```
  tindex(total(j) + count(j)) = k
END DO

! create origin and target datatypes for each get operation
DO i=1,p
  CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, oindex(total(i)+1),   &
                                     MPI_REAL, otype(i), ierr)
  CALL MPI_TYPE_COMMIT(otype(i), ierr)
  CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, tindex(total(i)+1),   &
                                     MPI_REAL, ttype(i), ierr)
  CALL MPI_TYPE_COMMIT(ttype(i), ierr)
END DO

! this part does the assignment itself
CALL MPI_WIN_FENCE(0, win, ierr)
DO i=1,p
  CALL MPI_GET(A, 1, otype(i), i-1, 0, 1, ttype(i), win, ierr)
END DO
CALL MPI_WIN_FENCE(0, win, ierr)

CALL MPI_WIN_FREE(win, ierr)
DO i=1,p
  CALL MPI_TYPE_FREE(otype(i), ierr)
  CALL MPI_TYPE_FREE(ttype(i), ierr)
END DO
RETURN
END
```

Example 11.2 A simpler version can be written that does not require that a datatype be built for the target buffer. But, one then needs a separate get call for each entry, as illustrated below. This code is much simpler, but usually much less efficient, for large arrays.

```
SUBROUTINE MAPVALS(A, B, map, m, comm, p)
USE MPI
INTEGER m, map(m), comm, p
REAL A(m), B(m)
INTEGER win, ierr
INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, sizeofreal

CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, sizeofreal, ierr)
CALL MPI_WIN_CREATE(B, m*sizeofreal, sizeofreal, MPI_INFO_NULL,  &
                    comm, win, ierr)

CALL MPI_WIN_FENCE(0, win, ierr)
DO i=1,m
  j = map(i)/m
  k = MOD(map(i),m)
  CALL MPI_GET(A(i), 1, MPI_REAL, j, k, 1, MPI_REAL, win, ierr)
```

```
END DO
CALL MPI_WIN_FENCE(0, win, ierr)
CALL MPI_WIN_FREE(win, ierr)
RETURN
END
```

11.3.4 Accumulate Functions

It is often useful in a put operation to combine the data moved to the target process with the data that resides at that process, rather then replacing the data there. This will allow, for example, the accumulation of a sum by having all involved processes add their contribution to the sum variable in the memory of one process.

MPI_ACCUMULATE(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, op, win)

IN	origin_addr	initial address of buffer (choice)
IN	origin_count	number of entries in buffer (non-negative integer)
IN	origin_datatype	datatype of each buffer entry (handle)
IN	target_rank	rank of target (non-negative integer)
IN	target_disp	displacement from start of window to beginning of target buffer (non-negative integer)
IN	target_count	number of entries in target buffer (non-negative integer)
IN	target_datatype	datatype of each entry in target buffer (handle)
IN	op	reduce operation (handle)
IN	win	window object (handle)

```
int MPI_Accumulate(void *origin_addr, int origin_count,
             MPI_Datatype origin_datatype, int target_rank,
             MPI_Aint target_disp, int target_count,
             MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
```

```
MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
             TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE,TARGET_RANK, TARGET_COUNT,
    TARGET_DATATYPE, OP, WIN, IERROR
```

```
void MPI::Win::Accumulate(const void* origin_addr, int origin_count, const
             MPI::Datatype& origin_datatype, int target_rank, MPI::Aint
             target_disp, int target_count, const MPI::Datatype&
             target_datatype, const MPI::Op& op) const
```

Accumulate the contents of the origin buffer (as defined by origin_addr, origin_count and origin_datatype) to the buffer specified by arguments target_count and target_datatype, at offset target_disp, in the target window specified by target_rank and win, using the operation op. This is like MPI_PUT except that data is combined into the target area instead of overwriting it.

Any of the predefined operations for MPI_REDUCE can be used. User-defined functions cannot be used. For example, if op is MPI_SUM, each element of the origin buffer is added to the corresponding element in the target, replacing the former value in the target.

Each datatype argument must be a predefined datatype or a derived datatype, where all basic components are of the same predefined datatype. Both datatype arguments must be constructed from the same predefined datatype. The operation op applies to elements of that predefined type. target_datatype must not specify overlapping entries, and the target buffer must fit in the target window.

A new predefined operation, MPI_REPLACE, is defined. It corresponds to the associative function $f(a, b) = b$; i.e., the current value in the target memory is replaced by the value supplied by the origin.

MPI_REPLACE can be used only in MPI_ACCUMULATE, not in collective reduction operations, such as MPI_REDUCE and others.

Advice to users. MPI_PUT is a special case of MPI_ACCUMULATE, with the operation MPI_REPLACE. Note, however, that MPI_PUT and MPI_ACCUMULATE have different constraints on concurrent updates. (*End of advice to users.*)

Example 11.3 We want to compute $\mathtt{B(j)} = \sum_{\mathtt{map(i)=j}} \mathtt{A(i)}$. The arrays A, B and map are distributed in the same manner. We write the simple version.

```
SUBROUTINE SUM(A, B, map, m, comm, p)
USE MPI
INTEGER m, map(m), comm, p, win, ierr
REAL A(m), B(m)
INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, sizeofreal

CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, sizeofreal, ierr)
CALL MPI_WIN_CREATE(B, m*sizeofreal, sizeofreal, MPI_INFO_NULL,  &
                    comm, win, ierr)

CALL MPI_WIN_FENCE(0, win, ierr)
DO i=1,m
  j = map(i)/m
  k = MOD(map(i),m)
  CALL MPI_ACCUMULATE(A(i), 1, MPI_REAL, j, k, 1, MPI_REAL,   &
                      MPI_SUM, win, ierr)
END DO
CALL MPI_WIN_FENCE(0, win, ierr)

CALL MPI_WIN_FREE(win, ierr)
RETURN
END
```