MOVING MPI APPLICATIONS TO THE NEXT LEVEL

Adrian Jackson
adrianj@epcc.ed.ac.uk
@adrianjhpc
MPI

- Core tool for computational simulation
  - De facto standard for multi-node computations
- Wide range of functionality
  - 4+ major revisions of the standard
  - Point-to-point communications
  - Collective communications
  - Single side communications
  - Parallel I/O
  - Custom datatypes
  - Custom communication topologies
  - Shared memory functionality
  - etc…
- Most applications only use a small amount of MPI
  - A lot are purely MPI 1.1, or MPI 1.1 + MPI I/O
- Fine but may leave some performance on the table
  - Especially at scale
Tip…

- Write your own wrappers to the MPI routines you’re using
  - Allows substituting MPI calls or implementations without changing application code
  - Allows auto-tuning for systems
  - Allows profiling, monitoring, debugging, without hacking your code
  - Allows replacement of MPI with something else (possibly)
  - Allows serial code to be maintained (potentially)

```fortran
! parallel routine
subroutine par_begin(size, procid)
  implicit none
  integer :: size, procid
  include "mpif.h"
  call mpi_init(ierr)
  call mpi_comm_size(MPI_COMM_WORLD, size, ierr)
  call mpi_comm_rank(MPI_COMM_WORLD, procid, ierr)
  procid = procid + 1
end subroutine par_begin

! dummy routine for serial machine
subroutine par_begin(size, procid)
  implicit none
  integer :: size, procid
  size = 1
  procid = 1
end subroutine par_begin
```
Performance issues

- Communication cost
  - Synchronisation
- Load balance
  - Decomposition
- Serial code
  - I/O
Synchronisation

- Synchronisation forces applications to run at speed of slowest process
  - Not a problem for small jobs
  - Can be significant issue for larger applications
  - Amplifies system noise

- MPI_Barrier is almost never required for correctness
  - Possibly for timing, or for asynchronous I/O, shared memory segments, etc.
  - Nearly all applications don’t need this or do this

- In MPI most synchronisation is implicit in communication
  - Blocking sends/receives
  - Waits for non-blocking sends/receives
  - Collective communications synchronise
Communication patterns

- A lot of applications have weak synchronisation patterns
  - Dependent on external data, but not on all processes
  - Ordering of communications can be important for performance
Common communication issues
Common communication issues
Standard optimisation approaches

- Non-blocking point to point communications
  - Split start and completion of sending messages
  - Split posting receives and completing receives
  - Allow overlapping communication and computation
  - **Post receives first**

```fortran
! Array of ten integers
integer, dimension(10) :: x
integer :: reqnum
integer, dimension(MPI_STATUS_SIZE) :: status

if (rank .eq. 1)
CALL MPI_ISSEND(x, 10, MPI_INTEGER, 3, 0,
  MPI_COMM_WORLD, reqnum, ierr)

if (rank .eq. 1)
CALL MPI_WAIT(reqnum, status, ierr)
```
Message progression

• However…
  • For performance reasons MPI library is (generally) not a stand alone process/thread
  • Simply library calls from the application
• Non-blocking messages theoretically can be sent asynchronously
  • Most implementations only send and receive MPI messages in MPI function calls

! Array of ten integers
integer, dimension(10) :: x
integer :: reqnum
integer, dimension(MPI_STATUS_SIZE) :: status

......

if (rank .eq. 1)
CALL MPI_ISSEND(x, 10, MPI_INTEGER, 3, 0,
  MPI_COMM_WORLD, reqnum, ierr)

......

if (rank .eq. 1)
CALL MPI_WAIT(reqnum, status, ierr)
Non-blocking for fastest completion

- However, non-blocking still useful….
  - Allows posting of receives before sending happens
    - Allows MPI library to efficiently receive messages (copy directly into application data structures)
  - Allows progression of messages that arrive first
    - Doesn’t force programmed message patterns on the MPI library
- Some MPI libraries can generate *helper* threads to progress messages in the background
  - i.e. Cray NEMESIS threads
  - Danger that these interfere with application performance (interrupt CPU access)
    - Can be mitigated if there are spare hyperthreads
- You can implement your own helper threads
  - OpenMP section, pthread implementation
  - Spin wait on MPI_Probe or similar function call
  - Requires thread safe MPI (see later)
- Also non-blocking collectives in MPI 3 standard
  - Start collective operations, come back and check progression later
Alternatives to non-blocking

- If non-blocking used to provide *optimal* message progression
  - i.e. no overlapping really possible

- Neighborhood collectives
  - MPI 3.0 functionality
  - Non-blocking collective on defined topology
  - Halo/neighbor exchange in a single call
  - Enables MPI library to optimise the communication

```c
MPI_NEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPe, COMM, IERROR)
```

```c
int MPI_Ineighbor_alltoall(const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf, int recvcount,
    MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
```
Topologies

- **Cartesian topologies**
  - each process is *connected* to its neighbours in a virtual grid.
    - boundaries can be cyclic
    - allow re-order ranks to allow MPI implementation to optimise for underlying network interconnectivity.
  - processes are identified by Cartesian coordinates.

```c
int MPI_Cart_create(MPI_Comm comm_old,
                    int ndims, int *dims, int *periods,
                    int reorder, MPI_Comm *comm_cart)
```

- **Graph topologies**
  - general graphs

- **Some MPI implementations will re-order ranks too**
  - Minimise communication based on message patterns
    - Keep MPI communications with a node wherever possible
Load balancing

- Parallel performance relies on sensible load balance
- Domain decomposition generally relies on input data set
- If partitions >> processes can perform load balancing
  - Use graph partitioning package or similar
  - i.e. metis
- Communication costs also important
  - Number and size of communications dependent on decomposition
- Can also reduce cost of producing input datasets
Sub-communicators

- MPI_COMM_WORLD fine but…
  - If collectives don’t need all processes it’s wasteful
  - Especially if data decomposition changes at scale
  - Can create own communicators from MPI_COMM_WORLD

```c
int MPI_Comm_split(MPI_Comm comm, int colour, int key, MPI_Comm *newcomm)
MPI_COMM_SPLIT(COMM, COLOUR, KEY, NEWCOMM, IERROR)
```
  - colour – controls assignment to new communicator
  - key – controls rank assignment within new communicator
Data decomposition

- May need to reconsider data decomposition decisions at scale
- May be cheaper to communicate data to subset of process and compute there
  - Rather than compute partial sums and do reductions on those
  - Especially if the same dataset is used for a set of calculation
Data decomposition

- May also need to consider damaging load balance (a bit) if you can reduce communications
Data decomposition

\[ \text{isgn} = 1 \quad \text{isgn} = 2 \]

\[ \text{ig} \quad \text{ig} \]

\[ \text{y} \]

\[ \begin{array}{cccc}
0 & 1 & 2 & 3 \\
\end{array} \]

\[ \text{xxf} \]

\[ \begin{array}{cccc}
0 & 1 & 2 & 3 \\
\end{array} \]

\[ \text{yxf} \]

\[ \begin{array}{cccc}
0 & 1 & 2 & 3 \\
\end{array} \]

\[ \begin{array}{cccc}
k-2 & k-1 & k & k+1 \\
\end{array} \]

\[ \begin{array}{cccc}
0 & 1 & 2 & 3 \\
\end{array} \]

\[ \begin{array}{cccc}
k-2 & k-1 & k & k+1 \\
\end{array} \]

\[ \begin{array}{cccc}
n-2 & n-1 \\
\end{array} \]

\[ \begin{array}{cccc}
k-2 & k-1 & k & k+1 \\
\end{array} \]

\[ \begin{array}{cccc}
n-2 & n-1 \\
\end{array} \]
Distributed Shared Memory (clusters)

- Dominant architecture is a hybrid of these two approaches: *Distributed Shared Memory.*
  - Due to most HPC systems being built from commodity hardware – trend to multicore processors.
  - Each Shared memory block is known as a *node.*
  - Usually 16-64 cores per node.
  - Nodes can also contain accelerators.

- Majority of users try to exploit in the same way as for a purely distributed machine
  - As the number of cores per node increases this can become increasingly inefficient…
  - …and programming for these machines can become increasingly complex
Hybrid collectives

- Sub-communicators allow manual construction of topology aware collectives
- One set of communicators within a node, or NUMA region
- Another set of communicators between nodes
- e.g.

```c
MPI_Allreduce(...,MPI_COMM_WORLD)
```

becomes

```c
MPI_Reduce(...,node_comm)
if(node_comm_rank == 0){
   MPI_Allreduce(...,internode_comm)
}
MPI_Bcast(...,node_comm)
```
Hybrid collectives

Split collective - Cray

My Allreduce (small)

MPI Allreduce (small)

My Allreduce (medium)

MPI Allreduce (medium)

My Allreduce (large)

MPI Allreduce (large)
Hybrid collectives

split collective - Infiniband cluster

My Allreduce (small)
MPI Allreduce (small)

My Allreduce (medium)
MPI Allreduce (medium)
Hybrid collectives

split collective - Xeon Phi Knights Landing

split collective - Xeon Phi Knights Landing

split collective - Xeon Phi Knights Landing

My Allreduce (large)

My Allreduce (small)

My Allreduce (medium)

MPI Allreduce (large)

MPI Allreduce (small)

MPI Allreduce (medium)
Shared memory

- Shared memory nodes provide shared memory 😊
- Potential for bypassing MPI library altogether in a node
  - MPI call have overheads; function call, message queues, progression, etc.
  - There are mechanisms for sharing memory between groups of processes
- Shared memory segments

```c
static double *data_area=NULL;
if(local_rank == 0){
    /* create a file for token generation */
    sprintf(fname,"/tmp/segsum.%d",getuid());
    fd = open(  fname,O_RDWR | O_CREAT, 0644);
    if( fd < 0 ){
        perror(fname);
        MPI_Abort(MPI_COMM_WORLD,601);
    }
    close(fd);
    segkey=ftok(fname, getpid());
    unlink(fname);
    shm_id =shmget(segkey,plan_comm.local_size*datasize*segsize,IPC_CREAT | 0644);
    if( shm_id == -1 ){
        perror("shmget");
        printf("%d\n",shm_id);
        MPI_Abort(MPI_COMM_WORLD,602);
    }
    MPI_Bcast(&shm_id,1,MPI_INT,0,plan_comm.local_comm);
    shm_seg =  shmat(shm_id,(void *) 0,0);
    if( shm_seg == NULL || shm_seg == (void *) -1 ){
        MPI_Abort(MPI_COMM_WORLD,603);
    }
    data_area = (double *)((char *)shm_seg);
```
Shared memory collectives

- Sub-communicators between nodes
- Shared memory within a node
- e.g.

\[
\text{MPI\_Allreduce(\ldots, MPI\_COMM\_WORLD)}
\]

becomes

```c
data_area[i*\text{node\_comm\_rank}] = a;
MPI\_Barrier(\text{node\_comm});
if(\text{node\_comm\_rank} == 0){
    for(i=1;i<\text{node\_comm\_size};i++){\text{\ldots}}
    \text{data\_area}[0] += \text{data\_area}[i];
}
\text{MPI\_Allreduce(\text{data\_area}[0], \ldots, internode\_comm)}
}\text{\ldots}
\text{MPI\_Barrier(\text{node\_comm});
\text{a=\text{data\_area}[0];}\n```
Shared memory collectives

shared memory collective - Xeon Phi Knights Landing

- My Allreduce (small)
- MPI Allreduce (small)

shared memory collective - Xeon Phi Knights Landing

- My Allreduce (medium)
- MPI Allreduce (medium)

shared memory collective - Xeon Phi Knights Landing

- My Allreduce (large)
- MPI Allreduce (large)
Shared memory

- Shared memory segments can be directly written/read by processes
  - With great power....
- Also somewhat non-portable, and segment clean-up can be an issue
  - Crashed programs leave segments lying around
  - Sysadmins need to have scripts to clean them up
- MPI 3 has shared memory functionality
  - MPI Windows stuff, building on previous single sided functionality
  - Portable shared memory

```c
MPI_Comm shmcomm;
MPI_Comm_split_type (MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, 0, MPI_INFO_NULL, &shmcomm)
MPI_Win_allocate_shared (alloc_length, 1, info, shmcomm, &mem, &win);
MPI_Win_lock_all (MPI_MODE_NOCHECK, win);
mem[0] = rank;
mem[1] = numtasks;
memcpy(mem+2, name, namelen);
MPI_Win_sync (win);
MPI_Barrier (shmcomm);
```
MPI + X

- Shared memory cluster
  - Hybrid architecture
  - Mixture of shared memory and distributed memory
- Hybrid parallelisation
  - Mixture of two different parallelisation strategies
  - Distributed memory and shared memory
  - Optimal communication structure
- (Potential) Benefits
  - Utilise fastest available communications
  - Share single resources within nodes
  - Scale limited decomposition/datasets
  - Address MPI library overheads
  - Efficiently utilise many-thread resources
MPI + OpenMP

(Potential) Drawbacks

- Hybrid parallel overheads
  - Two parallel overheads rather than one
  - Each OpenMP section costs
- Coverage
  - Struggle to completely parallelise

MPI libraries well optimised

- Communications as fast on-node as OpenMP
- A lot of applications not currently in region of problems with MPI library

Shared memory technology has costs

- Memory bandwidth
- NUMA costs
- Limited performance range
COSA – CFD code

COSA Hybrid Performance

- MPI
- Hybrid (4 threads)
- Hybrid (3 threads)
- Hybrid (2 threads)
- Hybrid (6 threads)
- MPI Scaling if continued perfectly
- MPI Ideal Scaling

Tasks (either MPI processes or MPI processes x OpenMP Threads)

Runtime (seconds)
COSA – Power efficiency

![Graph showing power efficiency vs. number of nodes for different systems.](image)
MPI+Threads

- How to handle MPI communications, what level of threaded MPI communications to support/require?
  - MPI_Init_thread replaces MPI_Init
    - Supports 4 different levels:
      - **MPI_THREAD_SINGLE** Only one thread will 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.
  - Where to do MPI communications:
    - Single or funneled:
      - Pros: Don’t have to change MPI implemented in the code
      - Cons: Only one thread used for communications leaves cores inactive, not parallelising all the code
    - Serialized
      - Pros: Can parallelism MPI code using OpenMP as well, meaning further parallelism
      - Cons: Still not using all cores for MPI communications, requires thread safe version of the MPI library
    - Multiple:
      - Pros: All threads can do work, not leaving idle cores
      - Cons: May requires changes to MPI code to create MPI communicators for separate threads to work on, and for collective communications. Can require ordered OpenMP execution for MPI collectives, experience shows fully threaded MPI implementations slower than ordinary MPI
MPI Hybrid Performance - Cray

![Graph showing MPI Hybrid Performance](image)
Using node resources

• Might be tempting to have a single MPI process per node

• Definitely needs multiple MPI processes per node
  • Certainly one per NUMA region
  • Possibly more to exploit network links/injection bandwidth
  • Need to care about process binding

• i.e. 2 processor node
  • At least 2 MPI processes, one per processor
  • may need 4 or more to fully exploit the network

• i.e. 1 KNL node
  • At least 4 MPI processes, one per quadrant
Manycore

• Hardware with many cores now available for MPI applications
  • Moving beyond SIMD units accessible from an MPI process
  • Efficient threading available

• Xeon Phi particularly attractive for porting MPI programs
  • Simply re-compile and run
  • Direct user access
  • Problem/Benefit

• Suggested model for Xeon Phi
  • OpenMP
  • MPI + OpenMP
  • MPI?.....
MPI Performance - PingPong

![Graph showing MPI performance for different latency scenarios and message sizes. The graph includes lines for different scenarios: Host latency 2 procs, KNC latency 2 procs, Host latency 16 procs, KNC latency 240 procs, and KNI latency 64 procs. The x-axis represents message size in bytes, ranging from 1 to 1,048,576, and the y-axis represents latency in microseconds, ranging from 0.1 to 10,000. Each line depicts the performance under varying conditions, highlighting trends and comparisons.]
MPI Performance - Allreduce

MPI_Allreduce KNC, KNL and host

Average time (microseconds)

Message size (bytes)
MPI Performance – PingPong – Memory modes

PINGPONG BANDWIDTH (MB/s)

MESSAGE SIZE (BYTES)

KNL Bandwidth 64 procs

KNL Fastmem bandwidth 64 procs
MPI Performance – PingPong – Memory modes

Latency (microseconds) vs Message size (Bytes)

- KNL latency 64 procs
- KNL Fastmem latency 64 procs
- KNL cache mode latency 64 procs
MPI_Allreduce KNL different memory modes for 2 and 64 processor benchmarks

- KNL 2 procs
- KNL 2 procs fastmem
- KNL 2 procs cache mode
- KNL 64 procs
- KNL 64 procs fastmem
- KNL 64 procs cache mode

Average time (microseconds) vs. Message size (bytes)
MPI + MPI

- Reduce MPI process count on node

- MPI runtime per node or NUMA region/network end point

- On-node collective optimisation
  - Shared-memory segment + planned collectives
  - http://www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0409.pdf
Planned Alltoallv performance - Cray

![Graph showing performance times for different Alltoallv implementations and MPI processes. The x-axis represents the number of MPI processes, and the y-axis represents time in microseconds. Different markers indicate different implementations: My Alltoallv (small), MPI AlltoAllv (small), My Alltoallv (medium), MPI AlltoAllv (medium).]
Planned Alltoallv performance – Infiniband cluster
Planned Alltoallv Performance - KNL
I/O

- Any serial portion of a program will limit performance
- I/O needs to be parallel
- Even simply reading a file from large process counts can be costly
- Example:
  - Identified that reading input is now significant overhead for this code
    - Output is done using MPI-I/O, reading is done serially
    - File locking overhead grows with process count
    - Large cases ~GB input files
  - Parallelised reading data
    - Reduce file locking and serial parts of the code
  - One or two orders of magnitude improvement in performance at large process counts
    - 1 minute down to 5 seconds
- Don’t necessarily need to use MPI-I/O
  - netCDF/HDF5/etc… can provide parallel performance
  - Best performance likely to be MPI-I/O
  - Also need to consider tuning filesystem (i.e. lustre striping, gfps)
Summary

• Basic MPI functionality fine for most
• Only need to optimise when scaling issues are apparent
  • Basic performance measuring/profiling essential before doing any optimisation

• MPI implementations do a lot of nice stuff for you
  • However, there can be scope for more involved communication work yourself

• Understanding your data decomposition and where calculated values are required essential
  • This may change at scale

• There are other things I could have talked about
  • Derived data types, persistent communications,…

• We’re looking for your tips, tricks, and gotchas for MPI
  • Please contact me if you have anything you think would be useful!