 15
- 3.10 Locking . . . 16

4 OpenSHMEM **19**
- 4.1 Version and Vendor Query . . . 19
- 4.2 Library Setup and Exit . . . 19
- 4.3 Accessibility Queries . . . 21
- 4.4 Memory Management . . . 22
- 4.5 Team Management . . . 26
- 4.6 Communication Management . . . 27
- 4.7 Remote Memory Access . . . 29
- 4.8 Atomic Memory Operations . . . 31
- 4.9 Signaling Operations . . . 40
- 4.10 Collective Operations . . . 43
- 4.11 Point-To-Point Synchronization . . . 49
- 4.12 Memory Ordering . . . 55
- 4.13 Distributed Locking . . . 56

4.14 Distributed Locking (Object-Oriented) . . . 57
4.15 Profiling Control . . . 57
4.16 Typing Support . . . 58

5 Indices and tables **59**

Index **61**

Abstract

This document describes the *OpenSHMEM for Python* package. *OpenSHMEM for Python* provides Python bindings for the *OpenSHMEM* standard, allowing Python applications to exploit multiple processors on workstations, clusters, and supercomputers using a *Partitioned Global Address Space* (PGAS) programming model.

CHAPTER

ONE

INTRODUCTION

1.1 OpenSHMEM

OpenSHMEM is a Partitioned Global Address Space (PGAS) programming model that provides low-latency, high-bandwidth communication for use in parallel applications. The OpenSHMEM project aims to standardize several implementations of the different SHMEM APIs.

OpenSHMEM programs follow a single program, multiple data (SPMD) style, where processing elements (PEs) perform computation on subdomains of the larger problem and communicate periodically to exchange information. The PEs all start at the same time, and they all run the same program. Typically, each PE performs computations on its own subdomain and communicates with other PEs to exchange information required for the next computation phase. OpenSHMEM is optimized for low-latency data transfers and supports one-sided communication, making it ideal for applications with irregular communication patterns involving small/medium-sized data transfers.

OpenSHMEM routines provide support for

- put operations - data transfer to a different PE
- get operations - data transfer from a different PE
- remote pointers - allow for direct references to data objects owned by another PE
- atomic memory operations - such as an atomic read-and-update operation, fetch-and-increment, on a remote or local data object
- barrier synchronization
- group synchronization
- data broadcast
- data reduction
- data collection
- distributed locking of critical regions
- data and process accessibility queries to other PEs

1.2 shmem4py

`shmem4py` is a Python wrapper for the OpenSHMEM API, and requires a working OpenSHMEM implementation installed. It is built using CFFI for Python-C interoperability, and uses NumPy arrays to represent data objects.

1.3 Symmetric variables

OpenSHMEM relies on the concept of symmetric variables. Those variables exist on all PEs and have the same size, type and relative address. Only symmetric variables can be accessed remotely by other PEs. In `shmem4py`, symmetric variables are allocated using routines such as *`alloc`* for raw memory allocations or *`array`*, *`empty`*, *`zeros`*, *`ones`*, and *`full`* for NumPy array allocations.

Tip: Python built-in data types such as `bool`, `int`, `float`, and `complex` are immutable, i.e., they cannot be modified after creation. As a consequence of this, `shmem4py` uses NumPy arrays to represent symmetric variables.

Warning:

Even though NumPy arrays are mutable, one has to be careful when addressing them, as the individual array elements are immutable:

```
>>> a = np.ones(2)
>>> a[0].flags.writeable
False
```

However, an array slice representing the same value returns a mutable array:

```
>>> a[0:1].flags.writeable
True
```

1.4 Resources

We do not aim to provide a comprehensive OpenSHMEM introduction in this documentation, focusing on the specifics of the Python bindings provided in `shmem4py`. For a more comprehensive introduction to OpenSHMEM, we refer to the following resources:

- OpenSHMEM.org
- OpenSHMEM.org Tutorials
- OpenSHMEM tutorial from the 2014 OpenSHMEM Workshop
- Parallel Research Kernels repository contains some C and Python OpenSHMEM examples

1.5 Acknowledgements

Our documentation is heavily-based on the OpenSHMEM 1.5 Specification. The Dockerfiles we use were initially based on Sandia OpenSHMEM's container specification. `shmem4py` relies on NumPy and CFFI.

CHAPTER

TWO

INSTALLATION

2.1 Requirements

A working OpenSHMEM implementation is required. Currently, Cray OpenSHMEMX, Open Source Software Solutions (OSSS) OpenSHMEM, Open MPI OpenSHMEM, OSHMPI, and Sandia OpenSHMEM are supported. Generally speaking, `shmem4py` will be installed using the OpenSHMEM implementation's `oshcc` wrapper found in the `$PATH`.

For an example setup of `shmem4py` using the OSHMPI/MPICH backend, see INSTALL.rst.

2.2 Containers

We encourage users to use Docker/Podman containers or follow the steps executed in the Dockerfiles. Containers based on those files are meant to show minimal configurations for building and running `shmem4py` with different OpenSHMEM implementations. Those images are used in GitHub Actions CI/CD and we consider them tested configurations. Currently, we test with OSSS OpenSHMEM, Open MPI OpenSHMEM, OSHMPI and Sandia OpenSHMEM on the latest releases of Fedora and Ubuntu.

2.3 Recommended versions

There exist many combinations of the operating system and software package versions that may work with `shmem4py`. We recommend to use the combinations which are tested in `shmem4py`'s CI/CD pipeline. As of 28/04/2023, the following package versions all work correctly:

OpenSHMEM distribution	Operating system	Required packages (package manager)	Required packages (make)	Dockerfile
Open MPI OpenSHMEM 4.1.4	Fedora 37	Python 3.11	UCX latest	osh-mem_fedora
	Ubuntu 22.04	Python 3.10	UCX latest	osh-mem_ubuntu
OSHMPI latest	Fedora 37	Python 3.11, MPICH 4.0		oshmpi_fedora
	Ubuntu 22.04	Python 3.10, MPICH 4.0		oshmpi_ubuntu
OSSS OpenSHMEM latest	Fedora 37	Python 3.11, PMIX 4.1.2	Open MPI 4.1.4, UCX latest	osss_fedora
	Ubuntu 22.04	Python 3.10, PMIX 4.1.2	Open MPI 4.1.4, UCX latest	osss_ubuntu
Sandia OpenSHMEM latest	Fedora 37	Python 3.11, MPICH 4.0	libfabric latest	sos_fedora
	Ubuntu 22.04	Python 3.10, MPICH 4.0	libfabric latest	sos_ubuntu
Cray OpenSHMEMX 9.1.2	Cray XC SLES 15	cray-openshmemx/9.1.2 module		

2.4 Installing shmem4py

Once a working OpenSHMEM implementation is installed, `shmem4py` can be installed using `pip`:

```
git clone https://github.com/mpi4py/shmem4py
cd shmem4py
python -m pip install .
```

You sure then test if everything works as expected:

```
make test-1
make test-2
```

2.5 Next steps

With the installation complete, you can now proceed to run the *Usage examples*. and try to base your code on them.

CHAPTER

THREE

USAGE EXAMPLES

3.1 Hello world

The simplest "Hello world" example analog to that of C implementation reads:

```
from shmem4py import shmem

mype = shmem.my_pe()
npes = shmem.n_pes()

print(f"Hello from PE {mype} of {npes}")
```

It should produce the following output:

```
$ oshrun -n 4 python -u hello.py
Hello from PE 1 of 4
Hello from PE 3 of 4
Hello from PE 2 of 4
Hello from PE 0 of 4
```

Note that unlike in C, initialization and finalization routines (*`init`* and *`finalize`*) do not need to be called explicitly.

3.2 Get a remote value

In the following example, each process (`mype`) out of `npes` processes, writes its rank into `src` and initializes an empty `dst` array. Then, each process fetches the value of `src` from the next process's (`mype + 1`) memory using *`get`* and stores it into its own `dst` array. The last process gets the value of `src` from the first process (`% npes`):

```
from shmem4py import shmem
import numpy as np

mype = shmem.my_pe()
npes = shmem.n_pes()
nextpe = (mype + 1) % npes

src = shmem.empty(1, dtype='i')
src[0] = mype

dst = np.empty(1, dtype='i')
```

(continues on next page)

(continued from previous page)

```
dst[0] = -1

print(f'Before data transfer rank {mype} src={src[0]} dst={dst[0]}')

shmem.barrier_all()
shmem.get(dst, src, nextpe)

assert dst[0] == nextpe
print(f'After data transfer rank {mype} src={src[0]} dst={dst[0]}')
```

The following output is expected:

```
$ oshrun -n 4 python -u rotget.py
Before data transfer rank 0 src=0 dst=-1
Before data transfer rank 3 src=3 dst=-1
Before data transfer rank 2 src=2 dst=-1
Before data transfer rank 1 src=1 dst=-1
After data transfer rank 0 src=0 dst=1
After data transfer rank 3 src=3 dst=0
After data transfer rank 1 src=1 dst=2
After data transfer rank 2 src=2 dst=3
```

Alternatively, the same could be achieved by using *put*, where each process can write its rank into a remote process's memory.

3.3 Broadcast an array from root to all PEs

The following code can be used to broadcast an array from a chosen rank (here **0**, the third argument of *broadcast* routine):

```
from shmem4py import shmem

mype = shmem.my_pe()
npes = shmem.n_pes()

source = shmem.zeros(npes, dtype="int32")
dest = shmem.full(npes, -999, dtype="int32")

if mype == 0:
    for i in range(npes):
        source[i] = i + 1

shmem.barrier_all()

shmem.broadcast(dest, source, 0)

print(f"{mype}: {dest}")

shmem.free(source)
shmem.free(dest)
```

The following output is expected:

```
$ oshrun -np 6 python -u broadcast.py
0: [1 2 3 4 5 6]
1: [1 2 3 4 5 6]
2: [1 2 3 4 5 6]
3: [1 2 3 4 5 6]
4: [1 2 3 4 5 6]
5: [1 2 3 4 5 6]
```

3.4 Approximate the value of Pi with reductions

The following example approximates the value of Pi following the C example given by Sandia SOS (pi_reduce.c):

```
from shmem4py import shmem
import random

RAND_MAX = 2147483647
NUM_POINTS = 10000

inside = shmem.zeros(1, dtype='i')
total = shmem.zeros(1, dtype='i')

myshmem_n_pes = shmem.n_pes()
me = shmem.my_pe()

random.seed(1+me)

for _ in range(0, NUM_POINTS):
    x = random.randint(0, RAND_MAX)/RAND_MAX
    y = random.randint(0, RAND_MAX)/RAND_MAX

    total[0] += 1
    if x*x + y*y < 1:
        inside[0] += 1

shmem.barrier_all()

shmem.sum_reduce(inside, inside)
shmem.sum_reduce(total, total)

if me == 0:
    approx_pi = 4.0*inside/total
    print(f"Pi from {total} points on {myshmem_n_pes} PEs: {approx_pi}")

shmem.free(inside)
shmem.free(total)
```

Here we can see that as the total number of points depends on the number of PEs, the more processes we use, the more accurate the approximation is:

```
$ oshrun -np 1 python -u pi.py
Pi from [10000] points on 1 PEs: [3.1336]
```

(continues on next page)

(continued from previous page)

```
$ oshrun -np 25 python -u pi.py
Pi from [250000] points on 25 PEs: [3.1392]
$ oshrun -np 100 python -u pi.py
Pi from [1000000] points on 100 PEs: [3.140364]
$ oshrun -np 250 python -u pi.py
Pi from [2500000] points on 250 PEs: [3.1413872]
```

3.5 Collect the same number of elements from each PE

Hint: MPI programmers will see the close resemblance of *`fcollect`* to MPI_Allgather.

The following example gathers one element from the `src` array from each PE into a single array available on all the PEs. It is a port of the C OpenSHMEM example (fcollect.c):

```python
from shmem4py import shmem

npes = shmem.n_pes()
me = shmem.my_pe()

dst = shmem.full(npes, 10101, dtype="int32")
src = shmem.zeros(1, dtype="int32")
src[0] = me + 100

print(f"BEFORE: dst[{me}/{npes}] = {dst}")

shmem.barrier_all()
shmem.fcollect(dst, src)
shmem.barrier_all()

print(f"AFTER: dst[{me}/{npes}] = {dst}")

shmem.free(dst)
shmem.free(src)
```

As we can see in the output, the results are available on every PE:

```
$ oshrun -np 6 python -u ./fcollect.py
 BEFORE: dst[0/6] = [10101 10101 10101 10101 10101 10101]
 BEFORE: dst[1/6] = [10101 10101 10101 10101 10101 10101]
 BEFORE: dst[2/6] = [10101 10101 10101 10101 10101 10101]
 BEFORE: dst[3/6] = [10101 10101 10101 10101 10101 10101]
 BEFORE: dst[4/6] = [10101 10101 10101 10101 10101 10101]
 BEFORE: dst[5/6] = [10101 10101 10101 10101 10101 10101]
 AFTER: dst[0/6] = [100 101 102 103 104 105]
 AFTER: dst[2/6] = [100 101 102 103 104 105]
 AFTER: dst[4/6] = [100 101 102 103 104 105]
 AFTER: dst[3/6] = [100 101 102 103 104 105]
 AFTER: dst[1/6] = [100 101 102 103 104 105]
 AFTER: dst[5/6] = [100 101 102 103 104 105]
```

3.6 Collect a different number of elements from each PE

Hint: MPI programmers will see the close resemblance of *collect* to MPI_Allgatherv.

The following example gathers a different number of elements from each PE into a single array available on all the PEs. It is a port of the C OpenSHMEM example (collect64.c). Each PE has a symmetric array of 4 elements (`[11, 12, 13, 14]`). `me+1` elements from each PE are collected into a single array:

```
from shmem4py import shmem

npes = shmem.n_pes()
me = shmem.my_pe()

src = shmem.array([11, 12, 13, 14])
dst = shmem.full(npes*(1+npes)//2, -1)

shmem.barrier_all()

shmem.collect(dst, src, me+1)

print(f"AFTER: dst[{me}/{npes}] = {dst}")

shmem.free(src)
shmem.free(dst)
```

As we can see in the output, the results are available on every PE:

```
$ oshrun -np 4 python -u collect.py
AFTER: dst[0/4] = [11 11 12 11 12 13 11 12 13 14]
AFTER: dst[1/4] = [11 11 12 11 12 13 11 12 13 14]
AFTER: dst[2/4] = [11 11 12 11 12 13 11 12 13 14]
AFTER: dst[3/4] = [11 11 12 11 12 13 11 12 13 14]
```

3.7 Atomic conditional swap on a remote data object

This example is ported from the OpenSHMEM Specification (Example 21). In it, the first PE to execute the conditional swap will successfully write its PE number to `race_winner` array on PE 0:

```
from shmem4py import shmem

race_winner = shmem.array([-1])

mype = shmem.my_pe()
oldval = shmem.atomic_compare_swap(race_winner, -1, mype, 0)

if oldval == -1:
    print(f"PE {mype} was first")

shmem.free(race_winner)
```

As expected, the order of the PEs is not guaranteed:

```
$ oshrun -np 64 python -u race_winner.py
PE 0 was first
$ oshrun -np 64 python -u race_winner.py
PE 32 was first
$ oshrun -np 64 python -u race_winner.py
PE 32 was first
$ oshrun -np 64 python -u race_winner.py
PE 48 was first
```

3.8 Test if condition is met

Tip: Note the usage of `wait_vars[idx:idx+1]` to refer to a mutable slice containing one value of the array in this example. `wait_vars[idx]` would be a read-only value and cannot be updated.

This example is ported from the OpenSHMEM Specification (Example 40). In this example, each non-zero PE updates a value in an array on PE `0`. PE `0` returns once the first process completed the update:

```
from shmem4py import shmem

mype = shmem.my_pe()
npes = shmem.n_pes()
if npes == 1:
    exit(0) # test requires at least 2 PEs

wait_vars = shmem.zeros(npes, dtype='i')

if mype == 0:
    idx = 0
    while not shmem.test(wait_vars[idx:idx+1], shmem.CMP.NE, 0):
        idx = (idx + 1) % npes
    print(f"PE {mype} observed first update from PE {idx}")

else:
    shmem.atomic_set(wait_vars[mype:mype+1], mype, 0)

shmem.free(wait_vars)
```

As before, the order of the updates is not guaranteed:

```
$ oshrun -np 64 python -u race_winner_test.py
PE 0 observed first update from PE 12
$ oshrun -np 64 python -u race_winner_test.py
PE 0 observed first update from PE 3
```

3.9 All to all communication

This example is ported from the OpenSHMEM Specification (Example 31). All pairs of PEs exchange two integers:

```
from shmem4py import shmem

mype = shmem.my_pe()
npes = shmem.n_pes()

count = 2

source = shmem.zeros(count*npes, dtype="int32")
dest = shmem.full(count*npes, 9999, dtype="int32")

for pe in range(0, npes):
    for i in range(0, count):
        source[(pe*count) + i] = mype*npes + pe

print(f"{mype}: source = {source}")

team = shmem.Team(shmem.TEAM_WORLD)
team.sync()

shmem.alltoall(dest, source, 2, team)

print(f"{mype}: dest = {dest}")

# verify results
for pe in range(0, npes):
    for i in range(0, count):
        if dest[(pe*count) + i] != pe*npes + mype:
            print(f"[{mype}] ERROR: dest[{(pe*count) + i}]={dest[(pe*count) + i]}, should be {pe*npes + mype}")

shmem.free(dest)
shmem.free(source)
```

We see the transposition in the destination array:

```
$ oshrun -np 3 python -u alltoall.py
0: source = [0 0 1 1 2 2]
1: source = [3 3 4 4 5 5]
2: source = [6 6 7 7 8 8]
0: dest = [0 0 3 3 6 6]
1: dest = [1 1 4 4 7 7]
2: dest = [2 2 5 5 8 8]
```

3.10 Locking

This example is ported from the OpenSHMEM Specification (Example 45). A lock is used to make sure that only one process modifies the array on PE `0`:

```
from shmem4py import shmem

lock = shmem.new_lock()
mype = shmem.my_pe()

count = shmem.array([0], dtype='i')
val = shmem.array([0], dtype='i')

shmem.set_lock(lock)
shmem.get(val, count, 0)
print(f"{mype}: count is {val[0]}")
val[0] += 1
shmem.put(count, val, 0)
shmem.clear_lock(lock)

shmem.del_lock(lock)
shmem.free(count)
shmem.free(val)
```

Alternatively, `shmem4py` provides a more object-oriented interface to achieve the same:

```
from shmem4py import shmem

lock = shmem.Lock()
mype = shmem.my_pe()

count = shmem.array([0], dtype='i')
val = shmem.array([0], dtype='i')

lock.acquire()
shmem.get(val, count, 0)
print(f"{mype}: count is {val[0]}")
val[0] += 1
shmem.put(count, val, 0)
lock.release()

lock.destroy()
shmem.free(count)
shmem.free(val)
```

Both examples produce the same output:

```
$ oshrun -np 7 python -u lock_oo.py
4: count is 0
3: count is 1
2: count is 2
1: count is 3
0: count is 4
```

(continues on next page)

(continued from previous page)

```
5: count is 5
6: count is 6
```

CHAPTER

FOUR

OPENSHMEM

4.1 Version and Vendor Query

info_get_version()	Return the major and minor version of the library implementation.
info_get_name()	Return the name string of the library implementation.

shmem4py.shmem.**info_get_version**()

Return the major and minor version of the library implementation.

Return type
Tuple[int, int]

shmem4py.shmem.**info_get_name**()

Return the name string of the library implementation.

Return type
str

4.2 Library Setup and Exit

init()	Allocate and initialize the needed resources.
finalize()	Release all the used resources.
global_exit([status])	Force termination of an entire program.
init_thread([requested])	Initialize the library with support for the provided thread level.
query_thread()	Return the level of thread support provided by the library.
THREAD(value[, names, module, qualname, ...])	Threading support levels.

shmem4py.shmem.**init**()

Allocate and initialize the needed resources. Collective.

All PEs must call this routine, or *init_thread*, before any other OpenSHMEM routine. It must be matched with a call to *finalize* at the end of the program.

Return type
None

shmem4py.shmem.**finalize**()

Release all the used resources. Collective.

This only terminates the shmem portion of a program, not the entire program. All processes that represent the PEs will still exist after the call to *finalize* returns, but they will no longer have access to resources that have been released.

Return type
None

shmem4py.shmem.**global_exit**(*status=0*)

Force termination of an entire program. Can be called by any PE.

Parameters
status (*int*) – The exit status of the main program.

Return type
NoReturn

shmem4py.shmem.**init_thread**(*requested=THREAD_MULTIPLE*)

Initialize the library with support for the provided thread level.

Either *init* or *init_thread* should be used to initialize the program.

Parameters
requested (THREAD) – The thread level support requested by the user.

Returns
The thread level support provided by the implementation.

Return type
THREAD

shmem4py.shmem.**query_thread**()

Return the level of thread support provided by the library.

Return type
THREAD

class shmem4py.shmem.**THREAD**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Threading support levels.

SINGLE

A single-threaded program. A hybrid program should not request *SINGLE* at the initialization call of either OpenSHMEM or MPI but request a different thread level at the initialization call of the other model.

Type
int

FUNNELED

Allows only the main thread to make communication calls.

Type
int

SERIALIZED

Communication calls are not made concurrently by multiple threads.

Type
int

MULTIPLE

The program may be multithreaded and any thread may invoke the OpenSHMEM interfaces.

Type
int

4.3 Accessibility Queries

my_pe()	Return the number of the calling PE.
n_pes()	Return the number of PEs running in a program.
pe_accessible(pe)	Return whether a PE is accessible.
addr_accessible(addr, pe)	Return whether a local array is accessible from the specified remote PE.
ptr(target, pe)	Return a local view to a symmetric array on the specified PE.

shmem4py.shmem.**my_pe**()

Return the number of the calling PE.

Return type
int

shmem4py.shmem.**n_pes**()

Return the number of PEs running in a program.

Return type
int

shmem4py.shmem.**pe_accessible**(*pe*)

Return whether a PE is accessible.

Parameters
pe (*int*) – The PE number to check for accessibility from the local PE.

Return type
bool

shmem4py.shmem.**addr_accessible**(*addr*, *pe*)

Return whether a local array is accessible from the specified remote PE.

Parameters

- **addr** (*NDArray[Any]*) – Local array object to query.
- **pe** (*int*) – The id of a remote PE.

Return type
bool

shmem4py.shmem.**ptr**(*target*, *pe*)

Return a local view to a symmetric array on the specified PE.

Parameters

- **target** (*NDArray[T]*) – The symmetric destination array.
- **pe** (*int*) – The PE number on which `target` is to be accessed.

Returns
A local pointer to the remotely accessible `target` array is returned when it can be accessed using memory loads and stores. Otherwise, `None` is returned.

Return type
NDArray[*T*] | None

4.4 Memory Management

alloc(count[, size, align, hints, clear])	Return memory allocated from the symmetric heap.
free(mem)	Deallocate memory of `mem`.
fromalloc(mem[, shape, dtype, order])	Return a NumPy array interpreted from the buffer allocated in the symmetric memory.
new_array(shape[, dtype, order, align, ...])	Return a new NumPy array allocated in the symmetric memory.
del_array(a)	Delete the array.
array(obj[, dtype, order, align, hints])	Return a new NumPy array allocated in the symmetric memory and initialize contents with `obj`.
empty(shape[, dtype, order, align, hints])	Return a new empty NumPy array allocated in the symmetric memory.
zeros(shape[, dtype, order, align, hints])	Return a new `0`-initialized NumPy array allocated in the symmetric memory.
ones(shape[, dtype, order, align, hints])	Return a new `1`-initialized NumPy array allocated in the symmetric memory.
full(shape, fill_value[, dtype, order, ...])	Return a new `fill_value`-initialized NumPy array allocated in the symmetric memory.
MALLOC(value[, names, module, qualname, ...])	Memory allocation hints.

shmem4py.shmem.**alloc**(*count*, *size=1*, *align=None*, *hints=None*, *clear=True*)

Return memory allocated from the symmetric heap.

Parameters

- **count** (*int*) – Number of elements to allocate.
- **size** (*int*) – Size of each element in bytes.
- **align** (*int* | *None*) – Byte alignment of the block allocated from the symmetric heap.
- **hints** (*int* | *None*) – A bit array of hints provided by the user to the implementation. Valid hints are defined as enumerations in *MALLOC* and can be combined using the bitwise OR operator.
- **clear** (*bool*) – If `True`, the allocated memory is cleared to zero.

Return type
memoryview

shmem4py.shmem.**free**(*mem*)

Deallocate memory of `mem`.

Parameters
mem (*memoryview* | *NDArray[Any]*) – The object to be deallocated.

Return type
None

shmem4py.shmem.**fromalloc**(*mem*, *shape=None*, *dtype=None*, *order='C'*)

Return a NumPy array interpreted from the buffer allocated in the symmetric memory.

Parameters

- **mem** (*memoryview*) – The memory to be interpreted as a NumPy array.
- **shape** (*int | Sequence[int] | None*) – The shape of the array. If `None`, the shape is inferred from the size of the memory.
- **dtype** (*DTypeLike*) – The data type of the array. If `None`, the data type is inferred from the memory contents.
- **order** (*Literal['C', 'F']*) – The memory layout of the array. If `'C'`, the array is contiguous in memory (row major). If `'F'`, the array is Fortran contiguous (column major).

Return type
NDArray[*Any*]

shmem4py.shmem.**new_array**(*shape*, *dtype=float*, *order='C'*, ***, *align=None*, *hints=None*, *clear=True*)

Return a new NumPy array allocated in the symmetric memory.

Parameters

- **shape** (*int | Sequence[int]*) – The shape of the array.
- **dtype** (*DTypeLike*) – The data type of the array.
- **order** (*Literal['C', 'F']*) – The memory layout of the array. If `'C'`, the array is contiguous in memory (row major). If `'F'`, the array is Fortran contiguous (column major).
- **align** (*int | None*) – Byte alignment of the block allocated in the symmetric memory. Keyword argument only.
- **hints** (*int | None*) – A bit array of hints provided by the user to the implementation. Valid hints are defined as enumerations in *MALLOC* and can be combined using the bitwise OR operator. Keyword argument only.
- **clear** (*bool*) – If `True`, the allocated memory is cleared to zero. Keyword argument only.

Return type
NDArray[*Any*]

shmem4py.shmem.**del_array**(*a*)

Delete the array.

Parameters
a (*NDArray[Any]*) – The array to be deleted.

Return type
None

shmem4py.shmem.**array**(*obj*, *dtype=None*, ***, *order='K'*, *align=None*, *hints=None*)

Return a new NumPy array allocated in the symmetric memory and initialize contents with `obj`.

Parameters

- **obj** (*Any*) – The object from which a NumPy array is to be initialized.
- **dtype** (*DTypeLike*) – The data type of the array. If `None`, the data type is inferred from the memory contents.
- **order** (*Literal['K', 'A', 'C', 'F']*) – The memory layout of the array. See `numpy.array` for the explanation of the options. Keyword argument only.

- **align** (*int | None*) – Byte alignment of the block allocated in the symmetric memory. Keyword argument only.
- **hints** (*int | None*) – A bit array of hints provided by the user to the implementation. Valid hints are defined as enumerations in *MALLOC* and can be combined using the bitwise OR operator. Keyword argument only.

Return type
NDArray[*Any*]

shmem4py.shmem.**empty**(*shape*, *dtype=float*, *order='C'*, ***, *align=None*, *hints=None*)

Return a new empty NumPy array allocated in the symmetric memory.

Parameters

- **shape** (*int | Sequence[int]*) – The shape of the array.
- **dtype** (*DTypeLike*) – The data type of the array.
- **order** (*Literal['C', 'F']*) – The memory layout of the array. If `'C'`, the array is contiguous in memory (row major). If `'F'`, the array is Fortran contiguous (column major).
- **align** (*int | None*) – Byte alignment of the block allocated in the symmetric memory. Keyword argument only.
- **hints** (*int | None*) – A bit array of hints provided by the user to the implementation. Valid hints are defined as enumerations in *MALLOC* and can be combined using the bitwise OR operator. Keyword argument only.

Return type
NDArray[*Any*]

shmem4py.shmem.**zeros**(*shape*, *dtype=float*, *order='C'*, ***, *align=None*, *hints=None*)

Return a new `0`-initialized NumPy array allocated in the symmetric memory.

Parameters

- **shape** (*int | Sequence[int]*) – The shape of the array.
- **dtype** (*DTypeLike*) – The data type of the array.
- **order** (*Literal['C', 'F']*) – The memory layout of the array. If `'C'`, the array is contiguous in memory (row major). If `'F'`, the array is Fortran contiguous (column major).
- **align** (*int | None*) – Byte alignment of the block allocated in the symmetric memory. Keyword argument only.
- **hints** (*int | None*) – A bit array of hints provided by the user to the implementation. Valid hints are defined as enumerations in *MALLOC* and can be combined using the bitwise OR operator. Keyword argument only.

Return type
NDArray[*Any*]

shmem4py.shmem.**ones**(*shape*, *dtype=float*, *order='C'*, ***, *align=None*, *hints=None*)

Return a new `1`-initialized NumPy array allocated in the symmetric memory.

Parameters

- **shape** (*int | Sequence[int]*) – The shape of the array.
- **dtype** (*DTypeLike*) – The data type of the array.
- **order** (*Literal['C', 'F']*) – The memory layout of the array. If `'C'`, the array is contiguous in memory (row major). If `'F'`, the array is Fortran contiguous (column major).

- **align** (*int | None*) – Byte alignment of the block allocated in the symmetric memory. Keyword argument only.
- **hints** (*int | None*) – A bit array of hints provided by the user to the implementation. Valid hints are defined as enumerations in *MALLOC* and can be combined using the bitwise OR operator. Keyword argument only.

Return type
NDArray[*Any*]

shmem4py.shmem.**full**(*shape*, *fill_value*, *dtype=None*, *order='C'*, ***, *align=None*, *hints=None*)

Return a new `fill_value`-initialized NumPy array allocated in the symmetric memory.

Parameters

- **shape** (*int | Sequence[int]*) – The shape of the array.
- **fill_value** (*int | float | complex | number*) – The value to fill the array with.
- **dtype** (*DTypeLike*) – The data type of the array.
- **order** (*Literal['C', 'F']*) – The memory layout of the array. If `'C'`, the array is contiguous in memory (row major). If `'F'`, the array is Fortran contiguous (column major).
- **align** (*int | None*) – Byte alignment of the block allocated in the symmetric memory. Keyword argument only.
- **hints** (*int | None*) – A bit array of hints provided by the user to the implementation. Valid hints are defined as enumerations in *MALLOC* and can be combined using the bitwise OR operator. Keyword argument only.

Return type
NDArray[*Any*]

class shmem4py.shmem.**MALLOC**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Memory allocation hints.

ATOMICS_REMOTE

The allocated memory will be used for atomic variables.

Type
int

SIGNAL_REMOTE

The allocated memory will be used for signal variables.

Type
int

4.5 Team Management

Team([team])	Team management.
Team.destroy()	Destroy the team.
Team.split_strided([start, stride, size, config])	Return a new team from a subset of the existing parent team PEs.
Team.get_config()	Return the configuration parameters of the team.
Team.my_pe()	Return the number of the calling PE within the team.
Team.n_pes()	Return the number of PEs in the team.
Team.translate_pe([pe, team])	Translate a given PE number from one team to the corresponding PE number in another team.
Team.create_ctx([options])	Create a communication context from the team.
Team.sync()	Register the arrival of a PE at a synchronization point.

class shmem4py.shmem.**Team**(*team=None*)

Team management.

Parameters
team (*Optional[Union[Team, TeamHandle]]*)

Return type
Team

destroy()

Destroy the team.

Return type
None

split_strided(*start=0*, *stride=1*, *size=None*, *config=None*, ***kwargs*)

Return a new team from a subset of the existing parent team PEs.

This routine must be called by all PEs in the parent team.

Parameters

- **start** (*int*) – The lowest PE number of the subset of PEs from the parent team that will form the new team.
- **stride** (*int*) – The stride between team PE numbers in the parent team that comprise the subset of PEs that will form the new team.
- **size** (*int | None*) – The number of PEs from the parent team in the subset of PEs that will form the new team. If None, the size is automatically determined.
- **config** (*Mapping[str, int] | None*) – Configuration parameters for the new team. Currently, only SHMEM_TEAM_NUM_CONTEXTS key is supported.
- ****kwargs** (*int*) – Additional configuration parameters for the new team.

Return type
Team

get_config()

Return the configuration parameters of the team.

Return type
Dict[str, int]

my_pe()

Return the number of the calling PE within the team.

Return type
int

n_pes()

Return the number of PEs in the team.

Return type
int

translate_pe(*pe=None*, *team=None*)

Translate a given PE number from one team to the corresponding PE number in another team.

Parameters

- **pe** (*int* | *None*) – PE number in the source team. If None, defaults to the calling PE number.
- **team** (Team | *None*) – Destination team. If None, defaults to the world team.

Return type
int

create_ctx(*options=0*)

Create a communication context from the team.

Parameters
options (*int*) – The set of options requested for the given context. Valid options are the enumerations listed in the *CTX* class. Multiple options may be requested by combining them with a bitwise OR operation. 0 can be used if no options are requested.

Return type
Ctx

sync()

Register the arrival of a PE at a synchronization point.

This routine does not return until all other PEs in a given team or active set arrive at this synchronization point.

Return type
None

4.6 Communication Management

Ctx([ctx])	Communication context.
Ctx.create([options, team])	Return a new communication context.
Ctx.destroy()	Destroy the communication context.
Ctx.get_team()	Retrieve the team associated with the communication context.
Ctx.fence()	Ensure ordering of delivery of operations on symmetric data objects.
Ctx.quiet()	Wait for completion of outstanding operations on symmetric data objects issued by a PE.
CTX(value[, names, module, qualname, type, ...])	Context creation options.

class shmem4py.shmem.**Ctx**(*ctx=None*)

Communication context.

Parameters
ctx (*Optional[Union[Ctx, CtxHandle]]*)

Return type
Ctx

static create(*options=0*, *team=None*)

Return a new communication context.

Parameters

- **options** (*int*) – The set of options requested for the given context. Valid options are the enumerations listed in the *CTX* class. Multiple options may be requested by combining them with a bitwise OR operation. `0` can be used if no options are requested.
- **team** (Team | *None*) – If the team is specified, the communication context is created from this `team`.

Return type
Ctx

destroy()

Destroy the communication context.

Return type
None

get_team()

Retrieve the team associated with the communication context.

Return type
Team

fence()

Ensure ordering of delivery of operations on symmetric data objects.

All operations on symmetric data objects issued to a particular PE on the given context prior to the call to *fence* are guaranteed to be delivered before any subsequent operations on symmetric data objects to the same PE.

Return type
None

quiet()

Wait for completion of outstanding operations on symmetric data objects issued by a PE.

Ensures completion of all operations on symmetric data objects issued by the calling PE on the given context.

Return type
None

class shmem4py.shmem.**CTX**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Context creation options.

PRIVATE

The given context will be used only by the thread that created it.

Type
int

SERIALIZED

The given context is shareable but will not be used by multiple threads concurrently.

Type
int

NOSTORE

quiet and *fence* operations performed on the given context are not required to enforce completion and ordering of memory store operations.

Type
int

4.7 Remote Memory Access

put(target, source, pe[, size, ctx])	Copy data from local `source` to `target` on PE `pe`.
get(target, source, pe[, size, ctx])	Copy data from `source` on PE `pe` to local `target`.
iput(target, source, pe[, tst, sst, size, ctx])	Copy strided data from local `source` to `target` on PE `pe`.
iget(target, source, pe[, tst, sst, size, ctx])	Copy strided data from `source` on PE `pe` to local `target`.
put_nbi(target, source, pe[, size, ctx])	Nonblocking copy data from local `source` to `target` on PE `pe`.
get_nbi(target, source, pe[, size, ctx])	Nonblocking copy data from `source` on PE `pe` to local `target`.

shmem4py.shmem.**put**(*target*, *source*, *pe*, *size=None*, *ctx=None*)

Copy data from local `source` to `target` on PE `pe`.

Parameters

- **target** (*NDArray[T]*) – Symmetric destination array.
- **source** (*NDArray[T]*) – Local array containing the data to be copied.
- **pe** (*int*) – PE number of the remote PE.
- **size** (*int | None*) – Number of elements to copy.
- **ctx** (*Ctx | None*) – A context handle specifying the context on which to perform the operation.

Return type
None

shmem4py.shmem.**get**(*target*, *source*, *pe*, *size=None*, *ctx=None*)

Copy data from `source` on PE `pe` to local `target`.

Parameters

- **target** (*NDArray[T]*) – Local array to be updated.

- **source** (*NDArray[T]*) – Symmetric source array.
- **pe** (*int*) – PE number of the remote PE.
- **size** (*int | None*) – Number of elements to copy.
- **ctx** (*Ctx | None*) – A context handle specifying the context on which to perform the operation.

Return type
None

shmem4py.shmem.**iput**(*target*, *source*, *pe*, *tst=1*, *sst=1*, *size=None*, *ctx=None*)

Copy strided data from local `source` to `target` on PE `pe`.

Parameters

- **target** (*NDArray[T]*) – Symmetric destination array.
- **source** (*NDArray[T]*) – Local array containing the data to be copied.
- **pe** (*int*) – PE number of the remote PE.
- **tst** (*int*) – The stride between consecutive elements of the `target` array. The stride is scaled by the element size of the `target` array. A value of `1` indicates contiguous data.
- **sst** (*int*) – The stride between consecutive elements of the `source` array. The stride is scaled by the element size of the `source` array. A value of `1` indicates contiguous data.
- **size** (*int | None*) – Number of elements to copy.
- **ctx** (*Ctx | None*) – A context handle specifying the context on which to perform the operation.

Return type
None

shmem4py.shmem.**iget**(*target*, *source*, *pe*, *tst=1*, *sst=1*, *size=None*, *ctx=None*)

Copy strided data from `source` on PE `pe` to local `target`.

Parameters

- **target** (*NDArray[T]*) – Local array to be updated.
- **source** (*NDArray[T]*) – Symmetric source array.
- **pe** (*int*) – PE number of the remote PE.
- **tst** (*int*) – The stride between consecutive elements of the `target` array. The stride is scaled by the element size of the `target` array. A value of `1` indicates contiguous data.
- **sst** (*int*) – The stride between consecutive elements of the `source` array. The stride is scaled by the element size of the `source` array. A value of `1` indicates contiguous data.
- **size** (*int | None*) – Number of elements to copy.
- **ctx** (*Ctx | None*) – A context handle specifying the context on which to perform the operation.

Return type
None

shmem4py.shmem.**put_nbi**(*target*, *source*, *pe*, *size=None*, *ctx=None*)

Nonblocking copy data from local `source` to `target` on PE `pe`.

Parameters

- **target** (*NDArray[T]*) – Symmetric destination array.
- **source** (*NDArray[T]*) – Local array containing the data to be copied.
- **pe** (*int*) – PE number of the remote PE.
- **size** (*int | None*) – Number of elements to copy.
- **ctx** (*Ctx | None*) – A context handle specifying the context on which to perform the operation.

Return type
None

shmem4py.shmem.**get_nbi**(*target*, *source*, *pe*, *size=None*, *ctx=None*)

Nonblocking copy data from `source` on PE `pe` to local `target`.

Parameters

- **target** (*NDArray[T]*) – Local array to be updated.
- **source** (*NDArray[T]*) – Symmetric source array.
- **pe** (*int*) – PE number of the remote PE.
- **size** (*int | None*) – Number of elements to copy.
- **ctx** (*Ctx | None*) – A context handle specifying the context on which to perform the operation.

Return type
None

4.8 Atomic Memory Operations

atomic_op(target, value, op, pe[, ctx])	Perform operation `op` on `value` and `target` on PE `pe`.
atomic_fetch_op(target, value, op, pe[, ctx])	Perform operation `op` on `value` and `target` on PE `pe` and return `target`'s prior value.
atomic_fetch_op_nbi(fetch, target, value, op, pe)	Perform operation `op` on `value` and `target` on PE `pe` and fetch `target`'s prior value to `fetch`.
AMO(value[, names, module, qualname, type, ...])	Atomic Memory Operations.

atomic_set(target, value, pe[, ctx])	Write `value` into `target` on PE `pe`.
atomic_inc(target, pe[, ctx])	Increment `target` array element on PE `pe`.
atomic_add(target, value, pe[, ctx])	Add `value` to `target` on PE `pe` and atomically update `target`.
atomic_and(target, value, pe[, ctx])	Perform bitwise AND on `value` and `target` on PE `pe`.
atomic_or(target, value, pe[, ctx])	Perform bitwise OR on `value` and `target` on PE `pe`.
atomic_xor(target, value, pe[, ctx])	Perform bitwise XOR on `value` and `target` on PE `pe`.
atomic_fetch(source, pe[, ctx])	Return the value of a `source` on PE `pe`.
atomic_swap(target, value, pe[, ctx])	Write `value` into `target` on PE `pe` and return the prior value.
atomic_compare_swap(target, cond, value, pe)	Conditionally update `target` on PE `pe` and return its prior value.
atomic_fetch_inc(target, pe[, ctx])	Increment `target` on PE `pe` and return its prior value.
atomic_fetch_add(target, value, pe[, ctx])	Add `value` to `target` on PE `pe` and return its prior value.
atomic_fetch_and(target, value, pe[, ctx])	Perform a bitwise AND on `value` and `target` at PE `pe` and return `target`'s prior value.
atomic_fetch_or(target, value, pe[, ctx])	Perform a bitwise OR on `value` and `target` at PE `pe` and return `target`'s prior value.
atomic_fetch_xor(target, value, pe[, ctx])	Perform a bitwise XOR on `value` and `target` at PE `pe` and return `target`'s prior value.
atomic_fetch_nbi(fetch, source, pe[, ctx])	Fetch the value of `source` on PE `pe` to local `fetch`.
atomic_swap_nbi(fetch, target, value, pe[, ctx])	Write `value` into `target` on PE `pe` and fetch its prior value to `fetch`.
atomic_compare_swap_nbi(fetch, target, cond, ...)	Conditionally update `target` and fetch its prior value to `fetch`.
atomic_fetch_inc_nbi(fetch, target, pe[, ctx])	Increment `target` on PE `pe` and fetch its prior value to `fetch`.
atomic_fetch_add_nbi(fetch, target, value, pe)	Add `value` to `target` on PE `pe` and fetch its prior value to `fetch`.
atomic_fetch_and_nbi(fetch, target, value, pe)	Perform bitwise AND on `target` on PE `pe` and fetch its prior value to `fetch`.
atomic_fetch_or_nbi(fetch, target, value, pe)	Perform bitwise OR on `target` on PE `pe` and fetch its prior value to `fetch`.
atomic_fetch_xor_nbi(fetch, target, value, pe)	Perform bitwise XOR on `target` on PE `pe` and fetch its prior value to `fetch`.

shmem4py.shmem.**atomic_op**(*target*, *value*, *op*, *pe*, *ctx=None*)

Perform operation `op` on `value` and `target` on PE `pe`.

Parameters

- **target** (*NDArray[Any]*) – Symmetric array of size 1 containing the destination value.
- **value** (*int | float | complex | number*) – The operand to the operation.
- **op** (AMO) – The operation to be performed.
- **pe** (*int*) – The PE number on which `target` is to be updated.
- **ctx** (Ctx | *None*) – The context on which to perform the operation. If None, the default context is used.

Return type

None

shmem4py.shmem.**atomic_fetch_op**(*target*, *value*, *op*, *pe*, *ctx=None*)

Perform operation `op` on `value` and `target` on PE `pe` and return `target`'s prior value.

Parameters

- **target** (*NDArray[Any]*) – Symmetric array of size 1 containing the destination value.
- **value** (*int | float | complex | number*) – The operand to the operation.
- **op** (AMO) – The operation to be performed.
- **pe** (*int*) – The PE number on which `target` is to be updated.
- **ctx** (Ctx | *None*) – The context on which to perform the operation. If None, the default context is used.

Return type

int | float | complex | *number*

shmem4py.shmem.**atomic_fetch_op_nbi**(*fetch*, *target*, *value*, *op*, *pe*, *ctx=None*)

Perform operation `op` on `value` and `target` on PE `pe` and fetch `target`'s prior value to `fetch`.

Parameters

- **fetch** (*NDArray[T]*) – Local array of size 1 to be updated.
- **target** (*NDArray[T]*) – Symmetric array of size 1 containing the destination value.
- **value** (*int | float | complex | number*) – The operand to the operation.
- **op** (AMO) – The operation to be performed.
- **pe** (*int*) – The PE number on which `target` is to be updated.
- **ctx** (Ctx | *None*) – The context on which to perform the operation. If None, the default context is used.

Return type

None

class shmem4py.shmem.**AMO**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Atomic Memory Operations.

SET

Set.

Type

str

INC

Increment.

Type

str

ADD

Add.

Type

str

AND

Bitwise AND.

Type
str

OR

Bitwise OR.

Type
str

XOR

Bitwise XOR.

Type
str

shmem4py.shmem.**atomic_set**(*target*, *value*, *pe*, *ctx=None*)

Write `value` into `target` on PE `pe`.

Parameters

- **target** (*NDArray[Any]*) – Symmetric array of size 1 where data will be written.
- **value** (*int | float | complex | number*) – The operand to the atomic set operation.
- **pe** (*int*) – The PE number on which `target` is to be updated.
- **ctx** (*Ctx | None*) – The context on which to perform the operation. If `None`, the default context is used.

Return type
None

shmem4py.shmem.**atomic_inc**(*target*, *pe*, *ctx=None*)

Increment `target` array element on PE `pe`.

Parameters

- **target** (*NDArray[Any]*) – Symmetric array of size 1 containing the element that will be modified.
- **pe** (*int*) – The PE number on which `target` is to be updated.
- **ctx** (*Ctx | None*) – The context on which to perform the operation. If `None`, the default context is used.

Return type
None

shmem4py.shmem.**atomic_add**(*target*, *value*, *pe*, *ctx=None*)

Add `value` to `target` on PE `pe` and atomically update `target`.

Parameters

- **target** (*NDArray[Any]*) – Symmetric array of size 1 containing the element that will be modified.
- **value** (*int | float | complex | number*) – The operand to the atomic add operation.
- **pe** (*int*) – The PE number on which `target` is to be updated.
- **ctx** (*Ctx | None*) – The context on which to perform the operation. If `None`, the default context is used.

Return type
None

shmem4py.shmem.**atomic_and**(*target*, *value*, *pe*, *ctx=None*)

Perform bitwise AND on `value` and `target` on PE `pe`.

Parameters

- **target** (*NDArray[Any]*) – Symmetric array of size 1 containing the element that will be modified.
- **value** (*int | float | complex | number*) – The operand to the bitwise AND operation.
- **pe** (*int*) – The PE number on which `target` is to be updated.
- **ctx** (*Ctx | None*) – The context on which to perform the operation. If `None`, the default context is used.

Return type
None

shmem4py.shmem.**atomic_or**(*target*, *value*, *pe*, *ctx=None*)

Perform bitwise OR on `value` and `target` on PE `pe`.

Parameters

- **target** (*NDArray[Any]*) – Symmetric array of size 1 containing the element that will be modified.
- **value** (*int | float | complex | number*) – The operand to the bitwise OR operation.
- **pe** (*int*) – The PE number on which `target` is to be updated.
- **ctx** (*Ctx | None*) – The context on which to perform the operation. If `None`, the default context is used.

Return type
None

shmem4py.shmem.**atomic_xor**(*target*, *value*, *pe*, *ctx=None*)

Perform bitwise XOR on `value` and `target` on PE `pe`.

Parameters

- **target** (*NDArray[Any]*) – Symmetric array of size 1 containing the element that will be modified.
- **value** (*int | float | complex | number*) – The operand to the bitwise XOR operation.
- **pe** (*int*) – The PE number on which `target` is to be updated.
- **ctx** (*Ctx | None*) – The context on which to perform the operation. If `None`, the default context is used.

Return type
None

shmem4py.shmem.**atomic_fetch**(*source*, *pe*, *ctx=None*)

Return the value of a `source` on PE `pe`.

Parameters

- **source** – Symmetric array of size 1 containing the element that will be fetched.

- **pe** (*int*) – The PE number from which `source` is to be fetched.
- **ctx** (*Ctx | None*) – The context on which to perform the operation. If `None`, the default context is used.

Return type
int | float | complex | *number*

shmem4py.shmem.**atomic_swap**(*target*, *value*, *pe*, *ctx=None*)

Write `value` into `target` on PE `pe` and return the prior value.

Parameters

- **target** (*NDArray[Any]*) – Symmetric array of size 1 containing the destination value.
- **value** (*int | float | complex | number*) – The value to be atomically written to the remote PE.
- **pe** (*int*) – The PE number on which `target` is to be updated.
- **ctx** (*Ctx | None*) – The context on which to perform the operation. If `None`, the default context is used.

Return type
int | float | complex | *number*

shmem4py.shmem.**atomic_compare_swap**(*target*, *cond*, *value*, *pe*, *ctx=None*)

Conditionally update `target` on PE `pe` and return its prior value.

Parameters

- **target** (*NDArray[Any]*) – Symmetric array of size 1 containing the destination value.
- **cond** (*int | float | complex | number*) – `cond` is compared to the remote `target` value. If `cond` and the remote `target` are equal, then `value` is swapped into the `target`; otherwise, the `target` is unchanged.
- **value** (*int | float | complex | number*) – The value to be atomically written to the remote PE.
- **pe** (*int*) – The PE number on which `target` is to be updated.
- **ctx** (*Ctx | None*) – The context on which to perform the operation. If `None`, the default context is used.

Return type
int | float | complex | *number*

shmem4py.shmem.**atomic_fetch_inc**(*target*, *pe*, *ctx=None*)

Increment `target` on PE `pe` and return its prior value.

Parameters

- **target** (*NDArray[Any]*) – Symmetric array of size 1 containing the destination value.
- **pe** (*int*) – The PE number on which `target` is to be updated.
- **ctx** (*Ctx | None*) – The context on which to perform the operation. If `None`, the default context is used.

Return type
int | float | complex | *number*

shmem4py.shmem.**atomic_fetch_add**(*target*, *value*, *pe*, *ctx=None*)

Add `value` to `target` on PE `pe` and return its prior value.

Parameters

- **target** (*NDArray[Any]*) – Symmetric array of size 1 containing the destination value.
- **value** (*int | float | complex | number*) – The operand to the atomic fetch-and-add operation.
- **pe** (*int*) – The PE number on which `target` is to be updated.
- **ctx** (*Ctx | None*) – The context on which to perform the operation. If `None`, the default context is used.

Return type

int | float | complex | *number*

shmem4py.shmem.**atomic_fetch_and**(*target*, *value*, *pe*, *ctx=None*)

Perform a bitwise AND on `value` and `target` at PE `pe` and return `target`'s prior value.

Parameters

- **target** (*NDArray[Any]*) – Symmetric array of size 1 containing the destination value.
- **value** (*int | float | complex | number*) – The operand to the bitwise AND operation.
- **pe** (*int*) – The PE number on which `target` is to be updated.
- **ctx** (*Ctx | None*) – The context on which to perform the operation. If `None`, the default context is used.

Return type

int | float | complex | *number*

shmem4py.shmem.**atomic_fetch_or**(*target*, *value*, *pe*, *ctx=None*)

Perform a bitwise OR on `value` and `target` at PE `pe` and return `target`'s prior value.

Parameters

- **target** (*NDArray[Any]*) – Symmetric array of size 1 containing the destination value.
- **value** (*int | float | complex | number*) – The operand to the bitwise OR operation.
- **pe** (*int*) – The PE number on which `target` is to be updated.
- **ctx** (*Ctx | None*) – The context on which to perform the operation. If `None`, the default context is used.

Return type

int | float | complex | *number*

shmem4py.shmem.**atomic_fetch_xor**(*target*, *value*, *pe*, *ctx=None*)

Perform a bitwise XOR on `value` and `target` at PE `pe` and return `target`'s prior value.

Parameters

- **target** (*NDArray[Any]*) – Symmetric array of size 1 containing the destination value.
- **value** (*int | float | complex | number*) – The operand to the bitwise XOR operation.
- **pe** (*int*) – The PE number on which `target` is to be updated.

- **ctx** (Ctx | None) – The context on which to perform the operation. If None, the default context is used.

Return type
int | float | complex | *number*

shmem4py.shmem.**atomic_fetch_nbi**(*fetch*, *source*, *pe*, *ctx=None*)

Fetch the value of source on PE pe to local fetch.

Nonblocking. The operation is considered complete after a subsequent call to *quiet*.

Parameters

- **fetch** (*NDArray[T]*) – Local array of size 1 to be updated.
- **source** (*NDArray[T]*) – Symmetric array of size 1 containing the element that will be fetched.
- **pe** (*int*) – The PE number from which source is to be fetched.
- **ctx** (Ctx | None) – The context on which to perform the operation. If None, the default context is used.

Return type
None

shmem4py.shmem.**atomic_swap_nbi**(*fetch*, *target*, *value*, *pe*, *ctx=None*)

Write value into target on PE pe and fetch its prior value to fetch.

Nonblocking. The operation is considered complete after a subsequent call to *quiet*.

Parameters

- **fetch** (*NDArray[T]*) – Local array of size 1 to be updated.
- **target** (*NDArray[T]*) – Symmetric array of size 1 containing the destination value.
- **value** (*int | float | complex | number*) – The value to be atomically written to the remote PE.
- **pe** (*int*) – The PE number on which target is to be updated.
- **ctx** (Ctx | None) – The context on which to perform the operation. If None, the default context is used.

Return type
None

shmem4py.shmem.**atomic_compare_swap_nbi**(*fetch*, *target*, *cond*, *value*, *pe*, *ctx=None*)

Conditionally update target and fetch its prior value to fetch.

Nonblocking. The operation is considered complete after a subsequent call to *quiet*.

Parameters

- **fetch** (*NDArray[T]*) – Local array of size 1 to be updated.
- **target** (*NDArray[T]*) – Symmetric array of size 1 containing the destination value.
- **cond** (*int | float | complex | number*) – cond is compared to the remote target value. If cond and the remote target are equal, then value is swapped into the target; otherwise, the target is unchanged.
- **value** (*int | float | complex | number*) – The value to be atomically written to the remote PE.

- **pe** (*int*) – The PE number on which `target` is to be updated.
- **ctx** (*Ctx | None*) – The context on which to perform the operation. If `None`, the default context is used.

Return type
None

shmem4py.shmem.**atomic_fetch_inc_nbi**(*fetch*, *target*, *pe*, *ctx=None*)

Increment `target` on PE `pe` and fetch its prior value to `fetch`.

Nonblocking.

The operation is considered complete after a subsequent call to *quiet*.

Parameters

- **fetch** (*NDArray[T]*) – Local array of size 1 to be updated.
- **target** (*NDArray[T]*) – Symmetric array of size 1 containing the destination value.
- **pe** (*int*) – The PE number on which `target` is to be updated.
- **ctx** (*Ctx | None*) – The context on which to perform the operation. If `None`, the default context is used.

Return type
None

shmem4py.shmem.**atomic_fetch_add_nbi**(*fetch*, *target*, *value*, *pe*, *ctx=None*)

Add `value` to `target` on PE `pe` and fetch its prior value to `fetch`.

Nonblocking. The operation is considered complete after a subsequent call to *quiet*.

Parameters

- **fetch** (*NDArray[T]*) – Local array of size 1 to be updated.
- **target** (*NDArray[T]*) – Symmetric array of size 1 containing the destination value.
- **value** (*int | float | complex | number*) – The value to be the atomic fetch-and-add operation.
- **pe** (*int*) – The PE number on which `target` is to be updated.
- **ctx** (*Ctx | None*) – The context on which to perform the operation. If `None`, the default context is used.

Return type
None

shmem4py.shmem.**atomic_fetch_and_nbi**(*fetch*, *target*, *value*, *pe*, *ctx=None*)

Perform bitwise AND on `target` on PE `pe` and fetch its prior value to `fetch`.

Nonblocking. The operation is considered complete after a subsequent call to *quiet*.

Parameters

- **fetch** (*NDArray[T]*) – Local array of size 1 to be updated.
- **target** (*NDArray[T]*) – Symmetric array of size 1 containing the destination value.
- **value** (*int | float | complex | number*) – The operand to the bitwise AND operation.
- **pe** (*int*) – The PE number on which `target` is to be updated.

- **ctx** (*Ctx | None*) – The context on which to perform the operation. If `None`, the default context is used.

Return type
None

shmem4py.shmem.**atomic_fetch_or_nbi**(*fetch*, *target*, *value*, *pe*, *ctx=None*)

Perform bitwise OR on `target` on PE `pe` and fetch its prior value to `fetch`.

Nonblocking. The operation is considered complete after a subsequent call to *quiet*.

Parameters

- **fetch** (*NDArray[T]*) – Local array of size 1 to be updated.
- **target** (*NDArray[T]*) – Symmetric array of size 1 containing the destination value.
- **value** (*int | float | complex | number*) – The operand to the bitwise OR operation.
- **pe** (*int*) – The PE number on which `target` is to be updated.
- **ctx** (*Ctx | None*) – The context on which to perform the operation. If `None`, the default context is used.

Return type
None

shmem4py.shmem.**atomic_fetch_xor_nbi**(*fetch*, *target*, *value*, *pe*, *ctx=None*)

Perform bitwise XOR on `target` on PE `pe` and fetch its prior value to `fetch`.

Nonblocking. The operation is considered complete after a subsequent call to *quiet*.

Parameters

- **fetch** (*NDArray[T]*) – Local array of size 1 to be updated.
- **target** (*NDArray[T]*) – Symmetric array of size 1 containing the destination value.
- **value** (*int | float | complex | number*) – The operand to the bitwise XOR operation.
- **pe** (*int*) – The PE number on which `target` is to be updated.
- **ctx** (*Ctx | None*) – The context on which to perform the operation. If `None`, the default context is used.

Return type
None

4.9 Signaling Operations

new_signal()	Create a signal data object.
del_signal(signal)	Delete a signal data object.
signal_fetch(signal)	Fetch the signal update on a local data object.
put_signal(target, source, pe, signal, ...)	Copy local `source` to `target` on PE `pe` and update a remote flag to signal completion.
put_signal_nbi(target, source, pe, signal, ...)	Copy local `source` to `target` on PE `pe` and update a remote flag to signal completion.
SIGNAL(value[, names, module, qualname, ...])	Signal operations.

shmem4py.shmem.**new_signal**()

Create a signal data object.

Return type
SigAddr

shmem4py.shmem.**del_signal**(*signal*)

Delete a signal data object.

Parameters
signal (SigAddr) – A signal data object to be deleted.

Return type
None

shmem4py.shmem.**signal_fetch**(*signal*)

Fetch the signal update on a local data object.

Parameters
signal (SigAddr) – Local, remotely accessible signal variable.

Returns
The contents of the signal data object at the calling PE.

Return type
int

shmem4py.shmem.**put_signal**(*target*, *source*, *pe*, *signal*, *value*, *sigop*, *size=None*, *ctx=None*)

Copy local `source` to `target` on PE `pe` and update a remote flag to signal completion.

Parameters

- **target** (*NDArray[T]*) – The symmetric destination array to be updated on the remote PE.
- **source** (*NDArray[T]*) – Local array containing the data to be copied.
- **pe** (*int*) – PE number of the remote PE.
- **signal** (SigAddr) – Symmetric signal object to be updated on the remote PE as a signal.
- **value** (*int*) – The value that is used for updating the remote `signal` data object.
- **sigop** (SIGNAL) – Signal operator that represents the type of update to be performed on the remote `signal` data object.
- **size** (*int | None*) – Number of elements to copy.
- **ctx** (Ctx | *None*) – The context on which to perform the operation. If None, the default context is used.

Return type
None

shmem4py.shmem.**put_signal_nbi**(*target*, *source*, *pe*, *signal*, *value*, *sigop*, *size=None*, *ctx=None*)

Copy local `source` to `target` on PE `pe` and update a remote flag to signal completion. Nonblocking.

This routine returns after initiating the operation. The operation is considered complete after a subsequent call to *quiet*.

Parameters

- **target** (*NDArray[T]*) – The symmetric destination array to be updated on the remote PE.
- **source** (*NDArray[T]*) – Local array containing the data to be copied.

- **pe** (*int*) – PE number of the remote PE.
- **signal** (SigAddr) – Symmetric signal object to be updated on the remote PE as a signal.
- **value** (*int*) – The value that is used for updating the remote `signal` data object.
- **sigop** (SIGNAL) – Signal operator that represents the type of update to be performed on the remote `signal` data object.
- **size** (*int | None*) – Number of elements to copy.
- **ctx** (Ctx | *None*) – The context on which to perform the operation. If None, the default context is used.

Return type
None

class shmem4py.shmem.**SIGNAL**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Signal operations.

SET

An update to signal data object is an atomic set operation. It writes an unsigned 64-bit value as a signal into the signal data object on a remote PE as an atomic operation.

Type
int

ADD

An update to signal data object is an atomic add operation. It adds an unsigned 64-bit value as a signal into the signal data object on a remote PE as an atomic operation.

Type
int

4.10 Collective Operations

barrier_all()	Register the arrival of a PE at a barrier, complete updates, wait for others.
sync_all()	Register the arrival of a PE at a synchronization point, wait for all others.
sync([team])	Register the arrival of a PE at a synchronization point, wait for others.
broadcast(target, source, root[, size, team])	Copy the source from root to target on participating PEs.
collect(target, source[, size, team])	Concatenate blocks of data from multiple PEs to an array in every PE participating in the collective routine.
fcollect(target, source[, size, team])	Concatenate blocks of data from multiple PEs to an array in every PE participating in the collective routine.
alltoall(target, source[, size, team])	Exchange data elements with all other participating PEs.
alltoalls(target, source[, tst, sst, size, team])	Exchange strided data elements with all other participating PEs.
reduce(target, source[, op, size, team])	Perform a specified reduction across a set of PEs.
OP(value[, names, module, qualname, type, ...])	Reduction operation.
and_reduce(target, source[, size, team])	Perform a bitwise AND reduction across a set of PEs.
or_reduce(target, source[, size, team])	Perform a bitwise OR reduction across a set of PEs.
xor_reduce(target, source[, size, team])	Perform a bitwise exclusive OR (XOR) reduction across a set of PEs.
max_reduce(target, source[, size, team])	Perform a maximum-value reduction across a set of PEs.
min_reduce(target, source[, size, team])	Perform a minimum-value reduction across a set of PEs.
sum_reduce(target, source[, size, team])	Perform a sum reduction across a set of PEs.
prod_reduce(target, source[, size, team])	Perform a product reduction across a set of PEs.

shmem4py.shmem.**barrier_all**()

Register the arrival of a PE at a barrier, complete updates, wait for others.

This routine blocks the calling PE until all PEs have called *barrier_all*. Prior to synchronizing with other PEs, *barrier_all* ensures completion of all previously issued memory stores and remote memory updates issued on the default context.

Return type
None

shmem4py.shmem.**sync_all**()

Register the arrival of a PE at a synchronization point, wait for all others.

This routine blocks the calling PE until all PEs in the world team have called *sync_all*.

Return type
None

shmem4py.shmem.**sync**(*team=None*)

Register the arrival of a PE at a synchronization point, wait for others.

This routine does not return until all other PEs in a given team or active set arrive at this synchronization point.

Parameters
team (Team | *None*) – The team over which to perform the operation.

Return type
None

shmem4py.shmem.**broadcast**(*target*, *source*, *root*, *size=None*, *team=None*)

Copy the `source` from `root` to `target` on participating PEs.

Parameters

- **target** (*NDArray[T]*) – Symmetric destination array.
- **source** (*NDArray[T]*) – Symmetric source array.
- **root** (*int*) – PE number within the team or active set from which the data is copied.
- **size** (*int | None*) – The number of elements to be copied.
- **team** (*Team | None*) – The team over which to perform the operation.

Return type
None

shmem4py.shmem.**collect**(*target*, *source*, *size=None*, *team=None*)

Concatenate blocks of data from multiple PEs to an array in every PE participating in the collective routine.

size can vary from PE to PE; `MPI_Allgatherv` equivalent.

Performs a collective operation to concatenate `size` data items from the `source` array into the `target` array.

Parameters

- **target** (*NDArray[T]*) – Symmetric destination array large enough to accept the concatenation of the source arrays on all participating PEs.
- **source** (*NDArray[T]*) – Symmetric source array.
- **size** (*int | None*) – The number of elements to be communicated.
- **team** (*Team | None*) – The team over which to perform the operation.

Return type
None

shmem4py.shmem.**fcollect**(*target*, *source*, *size=None*, *team=None*)

Concatenate blocks of data from multiple PEs to an array in every PE participating in the collective routine.

size must be the same value in all participating PEs; `MPI_Allgather` equivalent.

Parameters

- **target** (*NDArray[T]*) – Symmetric destination array large enough to accept the concatenation of the source arrays on all participating PEs.
- **source** (*NDArray[T]*) – Symmetric source array.
- **size** (*int | None*) – The number of elements to be communicated.
- **team** (*Team | None*) – The team over which to perform the operation.

Return type
None

shmem4py.shmem.**alltoall**(*target*, *source*, *size=None*, *team=None*)

Exchange data elements with all other participating PEs.

The total size of each PE's `source` object and `target` object is `size` times the size of an element times `N`, where `N` equals the number of PEs participating in the operation. The source object contains `N` blocks of data (where the size of each block is defined by `size`) and each block of data is sent to a different PE.

Parameters

- **target** – Symmetric destination array large enough to receive the combined total of `size` elements from each PE in the active set.
- **source** – Symmetric source array that contains `size` elements of data for each PE in the active set, ordered according to destination PE.
- **size** – The number of elements to exchange for each PE.
- **team** – The team over which to perform the operation.

Return type
None

shmem4py.shmem.**alltoalls**(*target*, *source*, *tst=1*, *sst=1*, *size=None*, *team=None*)

Exchange strided data elements with all other participating PEs.

Parameters

- **target** (*NDArray[T]*) – Symmetric destination array large enough to receive the combined total of `size` elements from each PE in the active set.
- **source** (*NDArray[T]*) – Symmetric source array that contains `size` elements of data for each PE in the active set, ordered according to destination PE.
- **tst** (*int*) – The stride between consecutive elements of the `target` data object. The stride is scaled by the element size.
- **sst** (*int*) – The stride between consecutive elements of the `source` data object. The stride is scaled by the element size.
- **size** (*int | None*) – The number of elements to exchange for each PE.
- **team** (*Team | None*) – The team over which to perform the operation.

Return type
None

shmem4py.shmem.**reduce**(*target*, *source*, *op=OP_SUM*, *size=None*, *team=None*)

Perform a specified reduction across a set of PEs.

Parameters

- **target** (*NDArray[T]*) – Symmetric destination array of length at least `size` elements, where the result of the reduction routine will be stored.
- **source** (*NDArray[T]*) – Symmetric source array of length at least `size` elements, that contains one element for each separate reduction routine.
- **op** (*OP*) – The reduction operation to perform.
- **size** (*int | None*) – The number of elements to perform the reduction on.
- **team** (*Team | None*) – The team over which to perform the operation.

Return type
None

class shmem4py.shmem.**OP**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Reduction operation.

AND

Bitwise AND.

Type
str

OR

Bitwise OR.

Type
str

XOR

Bitwise XOR.

Type
str

MAX

Maximum value.

Type
str

MIN

Minimum value.

Type
str

SUM

Sum.

Type
str

PROD

Product.

Type
str

shmem4py.shmem.**and_reduce**(*target*, *source*, *size=None*, *team=None*)

Perform a bitwise AND reduction across a set of PEs.

Parameters

- **target** (*NDArray[T]*) – Symmetric destination array of length at least `size` elements, where the result of the reduction routine will be stored.
- **source** (*NDArray[T]*) – Symmetric source array of length at least `size` elements, that contains one element for each separate reduction routine.
- **size** (*int | None*) – The number of elements to perform the reduction on.
- **team** (*Team | None*) – The team over which to perform the operation.

Return type
None

shmem4py.shmem.**or_reduce**(*target*, *source*, *size=None*, *team=None*)

Perform a bitwise OR reduction across a set of PEs.

Parameters

- **target** (*NDArray[T]*) – Symmetric destination array of length at least `size` elements, where the result of the reduction routine will be stored.
- **source** (*NDArray[T]*) – Symmetric source array of length at least `size` elements, that contains one element for each separate reduction routine.
- **size** (*int | None*) – The number of elements to perform the reduction on.
- **team** (*Team | None*) – The team over which to perform the operation.

Return type
None

shmem4py.shmem.**xor_reduce**(*target*, *source*, *size=None*, *team=None*)

Perform a bitwise exclusive OR (XOR) reduction across a set of PEs.

Parameters

- **target** (*NDArray[T]*) – Symmetric destination array of length at least `size` elements, where the result of the reduction routine will be stored.
- **source** (*NDArray[T]*) – Symmetric source array of length at least `size` elements, that contains one element for each separate reduction routine.
- **size** (*int | None*) – The number of elements to perform the reduction on.
- **team** (*Team | None*) – The team over which to perform the operation.

Return type
None

shmem4py.shmem.**max_reduce**(*target*, *source*, *size=None*, *team=None*)

Perform a maximum-value reduction across a set of PEs.

Parameters

- **target** (*NDArray[T]*) – Symmetric destination array of length at least `size` elements, where the result of the reduction routine will be stored.
- **source** (*NDArray[T]*) – Symmetric source array of length at least `size` elements, that contains one element for each separate reduction routine.
- **size** (*int | None*) – The number of elements to perform the reduction on.
- **team** (*Team | None*) – The team over which to perform the operation.

Return type
None

shmem4py.shmem.**min_reduce**(*target*, *source*, *size=None*, *team=None*)

Perform a minimum-value reduction across a set of PEs.

Parameters

- **target** (*NDArray[T]*) – Symmetric destination array of length at least `size` elements, where the result of the reduction routine will be stored.
- **source** (*NDArray[T]*) – Symmetric source array of length at least `size` elements, that contains one element for each separate reduction routine.
- **size** (*int | None*) – The number of elements to perform the reduction on.
- **team** (*Team | None*) – The team over which to perform the operation.

Return type
None

shmem4py.shmem.**sum_reduce**(*target*, *source*, *size=None*, *team=None*)

Perform a sum reduction across a set of PEs.

Parameters

- **target** (*NDArray[T]*) – Symmetric destination array of length at least `size` elements, where the result of the reduction routine will be stored.
- **source** (*NDArray[T]*) – Symmetric source array of length at least `size` elements, that contains one element for each separate reduction routine.
- **size** (*int | None*) – The number of elements to perform the reduction on.
- **team** (*Team | None*) – The team over which to perform the operation.

Return type

None

shmem4py.shmem.**prod_reduce**(*target*, *source*, *size=None*, *team=None*)

Perform a product reduction across a set of PEs.

Parameters

- **target** (*NDArray[T]*) – Symmetric destination array of length at least `size` elements, where the result of the reduction routine will be stored.
- **source** (*NDArray[T]*) – Symmetric source array of length at least `size` elements, that contains one element for each separate reduction routine.
- **size** (*int | None*) – The number of elements to perform the reduction on.
- **team** (*Team | None*) – The team over which to perform the operation.

Return type

None

4.11 Point-To-Point Synchronization

wait_until(ivar, cmp, value)	Wait until a variable satisfies a condition.
wait_until_all(ivars, cmp, value[, status])	Wait until all variables satisfy a condition.
wait_until_any(ivars, cmp, value[, status])	Wait until any one variable satisfies a condition.
wait_until_some(ivars, cmp, value[, status])	Wait until at least one variable satisfies a condition.
wait_until_all_vector(ivars, cmp, values[, ...])	Wait until all variables satisfy the specified conditions.
wait_until_any_vector(ivars, cmp, values[, ...])	Wait until any one variable satisfies the specified conditions.
wait_until_some_vector(ivars, cmp, values[, ...])	Wait until at least one variable satisfies the specified conditions.
test(ivar, cmp, value)	Indicate whether a variable on the local PE meets a condition.
test_all(ivars, cmp, value[, status])	Indicate whether all variables on the local PE meet a condition.
test_any(ivars, cmp, value[, status])	Indicate whether any one variable on the local PE meets a condition.
test_some(ivars, cmp, value[, status])	Indicate whether at least one variable on the local PE meets a condition.
test_all_vector(ivars, cmp, values[, status])	Indicate whether all variables on the local PE meet the specified conditions.
test_any_vector(ivars, cmp, values[, status])	Indicate whether any one variable on the local PE meets its specified condition.
test_some_vector(ivars, cmp, values[, status])	Indicate whether at least one variable on the local PE meets its specified condition.
signal_wait_until(signal, cmp, value)	Wait for a variable on the local PE to change from a signaling operation.
CMP(value[, names, module, qualname, type, ...])	Comparison operator.

shmem4py.shmem.**wait_until**(*ivar*, *cmp*, *value*)

Wait until a variable satisfies a condition.

Blocks until the value `ivar` satisfies the condition `ivar cmp value` at the calling PE, where `cmp` is the comparison operator.

Parameters

- **ivar** (*NDArray[Any]*) – Symmetric array of size `1` containing the element that will be compared.
- **cmp** (*CMP*) – The comparison operator that compares `ivar` with `value`.
- **value** (*int | float | complex | number*) – The value to be compared with `ivar`.

Return type

None

shmem4py.shmem.**wait_until_all**(*ivars*, *cmp*, *value*, *status=None*)

Wait until all variables satisfy a condition.

Blocks until all values specified in `ivars` not excluded by `status` satisfy the condition `ivars[i] cmp value` at the calling PE, where `cmp` is the comparison operator.

Parameters

- **ivars** (*NDArray[Any]*) – Symmetric array of objects to be compared.

- **cmp** (CMP) – The comparison operator that compares elements of `ivars` with `value`.
- **value** (*int | float | complex | number*) – The value to be compared with elements of `ivars`.
- **status** (*Sequence[int] | None*) – An optional mask array of length `len(ivars)` indicating which elements of `ivars` are excluded from the wait set. Nonzero values exclude the corresponding element from the wait set.

Return type
None

shmem4py.shmem.**wait_until_any**(*ivars*, *cmp*, *value*, *status=None*)

Wait until any one variable satisfies a condition.

Blocks until any one entry in the wait set specified by `ivars` not excluded by `status` satisfies the condition `ivars[i] cmp value` at the calling PE, where `cmp` is the comparison operator.

Parameters

- **ivars** (*NDArray[Any]*) – Symmetric array of objects to be compared.
- **cmp** (CMP) – The comparison operator that compares elements of `ivars` with `value`.
- **value** (*int | float | complex | number*) – The value to be compared with elements of `ivars`.
- **status** (*Sequence[int] | None*) – An optional mask array of length `len(ivars)` indicating which elements of `ivars` are excluded from the wait set. Nonzero values exclude the corresponding element from the wait set.

Returns
The index of entry `i` of `ivars` that satisfies the condition.

Return type
int | None

shmem4py.shmem.**wait_until_some**(*ivars*, *cmp*, *value*, *status=None*)

Wait until at least one variable satisfies a condition.

Blocks until at least one entry in the wait set specified by `ivars` not excluded by `status` satisfies the condition `ivars[i] cmp value` at the calling PE, where `cmp` is the comparison operator.

Parameters

- **ivars** (*NDArray[Any]*) – Symmetric array of objects to be compared.
- **cmp** (CMP) – The comparison operator that compares elements of `ivars` with `value`.
- **value** (*int | float | complex | number*) – The value to be compared with elements of `ivars`.
- **status** (*Sequence[int] | None*) – An optional mask array of length `len(ivars)` indicating which elements of `ivars` are excluded from the wait set. Nonzero values exclude the corresponding element from the wait set.

Returns
Indices of entries of `ivars` that satisfy the condition.

Return type
List[int]

shmem4py.shmem.**wait_until_all_vector**(*ivars*, *cmp*, *values*, *status=None*)

Wait until all variables satisfy the specified conditions.

Blocks until all values specified in `ivars` not excluded by `status` satisfy the condition `ivars[i] cmp values[i]` at the calling PE, where `cmp` is the comparison operator.

Parameters

- **ivars** (*NDArray[Any]*) – Symmetric array of objects to be compared.
- **cmp** (CMP) – The comparison operator that compares elements of `ivars` with the elements of `values`.
- **values** (*Sequence[int | float | complex | number]*) – Local array containing values to be compared with the respective elements of `ivars`.
- **status** (*Sequence[int] | None*) – An optional mask array of length `len(ivars)` indicating which elements of `ivars` are excluded from the wait set. Nonzero values exclude the corresponding element from the wait set.

Return type
None

shmem4py.shmem.**wait_until_any_vector**(*ivars*, *cmp*, *values*, *status=None*)

Wait until any one variable satisfies the specified conditions.

Blocks until any one value specified in `ivars` not excluded by `status` satisfies the condition `ivars[i] cmp values[i]` at the calling PE, where `cmp` is the comparison operator.

Parameters

- **ivars** (*NDArray[Any]*) – Symmetric array of objects to be compared.
- **cmp** (CMP) – The comparison operator that compares elements of `ivars` with the elements of `values`.
- **values** (*Sequence[int | float | complex | number]*) – Local array containing values to be compared with the respective elements of `ivars`.
- **status** (*Sequence[int] | None*) – An optional mask array of length `len(ivars)` indicating which elements of `ivars` are excluded from the wait set. Nonzero values exclude the corresponding element from the wait set.

Returns
The index of entry `i` of `ivars` that satisfies the condition.

Return type
int | None

shmem4py.shmem.**wait_until_some_vector**(*ivars*, *cmp*, *values*, *status=None*)

Wait until at least one variable satisfies the specified conditions.

Blocks until any one value specified in `ivars` not excluded by `status` satisfies the condition `ivars[i] cmp values[i]` at the calling PE, where `cmp` is the comparison operator.

Parameters

- **ivars** (*NDArray[Any]*) – Symmetric array of objects to be compared.
- **cmp** (CMP) – The comparison operator that compares elements of `ivars` with the elements of `values`.
- **values** (*Sequence[int | float | complex | number]*) – Local array containing values to be compared with the respective elements of `ivars`.

- **status** (*Sequence[int] | None*) – An optional mask array of length `len(ivars)` indicating which elements of `ivars` are excluded from the wait set. Nonzero values exclude the corresponding element from the wait set.

Returns
Indices of entries of `ivars` that satisfy the condition.

Return type
List[int]

shmem4py.shmem.**test**(*ivar*, *cmp*, *value*)

Indicate whether a variable on the local PE meets a condition.

Parameters

- **ivar** (*NDArray[Any]*) – Symmetric array of size `1` containing the element that will be tested.
- **cmp** (*CMP*) – The comparison operator that compares `ivar` with `value`.
- **value** (*int | float | complex | number*) – The value to be compared with `ivar`.

Return type
bool

shmem4py.shmem.**test_all**(*ivars*, *cmp*, *value*, *status=None*)

Indicate whether all variables on the local PE meet a condition.

Parameters

- **ivars** (*NDArray[Any]*) – Symmetric array of objects to be tested.
- **cmp** (*CMP*) – The comparison operator that compares elements of `ivars` with `value`.
- **value** (*int | float | complex | number*) – The value to be compared with elements of `ivars`.
- **status** (*Sequence[int] | None*) – An optional mask array of length `len(ivars)` indicating which elements of `ivars` are excluded from the test set. Nonzero values exclude the corresponding element from the test set.

Return type
bool

shmem4py.shmem.**test_any**(*ivars*, *cmp*, *value*, *status=None*)

Indicate whether any one variable on the local PE meets a condition.

Parameters

- **ivars** (*NDArray[Any]*) – Symmetric array of objects to be tested.
- **cmp** (*CMP*) – The comparison operator that compares elements of `ivars` with `value`.
- **value** (*int | float | complex | number*) – The value to be compared with elements of `ivars`.
- **status** (*Sequence[int] | None*) – An optional mask array of length `len(ivars)` indicating which elements of `ivars` are excluded from the test set. Nonzero values exclude the corresponding element from the test set.

Returns
The index of entry `i` of `ivars` that satisfies the condition.

Return type
int | None

shmem4py.shmem.**test_some**(*ivars*, *cmp*, *value*, *status=None*)

Indicate whether at least one variable on the local PE meets a condition.

Parameters

- **ivars** (*NDArray[Any]*) – Symmetric array of objects to be tested.
- **cmp** (CMP) – The comparison operator that compares elements of `ivars` with `value`.
- **value** (*int | float | complex | number*) – The value to be compared with elements of `ivars`.
- **status** (*Sequence[int] | None*) – An optional mask array of length `len(ivars)` indicating which elements of `ivars` are excluded from the test set. Nonzero values exclude the corresponding element from the test set.

Returns

Indices of entries of `ivars` that satisfy the condition.

Return type

List[int]

shmem4py.shmem.**test_all_vector**(*ivars*, *cmp*, *values*, *status=None*)

Indicate whether all variables on the local PE meet the specified conditions.

Parameters

- **ivars** (*NDArray[Any]*) – Symmetric array of objects to be tested.
- **cmp** (CMP) – The comparison operator that compares elements of `ivars` with the elements of `values`.
- **values** (*Sequence[int | float | complex | number]*) – Local array containing values to be compared with the respective elements of `ivars`.
- **status** (*Sequence[int] | None*) – An optional mask array of length `len(ivars)` indicating which elements of `ivars` are excluded from the test set. Nonzero values exclude the corresponding element from the test set.

Return type

bool

shmem4py.shmem.**test_any_vector**(*ivars*, *cmp*, *values*, *status=None*)

Indicate whether any one variable on the local PE meets its specified condition.

Parameters

- **ivars** (*NDArray[Any]*) – Symmetric array of objects to be tested.
- **cmp** (CMP) – The comparison operator that compares elements of `ivars` with the elements of `values`.
- **values** (*Sequence[int | float | complex | number]*) – Local array containing values to be compared with the respective elements of `ivars`.
- **status** (*Sequence[int] | None*) – An optional mask array of length `len(ivars)` indicating which elements of `ivars` are excluded from the test set. Nonzero values exclude the corresponding element from the test set.

Returns

The index of entry `i` of `ivars` that satisfies the condition.

Return type

int | None

shmem4py.shmem.**test_some_vector**(*ivars*, *cmp*, *values*, *status=None*)

Indicate whether at least one variable on the local PE meets its specified condition.

Parameters

- **ivars** (*NDArray[Any]*) – Symmetric array of objects to be tested.
- **cmp** (CMP) – The comparison operator that compares elements of `ivars` with the elements of `values`.
- **values** (*Sequence[int | float | complex | number]*) – Local array containing values to be compared with the respective elements of `ivars`.
- **status** (*Sequence[int] | None*) – An optional mask array of length `len(ivars)` indicating which elements of `ivars` are excluded from the test set. Nonzero values exclude the corresponding element from the test set.

Returns

Indices of entries of `ivars` that satisfy the condition.

Return type

List[int]

shmem4py.shmem.**signal_wait_until**(*signal*, *cmp*, *value*)

Wait for a variable on the local PE to change from a signaling operation.

Parameters

- **signal** (SigAddr) – Local symmetric source signal variable.
- **cmp** (CMP) – The comparison operator that compares `signal` with `value`.
- **value** (*int | float | complex | number*) – The value against which the object pointed to by `signal` will be compared.

Returns

The contents of the signal data object, `signal`, at the calling PE that satisfies the wait condition.

Return type

int

class shmem4py.shmem.**CMP**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Comparison operator.

EQ

Equal to.

Type

int

NE

Not equal to.

Type

int

GT

Greater than.

Type

int

LE

Less than or equal to.

Type
int

LT

Less than.

Type
int

GE

Greater than or equal to.

Type
int

4.12 Memory Ordering

fence([ctx])	Ensure ordering of delivery of operations on symmetric data objects.
quiet([ctx])	Wait for completion of outstanding operations on symmetric data objects issued by a PE.

shmem4py.shmem.**fence**(*ctx=None*)

Ensure ordering of delivery of operations on symmetric data objects.

All operations on symmetric data objects issued to a particular PE on the given context prior to the call to *fence* are guaranteed to be delivered before any subsequent operations on symmetric data objects to the same PE.

Parameters
ctx (Ctx | *None*) – A context handle specifying the context on which to perform the operation. If None, defaults to the default context.

Return type
None

shmem4py.shmem.**quiet**(*ctx=None*)

Wait for completion of outstanding operations on symmetric data objects issued by a PE.

Ensures completion of all operations on symmetric data objects issued by the calling PE on the given context.

Parameters
ctx (Ctx | *None*) – A context handle specifying the context on which to perform the operation. If None, defaults to the default context.

Return type
None

4.13 Distributed Locking

new_lock()	Create a lock object.
del_lock(lock)	Delete a lock object.
set_lock(lock)	Acquire a mutual exclusion lock after waiting for the lock to be freed.
test_lock(lock)	Acquire a mutual exclusion lock only if it is currently cleared.
clear_lock(lock)	Release a lock previously set by *set_lock* or *test_lock*.

shmem4py.shmem.**new_lock**()

Create a lock object.

Return type
LockHandle

shmem4py.shmem.**del_lock**(*lock*)

Delete a lock object.

Parameters
lock (LockHandle) – A lock object to be deleted.

Return type
None

shmem4py.shmem.**set_lock**(*lock*)

Acquire a mutual exclusion lock after waiting for the lock to be freed.

Parameters
lock (LockHandle) – Symmetric scalar variable or an array of length 1.

Return type
None

shmem4py.shmem.**test_lock**(*lock*)

Acquire a mutual exclusion lock only if it is currently cleared.

By using this routine, a PE can avoid blocking on a set lock.

Parameters
lock (LockHandle) – Symmetric scalar variable or an array of length 1.

Returns
Returns `False` if the lock was originally cleared and this call was able to acquire the lock. `True` is returned if the lock had been set and the call returned without waiting to set the lock.

Return type
bool

shmem4py.shmem.**clear_lock**(*lock*)

Release a lock previously set by *set_lock* or *test_lock*.

Releases a lock after performing a *quiet* operation on the default context to ensure that all symmetric memory accesses that occurred during the critical region are complete.

Parameters
lock (LockHandle) – Symmetric scalar variable or an array of length 1.

Return type
None

4.14 Distributed Locking (Object-Oriented)

Lock()	Lock object.
Lock.destroy()	Destroy the lock object.
Lock.acquire([blocking])	Acquire the lock.
Lock.release()	Release the lock.

class shmem4py.shmem.**Lock**

Lock object.

destroy()

Destroy the lock object.

Return type
None

acquire(*blocking=True*)

Acquire the lock.

Parameters
blocking (*bool*) – `True` to wait until the lock is acquired.

Returns
If `blocking` is `True`, waits and returns `True` once the lock has been acquired. If `blocking` is `False`, returns `True` if the lock has been acquired and `False` otherwise (i.e., lock was already set).

Return type
bool

release()

Release the lock.

Releases a lock after performing a *quiet* operation on the default context to ensure that all symmetric memory accesses that occurred during the critical region are complete.

Return type
None

4.15 Profiling Control

pcontrol([level])	Set the profiling level.

shmem4py.shmem.**pcontrol**(*level=1*)

Set the profiling level.

Parameters
level (*int*) – The profiling level.

Return type
None

4.16 Typing Support

shmem4py.shmem.**Number**
Numeric type.

alias of int | float | complex | number

shmem4py.shmem.**SigAddr = shmem4py.shmem.SigAddr**
Signal address.

shmem4py.shmem.**CtxHandle = shmem4py.shmem.CtxHandle**
Context handle.

shmem4py.shmem.**TeamHandle = shmem4py.shmem.TeamHandle**
Team handle.

shmem4py.shmem.**LockHandle = shmem4py.shmem.LockHandle**
Lock handle.

ffi.**CData**
See ffi.CData

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

A

acquire() (*shmem4py.shmem.Lock method*), 57
ADD (*shmem4py.shmem.AMO attribute*), 33
ADD (*shmem4py.shmem.SIGNAL attribute*), 42
addr_accessible() (*in module shmem4py.shmem*), 21
alloc() (*in module shmem4py.shmem*), 22
alltoall() (*in module shmem4py.shmem*), 44
alltoalls() (*in module shmem4py.shmem*), 45
AMO (*class in shmem4py.shmem*), 33
AND (*shmem4py.shmem.AMO attribute*), 33
AND (*shmem4py.shmem.OP attribute*), 45
and_reduce() (*in module shmem4py.shmem*), 46
array() (*in module shmem4py.shmem*), 23
atomic_add() (*in module shmem4py.shmem*), 34
atomic_and() (*in module shmem4py.shmem*), 35
atomic_compare_swap() (*in module shmem4py.shmem*), 36
atomic_compare_swap_nbi() (*in module shmem4py.shmem*), 38
atomic_fetch() (*in module shmem4py.shmem*), 35
atomic_fetch_add() (*in module shmem4py.shmem*), 36
atomic_fetch_add_nbi() (*in module shmem4py.shmem*), 39
atomic_fetch_and() (*in module shmem4py.shmem*), 37
atomic_fetch_and_nbi() (*in module shmem4py.shmem*), 39
atomic_fetch_inc() (*in module shmem4py.shmem*), 36
atomic_fetch_inc_nbi() (*in module shmem4py.shmem*), 39
atomic_fetch_nbi() (*in module shmem4py.shmem*), 38
atomic_fetch_op() (*in module shmem4py.shmem*), 32
atomic_fetch_op_nbi() (*in module shmem4py.shmem*), 33
atomic_fetch_or() (*in module shmem4py.shmem*), 37
atomic_fetch_or_nbi() (*in module shmem4py.shmem*), 40
atomic_fetch_xor() (*in module shmem4py.shmem*), 37
atomic_fetch_xor_nbi() (*in module shmem4py.shmem*), 40
atomic_inc() (*in module shmem4py.shmem*), 34
atomic_op() (*in module shmem4py.shmem*), 32
atomic_or() (*in module shmem4py.shmem*), 35
atomic_set() (*in module shmem4py.shmem*), 34
atomic_swap() (*in module shmem4py.shmem*), 36
atomic_swap_nbi() (*in module shmem4py.shmem*), 38
atomic_xor() (*in module shmem4py.shmem*), 35
ATOMICS_REMOTE (*shmem4py.shmem.MALLOC attribute*), 25

B

barrier_all() (*in module shmem4py.shmem*), 43
broadcast() (*in module shmem4py.shmem*), 43

C

clear_lock() (*in module shmem4py.shmem*), 56
CMP (*class in shmem4py.shmem*), 54
collect() (*in module shmem4py.shmem*), 44
create() (*shmem4py.shmem.Ctx static method*), 28
create_ctx() (*shmem4py.shmem.Team method*), 27
CTX (*class in shmem4py.shmem*), 28
Ctx (*class in shmem4py.shmem*), 27
CtxHandle (*in module shmem4py.shmem*), 58

D

del_array() (*in module shmem4py.shmem*), 23
del_lock() (*in module shmem4py.shmem*), 56
del_signal() (*in module shmem4py.shmem*), 41
destroy() (*shmem4py.shmem.Ctx method*), 28
destroy() (*shmem4py.shmem.Lock method*), 57
destroy() (*shmem4py.shmem.Team method*), 26

E

empty() (*in module shmem4py.shmem*), 24
EQ (*shmem4py.shmem.CMP attribute*), 54

F

fcollect() (*in module shmem4py.shmem*), 44
fence() (*in module shmem4py.shmem*), 55

fence() (*shmem4py.shmem.Ctx method*), 28
ffi.CData (*in module shmem4py.shmem*), 58
finalize() (*in module shmem4py.shmem*), 19
free() (*in module shmem4py.shmem*), 22
fromalloc() (*in module shmem4py.shmem*), 22
full() (*in module shmem4py.shmem*), 25
FUNNELED (*shmem4py.shmem.THREAD attribute*), 20

G

GE (*shmem4py.shmem.CMP attribute*), 55
get() (*in module shmem4py.shmem*), 29
get_config() (*shmem4py.shmem.Team method*), 26
get_nbi() (*in module shmem4py.shmem*), 31
get_team() (*shmem4py.shmem.Ctx method*), 28
global_exit() (*in module shmem4py.shmem*), 20
GT (*shmem4py.shmem.CMP attribute*), 54

I

iget() (*in module shmem4py.shmem*), 30
INC (*shmem4py.shmem.AMO attribute*), 33
info_get_name() (*in module shmem4py.shmem*), 19
info_get_version() (*in module shmem4py.shmem*), 19
init() (*in module shmem4py.shmem*), 19
init_thread() (*in module shmem4py.shmem*), 20
iput() (*in module shmem4py.shmem*), 30

L

LE (*shmem4py.shmem.CMP attribute*), 54
Lock (*class in shmem4py.shmem*), 57
LockHandle (*in module shmem4py.shmem*), 58
LT (*shmem4py.shmem.CMP attribute*), 55

M

MALLOC (*class in shmem4py.shmem*), 25
MAX (*shmem4py.shmem.OP attribute*), 46
max_reduce() (*in module shmem4py.shmem*), 47
MIN (*shmem4py.shmem.OP attribute*), 46
min_reduce() (*in module shmem4py.shmem*), 47
MULTIPLE (*shmem4py.shmem.THREAD attribute*), 20
my_pe() (*in module shmem4py.shmem*), 21
my_pe() (*shmem4py.shmem.Team method*), 26

N

n_pes() (*in module shmem4py.shmem*), 21
n_pes() (*shmem4py.shmem.Team method*), 27
NE (*shmem4py.shmem.CMP attribute*), 54
new_array() (*in module shmem4py.shmem*), 23
new_lock() (*in module shmem4py.shmem*), 56
new_signal() (*in module shmem4py.shmem*), 40
NOSTORE (*shmem4py.shmem.CTX attribute*), 29
Number (*in module shmem4py.shmem*), 58

O

ones() (*in module shmem4py.shmem*), 24
OP (*class in shmem4py.shmem*), 45
OR (*shmem4py.shmem.AMO attribute*), 34
OR (*shmem4py.shmem.OP attribute*), 46
or_reduce() (*in module shmem4py.shmem*), 46

P

pcontrol() (*in module shmem4py.shmem*), 57
pe_accessible() (*in module shmem4py.shmem*), 21
PRIVATE (*shmem4py.shmem.CTX attribute*), 28
PROD (*shmem4py.shmem.OP attribute*), 46
prod_reduce() (*in module shmem4py.shmem*), 48
ptr() (*in module shmem4py.shmem*), 21
put() (*in module shmem4py.shmem*), 29
put_nbi() (*in module shmem4py.shmem*), 30
put_signal() (*in module shmem4py.shmem*), 41
put_signal_nbi() (*in module shmem4py.shmem*), 41

Q

query_thread() (*in module shmem4py.shmem*), 20
quiet() (*in module shmem4py.shmem*), 55
quiet() (*shmem4py.shmem.Ctx method*), 28

R

reduce() (*in module shmem4py.shmem*), 45
release() (*shmem4py.shmem.Lock method*), 57

S

SERIALIZED (*shmem4py.shmem.CTX attribute*), 29
SERIALIZED (*shmem4py.shmem.THREAD attribute*), 20
SET (*shmem4py.shmem.AMO attribute*), 33
SET (*shmem4py.shmem.SIGNAL attribute*), 42
set_lock() (*in module shmem4py.shmem*), 56
SigAddr (*in module shmem4py.shmem*), 58
SIGNAL (*class in shmem4py.shmem*), 42
signal_fetch() (*in module shmem4py.shmem*), 41
SIGNAL_REMOTE (*shmem4py.shmem.MALLOC attribute*), 25
signal_wait_until() (*in module shmem4py.shmem*), 54
SINGLE (*shmem4py.shmem.THREAD attribute*), 20
split_strided() (*shmem4py.shmem.Team method*), 26
SUM (*shmem4py.shmem.OP attribute*), 46
sum_reduce() (*in module shmem4py.shmem*), 47
sync() (*in module shmem4py.shmem*), 43
sync() (*shmem4py.shmem.Team method*), 27
sync_all() (*in module shmem4py.shmem*), 43

T

Team (*class in shmem4py.shmem*), 26
TeamHandle (*in module shmem4py.shmem*), 58
test() (*in module shmem4py.shmem*), 52

test_all() (*in module shmem4py.shmem*), 52
test_all_vector() (*in module shmem4py.shmem*), 53
test_any() (*in module shmem4py.shmem*), 52
test_any_vector() (*in module shmem4py.shmem*), 53
test_lock() (*in module shmem4py.shmem*), 56
test_some() (*in module shmem4py.shmem*), 52
test_some_vector() (*in module shmem4py.shmem*), 53
THREAD (*class in shmem4py.shmem*), 20
translate_pe() (*shmem4py.shmem.Team method*), 27

W

wait_until() (*in module shmem4py.shmem*), 49
wait_until_all() (*in module shmem4py.shmem*), 49
wait_until_all_vector() (*in module shmem4py.shmem*), 50
wait_until_any() (*in module shmem4py.shmem*), 50
wait_until_any_vector() (*in module shmem4py.shmem*), 51
wait_until_some() (*in module shmem4py.shmem*), 50
wait_until_some_vector() (*in module shmem4py.shmem*), 51

X

XOR (*shmem4py.shmem.AMO attribute*), 34
XOR (*shmem4py.shmem.OP attribute*), 46
xor_reduce() (*in module shmem4py.shmem*), 47

Z

zeros() (*in module shmem4py.shmem*), 24