Errata for 'Using Advanced MPI'

May 18, 2023

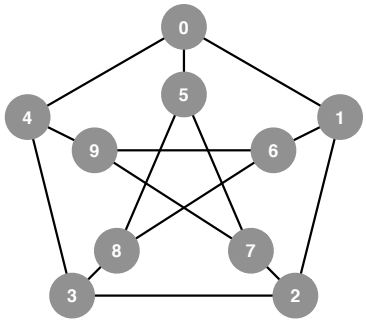
p 6 The formula in the computation for the number of neighbors in Conway's Game of Life has an error in it. The second term on the third line should be `u[i-1][j]`. That is, the computation is

```
nbrs = u[i+1][j+1] + u[i+1][j] + u[i+1][j-1] +
       u[i][j+1]   +             u[i][j-1]   +
       u[i-1][j+1] + u[i-1][j] + u[i-1][j-1];
```

Thanks to Yang Shangqin.

p 35 The size of the array `destinations` is $\sum_i degree_i$, not $\sum_i sources_i$. Thanks to Victor Eijkhout.

p 37 In Figure 2.11, swap the location of 7 and 8. The correct figure is



Thanks to Chan Park.

p 46 In Figure 2.16, the displacements need to be in bytes, and thus the use of `ind` must be multiplied by the extent of an `MPI_DOUBLE`. In addition, the third and fourth element need to be exchanged because of the ordering of processes in the Cartesian topology. Thus, replace these lines

```
MPI_Aint sdispls[4] = {ind(1,1), ind(1,by), ind(bx,1),
                       ind(1,1)},
         rdispls[4] = {ind(1,0), ind(1,by+1), ind(bx+1,1),
                       ind(0,1)};
```

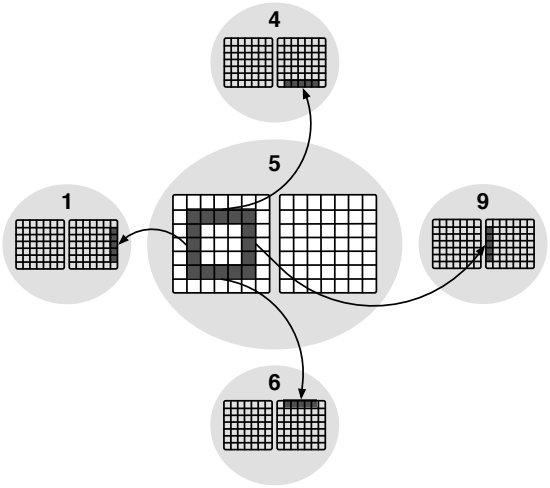
with these

```
MPI_Aint extent, lb;
MPI_Type_get_extent(MPI_DOUBLE, &lb, &extent);
MPI_Aint sdispls[4] = {ind(1,1)*extent, ind(1,by)*extent,
                       ind(1,1)*extent, ind(bx,1)*extent};
MPI_Aint rdispls[4] = {ind(1,0)*extent, ind(1,by+1)*extent,
                       ind(0,1)*extent, ind(bx+1,1)*extent};
```

Thanks to Chan Park.

p 47 In Table 2.24, `mpi_address_kind` should be in upper case and in bold, to be consistent with the presentation style.

p 47 In Figure 2.17, the source data in the mesh in central circle (labeled with a "5") should be move one cell inward from each side to represent the source of halo data. The following figure should replace 2.17:



Thanks to Chan Park.

p 130–134 Some readers have observed problems in using the MCS locks, described on pages 130–134. The code as presented is correct, but correct operation requires that the MPI implementation provide what is called asynchronous progress. This is required by the MPI standard; however, providing this may have negative performance consequences. Because of this, many MPI implementations by default do not enable asynchronous progress; some implementations may not provide a full implementation of asynchronous progress.

To use the MCS lock code, make sure that asynchronous progress is enabled in your MPI implementation. For MPICH, for example, this is done by setting the variable `MPIR_CVAR_ASYNC_PROGRESS=1` (and also building MPICH to support asynchronous progress). There is some evidence that some implementations still have problems with asynchronous progress, even when this is requested.

For implementations that do not correctly support asynchronous progress, the most common workaround is for each process to periodically call some MPI communication routine, such as `MPI_Iprobe`. In the case of the MCS lock release code, there is this block of code:

```
if (lmem[nextRank] == -1) {
  If-Block;
  // starts with MPI_Compare_and_swap(...)
}
```

The test in this code is used to avoid an unnecessary MPI communication operation; specifically, the `MPI_Compare_and_swap` call. However, because of the way that this code is written, the operation of the code is correct if the "If-Block" is unconditionally executed. In addition, the compare-and-swap call serves as an MPI communication call, so if asynchronous progress is not supported, unconditionally executing the "If-Block" can help (though not guarantee, depending on timing) the code to work properly.

Section 8.2 (Pages 243ff.) This is not exactly an errata, but the section describes several of the functions for working with `MPI_Count`, but omits one of the most important, `MPI_Get_elements_x`, the `MPI_Count` counterpart of `MPI_Get_elements`. This routine is described on page 113 in the MPI 3.1 standard.

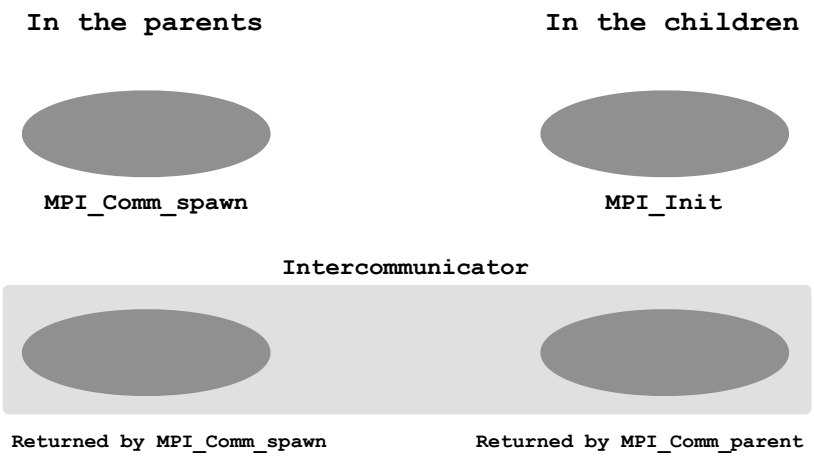
p 256–264 This is not exactly an errata, but the book was published before the `MPI_T_pvar_get_index` and `MPI_T_cvar_get_index` routines were officially adopted by the MPI Forum into MPI. Until those routines become part of MPI, it is necessary to search through all defined control or performance varibles by index to find the index that matches a name. Code for that is available on the web site, in the examples for the chapter "Support for Performance and Correctness Debugging."

p 256 The use of `MPI_Abort` swapped the arguments. It should be

```
if (err != MPI_SUCCESS) MPI_Abort(MPI_COMM_WORLD, 0);
```

Thanks to Chen-Yi GAO.

p 273 Figure 10.2, on the right, has `MPI_Int` where it should have `MPI_Init` (`MPI_Init_thread` would also be correct there). The figure should be:



and is available for download as advmpi-spawn.pdf .

Thanks to Jeff Hammond.

Section 11.1 (Page 307). The Fortran08 code in Example 11.1 is missing a call to `MPI_Type_commit` immediately after the call to `MPI_Type_contiguous`. The correct code (just around the missing call) should be:

```
...
call mpi_comm_dup(MPI_COMM_WORLD, comm)
call mpi_type_contiguous(100, MPI_REAL, rtype)
call mpi_type_commit(rtype)
call mpi_comm_rank(comm, myrank)
...
```

Thanks to Don Eliason.