Best Practice Guide to Hybrid MPI + OpenMP Programming

Version 1.2, 21st April 2017

Table of Contents

Index of Figures

Index of Tables

1 Introduction

Hybrid application programs using MPI + OpenMP are now commonplace on large HPC systems. There are essentially two main motivations for this combination of programming models:

1. Reduction in memory footprint, both in the application and in the MPI library (e.g. communication buffers).
2. Improved performance, especially at high core counts where the pure MPI scalability is running out.

Section 2 of this Best Practice Guide discusses these motivations in more detail and explains why and how these potential benefits can be realized in application codes. Section 3 discusses the possible downsides of MPI + OpenMP programs, covering software engineering issues and performance pitfalls. Section 4 covers the technical details of thread support in the MPI library, and Section 5 describes five different styles of MPI + OpenMP program and their relative advantages and disadvantages. Section 6 provides some best practice tips for developers of hybrid MPI + OpenMP applications.

2 Potential Advantages of MPI + OpenMP

There are two principal reasons why a hybrid MPI + OpenMP implementation of an application might be beneficial. The first of these is to reduce the memory requirements of an application; the second is to improve its performance. The following sections discuss these in more detail.

2.1 Reducing memory requirements

Some applications are constrained in the problems they can be used to solve by the amount of data that must be held in memory during execution. Modern supercomputers typically have a 1-2 GBytes of main memory installed per core – although having much more memory per core is technically feasible, this limit is imposed by installation and power costs. The amount of memory per core is not likely to increase in the near future – indeed it may even tend to decrease as the number of cores per node continues to go up.

Ideally, in a strong scaling scenario, where the same problem is run on increasing numbers of MPI processes, the total memory required would be constant. However this is often not the case. Equivalently, in a weak scaling scenario, where the problem size scales linearly with the number of MPI processes, the memory required per process does not remain constant, but increases as the number of MPI processes goes up.

The reason for this behaviour is that designing and implementing MPI applications generally requires that some data be replicated between MPI processes. A simple example might be a read-only look up table which every process has a copy of, and which would be difficult or inefficient to distribute across processes. Another common pattern in applications that use a spatial decomposition design is that of halo regions where each process mirrors the data on the boundary of its computational domain with its neighbouring processes. Another reason for loss of memory scalability can be due to MPI's internal data structures, for example communication buffers and internal storage for mappings of MPI ranks to physical locations in the machine.

Table 1 illustrates how the memory overhead of halo regions increases as a three-dimensional grid is decomposed across increasing numbers of processes, such that the local domain size decreases. With a local domain size of 50^3 grid points per processors, and a one-point deep halo region in every dimension, the halo regions account for 11% of the memory usage. When the local domain size shrinks to 10^3 grid points, the halos occupy 42% of the memory usage. This effect is even greater if halos are more than one grid point deep, or the decomposition is in more than three dimensions (as in some QCD simulations, for example).

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%

Table 1: Memory overheads in halo regions

The benefit of a hybrid MPI + OpenMP implementation is that only one copy of replicated data is required per process, and within a process, data can be shared by threads with no (or substantially less) replication.

An alternative way of looking at this issue, is that to give each MPI process sufficient memory to run a given problem, it might be necessary to run fewer MPI processes on each node than there are cores. This results in some cores being idle, and exploiting parallelism within a process using OpenMP threads gives an opportunity to recover some of the lost performance associated with having idle cores.

2.2 Improving performance

Few applications demonstrate perfect scaling behaviour up to the numbers of cores currently available in the largest HPC system. In fact many application codes run into scalability problems at much lower core counts on problems of scientific interest. At some number of cores, the overheads of parallelism start to become significant and the performance will eventually reach a maximum and then decrease again at higher core counts.

There are a number of scenarios in which using MPI + OpenMP can reduce these parallel overheads. However, a hybrid implementation should not be expected to outperform the pure MPI version in the regime where the scaling of the MPI version is still good. Replacing MPI processes with OpenMP threads will not make the computations run any faster: the best that can be hoped for is that it reduces some sources of parallel overhead, which, by definition, are not significant in the good scaling regime. Indeed in the regime where MPI is scaling well, the hybrid code is likely to introduce additional overheads that decrease performance.

Figure 1 illustrates typical performance curves for pure MPI and (successful) hybrid MPI + OpenMP versions of the same code. At low core counts, additional overheads (whose sources will de discussed in Section 3.4) mean that the performance of the pure MPI version is better. At some point, the curves cross over and the hybrid version performs better.

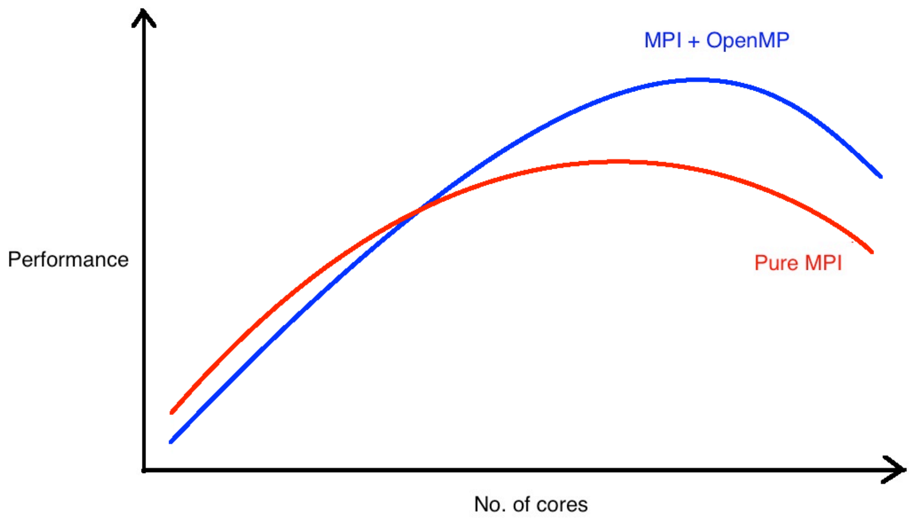

Figure 1: Typical performance curves for pure MPI and hybrid MPI + OpenMP versions of an application

Of course it is possible that the MPI + OpenMP is not successful in this respect: it might never outperform the pure MPI version, or it might only do so well beyond the core count at which the pure MPI performance is at its maximum, which is not useful.

There are several different types of overhead that may be reduced by introducing OpenMP: these are explored in the following Sections.

2.2.1 Exploiting additional layers of parallelism

Some MPI codes do not scale beyond a certain core count because they run out of available parallelism at the level that is used to distribute computation and data across MPI processes. However, there may be additional lower levels of parallelism in the application that can be exploited.

In principle, this could also be done using MPI, but in practice this can be hard, because the lower level parallelism may be hard to load balance, or may have irregular (or data-dependent) communication patterns. In addition, it may simply be hard to work around design decisions in the original MPI version, which did not take the lower levels of parallelism into account.

It may, therefore, for practical reasons be easier to exploit the additional level(s) of parallelism using OpenMP threads. It is possible to take an incremental (e.g. loop by loop) approach to adding OpenMP: this is maybe not performance optimal, but it keeps development time to a minimum.

Obviously, OpenMP parallelism cannot extend beyond a single node, but this may be enough for practical purposes: future systems seem likely to have more cores per node, rather than many more nodes.

2.2.2 Reducing communication overheads

It is natural to suppose that communicating data inside a node is faster between OpenMP threads than it is between MPI processes, because it avoids any copying in and out of MPI buffers, and there are no library call overheads. This is true, but there are a number of caveats – see Section 3.4 – and the cost of intra-node communication is rarely the most significant source of overhead.

In some circumstances, the overheads of collective communications can be improved by using MPI + OpenMP. In principle, the MPI library implementation of collective communication ought to be well optimised for clustered architectures, but this isn't always the case: it is difficult to do for **`MPI_AlltoAllv`**, for example, where every process sends a different amount of data to every other process. There can also be cases where MPI + OpenMP requires less data to be transferred, for example an AllReduce operation where every thread contributes to the sum, but only the master thread on each process uses the result. Compared to the pure MPI version, which would give a copy of the result to every process, some stages of the collective can be omitted.

In some cases, MPI codes actually communicate more data than is strictly necessary to satisfy the data dependencies in the underlying algorithm. This can happen where the data dependencies may be irregular and/or dependent on values determined at run time. It may make the implementation much simpler if all the data that *could possibly* be needed by (say) a neighbouring process is sent, instead of having to calculate, and communicate between the processes just to determine what data is really required.

As an example, consider the IFS weather forecasting code from European Centre for Medium-Range Weather Forecasts (ECMWF). One of the important communication phases in the code implements a semi-Lagrangian advection scheme, which handles the transport of atmospheric properties (e.g. momentum, temperature and humidity) by the wind. The forecast model divides the earth's surface up into a two-dimensional grid, and assigns patches of grid points to different MPI processes. For the semi-Lagrangian scheme, each grid point requires data from neighbouring grid points that are upwind, i.e. in the direction from where the local wind is blowing. However, to make the implementation tractable, every process sends halo values to all its neighbours, regardless of whether the data is actually used or not. This is illustrated in Figure 2, which shows a central grid patch assigned to a process and its eight nearest neighbours. The red arrows represent the wind velocity at each grid point in the patch. The semi-Langrangian algorithm requires only data to be communicated from neighbouring processes in an upwind direction, i.e. the green data points. However, the MPI implementation communicates not only the green data points, but all the purple ones as well, whose data *could* have been required if the wind velocities were different.

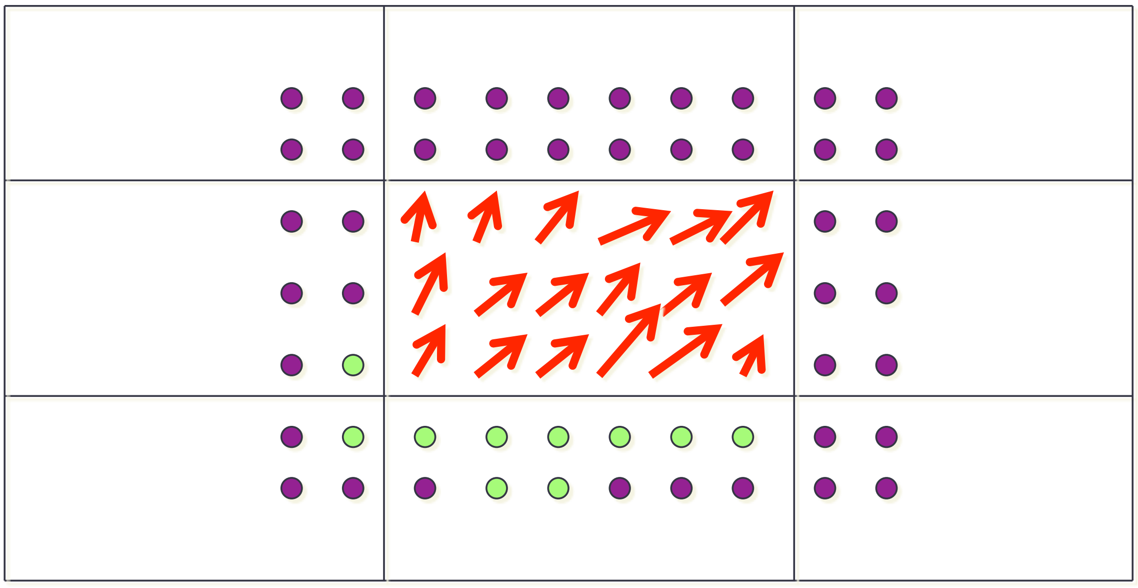

Figure 2: Communication pattern in IFS semi-Lagrangian advection scheme

Now consider the case in a hybrid MPI-implementation where all nine grid patches are handled by different threads on the same node. All the threads have access to the other threads' grid points as data which is shared between them, so the thread handling the central patch can simply read the data it requires (green points) and ignore that which it does not (purple points).

Overall, the MPI + OpenMP implementation significantly reduces the total volume of data exchanged, and improves the scalability of the code [1]. (It is worth noting that this problem could potentially also be addressed by using single-sided RDMA-style message passing. Although the MPI standard does support this, the performance of implementations has so far been disappointing. Other APIs such as Fortran co-arrays, UPC, OpenSHMEM or GASPI offer better performance, but lack the availability, enjoyed by MPI, of robust implementations on a wide range of platforms. In the case of IFS, experiments have been carried out using both approaches together, with hybrid MPI + Co-arrays +OpenMP and MPI + GASPI + OpenMP)

Similar communication patterns occur in other applications, e.g. particle simulations, where all the particles near the edge of a process's sub domain are communicated to neighbouring domains, since they *could possibly* interact with particles there in the next time step.

2.2.3 Reducing load imbalance

Load balancing between MPI processes can be hard and costly, because of the need to transfer both computational tasks and data from overloaded to underloaded processes. Transferring small tasks may not be beneficial, and having a global view of the loads on all processors (for example if all processes send their current load status to process 0) will not scale well at high process counts. To ensure scalability, it may be necessary to restrict the application to transferring load only between neighbours in some topology (usually the underlying communication topology), for example in diffusion based load balancing algorithms. However, this is often less effective at balancing the load quickly, as it takes multiple steps for load to migrate to distant underloaded processes. Typically, as the number of MPI processes increases, the overhead of balancing the load increases, and the consequent reduction in load imbalance decreases.

In contrast, load balancing between threads is much easier: load balancing algorithms only need to transfer tasks, and the data will follow via shared memory and the cache coherency mechanism. The overheads of balancing threads are lower, so fine grained balancing is possible, and thanks to shared variables and locks or atomic operations, it is easier to both have a global view of the load and to balance it dynamically as the computations progress.

For applications with load balance problems, keeping the number of MPI processes as small as possible, and exploiting OpenMP's inbuilt load balancing capabilities (via loop schedules or tasks) can therefore reduce the overheads due to load imbalance.

The design of many MPI applications is often based on the assumption that executing the same computation on different data on every core will take the same time on every core. As future hardware designs evolve more sophisticated methods of power-saving, this assumption may start to break down, resulting in load imbalance becoming a problem in applications which have so far not experienced it. Hybrid MPI + OpenMP implementation may be able to mitigate these effects by using dynamic scheduling within each node. The Asynchronous Task style of hybrid programming (see Section 5) is especially interesting in this regard, as it attempts to minimise synchronisation across the whole application, and therefore be more tolerant of load imbalance.

2.2.4 Reducing memory access costs

As described in Section 2.1, hybrid MPI + OpenMP implementation can reduce the memory requirements of applications. As well as allowing large problem sizes to be run on the same number of nodes, this can also have a performance benefit as a side effect.

If the memory footprint per core is reduced, this can improve data locality by allowing more of the current working data to fit into cache memory, and reducing the demands on memory bandwidth.

3 Potential disadvantages of MPI + OpenMP

In the previous Section, the potential benefits of MPI + OpenMP programs were discussed. In this Section, in contrast, we consider the potential disadvantages, covering both software engineering concerns and reasons why hybrid MPI + OpenMP codes may actually perform worse than the pure MPI version.

3.1 Development and maintenance costs

Hybrid MPI + OpenMP codes will almost always incur higher development and maintenance costs than a pure MPI version. Although OpenMP programming is generally easier (or at least quicker) than MPI programming, it is still parallel programming and therefore not trivial. Developers need to acquire an additional skill set.

OpenMP, as with all threaded programming APIs, is prone to race conditions and non-deterministic bugs (this is also true of MPI, but here race conditions are restricted to message data, and therefore typically easier to track down). This makes testing and assuring correctness more difficult.

It is normally desirable for it to remain possible to build a pure MPI version from the hybrid MPI + OpenMP source code: OpenMP has a number of conditional compilation facilities, including a standard pre-processing macro, to help make this possible. In general, the presence of OpenMP library calls means it is not just a case of ignoring directives.

There is reasonable tool support for performance analysis and debugging of hybrid MPI + OpenMP, but there are nevertheless some tools which support MPI but not OpenMP.

3.2 Portability

Both MPI and OpenMP are highly portable as standalone APIs. The combination of MPI + OpenMP is slightly less so, if the code assumes full thread safety of the MPI library (i.e. **`MPI_THREAD_MULTIPLE`** support, see Section 4 for details). Some MPI library implementations do not support this, or MPI installations on certain platforms may not be configured to support it. As a rule, this is gradually becoming less of a problem, since the two prevalent MPI library code bases, from which many implementations are derived (MPICH and OpenMPI) do have good **`MPI_THREAD_MULTIPLE`** support.

3.3 Libraries

Many scientific codes make use of third-party libraries, e.g. for numerical computations, such as linear algebra and FFTs, or for parallelism support, such as mesh partitioning and load balancing. When converting an application that spends any significant amount of execution time in library routines from pure MPI to hybrid MPI + OpenMP, care must be taken to ensure that suitable versions of the library (or alternative libraries) are available. Several different scenarios can be identified:

- The pure MPI code uses a distributed-memory library. In this case, a hybrid MPI + threads version of the library will be required, and will be called from the OpenMP master thread only.
- The pure MPI code uses a sequential library. In this case, there are two possibilities:
    - a threaded version of the library routine is required, and will be called from the OpenMP master thread, or
    - a thread-safe sequential version of the library is required, and will be called by multiple threads inside parallel regions.

It is important to note that hybrid or threaded versions of libraries do not necessarily use OpenMP internally: they may use other threading APIs, such as Posix threads or

Intel TBB. If this is the case, then care must be taken to correctly specify how many threads are to be used inside the library, as this may not be controlled by OpenMP's internal control variable settings. Furthermore, it can be easy to create more threads in total than there are available cores, resulting in oversubscription of cores and high context switching overheads. The behaviour may also differ depending on which compilers are used to compile the user application code and the library code.

This is an instance of a more general interoperability problem when multiple runtimes are active on a node: see the INTERTWinE Best Practice Guide for Writing OpenMP/OmpSs/StarPU + Multi-threaded Libraries Interoperable Programs for a more detailed discussion of this type of problem.

Unfortunately some well-known scientific libraries, for example PETSc [2], have very limited threading support. It is also worth noting that applications relying on high quality pseudo-random number generation must be very careful when using multiple threads not to compromise the quality of the random number distributions obtained.

3.4 Performance pitfalls

It is not easy to write a hybrid MPI + OpenMP version of an application code that outperforms the pure MPI version. In this Section some of the possible reasons for poor performance of the hybrid version are discussed.

3.4.1 Idle threads

The simplest approach to adding OpenMP to an MPI code is to add OpenMP directives incrementally to the more computationally intense parts of the application (typically parallel loops). However, using this approach can make it difficult to eliminate sequential sections where only the master thread is executing, especially if the code is written in Master-Only style (see Section 5.1), where all MPI calls occur outside of OpenMP parallel regions. Idle threads can be a significant source of overhead: this is nothing more that a statement of Amdahl's Law applied inside individual MPI processes.

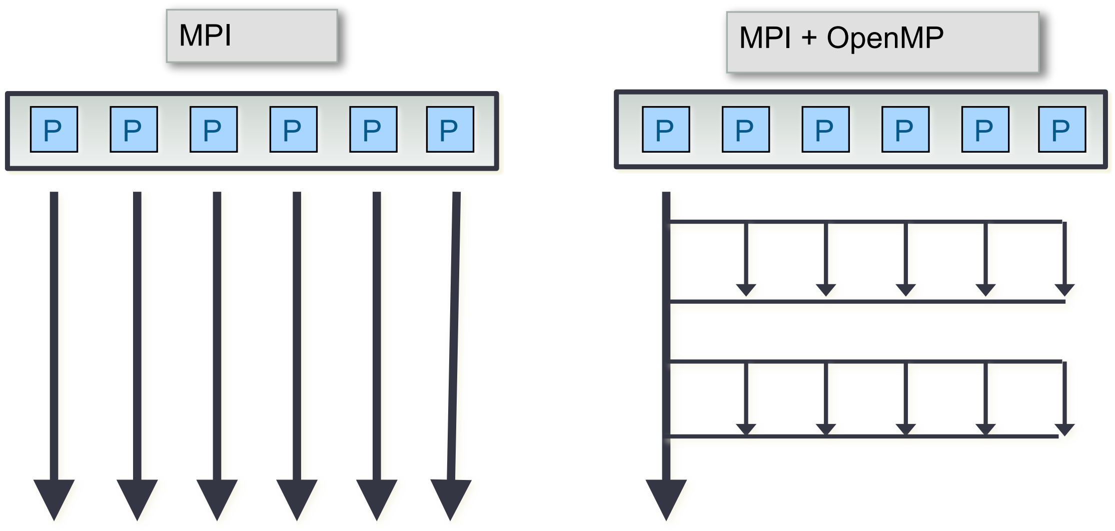

Figure 3: Amdahl's Law inside an MPI process: incomplete OpenMP coverage of computationally intensive code results in idle threads

3.4.2 Synchronisation

In MPI, synchronisation between processes is tightly coupled to communication: processes block until messages have been received or collective communication completed. In OpenMP, synchronisation and communication between threads are more decoupled. In OpenMP, a common synchronisation mechanism is the barrier: there are implicit barriers at the end of parallel loops and parallel regions. OpenMP does not have high-level point-to-point synchronisation mechanisms, whereby one thread can wait for another thread to reach a given point in the program. Synchronising threads at barriers also introduces the risk of load imbalance, if all the threads do not reach the barrier at the same time. Adding OpenMP to an MPI code therefore typically introduces a number of these thread barriers, which may be more expensive than point-to-point message passing inside a node.

If the hybrid code calls MPI from more than one thread, then there will be additional synchronisation overheads to prevent race conditions inside the MPI library. Depending on the level of thread-safety assumed, this synchronisation may either consist of OpenMP synchronisation constructs in the application code, or internal synchronisation inside the MPI library.

3.4.3 MPI derived datatypes

Use of derived datatypes is fairly common in MPI applications. Derived datatypes allow the programmer to specify message data that is not contiguous in memory (for example, slices of multidimensional arrays) to be passed to MPI communication routines. The MPI library is then responsible for (on the sender side) packing the data into a contiguous internal buffer before sending the message and (on the receiver side) unpacking the data from another contiguous internal buffer into user memory.

This packing and unpacking can be quite expensive: the copying consumes CPU, cache and memory bandwidth resources, and in some circumstances can be more expensive than the actual massage transfer across the network. However, these costs are hard to determine, as the time spent packing and unpacking may not show up separately in any profiling or performance analysis tools.

If MPI communication calls take place outside OpenMP parallel regions, then the packing and unpacking takes place on only one thread: the MPI library cannot recruit idle OpenMP threads to assist, nor do MPI implementations normally create internal threads to try to speed up the pack and unpack. It may therefore be necessary to avoid using derived datatypes, and write explicit loops to pack and unpack the message data into and out of contiguous buffers in user memory, which can then be effectively multithreaded with OpenMP directives.

3.4.4 Memory effects and placement

OpenMP codes can suffer from *false sharing* - cache-to-cache data transfers caused by multiple threads accessing different words in the same cache block. By contrast, MPI naturally avoids this since each process operates in a separate address space.

On systems with NUMA nodes (i.e. nodes with more than one processor socket), the placement of data in main memory becomes important. On these systems, although all main memory is accessible by all the cores on a node, the memory is physically distributed between the sockets. The operating system is responsible for allocating virtual memory pages to physical memory locations. On NUMA systems it has to choose a socket for every page to be stored in.

A common page placement policy (which is the default in Linux, for example) is *first touch* – pages are allocated on the socket where the first read/write to that page comes from. This is clearly a good option for pure MPI codes: every process will have its data allocated on the socket where it is executing. However, it is the worst possible option for OpenMP codes if data initialisation is not parallelised, and threads belonging to the same MPI process are executing on different sockets. All the data goes onto one socket: the one where the master thread is executing. When all the threads try to access this data, the memory bandwidth on a single socket can become a bottleneck. Note that in most cases it is the bandwidth bottleneck that is important, rather than the slightly increased latency of accesses memory on a remote socket.

Solving this problem requires either changing the page placement policy, if the operating system supports this, or careful multithreading of data initialisation. The latter approach is more robust across different systems, and may be straightforward if the main data structures are arrays, but can be much harder for irregular or dynamically created structures.

In practice, these NUMA effects can limit the scalability of OpenMP within a node: and it is often advantageous to run (at least) one MPI process per socket, rather than just one MPI process per node. In this situation, it is important to bind MPI processes and threads to cores in such a way that MPI processes are spread evenly across sockets, and every thread executes on the same socket as its parent process. Not all operating systems and batch schedulers offer good support for this (or may not have been locally configured correctly to do so). This is especially true if applications make use of hardware threads/virtual cores, and it can be hard to fix incorrect bindings. Although OpenMP provides the **`OMP_PLACES`** environment variable as a mechanism to specify where on the hardware threads should run, this is not generally useful for hybrid codes because it is not possible to arrange for different MPI processes to have different values of this variable.

3.5 How many threads?

Another disadvantage of hybrid MPI + OpenMP programming is that it introduces another tunable parameter which can significantly affect performance – the number of threads per MPI process. If the main motivation for hybrid programming is reduced memory usage, then it often makes sense to use the fewest number of threads than allow the problem to fit in memory.

Unfortunately the optimal value of this parameter with respect to performance is hard to predict, and may require significant benchmarking effort to determine correctly. It can depend on the application, the input data, the hardware platform, the number of nodes being used, the compiler and the MPI library implementation.

For best performance, there are some OpenMP environment variable settings that should be used, whose default values are implementation dependent and may not be set as you expect:

- **`OMP_WAIT_POLICY=active`** - encourages idle threads to spin rather than sleep (though beware that this may be counterproductive if the master thread calls libraries using other threading APIs.)
- **`OMP_DYNAMIC=false`** - don't let the runtime deliver fewer threads than you asked for
- **`OMP_PROC_BIND=true`** - prevents threads migrating between cores

4 MPI support for threads

In general making libraries thread-safe can be difficult. Since there is no universal definition of what is meant by "thread-safe", a library also has to document what behaviour can be expected under different circumstances when calls to routines are made from multiple threads.

Difficulties often arise when trying to retrospectively add thread safety to a library whose original design did not take account of this (which is the case for the major MPI library implementations such as MPICH and OpenMPI). Internal data structures in the library either have to be replicated (for example, one per thread), or else accesses to these structures must be protected by some form of synchronisation (typically locks).

This may add significant overheads, which can affect the performance of the library even when only one thread is being used. The MPI standard defines various classes of thread usage. The user can request a certain class for their application, and the implementation can return the class that is actually supported.

The MPI specification defines the following four classes of thread safety:

- **`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.

A program that intends to use multiple threads and MPI should call **`MPI_Init_thread`** instead of **`MPI_Init`** to initialise the MPI library. This function takes two additional arguments: a requested level of thread support (**`required`**) as input and a supplied level (**`provided`**) as output. Both arguments take one of the four levels specified above. The syntax for this function is as follows:

C/C++:

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

Fortran:

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

The **`required`** and **`provided`** arguments can take one of the four class values listed above which are guaranteed to be monotonic increasing, i.e.

`MPI_THREAD_SINGLE < MPI_THREAD_FUNNELED < MPI_THREAD_SERIALIZED < MPI_THREAD_MULTIPLE`

This makes it possible to test that the library provides at least the level of support required by the application, and to exit gracefully, for example like this:

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

It is also possible to retrieve the support level at any point in the program after the call to **`MPI_Init_thread()`** by calling **`MPI_Query_thread()`** which returns the **`provided`** value:

C/C++:

```
int MPI_Query_thread(int *provided);
```

Fortran:

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

Most MPI implementations can support at least **`MPI_THREAD_SERIALIZED`** by default: to get **`MPI_THREAD_MULTIPLE`** it may be necessary to set an environment variable or link with a different version of the library.

5 MPI + OpenMP styles

It is possible to classify MPI + OpenMP programs into five distinct styles, depending on if and how multiple OpenMP threads make MPI library calls.

- **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 but can be inside parallel regions.
- **Serialized**: MPI calls can be made by any thread, but only one thread makes MPI calls at any one time.
- **Multiple**: MPI communication can be made simultaneously in more than one thread.
- **Asynchronous Tasks**: MPI communications can take place from every thread, and MPI calls take place inside OpenMP tasks.

The following Sections describe the advantages and disadvantages of these different styles.

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

call MPI_Send(…)

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

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

Figure 4: Outline code in Master-Only style

5.1 Master-Only

In Master-Only style all MPI calls are made by the OpenMP master thread, outside of parallel regions. Note that strictly speaking, this requires the `MPI_THREAD_FUNNELED` level of thread support, since other threads will be executing, but not calling MPI. Figure 4 shows outline code for this style of MPI + OpenMP program.

The advantages of Master-Only style are:

- It is relatively simple to write and maintain, since there is clear separation between outer (MPI) and inner (OpenMP) levels of parallelism. It is the natural style that arises if an incremental, loop-wise approach is taken to adding OpenMP to an existing MPI code, by identifying the most time consuming loops and using parallel do/for constructs to mutlithread them.
- There are no concerns about synchronising threads before/after sending messages, since there is an implicit barrier at the end of each parallel region, and no thread may start the next parallel region until the master thread encounters the parallel construct (i.e. after any MPI calls).

On the other hand there are a number of disadvantages:

- Threads other than the master are necessarily idle during MPI calls, and cannot do any useful computation.

- This problem is made worse if the MPI code uses derived datatypes. The packing/unpacking of derived datatypes can be costly, especially if the memory access patterns are complex and/or sparse, and since the packing/unpacking occurs during MPI library calls, it will also be sequentialised as well as the actual message transfers.
- Data locality is poor, since all communicated data passes through the cache where the master thread is executing, as it is packed and unpacked into/from MPI internal buffers.
- For pure MPI codes running on shared memory clusters, some messages may be exchanged within the same node (most MPI implementations will do this using shared memory buffers, instead of using the network). In a communication phase where many messages are being exchanged it is therefore natural for the inter-node messages to overlap in time with intra-node ones. However, in Master-Only style MPI messages cannot overlap in time with inter-thread communication, since the latter occurs on-demand via the cache coherency mechanism, and can therefore only occur during OpenMP parallel regions.
- The only way to synchronise threads before and after message transfers is by closing and opening parallel regions, which has a relatively high overhead. To give some idea of the relative costs, the overhead of an OpenMP parallel region and the cost of sending a 1Kib message between nodes are roughly comparable: both are typically of the order of a few microseconds.

To illustrate some of these points consider the simple MPI code fragment in Figure 5, that implements a one-sided one-dimensional halo swap. Each MPI process sends `A(N)` to its neighbour to the right, and receives `A(0)` from its neighbour to the left.

```
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
```

Figure 5: Pure MPI code fragment implementing a one-sided one-dimensional halo exchange

If neighbouring processes are running on the same node, some of the MPI messages will be intra-node messages, while others will be inter-node messages. However it is possible for all these messages to be in flight at the same time.

Now consider the hybrid MPI + OpenMP, Master-Only style, version of the same fragment in Figure 6. Note that we have increased the local domain size by a factor of the number of threads (and implicitly reduced the number of MPI processes) to keep the total problem size constant, and parallelised both loops with parallel do constructs. The overheads of these parallel regions cannot overlap with the MPI messages. Furthermore, the inter-thread communincation takes place during the second loop: these are values of `A` for which the read of `A(I-1)` in the second loop comes from a different thread than the one that wrote this element of `A` in the first loop. This will likely result in a cache coherency miss when the read happens.

```
      NN = N*NTHREADS
!$OMP PARALLEL DO
      DO I=1,NN
         A(I) = B(I) + C(I)
      END DO

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

!$OMP PARALLEL DO
      DO I = 1,NN
         D(I) = A(I-1) + A(I)
      END DO
```

Figure 6: Hybrid MPI +OpenMP code fragment implementing a one-sided one-dimensional halo exchange

5.2 Funneled

In Funneled style, all MPI calls are made by the OpenMP master thread, but this may include calls from inside OpenMP parallel regions. An outline of code in this style can be found in Figure 7.

```
!$OMP parallel

… work

!$OMP barrier

!$OMP master
  call MPI_Send(…)
!$OMP end master

!$OMP barrier

.. work

!$OMP end parallel
```

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

Figure 7: Outline code in Funneled style

The advantages of Funneled style are:

- The code is still relatively simple to write and maintain.
- There are now cheaper ways available to synchronise threads before and after message transfers than closing and opening parallel regions.
- It is possible for other threads to do useful computation while the master thread is executing MPI calls.

However, there are still disadvantages with this style:

- The separation between outer (MPI) and inner (OpenMP) levels of parallelism is less clear than in Master-Only style, and the code will likely require some refactoring as well as the addition of OpenMP loop directives.
- All communicated data still passes through the cache where the master thread is executing, and datatype packing is sequentialised.
- As with Master-Only, it is hard to overlap inter-process and inter-thread communication.
- Although threads other than the master need not be idle while the master is executing MPI library calls, load balancing between the master thread and the other threads may need careful consideration.

The final point here is worth further discussion. If the master thread is executing an MPI call, the remaining threads cannot participate in an OpenMP worksharing construct at the same time, since worksharing constructs must be encountered by all threads in the parallel region (or none of them). Therefore any division of work between the other threads has to be achieved via other, less convenient mechanisms, such as explicitly calculating lower and upper loop bounds for each thread. Version 4.5 of the OpenMP specification has, however, introduced the **`taskloop`** construct, which could be successfully employed to solve this problem.

5.3 Serialized

In Serialized style, any thread inside an OpenMP parallel region may make calls to the MPI library, but the threads must be synchronised in such a way that only one thread at a time may be in an MPI call. This style requires **`MPI_THREAD_SERIALIZED`** support. Figure 8 shows outline code for this style.

The advantages of this style are:

- It is now possible to arrange for threads to communicate only their “own” data (e.g. to send data they previously wrote and/or receive data they will subsequently read). This improves locality, since the message data is not all being cycled through one cache.
- The enforced asymmetry between threads in Funneled style can be avoided.
- On some architectures, splitting messages up into smaller ones coming from different threads may improve the utilisation of the network interfaces, either by increasing the useable bandwidth or by increasing the available message injection rate.

On the other hand, disadvantages include:

- This style may be harder to write/maintain, as the MPI and OpenMP parallelism are more interlinked than in Master-Only or Funneled.
- Ensuring threads do not enter MPI calls at the same time may result in idle threads (for example, blocking on entry to a critical region), and arranging for threads to do useful computation instead may be difficult.
- Datatype packing is still serialised.
- Care is required to synchronise threads correctly, both to avoid multiple threads calling MPI at the same time, and to avoid race conditions on message contents.
- Splitting messages up into smaller ones coming from different threads will incur additional latency overheads.
- It is now often necessary to use tags or communicators to distinguish between messages from (or to) different threads in the same MPI process.

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

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

Figure 8: Outline code in Serialized style

Using tags to distinguish the source and destination threads for each message can be necessary because there could be, for example, a receive posted by each thread, all of which might match an incoming message from another rank, and the ordering of the sends and receives posted by different threads may be non-deterministic. This can become messy if tags are already being used to distinguish between different types of messages. One tactic is to add a large constant multiplied by the thread ID to the original tag, taking care that there are no overlapping values, and that the maximum tag value is not exceeded.

It is also possible to use different MPI communicators to make this distinction, but this can rapidly get out of hand if the communication pattern is not simple and symmetrical. Communicators may also consume significant memory resources inside the MPI library, which tags do not.

5.4 Multiple

In Multiple style, any thread inside (or outside) a parallel region may call MPI, and there are no restrictions on how many threads may be executing MPI calls at the same time. This requires **`MPI_THREAD_MULTIPLE`** support, and Figure 9 shows outline code for this style.

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

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

Figure 9: Outline code in Multiple style

The advantages of this style are:

- Messages from different threads can (in theory) be sent and/or received at the same time, though many MPI implementations may serialise them internally to some extent.
- As with Serialized, it is quite natural for threads to communicate only their “own” data.
- It is possible (but not guaranteed) that the MPI library can pack or unpack separate datatypes on more than one thread concurrently.

- Compared to Serialized, the user has fewer concerns about synchronising threads correctly as some of this responsibility has been passed to the MPI library.

However, some potential disadvantages are:

- From the development and maintainability point of view, this style has most of the disadvantages of Serialized.
- Not all MPI implementations support **`MPI_THREAD_MULTIPLE`** so applications written in this style will have some loss of portability.
- Some MPI implementations do not perform well in this style. If thread safety is implemented crudely using global locks, the locking overheads inside the library and serialisation of MPI calls may outweigh any potential performance benefits.

The performance of some MPI implementations with **`MPI_THREAD_MULTIPLE`** support has improved in recent releases, compared to the situation a few years ago, but it is still possible to observe poor behaviour. Compared to, say, Master-Only or Funneled style, the internal synchronisation overheads of Multiple can easily degrade the observed communication performance, even though the cache locality is better and there may be fewer (or no) OpenMP barriers required. This is especially true for short message lengths, where the time spent in the MPI software stack is significant compared to the time transferring bytes through the network.

A proposed extension to MPI standard, called “endpoints”, currently under discussion in the MPI Forum, which is designed to improve both usability and performance of Multiple style is described in [3].

5.5 Asynchronous Tasks

In Asynchronous Tasks style, MPI calls occur inside OpenMP dependent tasks. Since multiple threads may make concurrent MPI calls, this is a special case of Multiple style, and thus requires **`MPI_THREAD_MULTIPLE`** support. Figure 10 and Figure 11 show outline code for this style.

```
!$OMP parallel
!$OMP single
!$OMP task depend (out:sbuf)
…  write sbuf
!$OMP end task
… more tasks
!$OMP task depend(in:sbuf) depend(out:rbuf)
  call MPI_SendRecv(sbuf,…,rbuf,…)
!$OMP end task
… more tasks
!$OMP task depend (in:rbuf)
…  use rbuf
!$OMP end task
!$OMP end single
!$OMP end parallel
```

Figure 10: Outline Fortran code in Asynchronous style

```
#pragma omp parallel
{
#pragma omp single
{
#pragma omp task depend (out:sbuf)
   {  … write sbuf }
… more tasks
#pragma omp task depend (in:sbuf) depend (out:rbuf)
   { MPI_SendRecv(sbuf,…,rbuf,…); }
… more tasks
#pragma omp task depend (in:rbuf)
   {  … use rbuf }
}// end single
}// end parallel
```

Figure 11: Outline C code in Asynchronous style

The idea of using OpenMP dependent tasks is to minimise the synchronisation between threads and allow the OpenMP runtime as much flexibility as possible in scheduling computation and communication. This style of MPI + OpenMP programming is relatively new, as OpenMP implementations supporting task dependencies have only recently appeared. For a full discussion of this style of MPI + OpenMP programming, please refer to the separate INTERTWinE Best Practice Guide to MPI + OmpSs/OpenMP Tasks.

6 Best practice tips

- Before starting a hybrid MPI + OpenMP implementation, be clear about your motivation: is it to reduce memory requirements, or improve performance (or both)?
- If the main motivation is to reduce memory requirements, try to estimate the potential savings from reducing data replication.
- If the main motivation is performance, make sure you have a good understanding of the bottlenecks in the MPI code (e.g. is the lack of scalability due to load imbalance, or insufficient parallelism, or communication overheads?), and that you can sensibly argue why the hybrid version could help to alleviate them.
- If your application consumes a significant fraction of execution time in calls to external libraries, determine whether there are suitable hybrid/threaded/thread-safe versions of the library, or of alternative libraries.
- Make sure you have access to performance analysis and debugging tools that support hybrid MPI + OpenMP.
- Profile the pure MPI code at or around the core count where scalability is being lost to ensure you have a good picture of which routines need to be effectively parallelised with OpenMP: you should be looking to cover at least 90% of the execution time inside OpenMP parallel regions.
- Consider adding timing calls (e.g. using **`omp_get_wtime()`**) around every OpenMP parallel region so that parts of the code with poor OpenMP performance can be readily identified.
- Consider Master-Only or Funneled style for the initial implementation, and only later migrate (possible just parts of the code) to other styles if performance is inadequate.
- If you intend to run the hybrid code to run on systems with NUMA nodes, consider parallelising the initialisation of large data structures with OpenMP, to take advantage of the first touch allocation policy.
- Make use of OpenMP's conditional compilation features to ensure that the application can still be built without OpenMP.
- If the application makes use of derived datatypes to pack/unpack non-contiguous data, consider replacing these with user-level pack/unpack routines which can be parallelised with OpenMP.
- Make sure the OpenMP environment variables listed in Section 3.5 have the correct settings.
- If you are running on a system with NUMA nodes, check that the batch system and/or MPI launcher are correctly configured to place and bind processes and threads sensibly. A test program that uses, e.g. **`sched_getaffinity()`** to print out the core affinity of every thread and process can be very helpful. See https://github.com/olcf/XC30-Training/blob/master/affinity/Xthi.c for an example.

7 References

[1] Mozdzynski, G., Bull M., Richardson, R., A codesign effort to get ECMWF's IFS model to an extreme O(100) OpenMP threads per MPI task for the [Peta,Exa]Scale, 17th Workshop on High Performance Computing in Meteorology, http://www.ecmwf.int/sites/default/files/elibrary/2016/16794-codesign-effort-get-ecmwfs-ifs-model-extreme-o100-openmp-threads-mpi-task-petaexascale.pdf

[2] Threads and PETSc, https://www.mcs.anl.gov/petsc/miscellaneous/petscthreads

[3] Dinan, J., Balaji, P., Goodell, D., Miller, D., Snir, M., Thakur, R., Enabling MPI Interoperability Through Flexible Communication Endpoints, in Proceedings of EuroMPI 2013, http://www.mcs.anl.gov/papers/P4080-0613-1.pdf