Non-blocking RMA operations

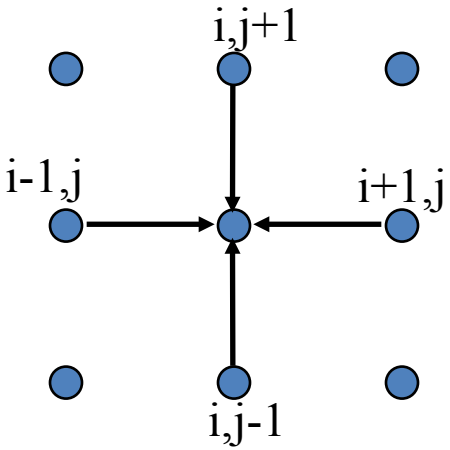
- Add _nbi (_NBI in Fortran) to any PUT and GET call
  - The transfer order is issued, but no assumptions about the data transfers should be made until the next ***shmem_quiet***.
  - No order between operations is enforced in the absence of more specific synchronizations (such as fence).

Remote Memory Access

- Put vs. Get
  - Put call completes when data is “being sent”
  - Get call completes when data is “stored locally”
- Cannot assume put data has been transferred until later synchronization
  - Data still in transit
  - Partially written at target
  - The delivery of words in a put operation can happen in any order
- Puts allow overlap
  - Communicate / Compute / Synchronize

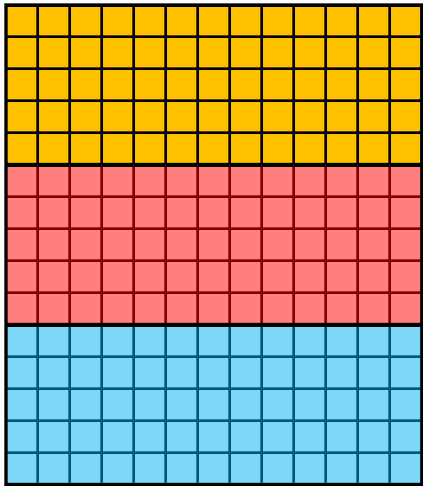
$$U_{i,j}^{n+1} = \frac{1}{4}\left(U_{i-1,j}^{n} + U_{i+1}^{n} + U_{i,j-1}^{n} + U_{i,j+1}^{n}\right)$$

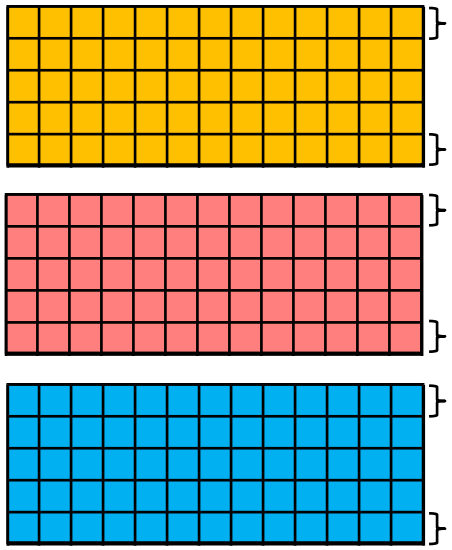
Laplace's equation – OpenSHMEM

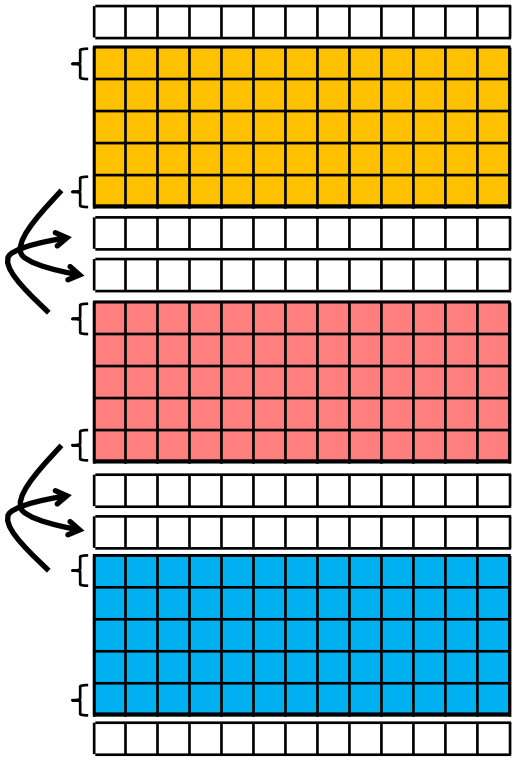



```
for j = 1 to jmax
  for i = 1 to imax
    Unew(i,j) = 0.25 * ( U(i-1,j) + U(i+1,j)
                       +   U(i,j-1) + U(i,j+1))

  end for
end for
```

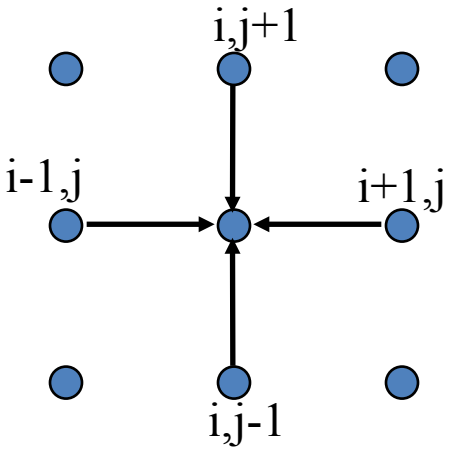






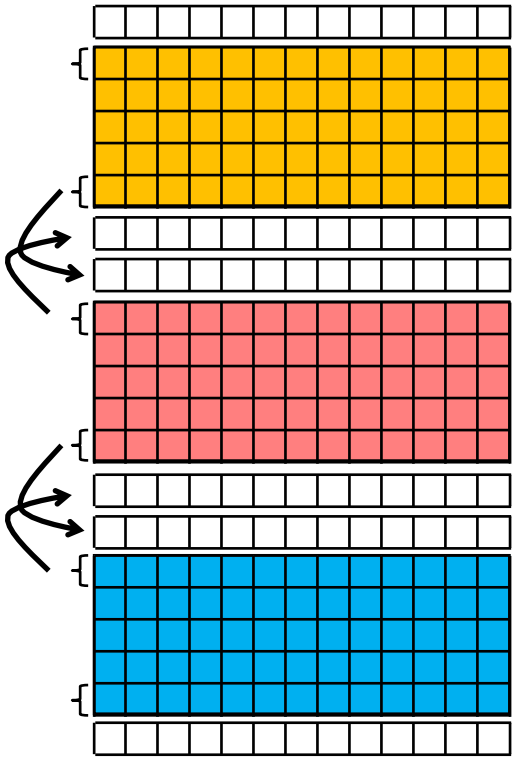
$$U_{i,j}^{n+1} = \frac{1}{4}\left(U_{i-1,j}^{n} + U_{i+1}^{n} + U_{i,j-1}^{n} + U_{i,j+1}^{n}\right)$$

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
```

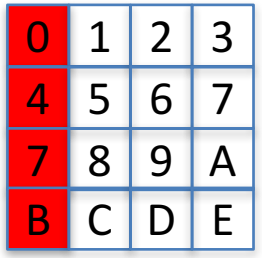
- How to implement using only PUT operations ?
- How to implement using only GET operations ?
- What is the main factor limiting performance ?



Collective Operations: Barrier_all

- void shmem_barrier_all(void)
  - Barrier between all PE. All operations issued before the barrier are completed upon return.
  - This operation complete al remote shmem_<type>_add and put.

Active Sets



- What if not all processes want to be involved in an operation ?
  - Think 2D matrices where collective behavior is desired by line or by column
- It provides a regular definition of a group of processes
  - Composed by a tuple
    (start, log stride[power of 2], size)
  - PE_start = 0, logPE_stride = 0, PE_size = 4
    Set: PE0, PE1, PE2, PE3
  - PE_start = 0, logPE_stride = 1, PE_size = 4
    Set: PE0, PE2, PE4, PE6
  - PE_start = 2, logPE_stride = 2, PE_size = 3
    Set: PE2, PE6, PE10
  - {PEx, where x = PE_start + k * 2 ^ logPE_stride,
    with k = 0 .. PE_size}

Collective Operations: Barrier

- void shmem_barrier(int PE_start,
  int logPE_stride, int PE_size, long *pSync)
- Define a barrier on a log (base 2) group of PE
- pSync: must be a symmetric array of type long, that is dedicated for the operation (of size __SHMEM_BARRIER_SYNC_SIZE). Upon entry it must contain __SHMEM_SYNC_VALUE. Upon return it will contain the same value.
- pSync is used internally for coordination and should not be modified during the operation on any PE.

Example: Barrier

```
#include <stdio.h>
#include <shmem.h>
long pSync[_SHMEM_BARRIER_SYNC_SIZE];
int x = 10101;

int main(void)
{
   int me, npes;
   for (int i = 0;
        i < _SHMEM_BARRIER_SYNC_SIZE; i += 1) {
      pSync[i] = _SHMEM_SYNC_VALUE;
   }
   start_pes(0);
   me = _my_pe();
   npes = _num_pes();
   if(me % 2 == 0) {
      x = 1000 + me;
      /*put to next even PE in a circular fashion*/
      shmem_int_p(&x, 4, me+2%npes);
      /*synchronize all even pes*/
      shmem_barrier(0, 1, (npes/2 + npes%2), pSync);
   }
   printf("%d: x = %d\n", me, x);
   return 0;
}
```

Collective Operations: Broadcast

- void shmem_broadcastXX(void *target,
  const void *source, size_t nlong,
  int PE_root, int PE_start, int logPE_stride,
  int PE_size, long *pSync);
  - XX can be 32 or 64
  - Similar concept to MPI_Bcast: broadcast a block of data from one PE to others PE
  - The participants group is defined bu the PE_root, PE_start, logPE_stride and PE_size.
  - The PE_root is a zero-based ordinal with respect to the active set of participants
  - pSync should follow the same rules as for the barrier

Collective Operations: Reductions

- void shmem_<type>_<op>_to_all(
  <type> *dest, <type>*source, int nreduce,
  int PE_start, int logPE_stride, int PE_size,
  <type>*pWrk, long *pSync);
  - Type might be: short, int, long, longlong, float, double
  - Op might be: and, or, xor, max, min, sum, prod
  - Dest and source must not overlap
  - pWrk must be a symmetric array of the same size as dest

Collective Operations: Gather

- void shmem_collectXX(void *target,
  const void *source, size_t nelems,
  int PE_start, int logPE_stride, int PE_size,
  long *pSync);
  - In C XX might be 32 or 64 (In fortran 4, 8, 16, 32, 64)
- Concatenates blocks of data from multiple PEs to an array in every PE (similar to MPI_Allgather)
- The group of participants is defined by the PE_start, logPE_stride and PE_size
- The data is concatenated based on the PE index in the active set
- 2 versions depending if the number of elements is the same on all PE (shmem_fcollectXX) or if they are different (shmem_collectXX)

Collective Operations: AlltoAll

- void shmem_alltoallXX(void *dest,
  const void *source, size_t nelems,
  int PE_start, int logPE_stride, int PE_size,
  long *pSync);
  - In C XX might be 32 or 64 (same in Fortran)
- each PE exchanges a fixed amount of data with all other PEs in the *Active set* (similar to MPI_Alltoall)
- The group of participants is defined by the PE_start, logPE_stride and PE_size
- The data is concatenated based on the PE index in the active set
- A strided version exists (shmem_alltoallsXX) where you can specify a stride for both the source and dest buffers (basically a vector of length 1 with a specified stride)

Point-to-point synchronizations

- void shmem_<type>_wait(<type> *var,
  int value);
  void shmem_<type>_wait_until(<type> *var,
  int cond, int value);
- Blocking function waiting until the condition on the *var is true with respect to the value
- The condition can be: equal, not equal, greater than, less or equal than, less than, greater or equal to

Example

```
#include <shmem.h>
#define GREEN 1
#define RED 0

int light=RED;

int main(int argc, char **argv) {
   int me; start_pes(0);
   me= _my_pe();
   if( me == 0 ) {

      printf("me:%d. Stop on Red Light\n", me);
      shmem_int_wait(&light, RED); /* Is the light still red? */
      printf("me:%d. Now I may proceed\n", me);

   } else if( me == 1 ){

      sleep(10);
      light=GREEN;
      printf("me:%d. I've turn light to green.\n", me);
      shmem_int_put(&light, &light, 1, 0); }

      return 0;
}
```

Output:

me:0. Stop on Red Light

me:1. I've turned light to green

me:0. Now I may proceed

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)

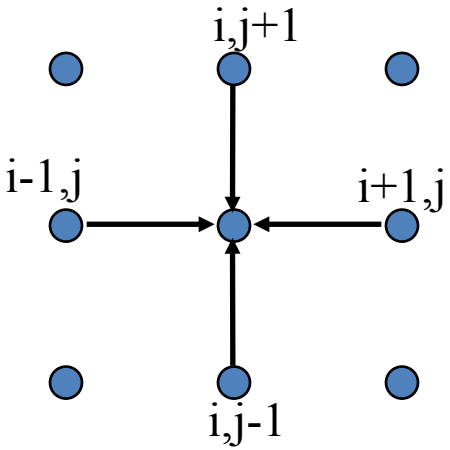
Example

```
#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;
}
```

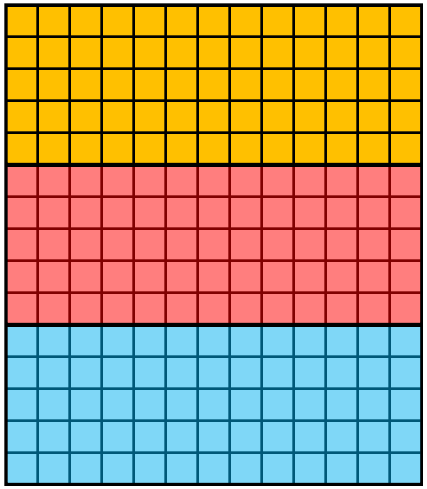
$$U_{i,j}^{n+1} = \frac{1}{4}\left(U_{i-1,j}^{n} + U_{i+1}^{n} + U_{i,j-1}^{n} + U_{i,j+1}^{n}\right)$$

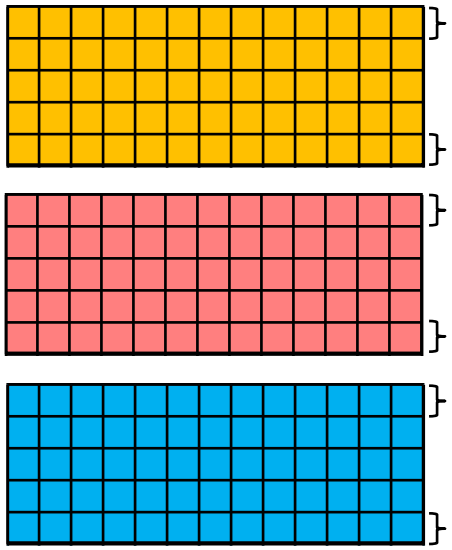
Laplace's equation – OpenSHMEM

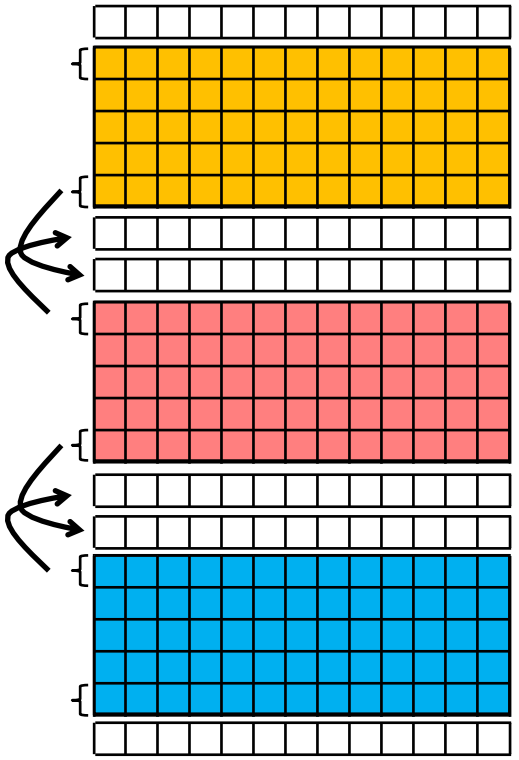



```
for j = 1 to jmax
  for i = 1 to imax
    Unew(i,j) = 0.25 * ( U(i-1,j) + U(i+1,j)
                       +   U(i,j-1) + U(i,j+1))

  end for
end for
```

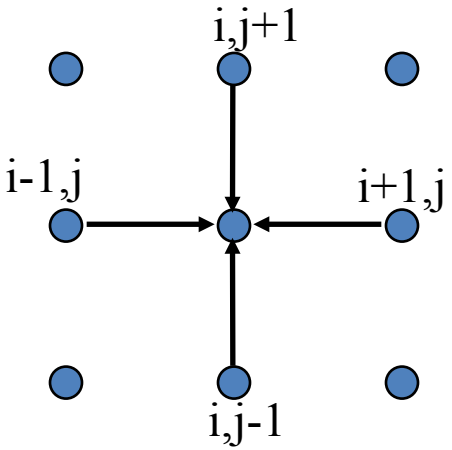






$$U_{i,j}^{n+1} = \frac{1}{4}\left(U_{i-1,j}^{n} + U_{i+1}^{n} + U_{i,j-1}^{n} + U_{i,j+1}^{n}\right)$$

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 using only PUT operations ?
- How to implement using only GET operations ?
- What is the main factor limiting performance ?