High Performance Computing: Tools and Applications

Edmond Chow
School of Computational Science and Engineering
Georgia Institute of Technology

Lecture 20

Shared memory computers and clusters

- On a shared memory computer, use MPI or a shared memory mechanism (e.g., OpenMP)?
- Disadvantage of MPI: may need to replicate common data structures; not scalable memory use (scales with number of processes)
  - also, may allow you to use larger “blocks” in algorithms that are more efficient this way, e.g., Jacobi-Schwarz with larger and fewer blocks
  - cannot continue to use MPI for compute nodes as number of cores increases and memory and network bandwidth per core decreases
  - MPI uses data copying in its protocols; should not be necessary with shared memory
- Advantage of MPI: do not have threads that might interfere with each other (sharing of heap);
  - also, forces you to decompose problems for locality

Shared memory computers and clusters (continued)

- In general, on clusters, and even Intel Xeon Phi, MPI may be combined with OpenMP
  - reduce number of MPI processes (MPI processes have overhead)
  - max number of MPI processes may be limited

MPI shared memory

MPI-3 provides for shared memory programming within a compute node (use load/store instead of get/put)

- Alternative to MPI+X, which might not interoperate well, e.g., X=OpenMP
- MPI+MPI has no interoperability issues

Recall MPI RMA

- define memory regions
- use put/get
- synchronize

MPI-3 shared memory programming extends some RMA ideas, but uses load/store instead of get/put

Shared memory programming with processes?

- processes have their own address space
- a shared memory region may have different addresses on each process (but is physically the same memory – copies are avoided)

Recall creating memory windows for RMA

```
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit,
  MPI_Info info, MPI_Comm comm, MPI_Win *win);
```

- `base` is the local address of beginning of window
- this memory can be any memory, including memory allocated by `MPI_Alloc_mem` (and freed by `MPI_Free_mem`)

Allocating memory and creating a memory window at the same time

```
MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info
  MPI_Comm comm, void *baseptr, MPI_Win *win);
```

Call it this way:

```
double *baseptr;
MPI_Win win;
MPI_Win_allocate(..., &baseptr, &win);
```

Free the window (also frees the allocated memory)

```
MPI_Win_free(&win);
```

Philosophy: shared memory windows

- data is private by default (like MPI programs running in different processes)
- data is made public explicitly through shared memory windows
- allows graceful migration of “pure” MPI programs to use multicore processors more efficiently
- “communication” (load/store) via shared memory does not involve extra copies
- creating a shared memory window is a collective operation (done at the same time on all ranks, allowing the optimization of the memory layout)

Allocating shared memory windows

Window on a process's memory that can be accessed by other processes on the same node

```
int MPI_Win_allocate_shared(MPI_Aint size, int disp_unit,
  MPI_Info info, MPI_Comm comm, void *baseptr, MPI_Win *win)
```

- called collectively by processes in `comm`
- processes in `comm` *must* be those that can access shared memory (e.g., processes on the same compute node)
- by default, a *contiguous* region of memory is allocated and shared (noncontiguous allocation is also possible, and may be more efficient as each contributed region could be page aligned)
- each process contributes `size` bytes to the contiguous region; `size` can be different for each process and can be zero
- the contribution to the shared region is in order by rank
- `baseptr` is the pointer to a process's contributed memory (not the beginning of the shared region) in the address space of the process

Allocating shared memory windows (continued)

- the shared region can now be used using load/store, with all the usual caveats about race conditions when accessing shared memory from different *processes*
- the shared region can also be accessed using RMA operations (particularly for synchronization)

How it works

- Process with rank 0 in `comm` allocates the entire shared memory region for all processes
- Other processes attach to this shared memory region
- The entire memory region may reside in a single locality domain, which may not be desirable
- Therefore, using noncontiguous allocation may be advantageous (set the `alloc_shared_noncontig` info key to true)

Address of shared memory contributed by another process

```
int MPI_Win_shared_query(MPI_Win win, int rank,
  MPI_Aint *size, int *disp_unit, void *baseptr);
```

- `baseptr` returns the address (in the local address space) of the beginning of the shared memory segment contributed by another process, the target `rank`
- also returns the size of the segment and the displacement unit
- if rank is `MPI_PROC_NULL`, then the address of the beginning of the first memory segment is returned
- this function could be useful if processes contribute segments of different sizes (so addresses cannot be computed locally), or if noncontiguous allocation is used
- in many programs, knowing the “owner” of each segment may not be necessary

Extension to MPI+MPI

- Function for determining which ranks are common to a compute node:

```
MPI_Comm_split_type (comm, MPI_COMM_TYPE_SHARED, 0,
     MPI_INFO_NULL, &shmcomm);
```

- Function for mapping group ranks to global ranks:
  `MPI_Group_translate_ranks`