HPCSE - I

«MPI Programming Model - Part III»

Panos Hadjidoukas

Schedule and Goals

- 01.12.2017: MPI - part 3
  - uncovered topics on MPI communication
  - parallel I/O
  - nested OpenMP parallelism
  - hybrid MPI + OpenMP programming model

Outline

- Send modes revisited
- Non-blocking neighboring communication
- More on collective operations
  - Broadcast of multiple values
  - `MPI_Alltoall`: good to know, hopefully not needed
  - `MPI_Scan`, `MPI_Exscan`: will be needed
- Parallel I/O
- OpenMP nested parallelism
- Hybrid MPI + OpenMP

I. Blocking and non-blocking sends

- blocking: return only when the buffer is ready to be reused
- non-blocking: return immediately

```
int MPI_Ssend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
// synchronous send: returns when the destination has started to receive the message

int MPI_Bsend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
// buffered send: returns after making a copy of the buffer. The destination might not yet
//                have started to receive the message

int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
// standard send: can be synchronous or buffered, depending on message size

int MPI_Rsend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
// ready send: an optimized send if the user can guarantee that the destination has already
//             posted the matching receive
```

```
int MPI_Issend(void *buf, int count, MPI_Datatype datatype, int dest, int tag,
               MPI_Comm comm, MPI_Request *request)

int MPI_Ibsend(void *buf, int count, MPI_Datatype datatype, int dest, int tag,
               MPI_Comm comm, MPI_Request *request)

int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag,
              MPI_Comm comm, MPI_Request *request)

int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
              MPI_Comm comm, MPI_Request *request)
```

MPI's Send modes

- Represent different choices of
  - buffering (where is the data kept until it is received)
  - synchronization (when does a send complete)

- `MPI_Send`: will not return until you can use the send buffer. It may or may not block (it is allowed to buffer, either on the sender or receiver side, or to wait for the matching receive).
- `MPI_Bsend`: May buffer; returns immediately and you can use the send buffer. A late add-on to the MPI specification. Should be used only when absolutely necessary.
- `MPI_Ssend`: will not return until matching receive posted
- `MPI_Rsend`: May be used ONLY if matching receive already posted. User responsible for writing a correct program.

Source: http://www.mcs.anl.gov/research/projects/mpi/sendmode.html

MPI's Send modes

- `MPI_Isend`: Nonblocking send. But not necessarily asynchronous. You can NOT reuse the send buffer until either a successful, wait/test or you KNOW that the message has been received. An immediate send must return to the user without requiring a matching receive at the destination.
- `MPI_Ibsend`: buffered nonblocking
- `MPI_Issend`: Synchronous nonblocking. Note that a Wait/Test will complete only when the matching receive is posted.
- `MPI_Irsend`: As with MPI_Rsend, but nonblocking.

Source: http://www.mcs.anl.gov/research/projects/mpi/sendmode.html

MPI's Send modes - Recommendations

- The best performance is likely if you can write your program so that you could use just `MPI_Ssend`; in that case, an MPI implementation can completely avoid buffering data.
- Use `MPI_Send` instead; this allows the MPI implementation the maximum flexibility in choosing how to deliver your data.
- If nonblocking routines are necessary, then try to use `MPI_Isend` or `MPI_Irecv`.
- Use `MPI_Bsend` only when it is too inconvenient to use `MPI_Isend`.
- The remaining routines, `MPI_Rsend`, `MPI_Issend`, etc., are rarely used but may be of value in writing system-dependent message-passing code entirely within MPI.

Source: http://www.mcs.anl.gov/research/projects/mpi/sendmode.html

II. Non-blocking neighbor communication

- Periodic domain

```
int main(int argc, char *argv[])  {
 int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
 MPI_Request reqs[4];
 MPI_Status stats[4];

 MPI_Init(&argc,&argv);
 MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);

 prev = rank-1;
 next = rank+1;
 if (rank == 0)  prev = numtasks - 1;
 if (rank == (numtasks - 1))  next = 0;

 MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[2]);
 MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[3]);

 MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[0]);
 MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[1]);

 { /* do some work */ }

 MPI_Waitall(4, reqs, stats);

 MPI_Finalize();
 return 0;
}
```

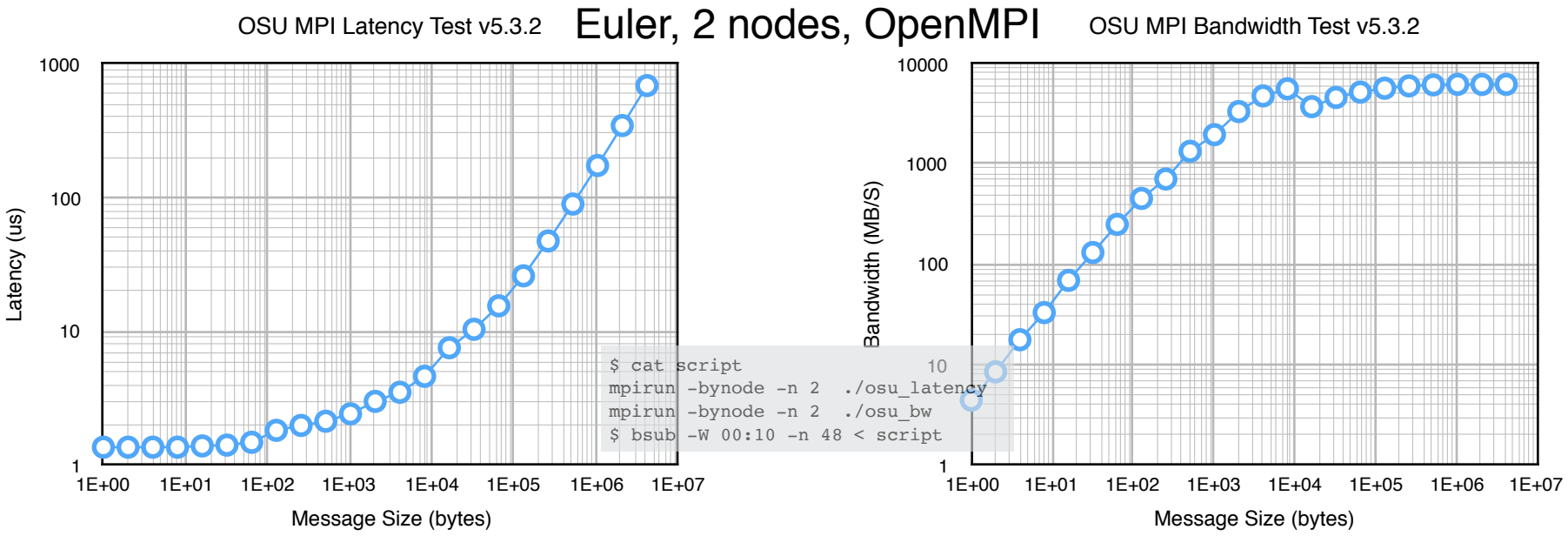
II. Non-blocking neighbor communication

- Non-periodic domain and use of MPI_PROC_NULL

```
int main(int argc, char *argv[])  {
 int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
 MPI_Request reqs[4];
 MPI_Status stats[4];

 MPI_Init(&argc,&argv);
 MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);

 prev = rank-1;
 next = rank+1;
 if (rank == 0)  prev = MPI_PROC_NULL;
 if (rank == (numtasks - 1))  next = MPI_PROC_NULL;

 MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[2]);
 MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[3]);

 MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[0]);
 MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[1]);

 { /* do some work */ }

 MPI_Waitall(4, reqs, stats);

 MPI_Finalize();
 return 0;
}
```

OSU Benchmarks

- http://mvapich.cse.ohio-state.edu/benchmarks/



How do these numbers relate to Computation-Transfer (CT) overlap?

- Based on the total message size exchanged between two neighboring ranks for the ghost data, we can estimate the required communication time ($T_{transfer}$).
- We can also measure the time for processing the inner domain ($T_{computation}$)
- For perfect CT overlap: $T_{transfer} < T_{computation}$
  - the time of `MPI_Waitall` should be almost zero!

III. More on collective operations

- Parallelizing Simpson integration: only the master rank (0) reads the input data. How do we share it?

```
int main(int argc, char** argv)
{
  MPI_Init(&argc,&argv);

  int size;
  int rank;
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  double a;            // lower bound of integration
  double b;            // upper bound of integration
  int nsteps; // number of subintervals for integration

  // read the parameters on the master rank
  if (rank==0)
    std::cin >> a >> b >> nsteps;

  // we need to share the parameters with the other ranks
  ???

  // integrate just one part on each thread
  double delta = (b-a)/size;
  double result = simpson(func,a+rank*delta,a+(rank+1)*delta,nsteps/size);

  //  collect all to the master (rank 0)
  MPI_Reduce(rank == 0 ? MPI_IN_PLACE : &result, &result, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

   // the master prints
  if (rank==0)
    std::cout << result << std::endl;

  MPI_Finalize();
  return 0;
}
```

Broadcast of multiple values

- We can use this to broadcast the data

```
// and then broadcast the parameters to the other ranks
 MPI_Bcast(&a, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
 MPI_Bcast(&b, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
 MPI_Bcast(&nsteps, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

- This is inefficient since we use three broadcasts.
- "Ugly" solution: change `nsteps` to double and broadcast an array of 3 doubles
- Another solution: pack it all into a struct and send it bitwise

```
// define a struct for the parameters
struct parms {
  double a;            // lower bound of integration
  double b;            // upper bound of integration
  int nsteps; // number of subintervals for integration
};

parms p;

// read the parameters on the master rank
if (rank==0)
  std::cin >> p.a >> p.b >> p.nsteps;

// broadcast the parms as bytes - warning, not portable on heterogeneous machines
MPI_Bcast(&p, sizeof(parms), MPI_BYTE, 0, MPI_COMM_WORLD);
```

- Can be dangerous because it assumes a homogeneous machine with identical integer and floating point formats.

Packing and unpacking

- Allocate a sufficiently large buffer and then pack the data into it
- Send/receive the packed buffer with type `MPI_PACKED`
- Finally unpack it on the receiving side

```
int MPI_Pack(void *inbuf, int incount, MPI_Datatype datatype,
             void *outbuf, int outcount, int *position, MPI_Comm comm)
// packs the data given as input into the outbuf buffer starting at a given position.
// outcount is the size of the buffer and position gets updated to point to the first
// free byte after packing in the data.
// An error is returned if the buffer is too small.

int MPI_Unpack(void *inbuf, int insize, int *position,
               void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)
// unpack data from the buffer starting at given position into the buffer outbuf.
// position is updated to point to the location after the last byte read

int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm, int *size)
// returns in size an upper bound for the number of bytes needed to pack incount
// values of type datatype. This can be used to determine the required buffer size
```

Packing data into a buffer

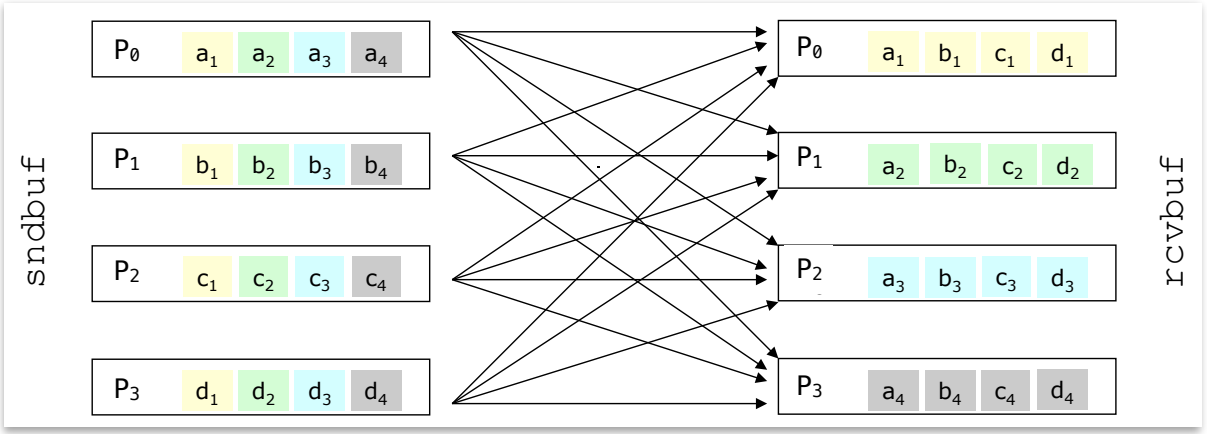
- Pack the input data, broadcast it and unpack

```
// create a buffer and pack the values.
// first get the size for the buffer and allocate a buffer
int size_double, size_int;
MPI_Pack_size(1, MPI_DOUBLE, MPI_COMM_WORLD,&size_double);
MPI_Pack_size(1, MPI_INT,    MPI_COMM_WORLD,&size_int);
int buffer_size = 2*size_double+size_int;
char* buffer = new char[buffer_size];

// pack the values into the buffer on the master
if (rank==0) {
  int pos=0;
  MPI_Pack(&a, 1, MPI_DOUBLE, buffer, buffer_size, &pos, MPI_COMM_WORLD);
  MPI_Pack(&b, 1, MPI_DOUBLE, buffer, buffer_size, &pos, MPI_COMM_WORLD);
  MPI_Pack(&nsteps, 1, MPI_INT, buffer, buffer_size, &pos, MPI_COMM_WORLD);
  assert ( pos <= buffer_size );
}

// broadcast the buffer
MPI_Bcast(buffer, buffer_size, MPI_PACKED, 0, MPI_COMM_WORLD);

// and unpack on the receiving side
int pos=0;
MPI_Unpack(buffer, buffer_size, &pos, &a, 1, MPI_DOUBLE, MPI_COMM_WORLD);
MPI_Unpack(buffer, buffer_size, &pos, &b, 1, MPI_DOUBLE, MPI_COMM_WORLD);
MPI_Unpack(buffer, buffer_size, &pos, &nsteps, 1, MPI_INT, MPI_COMM_WORLD);
assert ( pos <= buffer_size );

// and finally delete the buffer
delete[] buffer;
```

All-to-all

- `MPI_Alltoall`: *n*-th rank sends *k*-th portion of its data to rank *k* and receives *n*-th portion from node *k*.
  - Everyone scatters and gathers at the same time
  - like a matrix transpose. Attention: slow!

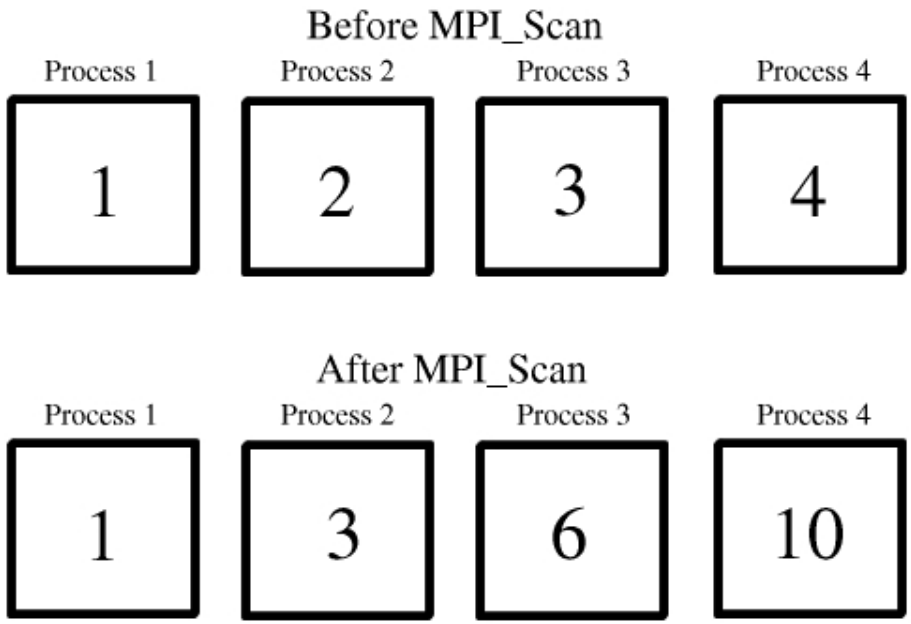
```
int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```



MPI_Scan + MPI_Exscan

- A scan or prefix-reduction operation performs partial reductions on distributed data.

```
int MPI_Scan(const void *sbuf, void *rbuf, int count, MPI_Datatype datatype,
             MPI_Op op, MPI_Comm comm)
// Computes the scan (partial reductions) of data on a collection of processes

int MPI_Exscan(const void *sbuf, void *rbuf, int count, MPI_Datatype datatype,
             MPI_Op op, MPI_Comm comm)
// Computes the exclusive scan (partial reductions) of data on a collection of processes
// MPI_Exscan is like MPI_Scan, except that the contribution from the calling process is
// not included in the result at the calling process (it is contributed to the subsequent
// processes, of course).
```



MPI_Scan + MPI_Exscan

- Example code

```
#include <mpi.h>
#include <math.h>
#include <stdio.h>

int main(int argc, char *argv[])
  int rank;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  double indata = 10.0*(rank+1);
  double outdata = 0;

  MPI_Scan(&indata, &outdata, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  //MPI_Exscan(&indata, &outdata, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

  printf("process %d: %f -> %f\n", rank, indata, outdata);

  MPI_Finalize();

  return 0;
}
```

```
$ mpirun -n 4 ./scan
process 0: 10.000000 -> 10.000000
process 1: 20.000000 -> 30.000000
process 2: 30.000000 -> 60.000000
process 3: 40.000000 -> 100.000000
```

```
$ mpirun -n 4 ./exscan
process 0: 10.000000 -> 0.000000
process 1: 20.000000 -> 10.000000
process 2: 30.000000 -> 30.000000
process 3: 40.000000 -> 60.000000
```

IV. HPC Simulations and I/O

- HPC applications perform I/O for many reasons
  - Write simulation data
    - for post processing and visualization
    - for checkpointing (fault tolerance, restarts)
  - Read data for processing
- The overhead of I/O can be significant and affect the overall simulation time
- File systems are a very important component of any HPC infrastructure

Parallel I/O

- A number of intermediate layers on top of which parallel applications can be developed
- They hide details of the low-level hardware layer, similarly to OpenMP and MPI
- Examples
  - MPI-I/O
  - parallel HDF5
  - parallel netCDF

Approaches to parallel I/O

- Sequential I/O : a master process collects and writes data to the file
  - Not scalable
- Parallel I/O: each process writes to its own file
  - Thousand of files
- Parallel I/O: multiple processes read from / write to a common file in parallel
  - The most common approach

MPI I/O

- Introduced with MPI-2 in 1997
- Idea: message passing from and to the parallel I/O system
  - receive = read
  - send = write
- Support of independent and collective operations
- Each MPI process reads or writes specific portions of a common file

Quick overview

- Available in mpio.h

```
int MPI_File_open(MPI_Comm comm, const char *filename,
                  int amode, MPI_Info info, MPI_File *fh);

#define MPI_MODE_RDONLY              2  /* ADIO_RDONLY */
#define MPI_MODE_RDWR                8  /* ADIO_RDWR  */
#define MPI_MODE_WRONLY              4  /* ADIO_WRONLY  */
#define MPI_MODE_CREATE              1  /* ADIO_CREATE */
#define MPI_MODE_EXCL               64  /* ADIO_EXCL */
#define MPI_MODE_DELETE_ON_CLOSE    16  /* ADIO_DELETE_ON_CLOSE */
#define MPI_MODE_UNIQUE_OPEN        32  /* ADIO_UNIQUE_OPEN */
#define MPI_MODE_APPEND            128  /* ADIO_APPEND */
#define MPI_MODE_SEQUENTIAL        256  /* ADIO_SEQUENTIAL */


int MPI_File_close(MPI_File *fh);
// Opens a file

int MPI_File_delete(const char *filename, MPI_Info info);
// Deletes a file

int MPI_File_set_size(MPI_File fh, MPI_Offset size);
// Sets the file size
// size [in]: size to truncate or expand file (nonnegative integer)

int MPI_File_preallocate(MPI_File fh, MPI_Offset size);
// Preallocates storage space for a file
// size [in]: size to preallocate (nonnegative integer)
```

Access modes

- The following access modes are supported (specified in amode, a bit vector OR of the following integer constants):
  - `MPI_MODE_RDONLY` : read only
  - `MPI_MODE_RDWR` : reading and writing
  - `MPI_MODE_WRONLY` : write only
  - `MPI_MODE_CREATE` : create the file if it does not exist
  - `MPI_MODE_EXCL` : error if creating file that already exists
  - `MPI_MODE_DELETE_ON_CLOSE` : delete file on close
  - `MPI_MODE_UNIQUE_OPEN` : file will not be concurrently opened elsewhere
  - `MPI_MODE_SEQUENTIAL` : file will only be accessed sequentially
  - `MPI_MODE_APPEND` : set initial position of all file pointers to end of file

Source: MPI specifications: http://mpi.deino.net/mpi_functions/MPI_File_open.html
http://www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-2.0/node175.htm

Possible approaches

1. Independent I/O
   - `MPI_File_seek +{MPI_File_read,MPI_File_write}`
   - `MPI_File_read_at, MPI_File_write_at`

2. Collective I/O
   - `MPI_File_seek +{MPI_File_read_all,MPI_File_write_all}`
   - `MPI_File_read_at_all, MPI_File_write_at_all`

3. Asynchronous independent I/O, e.g.
   - `MPI_File_seek +{MPI_File_iread,MPI_File_iwrite}`
   - `MPI_File_iread_at, MPI_File_iwrite_at`

4. Asynchronous collective I/O, e.g.
   - `MPI_File_seek +{MPI_File_iread_all,MPI_File_iwrite_all}`
   - `MPI_File_iread_at_all, MPI_File_iwrite_at_all`

Independent I/O

```
int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence);
// Updates the individual file pointer

#define MPI_SEEK_SET 600 // the pointer is set to offset
#define MPI_SEEK_CUR 602 // the pointer is set to the current pointer position plus offset
#define MPI_SEEK_END 604 // the pointer is set to the end of file plus offset (can be negative)

int MPI_File_read(MPI_File fh, void *buf, int count,  MPI_Datatype datatype,
                MPI_Status *status);
// Read using individual file pointer Updates the individual file pointer

int MPI_File_write(MPI_File fh, const void *buf, int count, MPI_Datatype datatype,
                MPI_Status *status);
// Read using individual file pointer
// status: contains the number of bytes successfully written



int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
                MPI_Datatype datatype, MPI_Status *status);
// Reads a file beginning at the position specified by offset

int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void * buf, int count,
                MPI_Datatype datatype, MPI_Status *status);
// Writes a file beginning at the position specified by offset
```

Collective I/O

```
int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence);

#define MPI_SEEK_SET            600
#define MPI_SEEK_CUR            602
#define MPI_SEEK_END            604


int MPI_File_read_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
                MPI_Status *status);
// Collective read using individual file pointer

int MPI_File_write_all(MPI_File fh, const void *buf, int count, MPI_Datatype datatype,
                MPI_Status *status);
// Collective write using individual file pointer




int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void * buf, int count,
                MPI_Datatype datatype, MPI_Status *status);
// Collective read using explicit offset

int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf, int count,
                MPI_Datatype datatype, MPI_Status *status);
// Collective write using explicit offset
```

Asynchronous independent I/O

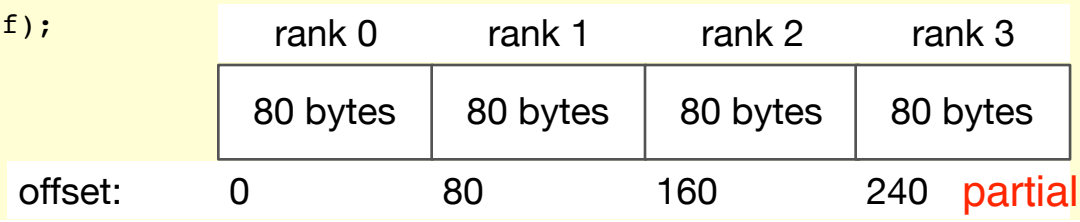
```
int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence);

#define MPI_SEEK_SET            600
#define MPI_SEEK_CUR            602
#define MPI_SEEK_END            604


int MPI_File_iread(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
            MPI_Request *request);
// Nonblocking read using individual file pointer

int MPI_File_iwrite(MPI_File fh, const void *buf, int count, MPI_Datatype datatype,
            MPI_Request *request);
// Nonblocking write using individual file pointer




int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
            MPI_Datatype datatype, MPI_Request *request);
// Nonblocking write using explicit offset

int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, const void * buf, int count,
            MPI_Datatype datatype, MPI_Request *request);
// Nonblocking write using explicit offset

#define MPIO_Wait MPI_Wait
#define MPIO_Test MPI_Test
```

Asynchronous collective I/O

```
int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence);

#define MPI_SEEK_SET            600
#define MPI_SEEK_CUR            602
#define MPI_SEEK_END            604


int MPI_File_iread_all(MPI_File fh, void *buf, int count,  MPI_Datatype datatype,
            MPI_Request *request);
// Nonblocking collective read using individual file pointer

int MPI_File_iwrite_all(MPI_File fh, const void *buf, int count, MPI_Datatype datatype,
            MPI_Request *request);
// Nonblocking collective write using individual file pointer




int MPI_File_iread_at_all(MPI_File fh, MPI_Offset offset, void *buf, int count,
            MPI_Datatype datatype, MPI_Request *request);
// Nonblocking collective read using explicit offset

int MPI_File_iwrite_at_all(MPI_File fh, MPI_Offset offset, const void * buf, int count,
            MPI_Datatype datatype, MPI_Request *request);
// Nonblocking collective write using explicit offset

#define MPIO_Wait MPI_Wait
#define MPIO_Test MPI_Test
```

Example 1a - Writing a binary file

```
int main(int argc, char **argv)
{
   int rank;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   const int nlocal = 10;
   double data[nlocal];  // every rank will write 10 doubles

   for (int i = 0; i < nlocal; i++) data[i] = 100*rank + i;  // initialization

   int step = 0;  // always zero, but not if it denotes the time step (e.g. diffusion)
   char filename[256];
   sprintf(filename, "mydata_%05d.bin", step);

   MPI_File f;
   MPI_File_open(MPI_COMM_WORLD, filename , MPI_MODE_WRONLY | MPI_MODE_CREATE, MPI_INFO_NULL, &f);

   MPI_File_set_size (f, 0);
   MPI_Offset base;
   MPI_File_get_position(f, &base);  // zero in this case

   MPI_Offset len = nlocal*sizeof(double);    // HOW MANY BYTES each rank will write
   MPI_Offset offset = rank*len;              // WHERE each rank will write
   MPI_Status status;

   MPI_File_write_at_all(f, base + offset, data, nlocal, MPI_DOUBLE, &status); // write

   MPI_File_close(&f);

   MPI_Finalize();

   return 0;
}
```



Example 1a - Additional options

- Collective I/O

```
MPI_File_write_at_all(f, base + offset, data, nlocal, MPI_DOUBLE, &status);
```

- Independent I/O

```
MPI_File_write_at(f, base + offset, data, nlocal, MPI_DOUBLE, &status);
```

- Asynchronous independent I/O

```
MPI_Request request;
MPI_File_iwrite_at(f, base + offset, data, nlocal, MPI_DOUBLE, &request); //at_all() for collective
/* do something useful here */
MPI_Wait(&request, &status);
```

If we perform I/O multiple times, then we can postpone the call to `MPI_Wait` until the next call to `MPI_File_iwrite_at`

Example 1b - Reading the binary file

```
const int nlocal = 10;
double data[nlocal]; // every rank will read 10 doubles

int step = 0;
char filename[256];

sprintf(filename, "mydata_%05d.bin", step);

MPI_File f;
MPI_File_open(MPI_COMM_WORLD, filename , MPI_MODE_RDONLY, MPI_INFO_NULL, &f);

MPI_Offset base;
MPI_File_get_position(f, &base);

MPI_Offset len = nlocal*sizeof(double);
MPI_Offset offset = rank*len;
MPI_Status status;

MPI_File_read_at_all(f, base + offset, data, nlocal, MPI_DOUBLE, &status);

MPI_File_close(&f);

if (rank == rank_to_print) // one rank prints what it read before
     for (int i = 0; i < nlocal; i++) {
        printf("data[%d] = %f\n", i, data[i]);
     }
```

```
$ ./printdata.sh mydata_00000.bin
0.000000 1.000000
2.000000 3.000000
4.000000 5.000000
6.000000 7.000000
8.000000 9.000000
100.000000 101.000000
102.000000 103.000000
104.000000 105.000000
106.000000 107.000000
108.000000 109.000000
200.000000 201.000000
202.000000 203.000000
204.000000 205.000000
206.000000 207.000000
208.000000 209.000000
300.000000 301.000000
302.000000 303.000000
304.000000 305.000000
306.000000 307.000000
308.000000 309.000000
```

You can also use the printdata.sh script to print the contents of the binary file.
https://gitlab.ethz.ch/hpcse17/hs2017/blob/master/examples/mpi3/mpiio/printdata.sh

Example 2a - Writing a binary file

```
const int nlocal = lrand48()%20;
double data[nlocal]; // every rank will write up to 20 doubles
for (int i = 0; i < nlocal; i++) data[i] = 100*rank + i;

int step = 0;
char filename[256];
sprintf(filename, "mydata_%05d.bin", step);

MPI_File f;
MPI_File_open(MPI_COMM_WORLD, filename , MPI_MODE_WRONLY | MPI_MODE_CREATE, MPI_INFO_NULL, &f);

MPI_File_set_size (f, 0);
MPI_Offset base;
MPI_File_get_position(f, &base);

MPI_Offset len = nlocal*sizeof(double);
MPI_Offset offset = 0;
MPI_Exscan(&len, &offset, 1, MPI_OFFSET, MPI_SUM, MPI_COMM_WORLD); // necessary exscan operation

MPI_Status status;
MPI_File_write_at_all(f, base + offset, data, nlocal, MPI_DOUBLE, &status);
```

Missing information!
How can the processes read the stored data without knowing the size?

Example 2b - Writing a binary file

```
const int nlocal = lrand48()%20;
double data[nlocal]; // every rank will write up to 20 doubles
for (int i = 0; i < nlocal; i++) data[i] = 100*rank + i;

int step = 0;
char filename[256];
sprintf(filename, "mydata_%05d.bin", step);

MPI_File f;
MPI_File_open(MPI_COMM_WORLD, filename , MPI_MODE_WRONLY | MPI_MODE_CREATE, MPI_INFO_NULL, &f);

MPI_File_set_size (f, 0);
MPI_Offset base;
MPI_File_get_position(f, &base);

MPI_Status status; // every rank writes first its number of doubles
MPI_File_write_at_all(f, base + rank*sizeof(int), &nlocal, 1, MPI_INT, &status);
base = base + nranks * sizeof(int); // new base: right after the header

MPI_Offset len = nlocal*sizeof(double);
MPI_Offset offset = 0;
MPI_Exscan(&len, &offset, 1, MPI_OFFSET, MPI_SUM, MPI_COMM_WORLD); // necessary exscan operation

MPI_File_write_at_all(f, base + offset, data, nlocal, MPI_DOUBLE, &status);
```

MPI_Exscan is needed here!

Example 2c - Reading the binary file

```
int nlocal;
double *data; //[nlocal];  // ranks do not know beforehand the number of doubles

int step = 0;
char filename[256];

sprintf(filename, "mydata_%05d.bin", step);

MPI_File f;
MPI_File_open(MPI_COMM_WORLD, filename, MPI_MODE_RDONLY, MPI_INFO_NULL, &f);

MPI_Offset base;
MPI_File_get_position(f, &base);

MPI_Status status;  // every rank reads the stored value of nlocal
MPI_File_read_at_all(f, base + rank*sizeof(int), &nlocal, 1, MPI_INT, &status);
base = base + nranks * sizeof(int); // new base: right after the header


MPI_Offset len = nlocal*sizeof(double); // size of bytes to read
MPI_Offset offset = 0;
MPI_Exscan(&len, &offset, 1, MPI_OFFSET, MPI_SUM, MPI_COMM_WORLD); // compute the offset

data = malloc(nlocal*sizeof(double)); // allocate the buffer and read your data
MPI_File_read_at_all(f, base + offset, data, nlocal, MPI_DOUBLE, &status);

MPI_File_close(&f);
```

How can we improve this?
Can we avoid MPI_Exscan in the reader?

Example 3a - Writing a binary file

```
typedef struct header_s
{
   int nlocal;
   MPI_Offset offset;
} header_t;

MPI_File f;
MPI_File_open(MPI_COMM_WORLD, filename , MPI_MODE_WRONLY | MPI_MODE_CREATE, MPI_INFO_NULL, &f);

MPI_File_set_size (f, 0);


MPI_Offset base;
MPI_File_get_position(f, &base);

MPI_Offset len = nlocal*sizeof(double);
MPI_Offset offset = 0;
MPI_Exscan(&len, &offset, 1, MPI_OFFSET, MPI_SUM, MPI_COMM_WORLD);

MPI_Status status;

header_t info;

info.nlocal = nlocal;  // number of elements
info.offset = offset;  // offset after the header

MPI_File_write_at_all(f, base + rank*sizeof(info), &info, sizeof(info), MPI_CHAR, &status);
base = base + nranks * sizeof(info);  // new base

MPI_File_write_at_all(f, base + offset, data, nlocal, MPI_DOUBLE, &status);
```

Header: number of elements + offset (for each rank)

One more improvement to add: precompute the size of the file and preallocate it (if possible)

Example 3b - Writing a binary file

```
typedef struct header_s
{
   int nlocal;
   MPI_Offset offset;
} header_t;

MPI_File f;
MPI_File_open(MPI_COMM_WORLD, filename , MPI_MODE_WRONLY | MPI_MODE_CREATE, MPI_INFO_NULL, &f);

MPI_Offset ntotal = 0;
MPI_Offset nbytes = sizeof(header_t) + nlocal*sizeof(double); // total bytes per rank

MPI_Allreduce(&nbytes, &ntotal, 1, MPI_OFFSET, MPI_SUM, MPI_COMM_WORLD);
MPI_File_preallocate(f, ntotal);

MPI_Offset base;
MPI_File_get_position(f, &base);

MPI_Offset len = nlocal*sizeof(double);
MPI_Offset offset = 0;
MPI_Exscan(&len, &offset, 1, MPI_OFFSET, MPI_SUM, MPI_COMM_WORLD);

MPI_Status status;

header_t info;

info.nlocal = nlocal;  // number of elements
info.offset = offset;  // offset after the header

MPI_File_write_at_all(f, base + rank*sizeof(info), &info, sizeof(info), MPI_CHAR, &status);
base = base + nranks * sizeof(info);  // new base

MPI_File_write_at_all(f, base + offset, data, nlocal, MPI_DOUBLE, &status);
```

Example 3c - Reading the binary file

```
typedef struct header_s
{
   int nlocal;
   MPI_Offset offset;
} header_t;  // header for the metadata

int nlocal;
double *data;

int step = 0;
char filename[256];

sprintf(filename, "mydata_%05d.bin", step);

MPI_File f;
MPI_File_open(MPI_COMM_WORLD, filename , MPI_MODE_RDONLY, MPI_INFO_NULL, &f);

MPI_Offset base;
MPI_File_get_position(f, &base);

MPI_Status status;
header_t info; // every rank reads and parses the corresponding information of the header
MPI_File_read_at_all(f, base + rank*sizeof(info), &info, sizeof(info), MPI_CHAR, &status);
int nlocal = info.nlocal;
MPI_Offset offset = info.offset;

data = malloc(nlocal*sizeof(double)); // allocate the buffer

base = base + nranks * sizeof(info);
MPI_File_read_at_all(f, base + offset, data, nlocal, MPI_DOUBLE, &status);

MPI_File_close(&f);
```

No MPI_Exscan in this case!

Example 4a - Writing a text file (C version)

- Fixed buffer size
  - With up to 32 characters for each number (on average)
- As before but we can see the contents of the output file

```
MPI_File_set_size (f, 0);
MPI_Offset base;
MPI_File_get_position(f, &base);

int bufsize = nlocal*32*sizeof(char);
char *sbuf = (char *)malloc(bufsize);
memset(sbuf, '\0', bufsize);

sprintf(sbuf, "%d ", rank);  // the buffer starts with the rank number
for(int i = 0; i < nlocal; ++i)
{
 char tmpbuf[32];
 sprintf(tmpbuf, "%.2f ", data[i]); // print the number to a string
 strcat(sbuf, tmpbuf); // and append the string to the buffer
}

printf("sbuf = %s\n", sbuf);

sbuf[strlen(sbuf)]=' ';  // some minor adjustments
sbuf[bufsize-1] = '\n';

MPI_Offset len = bufsize;   // number of bytes to write
MPI_Offset offset = rank*len; // offset
MPI_Status status;

MPI_File_write_at_all(f, base + offset, sbuf, len, MPI_CHAR, &status);
```

Example 4b - Writing a text file (C++ version)

- Dynamic buffer size using `std::stringstream` (C++)
- Different buffer size for each rank
  - MPI_Exscan is necessary

```
MPI_File_set_size (f, 0);
MPI_Offset base;
MPI_File_get_position(f, &base);

std::stringstream ss;
ss << rank << " ";
for(int i = 0; i < nlocal; ++i)
     ss << data[i] << " ";
ss << "\n";

string content = ss.str();

MPI_Offset len = content.size();
MPI_Offset offset = 0;
MPI_Exscan(&len, &offset, 1, MPI_OFFSET, MPI_SUM, MPI_COMM_WORLD);

MPI_Status status;

MPI_File_write_at_all(f, base + offset, const_cast<char *>(content.c_str()), len, MPI_CHAR,&status);
```

MPI I/O: additional notes

- The individual file pointer routines have the same semantics as the data access with explicit offset routines with the following modification:
  - the offset is defined to be the current value of the MPI-maintained individual file pointer
- After an individual file pointer operation is initiated, the individual file pointer is updated to point to the next element after the last one that will be accessed. The file pointer is updated relative to the current view of the file.
  - check the examples `file_offset_seek.c`, `file_offset_explicit.c` at the gitlab repository
- Write operations (e.g. `MPI_File_write`) return when the memory buffer (buf) can be safely reused (similarly to `MPI_Send`)
  - Return from the call does not guarantee that the data has been written to the storage device (or devices).

V. OpenMP nested parallelism

- `OMP_NESTED`: if the environment variable is set to TRUE, nested parallelism is enabled.
- In this case, each parallel directive creates a new team of threads.

```
#include <stdio.h>
#include <omp.h>
void nesting()
{
  #pragma omp parallel
  {
    int tid1 = omp_get_thread_num();
    #pragma omp parallel
    {
      int tid2 = omp_get_thread_num();
      #pragma omp critical
      printf("tid1 = %d, tid2 = %d\n", tid1, tid2);
    }
  }
}
```

nested parallelism can easily lead to processor oversubscription: #threads > #cores

Number of OpenMP threads

```
#include <omp.h>
#include <stdio.h>

void report_num_threads(int level)
{
    #pragma omp single
    {
        printf("Level %d: number of threads in the team = %d\n",
                  level, omp_get_num_threads());
    }
 }

int main()
{
    omp_set_dynamic(0);
    #pragma omp parallel num_threads(2)
    {
        report_num_threads(1);
        #pragma omp parallel num_threads(2)
        {
            report_num_threads(2);
            #pragma omp parallel num_threads(2)
            {
                report_num_threads(3);
            }
        }
    }
    return(0);
}
```

Output

```
$ export OMP_NESTED=false; ./nested_numthreads | sort
Level 1: number of threads in the team = 2
Level 2: number of threads in the team = 1
Level 2: number of threads in the team = 1
Level 3: number of threads in the team = 1
Level 3: number of threads in the team = 1

$ export OMP_NESTED=true; ./nested_numthreads | sort
Level 1: number of threads in the team = 2
Level 2: number of threads in the team = 2
Level 2: number of threads in the team = 2
Level 3: number of threads in the team = 2
Level 3: number of threads in the team = 2
Level 3: number of threads in the team = 2
Level 3: number of threads in the team = 2

$ export OMP_NESTED=true; export OMP_THREAD_LIMIT=6; ./nested_numthreads| sort
Level 1: number of threads in the team = 2
Level 2: number of threads in the team = 2
Level 2: number of threads in the team = 2
Level 3: number of threads in the team = 1
Level 3: number of threads in the team = 1
Level 3: number of threads in the team = 2
Level 3: number of threads in the team = 2
```

Nested loop parallelization - I

```
void work(int i, int j);

void nesting(int n)
{
  #pragma omp parallel
  {
    #pragma omp for
    for (int i=0; i<n; i++) {
      #pragma omp parallel
      {
        #pragma omp for
        for (int j=0; j<n; j++) {
          work(i, j);
        }
      }
    }
  }
}
```

several implicit barriers

Nested loop parallelization - II

```
void work(int i, int j);

void nesting(int n)
{
  #pragma omp parallel for
  for (int i=0; i<n; i++) {
    #pragma omp parallel for
    for (int j=0; j<n; j++) {
      work(i, j);
    }
  }
}
```

nested parallel regions

we avoided some implicit barriers

Nested loop parallelization - III

```
void work(int i, int j);

void nesting(int n)
{
  #pragma omp parallel for
  for (int k=0; k<n*n; k++) {
    int i = k / n;
    int j = k % n;
    work(i, j);
 }
}
```

loop fusion: we avoided nested parallelism

Basic loop transformations

- interchange: inner loops are exchanged with outer loops (see exercise 01)
- unrolling: the body of the loop is duplicated multiple times
- fusion: multiple loops are replaced with a single one (see above)
- fission: a single loop is broken into multiple loops over the same index range

Nested loop parallelization - IV

```
void work(int i, int j);

void nesting(int n)
{
  #pragma omp parallel for collapse(2)
  for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) {
      work(i, j);
    }
  }
}
```

collapse clause: let the OpenMP compiler do it for us

Unfortunately the collapse clause cannot be applied very often:

- if there is code between the for loops
- if the inner loop is in another function / library

Parallel loop and time evolution

- Identify any issues in the following codes

```
/* case 1 */
for (int step = 0; step < Nsteps; step++) {
  #pragma omp parallel for
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      A[i*N+j] = some_value(i, j, step);
    }
  }
}

/* case 2 */
#pragma omp parallel
{
  for (int step = 0; step < Nsteps; step++) {
    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
        A[i*N+j] = some_value(i, j, step);
      }
    }
  }
}
```

Overhead of parallel

Many issues here (see next)

Parallel loop and time evolution

- Understand the behavior of the following code

```
/* case 2 */
#pragma omp parallel
{
  for (int step = 0; step < Nsteps; step++) {
    <what should I put here?>
    for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
        A[i*N+j] = some_value(i, j, step);
      }
    }
  }
}
```

- Three options:
  - nothing — no work sharing
  - **#pragma omp for** — correct
  - **#pragma omp parallel for** — no actual work sharing

VI. Hybrid codes

- We finally want to combine all we used so far
  - Message passing between nodes
  - Multithreading on a single node
  - SIMD vectorization on a single core
- Example for PDGEMM
  - Distribute the matrices over nodes
  - Split the matrix into smaller blocks on each node
  - Finally vectorize the in-cache multiplication of the smallest blocks
- Potential problem: is MPI communication thread-safe?
  - Your MPI library might not care about thread-safety and you thus cannot make concurrent MPI calls
  - It can be worse: MPI might use an incompatible threading library to implement asynchronous communication. Your code might crash if it tries to launch a thread

Using MPI in a multithreaded context

- You need to call a special initialization function to use MPI with threads instead of the standard `MPI_Init`:

```
int MPI_Init_thread( int *argc, char ***argv, int required, int *provided )
// required is the threading support you desire
// provided is what the library supports and can be less
```

Level of thread support	Description
MPI_THREAD_SINGLE	Only a single thread can execute
MPI_THREAD_FUNNELED	The process may be multi-threaded, but only the main thread will make MPI calls (all MPI calls are funneled to the main thread).
MPI_THREAD_SERIALIZED	The process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads (all MPI calls are serialized).
MPI_THREAD_MULTIPLE	Multiple threads may call MPI, with no restrictions.

Why use hybrid MPI?

- Less memory use since threads can share data
  - N-body codes: no need to duplicate particle positions of other threads
  - PDE codes: no need for ghost cells within a node
- “Easier” to program
  - MPI requires explicit communication
  - Using threading within a node we can keep the MPI communication at a more coarse grained level
- Performance advantages
  - use multi-threaded libraries on a node, e.g. multi-threaded BLAS libraries

Hybrid programming styles

- Many ways to combine MPI processes and threads
  - One MPI process per node
  - One MPI process per socket (avoids NUMA issues)
  - Multiple MPI processes per socket, each with threads
- Many ways to use threads
  - “vector mode”: communication regions done by one thread followed by parallel loops done by all threads. Similar to using vector instructions.
  - “task mode”: one or more thread are responsible for communication, others do computation

Hybrid hello world

- No communication takes place

```
#include <stdio.h>
#include <mpi.h>
#include <omp.h>

int main(int argc, char** argv)
{
  int rank, size;

  MPI_Init(&argc,&argv);   // better use MPI_Init_thread() instead of this

  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  omp_set_num_threads(2);

  #pragma omp parallel
  {
    #pragma omp critical
    printf("Hello, world.  I am %d of %d and thread=%d\n", rank, size, omp_get_thread_num());
  }

  MPI_Finalize();
  return 0;
}
```

Hybrid integration example

- Use OpenMP in Simpson integration

```
inline double simpson(double (*f) (double), double a, double b, unsigned int N)
{
  double      h = (b-a)/N;
  double result = ( f(a) + 4*f(a+h/2) + f(b) ) / 2.0;
  #pragma omp parallel for reduction(+ : result)
  for ( unsigned int i = 1; i <= N-1; ++i )
        result += f(a+i*h) + 2*f(a+(i+0.5)*h);
    return result * h / 3.0;
}
```

- And check for thread support in MPI part

```
int main(int argc, char** argv)
{
  int provided;
  MPI_Init_thread(&argc,&argv,MPI_THREAD_FUNNELED,&provided);
  // we need to be able to communicate at least from the main thread
  assert(provided >= MPI_THREAD_FUNNELED);

  ...

  double delta = (p.b-p.a)/size;
  double result = simpson(func,p.a+rank*delta,p.a+(rank+1)*delta,p.nsteps/size);
  MPI_Reduce(rank==0 ? MPI_IN_PLACE : &result, &result, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

  if (rank==0)
    std::cout << result << std::endl;

  MPI_Finalize();
  return 0;
}
```

Hybrid programming style

- How do we spawn a special OpenMP thread for communication?

```
#include <omp.h>

int main(int argc, char *argv[])
{
  // initialize the MPI library with thread support

  omp_set_dynamic(0);
  #pragma omp parallel num_threads(2)
  {
    if (omp_getthread_num()==0)
    {
      // do communication
      ...
    }
    else
    {
    #pragma omp parallel
      {
        // do parallel work with remaining threads
        ...
      }

    }
  }
}
```

Parallel dot product

Sequential code

```
#include <iostream>
#include <vector>

int main(int argc, int char** argv) {
    const unsigned int n = 1<<12;

    std::vector<double> v(n, 2.);
    std::vector<double> w(n, 1.5);

    double result = 0.;
    for (unsigned int i=0; i<n; ++i) {
        result += v[i]*w[i];
    }

    std::cout << "Result: " << result << std::endl;

    return 0;
}
```

Parallel dot product

OpenMP code

```
#include <iostream>
#include <vector>

int main(int argc, int char** argv) {
    const unsigned int n = 1<<12;

    std::vector<double> v(n, 2.);
    std::vector<double> w(n, 1.5);

    double result = 0.;
#pragma omp parallel for reduction(+:result)
    for (unsigned int i=0; i<n; ++i) {
        result += v[i]*w[i];
    }

    std::cout << "Result: " << result << std::endl;

    return 0;
}
```

Parallel dot product

MPI code

```
#include <iostream>
#include <vector>
#include <mpi.h>

int main(int argc, int char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    const unsigned int n = 1<<12;

    const unsigned int start = n*rank / size;
    const unsigned int end   = n*(rank+1) / size;
    const unsigned int chunk_size = end - start;

    std::vector<double> v(chunk_size, 2.);
    std::vector<double> w(chunk_size, 1.5);

    double local_result = 0.;
    for (unsigned int i=0; i<chunk_size; ++i) {
        local_result += v[i]*w[i];
    }

    double result;
    MPI_Reduce(&local_result, &result, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0)
      std::cout << "Result: " << result << std::endl;

    MPI_Finalize();

    return 0;
}
```

Parallel dot product

Hybrid MPI+OpenMP code

```
#include <iostream>
#include <vector>
#include <mpi.h>

int main(int argc, int char** argv) {
    int provided;
    MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);

    if (provided < MPI_THREAD_FUNNELED) {
        std::cerr << "Error: MPI threading level is not enough." << std::endl;
        MPI_Finalize();
        return 1;
    }

    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    const unsigned int n = 1<<12;

    const unsigned int start = n*rank / size;
    const unsigned int end   = n*(rank+1) / size;
    const unsigned int chunk_size = end - start;

    std::vector<double> v(chunk_size, 2.);
    std::vector<double> w(chunk_size, 1.5);

    double local_result = 0.;
#pragma omp parallel for reduction(+:local_result)
    for (unsigned int i=0; i<chunk_size; ++i) {
        local_result += v[i]*w[i];
    }

    double result;
    MPI_Reduce(&local_result, &result, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) std::cout << "Result: " << result << std::endl;

    MPI_Finalize();
    return 0;
}
```

Enabling hybrid MPI

- Many platforms require special linker or runtime options

Platform	Enabling multithreaded MPI
MPICH	The library must have been compiled with support for multi-threaded applications. Not all the communication channels support MPI_THREAD_MULTIPLE.
MPAVICH2 (Euler)	Set environment variable MV2_ENABLE_AFFINITY=0
OpenMPI (Euler)	The default module (open_mpi/1.6.5) does not support threads. You can use a newer version, e.g.: module load new gcc/4.8.2 open_mpi/1.10.0 For more information: https://scicomp.ethz.ch/wiki/OpenMPI https://scicomp.ethz.ch/wiki/Hybrid_jobs
Cray	Set the environment variable MPICH_MAX_THREAD_SAFETY to one of single, funneled, serialized, multiple