• In recent years there has been a trend towards clustered architectures

• Distributed memory systems, where each node consist of a traditional shared memory multiprocessor (SMP).
  – with the advent of multicore chips, every cluster is like this

• Single address space within each node, but separate nodes have separate address spaces.
Clustered architecture
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?
We need to consider:

- Development / maintenance costs
- Portability
- Performance
Development / maintenance

- In most cases, development and maintenance will be harder than for an MPI code, and much harder than for an OpenMP code.

- If MPI code already exists, addition of OpenMP may not be too much overhead.

- In some cases, it may be possible to use a simpler MPI implementation because the need for scalability is reduced.
  - e.g. 1-D domain decomposition instead of 2-D
• 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?)
• Making libraries thread-safe can be 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
  – see later
Performance

Four possible performance reasons for mixed OpenMP/MPI codes:

1. Replicated data
2. Poorly scaling MPI codes
3. Limited MPI process numbers
4. MPI implementation not tuned for SMP clusters
Replicated data

• Some MPI codes use a replicated data strategy
  – all processes have a copy of a major data structure
  – classical domain decomposition code have replication in halos
  – MPI 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

• Halo regions are a type of replicated data
  – can become significant for small domains (i.e. many processes)
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

<table>
<thead>
<tr>
<th>Local domain size</th>
<th>Halos</th>
<th>% of data in halos</th>
</tr>
</thead>
<tbody>
<tr>
<td>$50^3 = 125000$</td>
<td>$52^3 - 50^3 = 15608$</td>
<td>11%</td>
</tr>
<tr>
<td>$20^3 = 8000$</td>
<td>$22^3 - 20^3 = 2648$</td>
<td>25%</td>
</tr>
<tr>
<td>$10^3 = 1000$</td>
<td>$12^3 - 10^3 = 728$</td>
<td>42%</td>
</tr>
</tbody>
</table>
Poorly scaling MPI codes

• If the MPI version of the code scales poorly, then a mixed MPI/OpenMP version may scale better.

• May be true in cases where OpenMP scales better than MPI due to:

  1. Algorithmic reasons.
     - e.g. adaptive/irregular problems where load balancing in MPI is difficult.

  2. Simplicity reasons
     - e.g. 1-D domain decomposition
Load balancing

• 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
Limited MPI process numbers

• MPI library implementation may not be able to handle millions of processes adequately.
  – e.g. limited buffer space
  – Some MPI operations are hard to implement without $O(p)$ computation, or $O(p)$ storage in one or more processes
  – e.g. AlltoAllv, matching wildcards

• Likely to be an issue on very large systems.

• Mixed MPI/OpenMP implementation will reduce number of MPI processes.
• Some MPI implementations are not well optimised for SMP clusters
  – less of a problem these days
• Especially true for collective operations (e.g. reduce, alltoall)
• Mixed-mode implementation naturally does the right thing
  – reduce within a node via OpenMP reduction clause
  – then reduce across nodes with MPI_Reduce
• Mixed-mode code also tends to aggregate messages
  – send one large message per node instead of several small ones
  – reduces latency effects, and contention for network injection
Styles of mixed-mode programming

• 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
### Fortran

```fortran
 !$OMP parallel
  work...
!$OMP end parallel

call MPI_Send(...)

 !$OMP parallel
  work...
!$OMP end parallel
```

### C

```c
#pragma omp parallel
{
  work...
}

ierror=MPI_Send(...);

#pragma omp parallel
{
  work...
}
```
OpenMP Funneled

**Fortran**

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

**C**

```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

```fortran
!$OMP parallel
... work
call MPI_Send(...)
... work
!$OMP end parallel
```

C

```c
#pragma omp parallel
{
    ... work
    ierror=MPI_Send(...);
    ... work
}
```
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
  ```c
  int MPI_Init_thread(int *argc, char **(*argv)[], int required, int *provided);
  ```

- Fortran syntax
  ```fortran
  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.
• These integer values are monotonic; i.e.,
  – MPI_THREAD_SINGLE < MPI_THREAD_FUNNELED
  – MPI_THREAD_SERIALIZE < 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
  ```c
  int MPI_query_thread(int *provided);
  ```

- Fortran syntax
  ```fortran
  MPI_QUERY_THREAD(PROVIDED, IERROR)
  INTEGER PROVIDED, IERROR
  ```

- Need to compare the output manually, i.e.
  ```c
  if (provided < requested) {
    printf("Not a high enough level of thread support!\n");
    MPI_Abort(MPI_COMM_WORLD,1)
    ...etc.
  }
  ```
Pitfalls

• The OpenMP implementation may introduce additional overheads not present in the MPI code (e.g. synchronisation, false sharing, sequential sections).

• 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 may be replaced by (more expensive) barriers.

• 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.

• 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 placement becomes very important
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

```fortran
!$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
• 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.
```c
#pragma omp parallel
{
    ... work
    #pragma omp barrier
    if (omp_get_thread_num() == 0) {
        ierror=MPI_Ssend(...);
    }
    else {
        do some computation
    }
    #pragma omp barrier
    ... work
}
```
OpenMP Funneled with overlapping (2)

```c
#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
• 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)

• Disadvantages
  – Hard to write/maintain
  – Not all MPI implementations support this – loss of portability
  – Most MPI implementations don’t perform well like this
    – Thread safety implemented crudely using global locks.
End points

- A possible solution to permit more easier use and efficient implementations of Multiple is to extend MPI so that an MPI rank may have multiple source and destination identifiers (end points).
- e.g. if we want 4 threads per MPI process we could create an MPI communicator with 4 end points per rank
  - each thread can use a different end point
- Avoids need to use tags to identify threads
- Currently under discussion in MPI Forum
  - might appear in MPI 4.0?
Performance

• Conceptually easy to write
  – rather messy
  – hard to get good performance: cannot just concentrate on key kernels
Summary

• Hybrid programming still a major current research topic

• Many see it as the key to exascale, however …
  – will require MPI_THREAD_MULTIPLE style to avoid synchronisation
  – ... and end points to make this usable?

• Achieving correctness is hard
  – have to consider race conditions on messages

• 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