Programming of Supercomputers

Assignment 3:

MPI Point-to-Point and One-sided Communication

Prof. Michael Gerndt, Madhura Kumaraswamy

Technische Universität München

Informatik 10: Lehrstuhl für Rechnerarchitektur & Parallele Systeme

30.11.2018

Announcements

- Register for the exam on TUMOnline
- Delete all useless files/directories from /home on SuperMUC
- Next session: 21.12.2018

- **Deadlines:**
  - Assignment 3 deadline: 18.12.2018 at 23:59

Discussion: Assignment 1

- GPROF provided insight?

- Faster compiler: GCC or ICC?
  - Faster compiler flags?
  - Exhaustive evaluation of all compiler flags possible?

- Scalability:
  - OpenMP, MPI, Hybrid
- Best overall performance?
  - Predicted outcome? Expected or unexpected?
  - Best performance in the multi-threaded case?
  - OpenMP performance?
  - MPI performance?
  - Hybrid performance?

- Key lessons learned?

Discussion: Assignment 2

- Bugs in parallel programming
  - Were you familiar with them?
  - What did you learn about floating point arithmetic?
  - Are all of them parallel programming specific?

- What do you think about TotalView?
  - Is it worth learning?
  - Good OpenMP support?
  - Good MPI support?

- Difference between Score-P and Gprof

- What do you think about Vampir?
  - Is it worth learning?
  - Did you explore more than what was in the assignment?

Assignment 3

- Gaussian elimination
  - Solves a linear system of equations in the form: $Ax = b$
  - Upper triangular matrix form

- 3 parts:
  - Setting a baseline
    - Measurement problems
    - Set up an accurate baseline
    - Scaling
  - MPI point-to-point communication
    - Focus on blocking and non-blocking communication
    - Convert blocking to non-blocking communication
    - Try to achieve overlap between computation and communication
    - Perfect overlap is the target
  - MPI one-sided communication
    - Convert blocking point-to-point communication to one-sided

Introduction to MPI

- Covered in the Parallel Programming lecture
- **M**essage **P**assing **I**nterface: specification for writing message passing programs
- Standard mode: SPMD
- Large set of operations in the standard
- Types:
  - Point-to-point
    - Blocking/non-blocking
  - Collective

Code Example

```
// required MPI include file
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[]) {
    int  numtasks, rank, out_msg, in_msg, tag = 1;
    MPI_Status status;   // required variable for receive routines
    MPI_Init(&argc,&argv);  //Initialize MPI
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);  //Get number of ranks
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);  //Get my rank

    // Process 0 sends message to Process 1 and waits to receive a return message
    if (rank == 0) {
        out_msg = 0;
        MPI_Send(&out_msg, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);  //MPI_COMM_WORLD is the communicator
        MPI_Recv(&in_msg, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);
    }
    // Process 1 waits for message from Process 0, and then returns a message
    else if (rank == 1) {
        out_msg = 1;
        MPI_Recv(&in_msg, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
        MPI_Send(&out_msg, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
    }

    printf("Task %d: Received %d \n", rank, in_msg);

    MPI_Finalize();
}
```

Output:

```
Task 0: Received 1
Task 1: Received 0
```

MPI Point-to-Point Communication

- 2 buffers
  - System buffer
  - Application/user buffer
- Blocking:
  - The data must be successfully sent or copied to the system (internal) buffer
  - The application, or send/receive buffer that contained the sent/received data can be reused
- Non-blocking:
  - Returns immediately
  - Should not modify the send/receive buffer until completion
  - Check status of call via dedicated methods (wait/test)

MPI Point-to-Point Communication

- **Blocking:**
  - Send modes:
    1. MPI_Send
       - Will not return until the send buffer can be used
       - May/may not block
    2. MPI_Bsend
       - May buffer, and returns immediately
       - The send buffer can be used
    3. MPI_Ssend
       - Synchronous send
       - Will not return until matching receive posted
  - Receive modes:
    1. MPI_Recv
       - Will not return until the receive buffer contains the message

MPI Point-to-Point Communication

- **Non-blocking:**
  - Send modes:
    1. MPI_Isend
       - Nonblocking send, not necessarily asynchronous
       - A communication request handle (MPI_Request) is returned for querying the status of the pending message
       - Cannot reuse the send buffer until a successful test/wait
    2. MPI_Ibsend
       - Buffered non-blocking send
    3. MPI_Issend
       - Synchronous non-blocking send

  - Receive modes:
    1. MPI_Irecv
       - Returns immediately without waiting for the message to be received
       - MPI_Request is returned for querying the status of the pending message
       - Cannot reuse the receive buffer until a successful test/wait

MPI Point-to-Point Communication

- Testing for completion:
  - MPI_Wait

    ```
    int MPI_Wait(MPI_Request *request, MPI_Status *status)
    ```

    - Blocks until communication is completed
    - Applicable for MPI_Isend and MPI_Irecv
    - MPI_Waitany, MPI_Waitall, MPI_Waitsome
    - MPI_Irecv immediately followed by MPI_Wait = MPI_Recv

  - MPI_Test

    ```
    int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
    ```

    - Checks the status of the communication
    - Non-blocking
    - Useful for both sender and receiver of non-blocking communications
    - Parameter "flag" returns true (1) if the operation has completed, false (0) if not

MPI One-Sided Communication

- Point-to-point operations are two-sided
  - Require sender and receiver
  - Requires a matching recv for send or vice-versa
- One-sided communication involve 2 processes:
  - Decouple data transfer and process synchronization
  - Origin process that performs a 'put' or 'get' operation
  - Target process whose memory is being accesses
  - Target does not perform a counterpart of the origin's action
- One-sided MPI operations are known as RDMA or RMA operations
- Origin cannot simply access target data at arbitrary times
- Origin and target declare specific memory areas available for one-sided communications
  - A *window* is accessible to other processes to 'get' data or 'put' data

MPI One-Sided Communication

- How to create windows of memory?

  MPI_Win_create, MPI_Win_allocate, MPI_Win_create_dynamic (dynamically attach using MPI_Win_attach/detach): collectivey performed by a process group

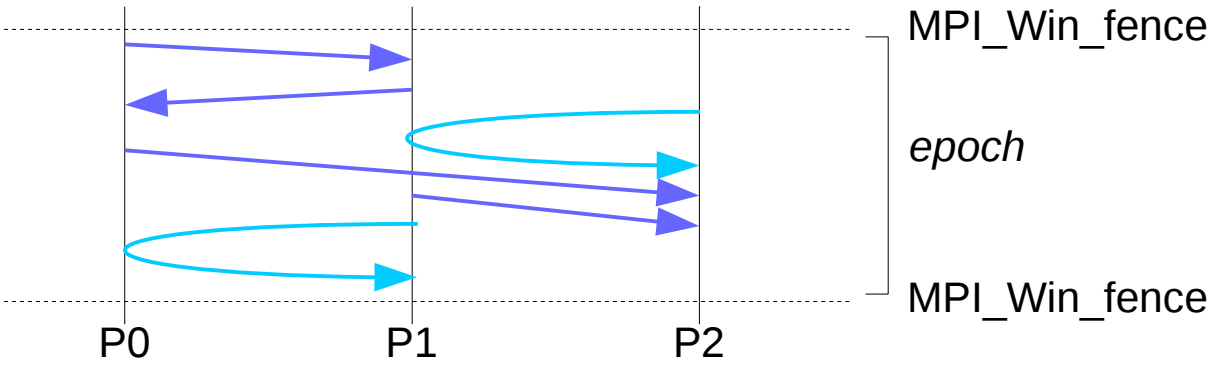
```
int main(int argc, char ** argv)
{
    int *arr;
    MPI_Win w;
    MPI_Init(&argc, &argv);
    arr[0] = 10; arr[1] = 20;

    /* create private memory */
    MPI_Alloc_mem(100*sizeof(int), MPI_INFO_NULL, &arr);

    /* all procs collectively declare a remotely accessible memory */
    MPI_Win_create(arr, 100*sizeof(int), sizeof(int), MPI_INFO_NULL, MPI_COMM_WORLD, &w);

    /* OR collectively create a remotely accessible memory */
    MPI_Win_allocate(100*sizeof(int), sizeof(int), MPI_INFO_NULL, MPI_COMM_WORLD, &arr, &w);

    MPI_Win_free(&w);
    MPI_Free_mem(arr);
    MPI_Finalize();
    return 0;
}
```

MPI One-Sided Communication

- How to read, write & updating this memory
  - MPI_Get, MPI_Put, MPI_Accumulate, MPI_Get_accumulate
  - Non-blocking
- How to synchronize the data
  - Active RMA/active target synchronization
    - Target sets boundaries on the time period/epoch for access to the window can be accessed
  - Passive RMA/passive target synchronization
    - Target process puts no limitation on the time for access to its window
    - Data is read/written at will
    - Hard to debug

MPI One-Sided Communication

- Synchronizing the data
  - Fence (active target)
    - Resembles an MPI_Barrier
    - All processes collectively call MPI_Win_fence



MPI One-Sided Communication

- Synchronizing the data
  - Post-start-complete-wait (generalized active target)
    - MPI_Win_fence requires the synchronization of the entire communicator
    - Post-start-complete-wait allows to specify which processes actually need to communicate

```
if (rank == 0) {
    /* Ranks 1 and 2 want to put data into rank 0 */
   /* Rank 0 begins an exposure epoch*/
    MPI_Win_post(group1_2,0,win);
    /* Wait for epoch to end */
    MPI_Win_wait(win);
}
else {
    /* Begin the access epoch */
    MPI_Win_start(group_0,0,win);
    /* Put into rank 0*/
    MPI_Put(...);
    /* Terminate the access epoch */
    MPI_Win_complete(win);
}
```

MPI One-Sided Communication

- Synchronizing the data
  - Lock/unlock (passive target)
    - Similar to mutex
    - MPI_Win_Lock, MPI_Win_Unlock

Instructions

- Data and source code will be uploaded on Moodle
- Intel compiler & Intel MPI
- Login:
  - Phase 1 thin nodes (Sandy Bridge): ssh -YC sb.supermuc.lrz.de
  - Phase 2 Haswell nodes: ssh -YC hw.supermuc.lrz.de

Questions?