Advanced OpenMP

MPI and OpenMP

Overview

- Motivation
- Potential advantages of MPI + OpenMP
- Problems with MPI + OpenMP
- Styles of MPI + OpenMP programming
  - MPI’s thread interface

Motivation

- With the ubiquity of multicore chips, almost all current CPU systems are *clustered architectures*
- Distributed memory systems, where each node consist of a shared memory multiprocessor (SMP).
- Single address space within each node, but separate nodes have separate address spaces.

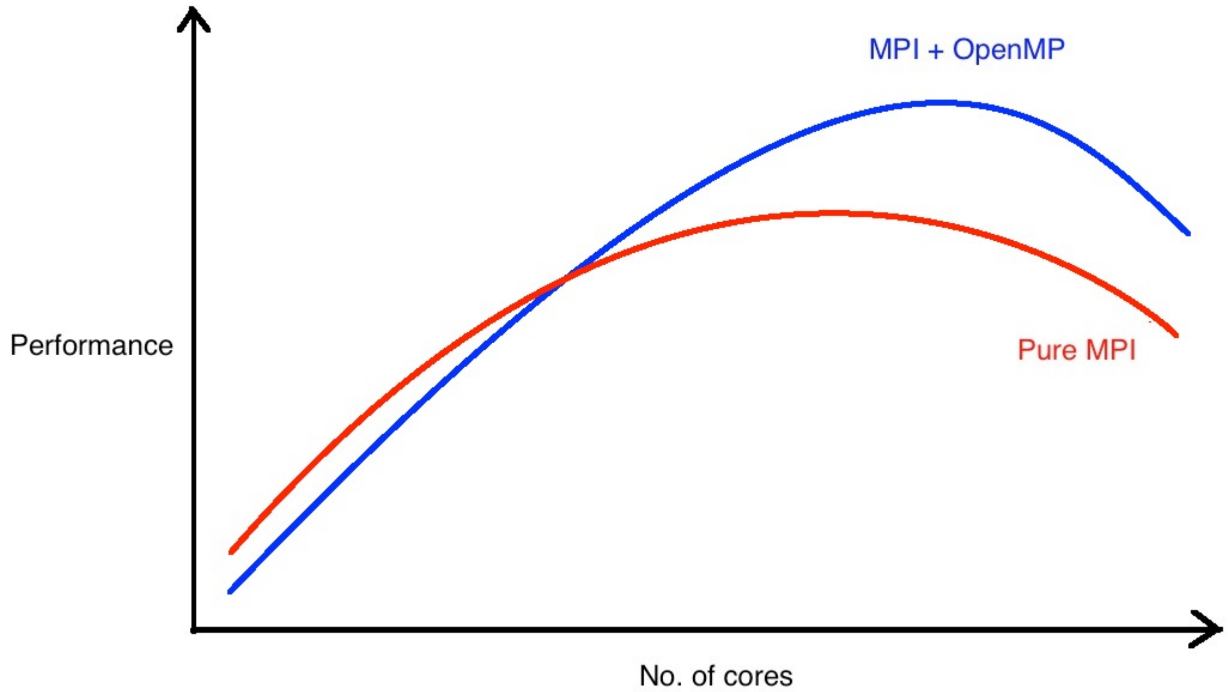
Programming clusters

- How should we program such a machine?
- Could use MPI across whole system
- Cannot (in general) use OpenMP/threads across whole system
  - requires support for single address space
  - this is possible in software, but inefficient
  - also possible in hardware, but expensive
- Could use OpenMP/threads within a node and MPI between nodes
  - is there any advantage to this?

Expectations

- In general, MPI + OpenMP does not improve performance (and may be worse!) in the regime where the MPI application is scaling well.
- Benefits come when MPI scalability (either in time or memory) starts to run out
- MPI + OpenMP may extend scalability to larger core counts

Typical performance curves



Potential advantages of MPI + OpenMP

- Reducing memory usage
- Exploiting additional levels of parallelism
- Reducing load imbalance
- Reducing communication costs

Reducing memory usage

- Some MPI codes use a replicated data strategy
  - all processes have a copy of a major data structure
- Classical domain decomposition codes have replication in halos
- MPI internal message buffers can consume significant amounts of memory
- A pure MPI code needs one copy per process/core.
- A mixed code would only require one copy per node
  - data structure can be shared by multiple threads within a process
  - MPI buffers for intra-node messages no longer required
- Will be increasingly important
  - amount of memory per core is not likely to increase in future

Effect of domain size on halo storage

- Typically, using more processors implies a smaller domain size per processor
  - unless the problem can genuinely weak scale
- Although the amount of halo data does decrease as the local domain size decreases, it eventually starts to occupy a significant amount fraction of the storage
  - even worse with deep halos or >3 dimensions

Local domain size	Halos	% of data in halos
$50^3 = 125000$	$52^3 – 50^3 = 15608$	11%
$20^3 = 8000$	$22^3 – 20^3 = 2648$	25%
$10^3 = 1000$	$12^3 – 10^3 = 728$	42%

Exploiting additional levels of parallelism

- Some MPI codes do not scale beyond a certain core count because they run of of available parallelism at the top level.
- However, there may be additional lower levels of parallelism that can be exploited.
- In principle, this could also be done using MPI.
- In practice this can be hard
  - The lower level parallelism may be hard to load balance, or have irregular (or runtime determined) communication patterns.
  - May be hard to work around design decisions in the original MPI version.

- It may, for practical reasons, be easier to exploit the additional level(s) of parallelism using OpenMP threads.
- Can take an incremental (e.g. loop by loop) approach to adding OpenMP
  - maybe not performance optimal, but keeps development cost/time to a minimum.
- Obviously OpenMP parallelism cannot extend beyond a single node, but this may be enough
  - future systems seem likely to have more cores per nodes, rather than many more nodes

Reducing load imbalance

- Load balancing between MPI processes can be hard
  - need to transfer both computational tasks and data from overloaded to underloaded processes
  - transferring small tasks may not be beneficial
  - having a global view of loads may not scale well
  - may need to restrict to transferring loads only between neighbours
- Load balancing between threads is much easier
  - only need to transfer tasks, not data
  - overheads are lower, so fine grained balancing is possible
  - easier to have a global view
- For applications with load balance problems, keeping the number of MPI processes small can be an advantage

Reducing communication costs

- It is natural to suppose that communicating data inside a node is faster between OpenMP threads between MPI processes.
    - no copying into buffers, no library call overheads
- True, but there are lots of caveats – see later.
- In some cases, MPI codes communicate more data than is actually required
    - where actual data dependencies may be irregular and/or data-dependent
    - makes implementation easier

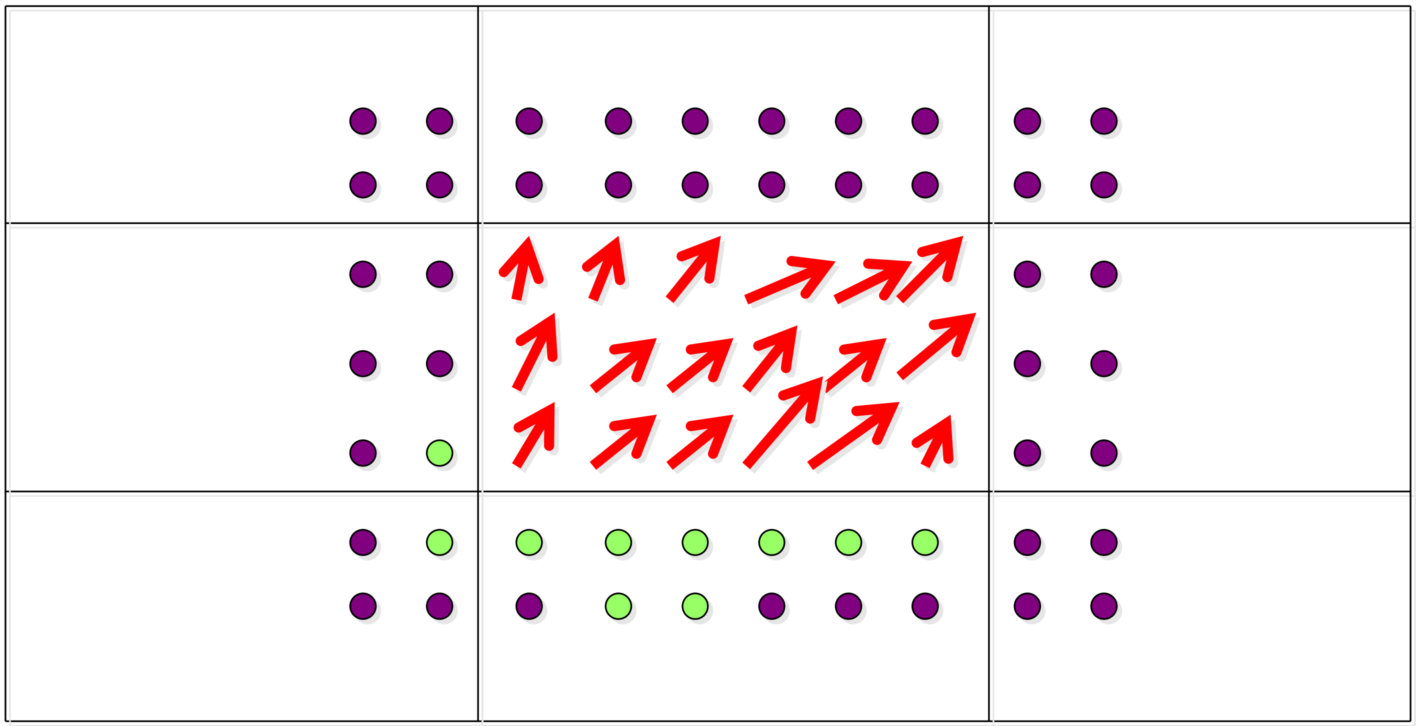
Collective communication

- In some circumstances, collective communications can be improved by using MPI + OpenMP
  - e.g. AllReduce, AlltoAll
- In principle, the MPI implementation ought to be well optimised for clustered architectures, but this isn't always the case.
  - hard to do for AlltoAllv, for example
- Can be cases where MPI + OpenMP transfers less data
  - e.g. AllReduce where every thread contributes to the sum, but only the master threads uses the result

Example

- ECMWF IFS weather forecasting code
- Semi-Lagrangian advection: require data from neighbouring grid cells only in an upwind direction.
- MPI solution – communicate all the data to neighbouring processors that *could possibly* be needed.
- MPI + OpenMP solution – within a node, only read data from other threads' grid point if it is actually required
  - Significant reduction in communication costs

IFS example



Problems with MPI + OpenMP

- Development/maintenance costs
- Portability
- Libraries
- Performance pitfalls

Development / maintenance costs

- In most cases, development and maintenance will be harder than for a pure MPI code.

- OpenMP programming is easier than MPI (in general), but it's still parallel programming, and therefore hard!
  - application developers need yet another skill set

- OpenMP (as with all threaded programming) is subject to subtle race conditions and non-deterministic bugs
  - correctness testing can be hard

Portability

- Both OpenMP and MPI are themselves highly portable (but not perfect).

- Combined MPI/OpenMP is less so
  - main issue is thread safety of MPI
  - if maximum thread safety is assumed, portability will be reduced

- Desirable to make sure code functions correctly (maybe with conditional compilation) as stand-alone MPI code (and as stand-alone OpenMP code?)

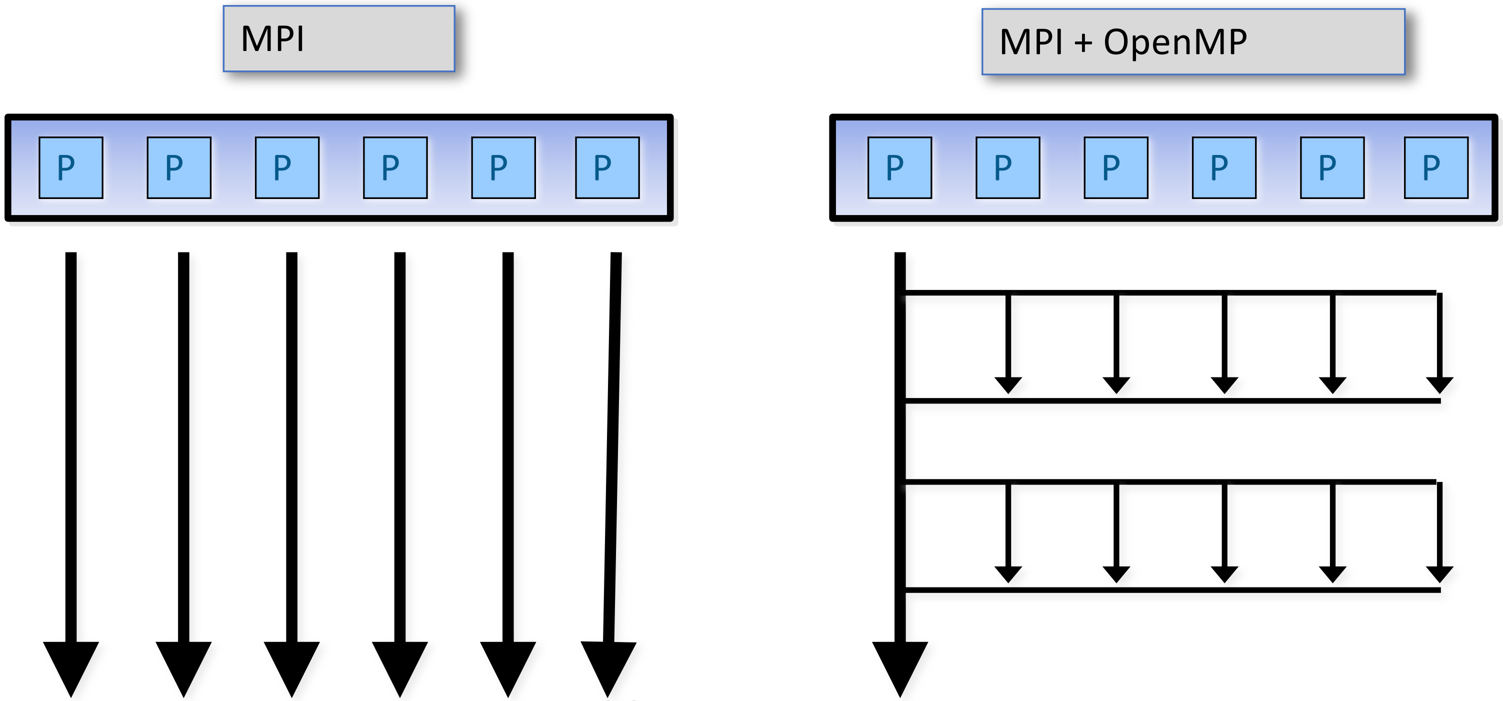
Libraries

- If the pure MPI code uses a distributed-memory library, need to replace this with a hybrid version.
- If the pure MPI code uses a sequential library, need to replace this with either a threaded version called from the master thread, or a thread-safe version called inside parallel regions.
- If thread/hybrid library versions use something other than OpenMP threads internally, can get problems with oversubscription.
  - Both the application and the library may create threads that might not idle nicely when not being used

Performance pitfalls

- Adding OpenMP may introduce additional overheads not present in the MPI code (e.g. synchronisation, false sharing, sequential sections, NUMA effects).
- Adding OpenMP introduces a tunable parameter – the number of threads per MPI process
  - optimal value depends on hardware, compiler, input data
  - hard to guess the right value without experiments
- Placement of MPI processes and their associated OpenMP threads within a node can have performance consequences.

- An incremental, loop by loop approach to adding OpenMP is easy to do, but it can be hard to get sufficient parallel coverage.
  - just Amdahl's law applied inside the node



More pitfalls...

- The mixed implementation may require more synchronisation than a pure OpenMP version, if non-thread-safety of MPI is assumed.
- Implicit point-to-point synchronisation via messages may be replaced by (more expensive) barriers.
  - loose thread to thread synchronisation is hard to do in OpenMP
- In the pure MPI code, the intra-node messages will often be naturally overlapped with inter-node messages
  - harder to overlap inter-thread communication with inter-node messages – see later
- OpenMP codes can suffer from false sharing (cache-to-cache transfers caused by multiple threads accessing different words in the same cache block)
  - MPI naturally avoids this

NUMA effects

- Nodes which have multiple sockets are NUMA: each socket has it’s own block of RAM.
- OS allocates virtual memory pages to physical memory locations
  - has to choose a socket for every page
- Common policy (default in Linux) is *first touch* – allocate on socket where the first read/write comes from
  - right thing for MPI
  - worst possible for OpenMP if data initialisation is not parallelised
  - all data goes onto one socket
- NUMA effects can limit the scalability of OpenMP: it may be advantageous to run one MPI process per NUMA domain, rather than one MPI process per node.

Process/thread placement

- On NUMA nodes need to make sure that:
  - MPI processes are spread out across sockets
  - OpenMP threads are on the same socket as their parent process

- Not all batch systems do a good job of this....
  - can be hard to fix this as a user
  - gets even more complicated if SMT (e.g. Hyperthreads) is used.

Styles of MPI + OpenMP programming

- Can identify 4 different styles of MPI + OpenMP programming, depending on when/how OpenMP threads are permitted to make MPI library calls

- Each has its advantages and disadvantages

- MPI has a threading interface which allow the programmer to request and query the level of thread support

The 4 styles

- Master-only
  - all MPI communication takes place in the sequential part of the OpenMP program (no MPI in parallel regions)
- Funneled
  - all MPI communication takes place through the same (master) thread
  - can be inside parallel regions
- Serialized
  - only one thread makes MPI calls at any one time
  - distinguish sending/receiving threads via MPI tags or communicators
  - be very careful about race conditions on send/recv buffers etc.
- Multiple
  - MPI communication simultaneously in more than one thread
  - some MPI implementations don't support this
  - …and those which do mostly don't perform well

OpenMP Master-only

Fortran

```
!$OMP parallel
 work…
!$OMP end parallel

call MPI_Send(…)

!$OMP parallel
 work…
!$OMP end parallel
```

C

```
#pragma omp parallel
{
   work…
}
ierror=MPI_Send(…);
#pragma omp parallel
{
   work…
}
```

OpenMP Funneled

Fortran

```
!$OMP parallel
… work
!$OMP barrier
!$OMP master
  call MPI_Send(…)
!$OMP end master
!$OMP barrier
.. work
!$OMP end parallel
```

C

```
#pragma omp parallel
{
  … work
  #pragma omp barrier
  #pragma omp master
  {
    ierror=MPI_Send(…);
  }
 #pragma omp barrier
  … work
}
```

OpenMP Serialized

Fortran

```
!$OMP parallel
… work
!$OMP critical
  call MPI_Send(…)
!$OMP end critical
… work
!$OMP end parallel
```

C

```
#pragma omp parallel
{
  … work
  #pragma omp critical
  {
    ierror=MPI_Send(…);
  }
  … work
}
```

OpenMP Multiple

Fortran

```
!$OMP parallel
… work
call MPI_Send(…)
… work
!$OMP end parallel
```

C

```
#pragma omp parallel
{
  … work
  ierror=MPI_Send(…);
  … work
}
```

Thread Safety

- Making MPI libraries thread-safe is difficult
  - lock access to data structures
  - multiple data structures: one per thread
  - …
- Adds significant overheads
  - which may hamper standard (single-threaded) codes
- MPI defines various classes of thread usage
  - library can supply an appropriate implementation

MPI_Init_thread

- MPI_Init_thread works in a similar way to MPI_Init by initialising MPI on the main thread.
- It has two integer arguments:
  - Required ([in] Level of desired thread support)
  - Provided ([out] Level of provided thread support)

- C syntax

```
int MPI_Init_thread(int *argc, char *((*argv)[]), int
  required, int *provided);
```

- Fortran syntax

```
MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR)
   INTEGER REQUIRED, PROVIDED, IERROR
```

MPI_Init_thread

- 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.

MPI_Init_thread

- These integer values are monotonic; i.e.,
  - MPI_THREAD_SINGLE < MPI_THREAD_FUNNELED < MPI_THREAD_SERIALIZED < MPI_THREAD_MULTIPLE
- Note that these values do not strictly map on to the four MPI/OpenMP Mixed-mode styles as they are more general (i.e. deal with Posix threads where we don't have "parallel regions", etc.)
  - e.g. no distinction here between Master-only and Funneled
  - see MPI standard for full details

MPI_Query_thread()

- MPI_Query_thread() returns the current level of thread support
  - Has one integer argument: provided [in] as defined for MPI_Init_thread()

C syntax

```
int MPI_query_thread(int *provided);
```

Fortran syntax

```
MPI_QUERY_THREAD(PROVIDED, IERROR)
  INTEGER PROVIDED, IERROR
```

Need to compare the output manually, i.e.

```
if (provided < requested) {
  printf("Not a high enough level of thread support!\n");
  MPI_Abort(MPI_COMM_WORLD,1)
    …etc.
}
```

Master-only

- Advantages
  - simple to write and maintain
  - clear separation between outer (MPI) and inner (OpenMP) levels of parallelism
  - no concerns about synchronising threads before/after sending messages
- Disadvantages
  - threads other than the master are idle during MPI calls
  - all communicated data passes through the cache where the master thread is executing.
  - inter-process and inter-thread communication do not overlap.
  - only way to synchronise threads before and after message transfers is by parallel regions which have a relatively high overhead.
  - packing/unpacking of derived datatypes is sequential.

Example

```
DO I = 1,N
   A(I) = B(I) + C(I)
END DO

CALL MPI_BSEND(A(N),1,.....)
CALL MPI_RECV(A(0),1,.....)


DO I = 1,N
   D(I) = A(I-1) + A(I)
END DO
```

Example

```
!$omp parallel do
DO I = 1,N * nthreads
   A(I) = B(I) + C(I)
END DO

CALL MPI_BSEND(A(N),1,.....)
CALL MPI_RECV(A(0),1,.....)

!$omp parallel do
DO I = 1,N * nthreads
   D(I) = A(I-1) + A(I)
END DO
```

Implicit barrier added here

Intra-node messages overlapped with inter-node

Inter-thread communication occurs here

Funneled

- Advantages
  - relatively simple to write and maintain
  - cheaper ways to synchronise threads before and after message transfers
  - possible for other threads to compute while master is in an MPI call
- Disadvantages
  - less clear separation between outer (MPI) and inner (OpenMP) levels of parallelism
  - all communicated data still passes through the cache where the master thread is executing.
  - inter-process and inter-thread communication still do not overlap.
  - awkward asymmetry between threads

OpenMP Funneled with overlapping (1)

```
#pragma omp parallel
{
      … work
  #pragma omp barrier
  if (omp_get_thread_num() == 0)  {
    ierror=MPI_Send(…);
  }
  else {
    do some computation
  }
 #pragma omp barrier
      … work
}
```

Can't use worksharing here!

OpenMP Funneled with overlapping (2)

```
#pragma omp parallel num_threads(2)
{
if (omp_get_thread_num() == 0)  {
    ierror=MPI_Send(…);
  }
  else {
#pragma omp parallel
    {
       do some computation
    }
  }
}
```

Higher overheads and harder to synchronise between teams

Serialised

- Advantages
  - easier for other threads to compute while one is in an MPI call
  - can arrange for threads to communicate only their “own” data (i.e. the data they read and write).
- Disadvantages
  - getting harder to write/maintain
  - more, smaller messages are sent, incurring additional latency overheads
  - need to use tags or communicators to distinguish between messages from or to different threads in the same MPI process.

Distinguishing between threads

- By default, a call to MPI_Recv by any thread in an MPI process will match an incoming message from the sender.
- To distinguish between messages intended for different threads, we can use MPI tags
  - if tags are already in use for other purposes, this gets messy
- Alternatively, different threads can use different MPI communicators
  - OK for simple patterns, e.g. where thread N in one process only ever communicates with thread N in other processes
  - more complex patterns also get messy

Multiple

- Advantages
  - Messages from different threads can (in theory) overlap
    - many MPI implementations serialise them internally.
  - Natural for threads to communicate only their “own” data
  - Fewer concerns about synchronising threads (responsibility passed to the MPI library)
- Disdavantages
  - Hard to write/maintain
  - Not all MPI implementations support this – loss of portability
  - Some MPI implementations don’t perform well like this
    - Thread safety implemented crudely using global locks.

Summary

- MPI + OpenMP programming is becoming standard practice
  - ~30% of consumed CPU hours on ARCHER
- Many see it as the key to exascale, however …
  - may require MPI_THREAD_MULTIPLE style to reduce overheads
- Achieving correctness is hard
  - have to consider race conditions on message buffers
- Achieving performance is hard
  - entire application must be threaded (efficiently!)
- Must optimise choice of
  - numbers of processes/threads
  - placement of processes/threads on NUMA architectures

Reusing this material

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

https://creativecommons.org/licenses/by-nc-sa/4.0/

This means you are free to copy and redistribute the material and adapt and build on the material under the following terms: You must give appropriate credit, provide a link to the license and indicate if changes were made. If you adapt or build on the material you must distribute your work under the same license as the original.

Note that this presentation contains images owned by others. Please seek their permission before reusing these images.