Memory Ordering Operations

- As most of the operations are not synchronizing there is a need for enforcing ordering
  - Basically a remote happen-before type of relationship between code blocks
  - void shmem_quiet(void): wait for completion of all outstanding Put, AMO and store operation issues by the PE
  - void shmem_fence(void): assure ordering of delivery of Put, AMO and store operations. All operation prior to the call to shmem_fence are guaranteed to be ordered to be delivered before any subsequent Put, AMO or store operation.
- Beware: the meaning of these synchronizations are purely local (i.e. barriers are needed for global scope)

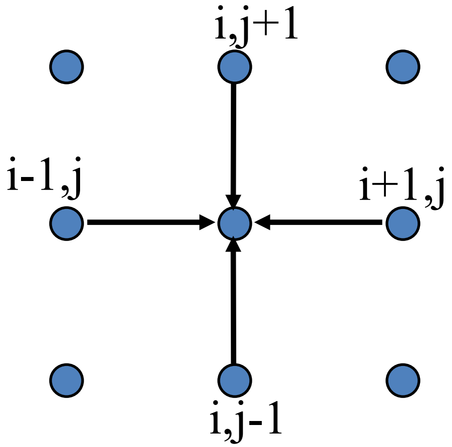
Example

```
#include <stdio.h>
#include <shmem.h>

long target[10] = {0};
int targ = 0;
int main(void)
{
    long source[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int src = 99;
    start_pes(0);
    if (_my_pe() == 0) {
        shmem_long_put(target, source, 10, 1); /*put1*/
        shmem_long_put(target, source, 10, 2); /*put2*/
        shmem_fence();
        shmem_int_put(&targ, &src, 1, 1); /*put3*/
        shmem_int_put(&targ, &src, 1, 2); /*put4*/
    }
    shmem_barrier_all(); /* sync sender and receiver */
    printf("target[0] on PE %d is %d\n", _my_pe(), target[0]);
    return 1;
}
```

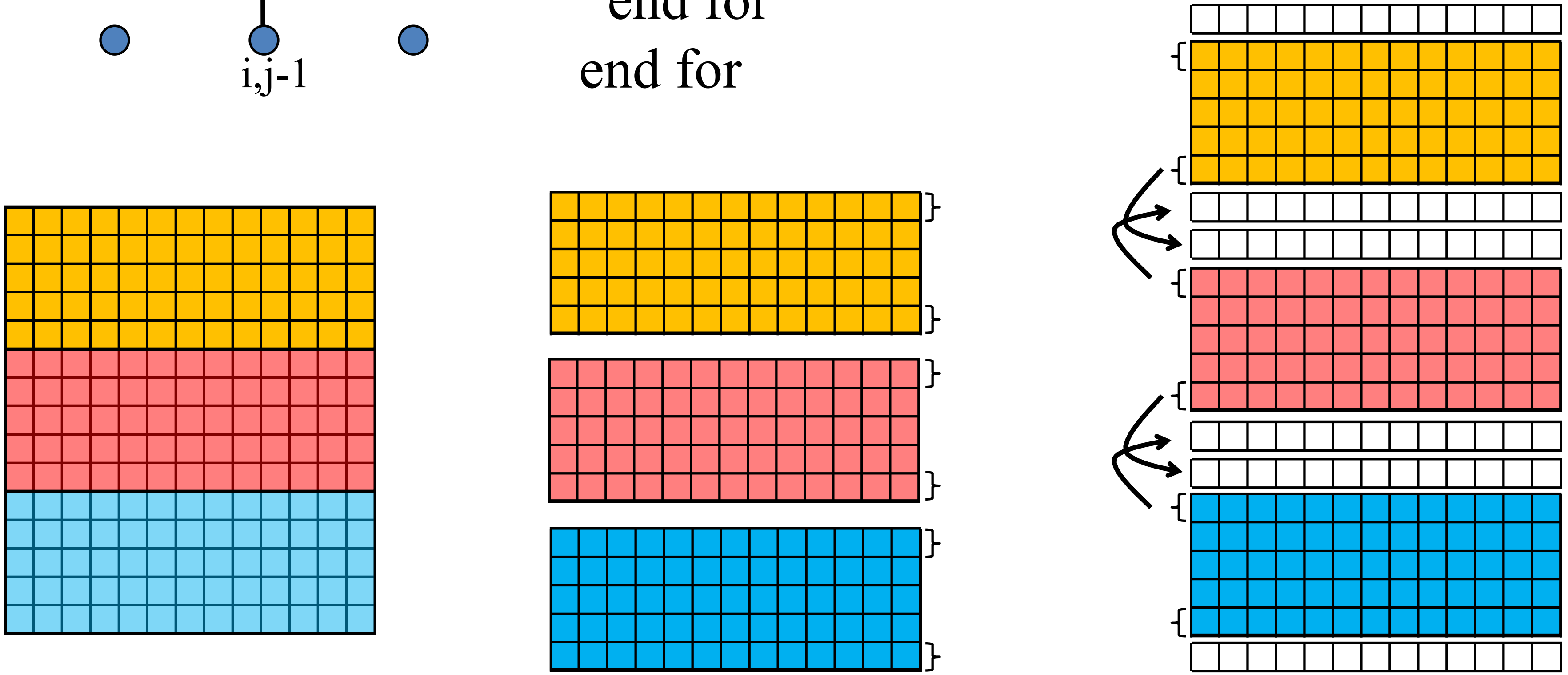
$$U_{i,j}^{n+1} = \frac{1}{4}\left(U_{i-1,j}^{n} + U_{i+1}^{n} + U_{i,j-1}^{n} + U_{i,j+1}^{n}\right)$$

Laplace's equation – OpenSHMEM



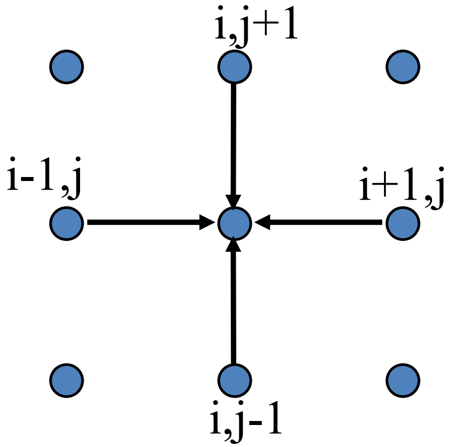
```
for j = 1 to jmax
  for i = 1 to imax
    Unew(i,j) = 0.25 * ( U(i-1,j) + U(i+1,j)
                     +   U(i,j-1) + U(i,j+1))

  end for
end for
```



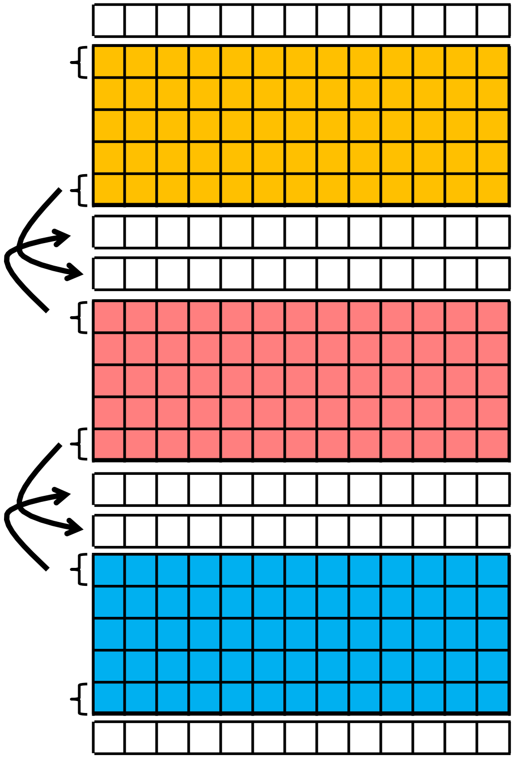
$$U_{i,j}^{n+1} = \frac{1}{4}\left(U_{i-1,j}^{n} + U_{i+1}^{n} + U_{i,j-1}^{n} + U_{i,j+1}^{n}\right)$$

Laplace’s equation – OpenSHMEM



```
for j = 1 to jmax
  for i = 1 to imax
    Unew(i,j) = 0.25 * ( U(i-1,j) + U(i+1,j)
                    +   U(i,j-1) + U(i,j+1))
  end for
end for
```

- How to implement without using global barriers ?
- Any particular issues due to synchronizations ?
- How to decrease synchronization pressure ?



Atomic Memory Operations (AMO)

- One-sided mechanism that combines memory update operations with atomicity guarantee
- Two types of AMO routines:
  - Non-fetch: update the remote memory in a single atomic operation. No completion is imposed as there is no local return value related to the operation.
  - Fetch-and-operate: combine memory update and fetch operations in a single atomic operation.The routines return after the data has been fetched and locally delivered.

AMO: fetch: CSWAP

- <type> shmem_<type>_cswap(<type>* target, <type> cond, <type>value, int pe);
  - type: int, long, longlong
  - The function returns the old value of *target
  - Target: remotely accessible integer data object to be updated
  - Cond: the value to be compared with. If the remote target and the cond value are equal, then value is swaped into the remote target. Otherwise, the remote target is unchanged.

AMO: fetch: SWAP

- <type> shmem_<type>_swap(<type>* target, <type>value, int pe);
  - type: float, double, int, long, longlong
  - The function returns the old value of *target
  - Target: remotely accessible integer data object to be updated
  - the remote target is swaped with value into the remote target

AMO: fetch: FINC, FADD

- <type> shmem_<type>_finc(<type> *target,
  int pe);
- <type> shmem_<type>_fadd(<type> *target,
  <type> value, int pe);
  - Atomic fetch-and-increment/add on the remote data object with 1/value
  - Returns the previous value in *target

AMO: non-fetch: INC, ADD

- void shmem_<type>_inc(<type> *target,
  int pe);
- void shmem_<type>_add(<type> *target,
  <type> value, int pe);
  - Atomic increment/add on the remote data object with 1/value
  - Returns … nothing

Locking Routines

- Similar to mutexes but for distributed settings
  - Work in First Come First serve mode
- **void** shmem_clear_lock(**volatile long** *lock);
  - Release the owned lock
- **void** shmem_set_lock(**volatile long** *lock);
  - Acquire the lock, blocks until the lock has been released by the prior owner and succesfully acquired by the PE
- **int** shmem_test_lock(**volatile long** *lock);
  - Return 1 if the lock is currently owned by another PE. Otherwise the lock is acquired and the return is 0.

Example

```
#include <shmem.h>

long L = 0;


int main(int argc, char **argv) {
  int me, slp = 1;
  shmem_init();
  me = shmem_my_pe();
  shmem_barrier_all();

  if (me == 1)
    sleep (3);

  shmem_set_lock(&L);
  printf("%d: sleeping %d second%s...\n", me, slp, slp == 1 ? "" : "s");
  sleep(slp);
  printf("%d: sleeping...done\n", me);
  shmem_clear_lock(&L);
  shmem_barrier_all();
  return 0; }
```