TIMO SCHNEIDER <TIMOS@INF.ETHZ.CH>

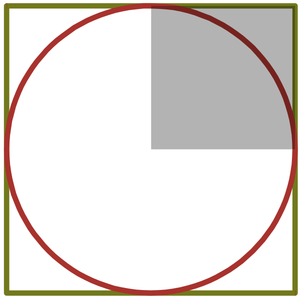
DPHPC Recitation Session 2
Advanced MPI Concepts

Recap

- **MPI is a widely used API to support message passing for HPC**
- **We saw that six functions are enough to write useful parallel programs in SPMD style**
  - MPI_Init() / MPI_Finalize() --- required for initialization
  - MPI_Send() / MPI_Recv() --- actually sending messages
  - MPI_Comm_rank() / MPI_Comm_size() --- Who am I?
- **We also looked at MPI collectives, e.g., MPI_Bcast()**
- **If six functions are enough, why are there ~300 in the standard?**
  - Optimization: Try to implement your own broadcast – should be hard to beat MPI performance.
  - Convenience: Do you really want to do this? Do you have too much time?
  - Performance Portability: Do you think your Broadcast will also be fast on a different cluster, which uses a different network?

Homework – Pi with MPI

- Idea: Circle with radius 1, in the middle of a rectangle with side length 2.



- Area of circle segment is: (Pi*r^2)/4
- Area of dark rectangle is: r^2
- Pi = 4 * Area of circle / Area of rectangle
- Get the ratio of areas by putting many points randomly inside the rectangle, and count how many are inside vs. outside of the circle.
- Point p = (x,y), if x^2+y^2 <= 1 it is in the circle (hit) otherwise not (miss)

Homework – Pi with MPI

- **Each MPI rank simulates some point throws, in the end they are added together**
- **Use MPI_Comm_size() to find out how many throws each ranks should do (to get to a predefined total)**
- **Assign num_iters % commsize to some rank (are there better ways?)**
- **Collect hits/misses in two variables**
- **Use MPI_Reduce() to get the sum of all hits**

Today – "Advanced" MPI features

Looking at those serves two purposes:

- **Telling you that they exist, so use it in your project (if suitable) to get good performance**
- **The focus today is on concepts not so much on details, so not every argument of every function will be explained**
- **The ideas behind them are important, so even if you don't use MPI you know where there might be some potential for optimization**

- **MPI Datatypes**
- **Non-blocking collectives**
- **MPI one sided**

MPI Datatypes – Basic Types

- **Basic Types: MPI_INT, MPI_CHAR, MPI_FLOAT, MPI_DOUBLE …**
- **Use them (and the count argument) to send the corresponding types in C. Avoid MPI_BYTE if possible**

- **Now assume we have a 2D matrix of N*N doubles in C**
- **C does not have multi-dimensional arrays built in**
- **Can emulate it using 1D array.**
  mat[i,j] = m[i*N+j] (row major layout) or
  mat[i, j] = m[j*N+i] (column major layout)

```
double* m = malloc(N*N*sizeof(double));

// fill with random data
for (int i=0; i<N; i++)
  for (int j=0; i<N; i++)
    m[i*N+j] = rand();
```

MPI Datatypes – Small messages

- **Now we want to send a column of our matrix stored in row-major layout to another process**

```
for (int row=0; i<N; i++)
  MPI_Send(&m[row*N+col], 1, MPI_DOUBLE, peer, tag, comm);
```

- **This will send N separate small messages**
- **Each message has to be matched by the receiver, and usually there is some overhead when sending small messages (i.e., minimum packet size on the network)**
- **So this will give bad performance! Do NOT do this!**

MPI Datatypes – Manual Packing

- So how about packing the column data into a send buffer?

```
double* buf = malloc(N*sizeof(double));
for (int row=0; i<N; i++) {
    sendbuf[row] = m[row*N+col];
}
MPI_Send(buf, 1, MPI_DOUBLE, peer, tag, comm);
```

- Works better in many cases
- Sadly, many people do this in real applications

- We added an extra copy of our data! Copying is not free!
- But what if your network is very good with small messages?
- Maybe a hybrid approach would be best, i.e., send in chunks of 100 doubles? Or 500?
- Idea: Let MPI decide how to handle this!

MPI Datatypes – Type creation

- **We need to tell MPI how the data is laid out**
- **MPI_Type_vector(count, blocklen, stride, basetype, newtype) will create a new datatype, which consists of count instances of blocklen times basetype, with a space of stride in between.**

stride = 4
count = 5
blocklen = 1

- **Before a new type can be used it has to be committed with MPI_Type_commit(MPI_Datatype* newtype)**

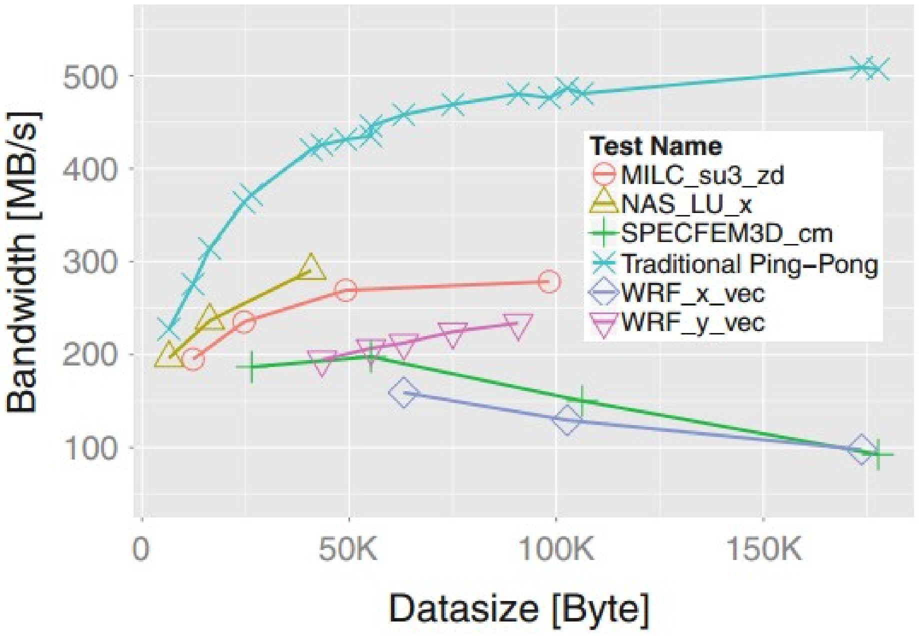
```
MPI_Datatype newtype;
MPI_Type_vector(N, blocklen, N, MPI_DOUBLE, &newtype);
MPI_Type_commit(&newtype);
MPI_Send(m, 1, newtype, peer, tag, comm);
```

MPI Datatypes – Composable

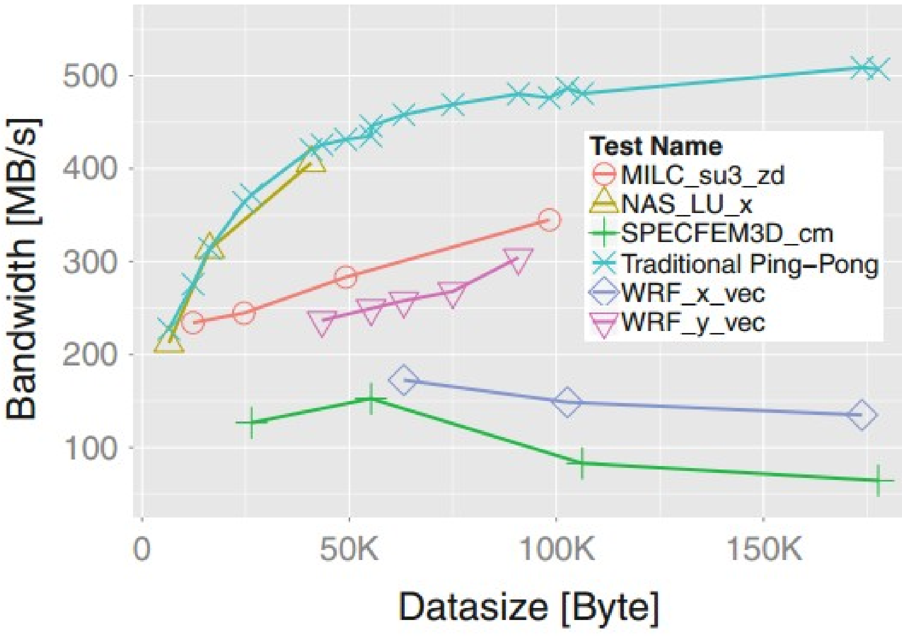
- **MPI Datatypes can are composable! - So you can create a vector of a vector datatype! (Useful for 3D matrices!)**
- **The MPI_Type_vector() is not the only type creation function**
  - MPI_Type_indexed() allows non-uniform strides
  - MPI_Type_struct() allows to combine different datatypes into one “object”
  - See MPI standard for complete list/definition if you need them!

Datatypes - Performance

Manual Packing



MPI Datatypes



Schneider/Gerstenberger: Application-oriented ping-pong benchmarking: how to assess the real communication overheads

Nonblocking Collectives

- We saw nonblocking versions of Send and Receive last week
- They allow us to do something useful (computation) while we wait for data to be transmitted
- MPI also defines nonblocking collectives
- Example: MPI_Ialltoall(void* senbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, MPI_Datatype recvtype, MPI_Comm comm, MPI_Request * request)
- Same as MPI_Alltoall, except for the request handle!
- We can use MPI_Test() / MPI_Wait() / MPI_Waitall() on this handle, just as we did with nonblocking point-to-point communication
- Many MPI implementations do not progress if you do not call MPI functions, i.e., MPI_Test()!

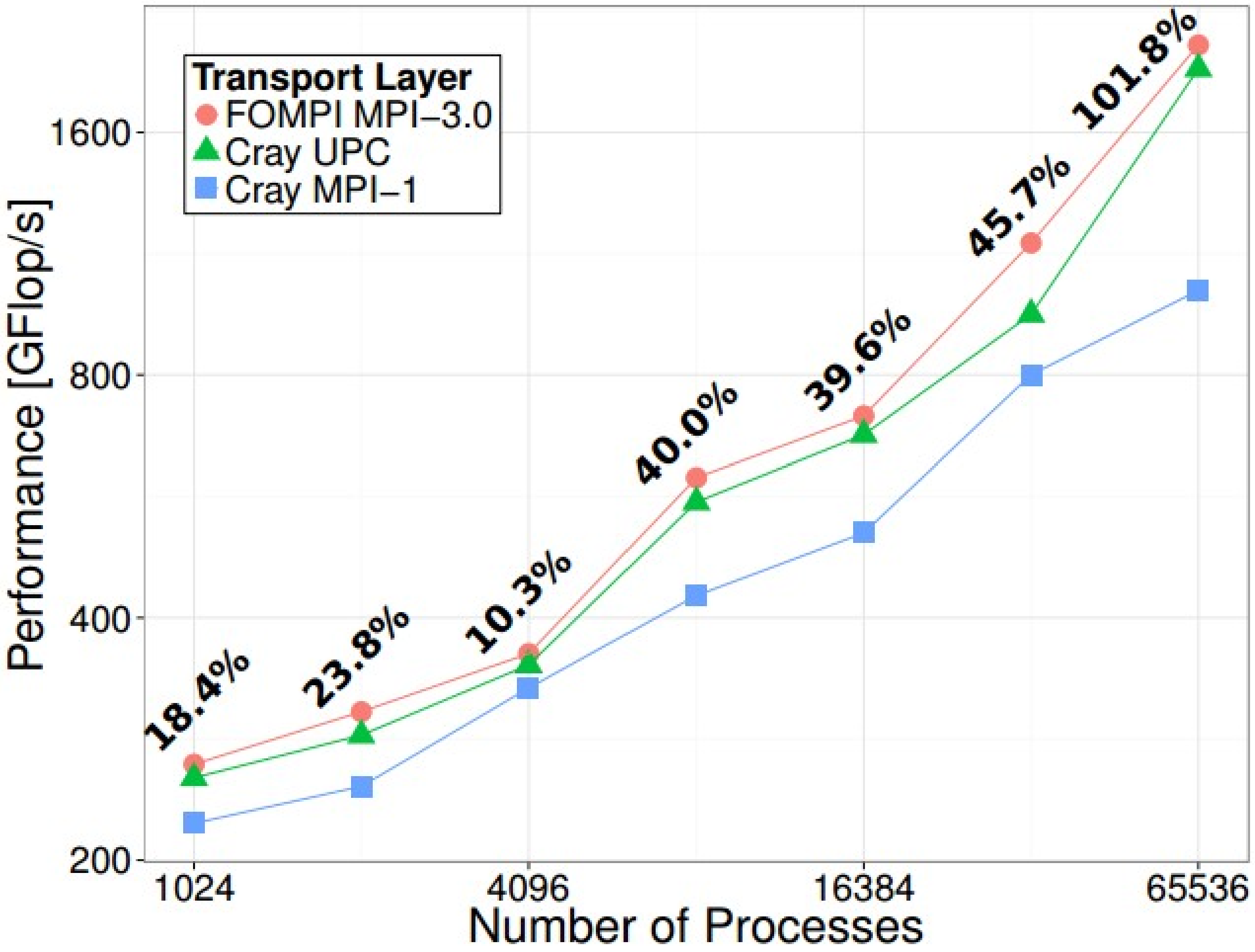
MPI-3 One-sided

- **Message passing is not the only programming model supported by MPI**
- **Since MPI version two it also supports one-sided communication, so only one process has to “do something” to transfer data**
- **The one-sided interface changed substantially in MPI-3, be aware of this when searching for documentation on your own**
- **Make sure you are using an MPI implementation which supports MPI-3 if you want to use the features described here, i.e., Open MPI does not!**

Benefits of the one-sided programming model

- **The semantics of message passing imply**
  - Messages are either buffered at the receiver until matching receive is called, this means the entire message has to be copied
  - Or sender waits until the receiver has called a matching receive, this means time is "wasted" where nothing is transmitted even though the data is available
  - Incoming messages need to be matched against "posted" receives. This is often implemented by traversing a queue of messages / stored receive info
- **Most of this is done in software on the CPU**
- **Most modern network cards support RDMA (Remote direct memory access)**
  - Data can be transferred to a remote memory address
  - The remote node does not need to do anything
- **The one-sided (or RMA) programming model is a better match for modern hardware, and gets rid of some of the overheads of message passing**
- **But is often harder to program**

MPI-3 One-sided Performance (MILC Code)



Gerstenberger/Besta/Hoefler: Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided

MPI One-Sided Concepts - Window

- Data is transferred with Get() and Put() calls
- Before we can access the memory of a remote node, this node has to expose a memory region
- In MPI terms such a region is called an MPI_Window
- We can either create a window from already allocated/used memory
  MPI_Win_create(void* base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win* win)
- Or let MPI allocate new memory for us (use this if you have a choice)
  MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, void** baseptr, MPI_Win* win)
- Window creation is collective!
- Third option: attach memory to an existing window (slow)

MPI One-Sided Concepts - Synchronization

- MPI RMA defines “epochs”
- Before communicating we open an epoch
- Then we use Put()/Get()
- Then we close the epoch
- Only now can we safely access the data in our window!

MPI One-Sided – Fence Synchronization

- **The simplest way to open/close an epoch is with MPI_Fence(int assert, MPI_Win win)**
- **A fence closes the previously opened epoch (if there was one) and opens a new one in a single call**

```
MPI_Win win;
int data;
If (rank == 0) data = 42;
MPI_Win_create(&data, sizeof(int), 1, MPI_INFO_NULL, comm, &win);
MPI_Win_fence(0, win);
if (rank != 0)
  MPI_Get(&data, 1, MPI_INT, 0, 0, 1, MPI_INT, &win);
MPI_Win_fence(0, win);
MPI_Win_free(&win);
```

MPI One-Sided – Post/Start/Complete/Wait

- **While easy to program, sometimes fence synchronization does too much**
  - It synchronizes the window for all ranks in the communicator
  - It does not differentiate between origin (caller of put/get) and target (peer in those calls) processes
  - Often as expensive as doing an MPI_Barrier()
- **MPI_Win_start() / MPI_Win_complete() start and end an epoch on the origin**
- **MPI_Win_post() / MPI_Win_wait() start and end an epoch on the target**
- **start/post call take not only the window, but also an MPI_Group argument, this specifies which ranks are included in the communication**
- **Groups can be created/manipulated by the MPI_Group_XXX() and MPI_Comm_group() functions**

MPI One-Sided – Lock/Unlock

- **In fence and PSCW synchronization, the target plays an active role, i.e., calls a synchronization function**
- **Therefore these modes are called "Active Target Mode"**
- **There is also a "Passive Target Mode" where the target does not need to do anything**
  - MPI_Win_lock_all() allows us to access the window of all other ranks (cf. Fence)
  - MPI_Win_lock() allows us to access the window of a specific rank (cf. PSCW)
  - Locks can be shared or exclusive
  - Epoch opened with lock/lock_all is closed via unlock/unlock_all
- **In passive target mode we can also use MPI_Win_flush() to finish all outstanding operations to a specific target rank**