Optimising Performance on Mellanox QDR and Intel Westmere Platforms

Dell/Cambridge HPC Solution Centre

Aleksander Korzynski, Dr Paul Calleja, Stuart Rankin

White paper structure

Abstract

This whitepaper describes the various factors affecting application performance using Mellanox QDR Infiniband interconnects in combination with Intel X5600-series-based platforms. The recommendations in this whitepaper are based on experience gained in operating a large HPC cluster at the University of Cambridge HPC Service. We demonstrate how application performance on Mellanox QDR can depend heavily on the choice of the MPI implementation and on the detailed configuration of the implementation, and we also show that no single MPI implementation is optimal for all applications. A methodology is given for comparing and selecting the optimal MPI implementation and the MPI configuration for a given application. We also demonstrate the magnitude of the performance change between MPI implementations and describe how to fine-tune Intel MPI in order to optimise performance. We also describe factors affecting performance on architectures having some element of Non-Uniform Memory Access (NUMA), which is a relatively new feature of the Intel 5600 system architecture. Unless an application is run in a certain way, NUMA-related issues may lead to suboptimal application performance. We also propose methods for optimising performance on these NUMA systems using features built into Intel MPI.

1.0 Introduction

Application performance on platforms based on the Intel X5600-series processor and using Mellanox QDR Infiniband depends heavily on system configuration and runtime environment. In Section 2, we describe how to optimise MPI communications on Mellanox QDR Infiniband. In Section 3, we discuss the issues associated with running applications on Intel Westmere NUMA platforms. In Appendix A we describe how to integrate Intel MPI with the PBS (Torque) queuing system, although this mechanism is easily portable to other scheduling systems. In Appendix B we present a checklist of the recommended settings, and in Appendix C we present the reasons for creating a custom tuning utility for Intel MPI.

2.0 Optimising MPI on Mellanox QDR Infiniband

In this section, we demonstrate how to optimise MPI communications, and we focus on Mellanox QDR Infiniband in particular. We present a methodology for comparing interconnects, MPI implementations and MPI configurations and selecting the optimal interconnect, MPI implementation and MPI configuration for a given application. The methodology is based on the IMB synthetic benchmark. Next, we demonstrate the correlation between IMB performance and the performance of a real-world HPC application. We also demonstrate the magnitude of difference between MPI implementations and MPI configurations on Mellanox QDR Infiniband. Finally, we describe how to further improve Intel MPI performance with fine-tuning.

2.1 Methodology: the IMB benchmark

We note that the most reliable method of finding the optimal interconnect, MPI implementation and MPI configuration for a given application is to simply test that application on every combination available. However, that methodology can be expensive in terms of man-hours and machine time.

2.1 .1 Background

The MPI standard defines a number of calls that represent different communication patterns, such as Alltoall, Gather, Scatter, etc. Most of those calls accept a message as an argument and the message has a certain size, usually from 0 to a few megabytes. Some MPI implementations offer the choice of the algorithm for a given call, message size, node count, core count and processes per node (ppn) value. Different algorithms are optimal on different platforms.

The Intel MPI Benchmarks (IMB) tests the most important MPI calls by simply invoking each call using a given message size, node count, core count and ppn value and measuring the time it takes to complete the call. The test is invoked multiple times and the minimum, maximum and average timings are obtained. IMB reports the timings but does not interpret them. We are more interested in the throughput for a given message size rather than the actual timing. The throughput is given by the following formula:
Throughput (bytes/second] = (message_size [bytes]/timing [microseconds])*1,000,000.

We propose the following methodology for choosing the optimal MPI system (i.e. interconnect, MPI implementation and MPI configuration) for a given application:
(1) obtain a profile of the application on a given test case, node count, core count and ppn value.
The profile should contain the information on

- time spent in MPI calls,
- MPI call type
- MPI call message size

(2) check the time spent in MPI communications. If it is small, it may be worthwhile to stop here and try optimising other aspects of the application rather than MPI communications.
(3) obtain IMB tests on the system in question. We recommend running IMB at least twice and choosing the better results, because random factors can sometimes slow individual timings down.
When comparing different choices of the software environment, we recommend running the tests on exactly the same nodes in order to mitigate the impact of differences between individual nodes.
The IMB results may already exist and be publically available so it may not be necessary to actually run them.

(4) use a spreadsheet to calculate the throughput for each message size and plot the data on a chart.
(5) compare the systems in question by looking at the IMB throughputs obtained on the MPI calls, message sizes, node counts, core counts and ppn values used by the application. The system with the best throughputs will often be the optimal choice for that application.
(6) if the application allows for configuring the message sizes that the application uses, look through the IMB throughputs for the peak value and configure the application accordingly.

In correlating IMB performance to application performance, it is important to keep in mind that sometimes there is a tradeoff between achieving the best MPI performance vs CPU overhead. So it must be noted that we may not always be able to predict application performance based on IMB numbers.

In the following sections, we demonstrate how the above method works on a real-world HPC application. We also demonstrate the magnitude of the performance delta between various MPI implementations and MPI configurations.

2.2 Correlation between IMB and real-world application performance

The impact of the choice of the MPI implementation and MPI configuration can be observed on CASTEP[2] 5.0. It is an application used in material science and developed at the University of Cambridge. Profiling of CASTEP reveals that it spends much of its time in the Alltoallv call. A significant amount of time can also be spent in Scatter and Scatterv calls. Under the default settings, the greater the core count, the smaller the Alltoallv message sizes that CASTEP uses. CASTEP provides the user with a limited control of the message size used for the Alltoallv call using a special message_size parameter to CASTEP.

Under the default message_size, every CASTEP process participates in the Alltoallv calls. When the message_size parameter to CASTEP is increased, fewer processes participate in Alltoallv and CASTEP handles communication with the remaining processes using gathering and scattering [1]. The message size used for Alltoallv is then greater.

The message_size parameter is in 16-byte units. However, the actual message sizes will not be exactly the configured one. For example, we have determined by profiling that setting the parameter to 128 on 256 cores produces message sizes around 128*16=2kB, but not exactly. On the other hand, setting a message size of 32768 on 256 cores produces message sizes near 1MB and 2MB, even though 32768*16=512K. Also, we have experimentally found that with the message_size parameter set to 32768 on 256 cores, CASTEP uses exactly 16 processes for Alltoallv and the communication with the remaining ones is done though gathering and scattering.

We present the IMB results for the Alltoallv call on 64 cores and 8 processes per node across many message sizes. We compare OpenMPI, mvapich2 and Intel MPI. We also compare two different algorithms offered by Intel MPI. Algorithm 1 is the default in version 4.0.1, but algorithm 2 was the default in 4.0.0. Two algorithms are also offered by OpenMPI and the default is algorithm 1. mvapich2 offers only one algorithm for Alltoallv. We have performed two series of tests: on ConnectX/2 cards and on a mixed environment of ConnectX/2 and the previous generation ConnectX cards.

Note also that we set the OpenMPIs mpi_leave_pinned parameter to 1, which is recommended for benchmarking.

Mixed ConnectX and ConnectX/2, Mellanox QDR, MPI_Alltoallv, 64 cores, 8 processes per node

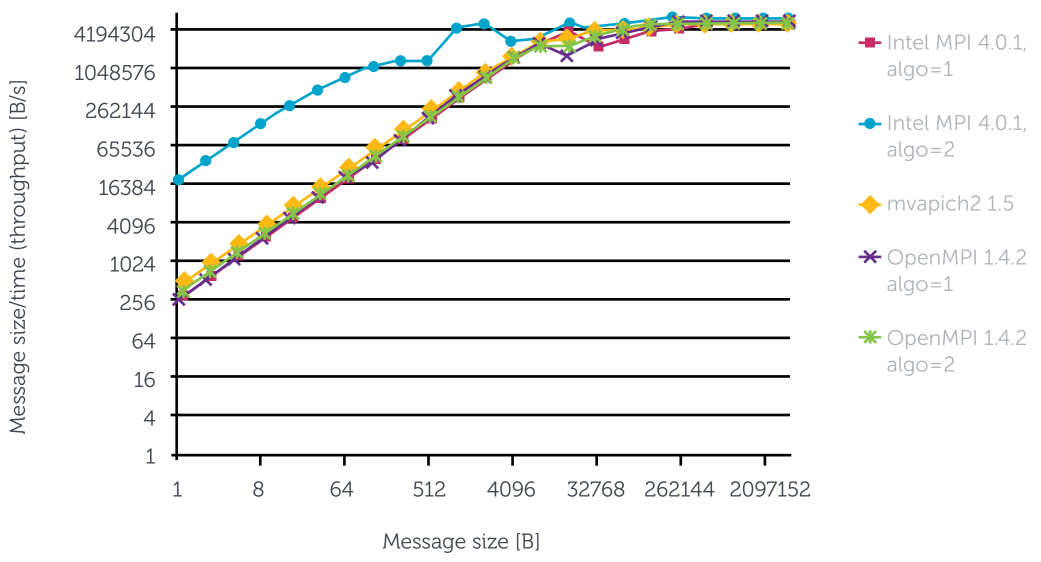

Figure 1: MPI_Alltoallv on 64 cores on Westmere, Mellanox QDR, mixed ConnectX and ConnectX/2.

ConnectX/2, Mellanox QDR, MPI_Alltoallv, 64 cores, 8 processes per node

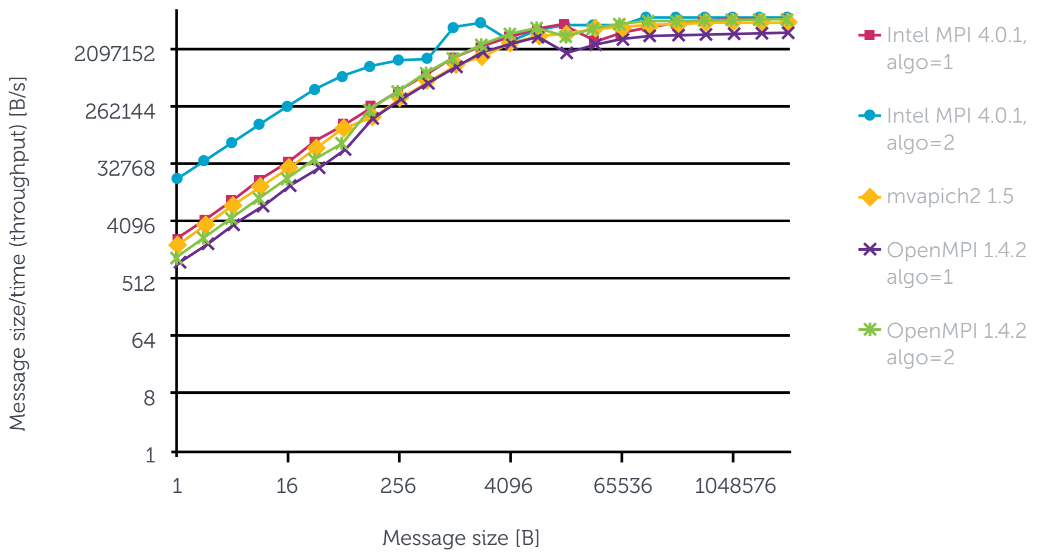

Figure 2: MPI_Alltoallv on 64 cores on Westmere, Mellanox QDR, ConnectX/2.

On the mixed ConnectX and ConnectX/2 variant, we can observe a massive difference between the MPI implementations on the Alltoallv call on small message sizes. The best variant, Intel MPI with the second algorithm (Plum's algorithm), is about 40 times as fast on small message sizes than the next best variant: mvapich2.

On ConnectX/2, Intel MPI with the Plum's algorithm yields the same performance as on the mixed ConnectX and ConnectX/2 variant, whereas the other variants yield higher performance on small message sizes. However, Intel MPI with the Plum's algorithm is still the fastest of them all. On small messages, it is about 10 times as fast as the next best MPI implementation: mvapich2.

Furthermore, we observe that there is a sweet spot in the Intel MPI Plum's algorithm somewhere near the message size of 2kB.

In order to demonstrate the correlation between IMB results and application performance, let's consider the mixed ConnectX and ConnectX/2 variant, because it shows a higher magnitude of difference between MPI implementations than the pure ConnectX/2 variant. We can predict that in order to obtain optimal CASTEP performance, one should use Intel MPI configured to use the Plum's algorithm and one should configure CASTEP's Alltoallv message size to around 2kB. In order to have CASTEP use that message size, we set the message_size parameter in CASTEP to 128 (because it is in 16-byte units). We also test CASTEP on Intel MPI with the default message size. If the user cannot use Intel MPI (because it is a commercial implementation), then either OpenMPI or mvapich2 may be the fastest, depending on the actual message sizes used by the application. However, the differences between OpenMPI and mvapich2 on Alltoallv are not as great as between Intel MPI and the other two. Furthermore, OpenMPI algo=2 should be slightly faster than algo=1, which is the default. For mvapich2 and OpenMPI, we test the default CASTEP message_size and also the message_size set to 32768 (on 256 cores, that corresponds to message sizes of 1MB and 2MB). On the following figure, we observe a high correlation of IMB results with CASTEP performance.

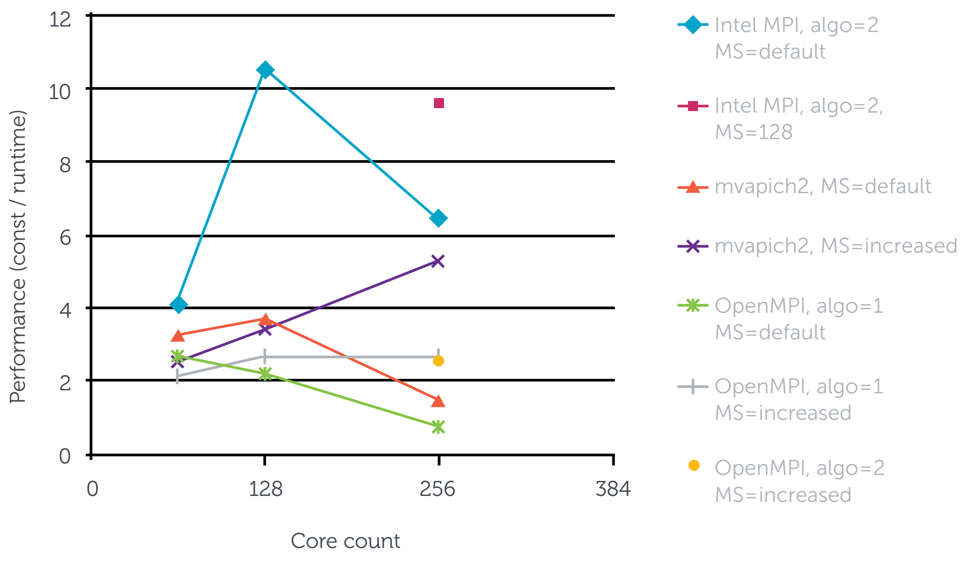

Figure 3: Tuning CASTEP's communications on Westmere/Mellanox QDR.

Firstly, the above results show that the performance delta between MPI implementations and MPI configurations is as high as 500%.

Secondly, let's consider the results with the default message size. When the default message size is used, all processes participate in Alltoallv in CASTEP. We have established by profiling that on 128 cores, CASTEP uses the message size of about 2kB by default. On 64 cores, the message size is larger and on 256 it is smaller. We observe that on all three core counts, 64, 128 and 256, the slowest result is OpenMPI algo=1, mvapich2 is in the middle and the fastest Intel MPI algo=2. That is consistent with IMB results. The greatest difference between the MPI implementations and MPI configurations is observed on 128 cores, that is when the message size is 2kB. That shows a strong correlation with IMB results: message size of 2kB corresponds to the sweet spot in Intel MPI performance.

Furthermore, we observe that we can significantly improve Intel MPI performance on 256 cores by explicitly setting the CASTEP message size to 2kB (message_size=128). That also shows a strong correlation with IMB results.

However, on increasing message size with OpenMPI and mvapich2, the correlation with the IMB results obtained on 64 cores is weak. mvapich2 performs significantly faster than OpenMPI algo=1 and OpenMPI algo=2 performs slightly slower than OpenMPI. That is not consistent with IMB results on 64 cores, but that is not a surprise, because, as explained earlier, in this case CASTEP uses 16 processes for Alltoallv with some arbitrary topology and the remaining communication is done through gathering and scattering. The optimal algorithm for Alltoallv for that core count and topology is not necessarily the same one as for 64 cores done for the IMB test, so the IMB test on 64 cores is not reliable in this case. Furthermore, with increased message size, a significant part of the run time is spent in gathering and scattering MPI calls and their performance also has an impact on CASTEP performance.

In summary, IMB can be a useful tool for predicting application performance, but since it only tests a small set of variables that application performance depends on, it has limitations in applicability.

2.3 MPI Implementations

It is virtually impossible to present a comprehensive comparison of MPI implementations because of the large number of variables that affect MPI performance. The performance of every MPI implementation depends on the node count, core count, ppn value, MPI call, message size, the configuration of that implementation and, of course, the underlying hardware. Furthermore, the performance of an MPI implementation changes with the version of the software.

As a result, the most reasonable approach is to compare MPI implementations for given job profiles. We will now demonstrate a methodology for comparing the implementations across multiple MPI calls. The user can apply the same methodology to compare MPI implementations in his environment in order to choose the fastest implementation for his job profiles.

We also needed to pick a single MPI implementation as the default one for the Westmere/Mellanox QDR cluster at the University of Cambridge. We have chosen Intel MPI. There were the following reasons for doing so:

- As we have shown on Figure 1, Intel MPI 4.0.1 offers significantly higher performance on the Alltoallv call on small messages than OpenMPI 1.4.2 and mvapich2 1.5. Alltoallv performance on small message sizes is very important for the application CASTEP, which is commonly used at the cluster at the University of Cambridge.
- There is commercial support for Intel MPI.

We have also focused our integration, configuration and tuning activities on the same MPI implementation.

Nonetheless, we will now demonstrate the methodology for comparing MPI implementations across multiple MPI calls. The following considerations assume the core count of 96, node count of 8, ppn value of 12 and two different message sizes: a small one, 32 bytes, and a large one, 1 megabyte. We will use Intel MPI 4.0.1, OpenMPI 1.4.2 and mvapich2 1.5.

We will also compare a default version of Intel MPI and a version with tuned MPI algorithms. The tuning procedure is described later in the document. Due to limited time, we have decided to tune only the MPI implementation chosen as the default. We note that means that OpenMPI and mvapich2 performance can potentially be improved through tuning. We also note that there are other tuneable variables in Intel MPI that we did not consider. That means that Intel MPI performance can also be potentially improved. Nonetheless, comparing the default configuration of the MPI implementations is a good starting point. The comparison will also demonstrate the magnitude of difference between the MPI implementations and MPI configurations.

The methodology of comparing MPI implementations is based on the Kiviat diagram, also known as the radar chart. The diagram allows for comparing the implementations against several criteria at once. We will now explain how to read the diagram. It consists of spokes extending from the centre. Each spoke corresponds to one of the variables. A point on the spoke corresponds to a value of the variable. The points corresponding to the same MPI implementation are connected with lines. The further away the lines are from the centre of the graph, the better.

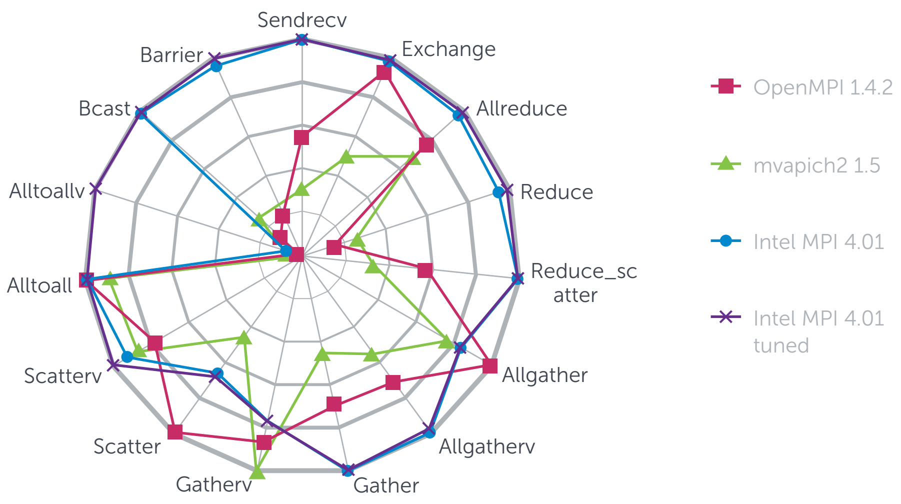

Figure 4: MPI implementations using a small message size: 32 bytes. The jobs used 96 cores, 8 node, 12 processes per node. The hardware was Mellanox QDR ConnectX/2.

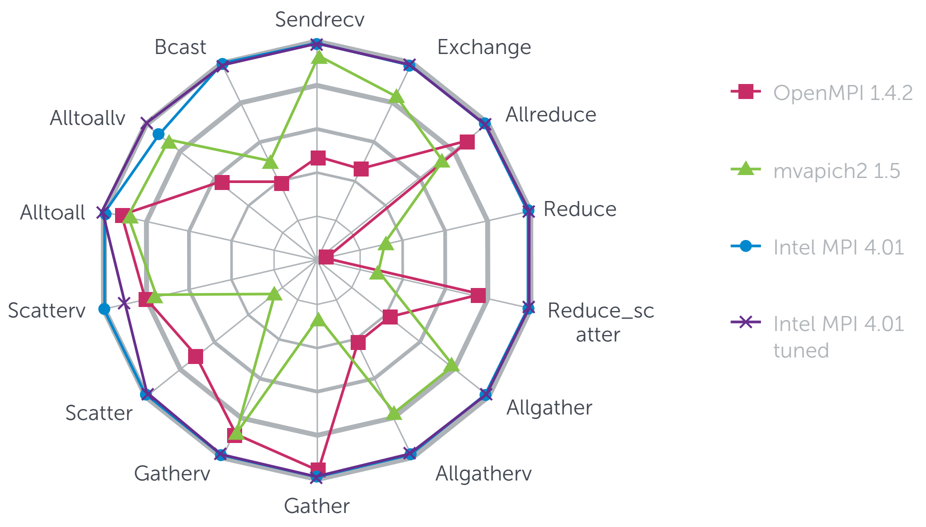

Figure 5: MPI implementations using a large message size: 1MB. The jobs used 96 cores, 8 node, 12 processes per node. The hardware was Mellanox QDR ConnectX/2

Our observations confirm that no single MPI implementation offers the highest performance on all MPI calls. Nevertheless, on the tested parameters, Intel MPI 4.0.1 offers the highest performance on most of the calls. That confirms that it is a reasonable choice for the default implementation.

We also observe that tuning of Intel MPI 4.0.1 makes the greatest difference on the Alltoallv call on small message sizes.

However, as explained earlier, one should always attempt to find the optimal MPI implementation and MPI configuration for the specific environment and parameters required by a given job.

2.4 Tuning MPI algorithms in Intel MPI

MPI implementations can be tuned, which can significantly improve performance. There are a wide variety of variables that can be tuned. One of the aspects that can be tuned is the choice of algorithm for each MPI call, message size, node count, core count and ppn value. We have found that the choice of algorithm can make a significant difference on Intel MPI running on Mellanox QDR. By default, Intel MPI uses various heuristics to choose the algorithms, but it is better to determine them experimentally. In this section, we will describe how to do that.

In Intel MPI, the choice of algorithms for a call can be specified using I_MPI_ADJUST_XXX environment variables, where XXX is the name of the MPI function. For example, for the Alltoall call, the variable is I_MPI_ADJUST_ALLTOALL. The algorithm can be configured in one of the following ways:

1. I_MPI_ADJUST_XXX=2
   Use algorithm 2.

2. I_MPI_ADJUST_XXX='2@36'
   Using algorithm 2 on the core count of 36.

3. I_MPI_ADJUST_XXX='1:0-599;3:600-4194304'
   Use algorithm 1 on message sizes 0-599 and algorithm 3 on message sizes 600-4194304.

4. I_MPI_ADJUST_XXX='1:0-599@36;4:600-4194304@36'
   Use algorithm 1 on message sizes 0-599 and core count of 36 and algorithm 4 of message sizes 600-4194304 and core count of 36.

The following variables can be configured using all four methods:

I_MPI_ADJUST_BCAST
I_MPI_ADJUST_GATHER
I_MPI_ADJUST_ALLGATHER
I_MPI_ADJUST_ALLGATHERV
I_MPI_ADJUST_SCATTER
I_MPI_ADJUST_REDUCE
I_MPI_ADJUST_ALLREDUCE
I_MPI_ADJUST_REDUCE_SCATTER
I_MPI_ADJUST_ALLTOALL

The following variables can be configured only using the first two methods, because the call does not accept a message as argument:

I_MPI_ADJUST_BARRIER

The following variables can be configured only using the first two methods because of a limitation in the MPI library:

I_MPI_ADJUST_GATHERV
I_MPI_ADJUST_SCATTERV
I_MPI_ADJUST_ALLTOALLV

The algorithm choices obtained using the methodology described in this section are intended to be optimal for most applications. However, a different selection of algorithms may still yield higher performance on a minority of applications. For example, as we could see above, the Gatherv, Scatterv and Alltoallv calls cannot be configured per message size but rather a single algorithm has to be selected for all message sizes at once. As a result, the optimal choice will depend on the message sizes used by the application. Nonetheless, the methodology presented in this section obtains reasonable defaults.

There is a tool called mpitune in Intel MPI that, among other uses, can establish optimal default algorithms for MPI calls. However, we have created our own tuning utility, the reasons for which are quite complex and are described in Appendix C. In short, we have made different design decisions to those made by the authors of Intel MPI. Our utility produces different tuning data to the original mpitune. The utility is called yampitune (Yet Another mpitune) and can be downloaded from http://www.hpc.cam.ac.uk/download/
The tool is intended to be used only with Intel MPI 4.0.1.
yampitune produces tuning data for the most important MPI calls across a variety of message sizes on a given combination of the number of nodes (NN), number of processes (NP) and processes per node (PPN). Intel MPI will then pick the tuning data that matches the NN, NP and PPN parameters of the job. Ideally, you should obtain the optimal configuration for as many combinations of NN, NP and PPN as possible, starting with the most popular ones among your users. They will usually be multiples of fully populated nodes or core counts that are powers of two.
To use the tool, run it as follows:

```
yampitune NN NP PPN
```

where:

NN: number of nodes
NP: number of cores
PPN: processes per node

You should run every combination of NN, PPN and NP in a separate directory. The script will produce a number of files, the most important of which will be mpiexec_FABRIC_nn_NN_np_NP_ppn_PPN.conf. Copy that file to the $INTEL_MPI_HOME/etc.

Intel MPI will pick up the tuning data if the -tune parameter is passed to mpirun:

```
mpirun -tune -ppn $PPN -np $NP <application> [options]
```

As a final note, we have noticed that Intel MPI version 4.0.1 always uses an inferior algorithm for Alltoallv on Mellanox QDR. For that reason, we suggest adding the following to the default environment:

```
I_MPI_ADJUST_ALLTOALLV=2
```

The above will ensure that the optimal algorithm will be used even if an mpiexec_FABRIC_nn_NN_np_NP_ppn_PPN.conf file is not available for a given combination of NN, NP and PPN.

3.0 Optimising applications on Intel Westmere NUMA platforms

NUMA stands for Non-Uniform Memory Access. It means that computer memory is divided into memory nodes and a given CPU core can access a given memory node faster than another memory node. The servers operated by the University of Cambridge HPC Service are dual-socket, hex-core Westmere machines. The architecture is illustrated on Figure 6. They have two memory nodes, one for each CPU socket. They also have three levels of cache: L1 (32 kB), L2 (256kB) and L3 (12 MB). There are twelve L1 and L2 caches, each one exclusive to a core. However, there are only two L3 caches, each one shared between cores located on the same socket.

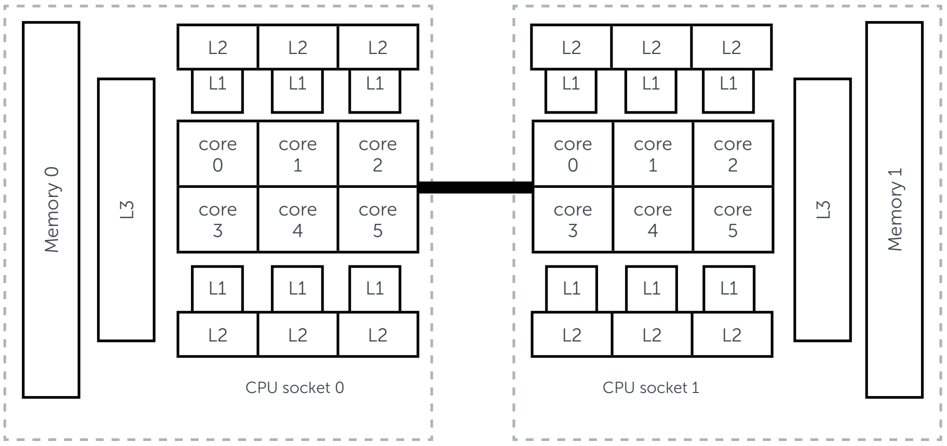

Figure 6: NUMA architecture of Intel Westmere

The non-uniform architecture poses the following challenges to optimisation:
- For a given process, as much memory as possible should be allocated on the local memory node.
That requires proper OS kernel configuration and managing the placement of each process on a socket.
- Application performance may depend on the pattern of placing processes or threads on sockets.
  - On the one hand, for some applications, keeping certain processes on the same socket may yield optimal performance because the sharing of resources may be beneficial. For example, the shared L3 cache may improve inter-process communication. On the other hand, for other applications or other processes of the same application, doing the opposite, that is keeping the processes on different sockets, may be optimal, because the resources available to each process are then maximised. For example, it may be beneficial to have maximal memory bandwidth, interconnect bandwidth and cache size available to each process. In order to obtain optimal performance from a given application, it is usually best to experimentally determine the optimal process placement pattern.
  - In case of multi-threaded applications, we recommend keeping threads on the same socket by default, because threads share the same heap and it is beneficial to keep the heap on the local memory node. However, it is still possible that for some applications performance will be higher when threads are spread between sockets.
  - When a node is under-populated, that is when there are fewer processes or threads running on it than there are cores, the process placement pattern can make a particularly strong difference on performance. We recommend paying special attention to process and thread placement when the node is under-populated.

The above challenges will be examined in more detail further in this document, solutions to the most common problems will be proposed and benchmark results will be presented.

3.1 Memory locality

Memory should be allocated on the local memory node whenever possible. The configuration of the kernel has a major impact on how memory is allocated. For example, under the default configuration in Scientific Linux 5.5, the kernel will allocate memory on a distant node if the local memory node is full with dirty filesystem caches.

Memory allocation behaviour of the kernel is configured using the vm.zone_reclaim_mode sysctl.
The value should be the result of the OR operation of the following:

1 = Zone reclaim on
2 = Zone reclaim writes dirty pages out
4 = Zone reclaim swaps pages

We recommend setting vm.zone_reclaim_mode=3 (reclaim on and write dirty pages out). The default in Scientific Linux 5.5 is the value of 0. We will later demonstrate the impact of changing mode 0 to mode 3 on a performance test.

We also note that in order to let the OS allocate memory on the local memory node, it is necessary to pin processes to cores. Note that it is not necessary to pin a process to a memory node (even though it is possible to do that in Linux). That is because with the kernel configured as above, memory will be allocated from the local memory node automatically, whenever possible.

For multi-threaded applications, it is necessary to pin processes to more than one core. That way each thread will run on its own core. If a process is pinned to more than 1 core, the set of cores to which it is pinned is called a domain. For example, there may be two MPI processes on a Westmere node, each with a 6-core domain, consisting of cores of a given socket.

As a side note, even if Westmere nodes were not NUMA, pinning processes to cores could still improve performance, because it prevents process migration, which in turn disrupts L1 and L2 cache utilisation.

Intel MPI can pin processes to cores and we recommend taking advantage of that functionality. We will describe later how to use it, and discuss the pros and cons in more detail.

We have measured the impact of setting vm.zone_reclaim_mode to 3 as compared to the value of 0. When filesystem caches are empty, the two settings yield the same performance. However, we have created a test that simulates an unfavourable scenario. In the test, we first run rsync for a considerable period of time in order to fill in the filesystem caches on both memory nodes. We then run a single-core, memory-hungry benchmark (StarSTREAM_Copy) on one of the sockets. The filesystem caches on the memory node local to that socket are then released. Next we run the single-core benchmark on the other socket, and we see a considerable difference between mode 3 and 0. In mode 0, the kernel allocates memory from the distant memory node because local memory is used by filesystem caches, and the distant memory is available. In mode 3, the kernel releases the filesystem caches on the local memory node and allocates the process memory locally.

As a comparison, we have also measured the performance of StarSTREAM_Copy under the most unfavourable scenario that is when all memory for each process is allocated on the distant memory node. We obtained that result by pinning each process to a core and pinning the memory of each process to its remote memory node. That scenario may occur when a user manages pinning improperly.

The results are shown in Figure 7.

Impact of memory locality on StarSTREAM_Copy

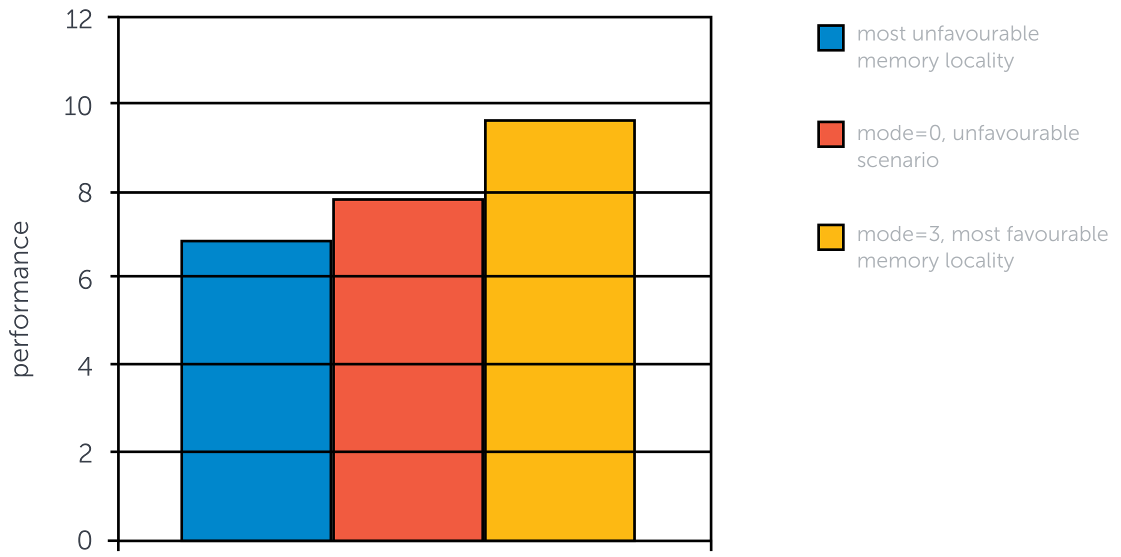

Figure 7: Impact of memory locality on StarSTREAM_Copy

We can observe that the performance delta between mode 0 and mode 3 in an unfavourable scenario can be as high as 25%. Furthermore, the performance delta between the most local and most remote memory placement is more than 40%.

3.2 Process and thread placement: impact on performance

We will present benchmark results that illustrate the impact of process placement pattern on performance. We have found that process placement makes the greatest difference when a node is under-populated. For that reason, we will demonstrate the impact of process placement on under-populated nodes.

We note that there could be many reasons for nodes being under-populated:

- An application supports only specific core counts, which are not multiples of the total number of cores in a node. That can often be the case on Westmere servers with 12 cores per node.
- Memory constraints
- Licensing cost: a license may be charged per core, but using ppn=12 may be only a little faster than ppn=10, meaning that using ppn=10 is more cost-efficient
- Benchmarking rules

We will use SENGA to demonstrate the impact of the process placement pattern on performance. SENGA is a commonly used HPC code developed at the University of Cambridge.

The tests have been obtained on a dual-socket, hexa-core Westmere node. The process count was 8. We have tested two process placement patterns:

- The scatter method, which means spreading the processes equally between sockets.
- The bunch method, which means aggregating the processes on as few sockets as possible.

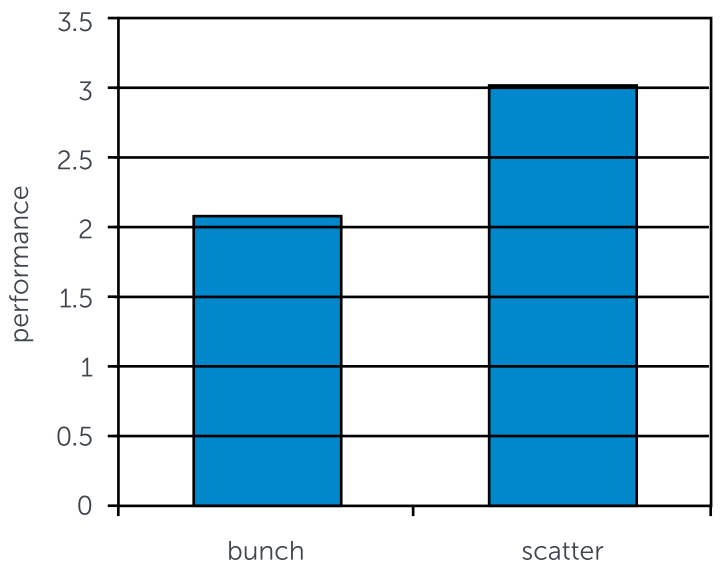

Figure 8: Impact of the pinning pattern on SENGA.

The scatter method yields 50% higher performance than the bunch pattern on SENGA. In conclusion, it is important to determine the optimal process placement pattern for a given application in order to obtain optimal performance. Occasionally where applications widely benefit from under-subscription, it may be worthwhile considering turning off cores in the BIOS to save power and simplify process scheduling.

3.3 Issue of core numbering when managing process placement

Before we describe how to control process placement, we draw attention to the problem of CPU core numbering. When creating custom pinning configurations, it is important to determine the CPU core numbering scheme on a given server. Different servers use different numbering schemes. For example, on some servers cores 1,3,5,7,9,11 are located on socket 0 and cores 0,2,4,6,8,10 on socket 1. On other servers, cores 0,1,2,3,4,5 are located on socket 0 and 6,7,8,9,10,11 on socket 1. Other combinations are possible as well. A useful tool for determining the numbering is cpuinfo from the Intel MPI package.

Below is the output from a Westmere server:

```
Intel(R) Xeon(R)  Processor (Intel64)
=====  Processor composition  =====
Processors(CPUs)  : 12
Packages(sockets) : 2
Cores per package : 6
Threads per core  : 1
=====  Processor identification  =====
Processor         Thread Id.         Core Id.Package Id.
0         0                  0                 1
1         0                  0                 0
2         0                  1                 1
3         0                  1                 0
4         0                  2                 1
5         0                  2                 0
6         0                  8                 1
7         0                  8                 0
8         0                  9                 1
9         0                  9                 0
10        0                  10                1
11        0                  10                0
=====  Placement on packages  =====
Package Id.       Core Id.Processors
1         0,1,2,8,9,10                0,2,4,6,8,10
0         0,1,2,8,9,10                1,3,5,7,9,11
=====  Cache sharing  =====
Cache   Size                Processors
L1  32  KB                  no sharing
L2  256 KB                  no sharing
L3  12  MB                  (0,2,4,6,8,10)(1,3,5,7,9,11)
```

The above output describes the Westmere NUMA architecture with two processor sockets. Cores 0,2,4,6,8,10 are located on socket 1 and cores 1,3,5,7,9,11 are located on socket 0.

3.4 Basics of controlling process placement in Intel MPI

We will now describe how to manage process placement within NUMA nodes using the process pinning mechanism in Intel MPI. We will describe the default pinning pattern in Intel MPI, and then describe how to create custom patterns.

There are two methods to pin processes to cores: by the mpd (multi-purpose daemon) or within the MPI library. It is necessary to use the mpd method in order to give the kernel a chance to allocate all memory of the process on the local memory node. That is not the default setting in version 4.0.1, so we recommend using the following environment variable:

```
I_MPI_PIN_MODE=mpd
```

The two most important variables for controlling process pinning in Intel MPI are I_MPI_PIN_PROCESSOR_LIST and I_MPI_PIN_DOMAIN. For more information about how to use the variables, see the Intel MPI reference manual. However, the following defaults will work for most applications.
The default process pinning pattern in Intel MPI 4.0.1 is I_MPI_PIN_DOMAIN=auto. Under that setting, Intel MPI creates as many domains as there are processes on a node and the size of each domain is obtained by dividing the number of cores by the number of processes. Since we do not know if processes spawn any threads, we recommend that setting as a reasonable default. Also, under that setting, processes will be scattered between sockets. The scatter pattern is a reasonable default, but some applications prefer the bunch pattern.

Alternatively, when all multi-threaded applications running on the cluster are OpenMP applications, a reasonable default is I_MPI_PIN_DOMAIN=omp. The domain size is then set to the value of the OMP_NUM_THREADS environment variable. The OMP_NUM_THREADS variable defines the number of threads per process in OpenMP applications.

In the following table, we describe in more detail how I_MPI_PIN_DOMAIN=auto works on 12-core Westmere nodes.

Number of processes	Default pinning pattern on a 12-core Westmere in Intel MPI
12	The first 6 ranks are pinned to consecutive cores on socket 0, then to consecutive cores on socket 1.
11-7	Ranks are scattered evenly between the sockets. The first half of the ranks is pinned to consecutive cores on socket 0 and the rest to consecutive cores on socket 1. The scatter pattern is a reasonable default, but some applications prefer the bunch pattern.
6-5	Processes are scattered evenly between sockets and each process is pinned to a two-core domain, where both cores are located on the same socket. The above pattern is primarily intended for the scenario when each process spawns 2 threads. Each thread can then run on a separate core. If a process does not spawn extra threads, the pinning pattern is also fine but not optimal. On the one hand, it is good, because the two cores are located on the same socket, so even if the process migrates between the two cores, memory access will still be to the local NUMA node. On the other hand, it would be better to pin each process to a single core, because process migration upsets L1 and L2 cache usage. Nonetheless, if we do not know in advance whether processes spawn threads, we recommend this pinning pattern as a reasonable default. Also, this pattern scatters the processes evenly between sockets, which is a reasonable default, but some applications prefer the bunch pattern.
4	Processes are scattered evenly between sockets and each process is pinned to a three-core domain located on one socket. For the same reasons as for 6 cores, we recommend this pinning pattern as a reasonable default.
3	Processes are scattered evenly between sockets and each process is pinned to a four-core domain located on one socket. For the same reasons as for 6 cores, we recommend this pinning pattern as a reasonable default.
2	Processes are scattered evenly between sockets and each process is pinned to a six-core domain located on one socket. For the same reasons as for 6 cores, we recommend this pinning pattern as a reasonable default.
1	No pinning is done. This is reasonable if we do not know if the process will spawn any threads.

When the above defaults are not optimal, they can be reconfigured using I_MPI_PIN_PROCESSOR_LIST or I_MPI_PIN_DOMAIN.
To have the MPI implementation report the pinning information (and some other useful information) use:

```
I_MPI_DEBUG=5
```

3.5 Process placement in Intel MPI when a node is shared by multiple jobs

At the cluster at the University of Cambridge HPCS, we do not allow the queuing system to schedule multiple jobs on one node. However, a user may aggregate multiple independent programmes into a single job script. When multiple independent programmes are aggregated into a single job, we call each such independent programme a joblet.

With Intel MPI, a joblet corresponds to an instance of mpiexec. Unfortunately, when a node is shared by multiple instances of mpiexec, the pinning mechanism in Intel MPI 4.0.1 does not always work properly. For example, under some circumstances, Intel MPI may pin multiple processes to the same core. It is therefore necessary to override the default pinning configuration when a node is shared by multiple instances of mpiexec. Unfortunately, in some edge cases, the required pinning configuration is quite complex.

Fortunately, our experience of running the cluster at the University of Cambridge tells us that the most common case of node sharing is a single-node job, where each joblet consists of the same number of processes. We have created a joblets script that manages process pinning in that case. The process placement pattern used by the script is to scatter processes evenly between sockets.

The joblets script interfaces with the PBS queuing system. It should be invoked as below:

```
export NUM_JOBLETS=N # number of joblets (customise this)
export NP_PER_JOBLET=K # number of processes per joblet (customise this)
joblets <application> [options]
```

The joblets script defines a unique JOBLET_ID variable for each joblet. Based on the JOBLET_ID, the user can implement a logic that runs a different application in each joblet. For example, if there are three joblets, the <application> in the above example can be a script similar to the following:

```
#!/bin/bash
if [[ "$JOBLET_ID" -eq "1" ]]
then
  cd directory1
  exec ./app1 options1
elif [[ "$JOBLET_ID" -eq "2" ]]
then
  cd directory2
  exec ./app2 options2
elif [[ "$JOBLET_ID" -eq "3" ]]
then
  cd directory3
  exec ./app3 options3
fi
```

The joblets script can also be used for non-MPI, serial, single-core jobs. Although it is not necessary to run such jobs under Intel MPI, we are of the opinion that it is better to use a uniform start-up and pinning mechanism for serial and parallel jobs.

When the user wants to submit multiple single-node jobs, each one consisting of multiple joblets, the PBS job array functionality can be used to extend the joblets functionality. In a job array, each job uses the same job script, but PBS will define a different value of the PBS_ARRAYID variable in each job. When each such job consists of multiple joblets, individual joblets can be identified by the JOBLET_ID variable.
The <application> can then be a script similar to the following:

```
#!/bin/bash
if [[ “$PBS_ARRAYID” -eq “1” ]] && [[ “$JOBLET_ID” -eq “1” ]]
then
  cd directory1
  exec ./app1 options1
elif [[ “$PBS_ARRAYID” -eq “1” ]] && [[ “$JOBLET_ID” -eq “2” ]]
then
  cd directory2
  exec ./app2 options2
elif [[ “$PBS_ARRAYID” -eq “2” ]] && [[ “$JOBLET_ID” -eq “1” ]]
then
  cd directory3
  exec ./app3 options3
elif [[ “$PBS_ARRAYID” -eq “2” ]] && [[ “$JOBLET_ID” -eq “2” ]]
then
  cd directory4
  exec ./app4 options4
fi
```

More information on using PBS job arrays can be found at
<http://www.clusterresources.com/torquedocs/2.1jobsubmission.shtml#jobarray>
The joblets script can be downloaded from http://www.hpc.cam.ac.uk/download/.
In other cases of node sharing, we recommend either creating the pinning configuration manually or disabling pinning entirely. To disable pinning entirely, use the following:

```
I_MPI_PIN=0
```

4.0 Summary

We have shown that the choice of the MPI implementation and MPI configuration can improve the performance of a synthetic benchmark by a factor of 40. We have also shown that on a real-world application, the performance delta between various choices of MPI implementation, MPI configuration and application's MPI-related parameters can be as high as 500%.

Furthermore, we have shown that on a Westmere NUMA system, tuning the kernel configuration on a popular Linux distribution can improve performance by 25% on a synthetic benchmark, due to better memory locality. We have also shown that the most favourable memory locality is 40% faster than the most unfavourable locality on that benchmark. Moreover, we have shown that the performance delta between different process placement patterns on a NUMA system can be as high as 50% on a real-world application.

Finally, we have made recommendations on optimal configuration choices on Mellanox QDR and Intel Westmere platforms. Further information on benchmarking and tuning is available widely and the authors note further best practice information and profiling details are available at the HPC Advisory Council [3].

Appendix A Integrating Intel MPI with PBS

We will describe issues associated with integrating Intel MPI with PBS/Torque.
Firstly, the TM interface of PBS should ideally be used to start processes on remote nodes. That allows the PBS to do full accounting of jobs. It also does not require ssh access between the nodes.
We have evaluated the third-party mpiexec tool from the Ohio Supercomputing Centre (OSC). It has some useful features, such as tight integration with PBS or easy aggregation of multiple independent jobs into one job script. However, we have rejected that option for a number of reasons:

- We have found that the latest version of OSC mpiexec, 0.84, does not work with the latest version of the Intel MPI library: 4.0.1.
- We have found that when Intel MPI 4.0.0 is started with the OSC mpiexec 0.84, the performance of the PingPong IMB test on the 128K message size is significantly lower: 3.3GBps instead of 7.63GBps. Apparently the choice of the start-up method affects the performance of MPI communications in Intel MPI.
- OSC mpiexec is a third-party tool that is not officially supported by the vendor of the MPI implementation.

For the above reasons, we prefer to use the native start-up method in Intel MPI. We have integrated Intel MPI with PBS as follows.
PBS provides the pbsdsh tool that allows starting a programme on remote nodes. A wrapper around pbsdsh is needed to simulate the ssh interface. The wrapper removes all the ssh switches and extracts the hostname from the fully qualified domain name:

```
pbsdsh_intelmpi_wrapper:
#!/bin/bash

. /etc/profile.d/modules.sh
module load torque

while [[ "${1:0:1}" == "-" ]]
```

```
MY_FQDN="$1"
shift

IFS_OLD="$IFS"
IFS="."
HOST_ARR=( $MY_FQDN )
IFS="$IFS_OLD"

exec pbsdsh -o -h "${HOST_ARR[0]}" "$@"
```

Intel MPI can be configured to use the wrapper using the following environment variable:
I_MPI_MPD_RSH=pbsdsh_intelmpi_wrapper

The wrapper can be placed anywhere in the path, for example in the $INTELMPI_HOME/bin64 directory.

Our experience with the above wrapper has been positive most of the time, but we have occasionally seen the start-up procedure fail. If the above start-up method is found to exhibit problems, it is also possible to bypass the TM interface and use ssh to start processes. In that case, use:

```
I_MPI_MPD_RSH=ssh
```

We also recommend using the following commands to start the job. The first two lines extract the core count and ppn values from the PBS nodefile and the last line invokes Intel MPI. The -tune option to mpirun enables tuning, which is described elsewhere in this document.

```
export np=$(cat $PBS_NODEFILE | wc -l)
export ppn=$(uniq -c "$PBS_NODEFILE" | head --lines=1 | sed -e 's/^ *\([0-9]\+\) .*$/\1/g')
mpirun -tune -ppn $ppn -np $np <application> [options]
```

Appendix B Checklist of recommended settings

This section contains a checklist of recommended settings discussed in this whitepaper. It also contains some minor recommendations that are not described elsewhere in the document.

B.1 Linux Kernel 2.6.18

- ✓ sysctl vm.zone_reclaim_mode=3
  Ensures memory locality by writing out dirty pages when necessary.

B.2 Intel MPI 4.0.1

- ✓ I_MPI_PIN_MODE=mpd
  Allows the memory to be collocated with the core. That is possible due to pinning at the level of the MPD daemon rather than the MPI library.

- ✓ I_MPI_FABRICS=shm:dapl
  Both shm:ofa and shm:dapl can be used, but we have tested shm:dapl extensively.

- ✓ I_MPI_DAPL_PROVIDER=ofa-v2-mlx4_0-1
  Various DAPL providers can be used, each with its own pros and cons: see the OFED DAPL release notes for details. We have chosen ofa-v2-mlx4_0-1, because it is fairly new but also well tested, and it scales well.

- ✓ I_MPI_FALLBACK=disable
  This ensures that a fallback interconnect, like Ethernet, is not used when Infiniband is down.

- ✓ I_MPI_JOB_STARTUP_TIMEOUT=60
  The increased timeout prevents occasional failures during start-up.

- ✓ I_MPI_MPD_RSH=pbsdsh_intelmpi_wrapper
  The above ensures tight integration between the PBS/Torque resource manager and Intel MPI.

- ✓ I_MPI_ADJUST_ALLTOALLV=2
  Enables the optimal algorithm for Alltoallv in Intel MPI version 4.0.1

- ✓ Tuning data files should be generated and located in $INTELMPI_HOME/etc. The mpirun script should be invoked with the -tune switch.

- ✓ Users need access to example job scripts as described in this document

B.3 OFED-1.5.1-mlnx9

- ✓ DAPL_ACK_RETRY=7
- ✓ DAPL_ACK_TIMER=20

Important variables that prevent occasional job failures with error code 12 (timeout) on large clusters, that is, having 512+ cores (source: OFED 1.5.1 DAPL release notes). The errors that the above variables prevent look similar to the following:

```
node-i06:31dd: DTO completion ERR: status 12, op OP_RDMA_WRITE, vendor_err 0x81 - 10.143.9.7
[504:node-i06.storage.cluster][../../dapl_module_poll.c:3972] Intel MPI fatal error: ofa-v2-mlx4_0-1 DTO operation posted for
[503:node-i07.storage.cluster] completed with error. status=0x8. cookie=0x0
Assertion failed in file ../../dapl_module_poll.c at line 3973: 0
internal ABORT - process 504
```

- ✓ DAPL_CM_ROUTE_TIMEOUT_MS 20000
- ✓ DAPL_CM_ARP_TIMEOUT_MS 10000
- ✓ DAPL_UCM_REP_TIME 800
- ✓ DAPL_UCM_RTU_TIME 400
- ✓ DAPL_UCM_RETRY 15

The above variables prevent job failures with error code 12 when the user changes the DAPL provider from ofa-v2-mlx4_0-1 to another one.

Appendix C Reasons for creating unofficial Intel MPI tuning utility: yampitune

This appendix describes the reasons we have created the yampitune utility and use it instead of the mpitune utility from Intel MPI 4.0.1.007. However, the issues have been reported to Intel and may be solved in future versions of Intel MPI.

C.1 Tuning Alltoallv, Gatherv and Scatterv

On a Westmere/Mellanox QDR system on 24 cores, mpitune makes the decision to pick algorithm 1 for Alltoallv. However, algorithm 2 is 100% faster than algorithm 1 on small messages, whereas algorithm 1 is 20-40% faster than algorithm 2 on large messages. We prefer to have algorithm 2 as default.

Alltoallv, just like Gatherv and Scatterv, cannot have the algorithm configured per message size but rather a single algorithm is chosen for all message sizes. We suspect that the way mpitune makes the decision is that it adds up all the timings and picks the algorithm with the smaller sum. However, the problem with that method is that for small messages the timings are very small and for large messages the timings are much larger. As a theoretical example, for 32 bytes it may be 60 microseconds and for 4 megabytes it may be 600,000 microseconds. When the numbers are added up, the timings on small message sizes have a very small impact on the sum.

Instead of adding up the timings, we use another method in yampitune. We normalise the timings first. We take the largest message size in the set, in this case 4MB. Then we multiply each timing by the division of the maximum message size and the message size of that timing. For example, for the message size of 32 bytes, we multiply the timing by 4 megabytes divided by 32 bytes:
60 * 4M / 32 = 7864320.

For the message size of 4 megabytes, we multiply the timing by 4 megabytes divided by 4 megabytes:

600,000 * 4M / 4M = 600,000.

Then we add up the results. The algorithm with the smaller sum of normalised timings can then be chosen as the default.

Using the above method can make a big improvement on the performance of codes that send a lot of small messages using the Alltoallv call (up to 100%), but codes that use large message sizes will be slower by only up to 20-40%.

C.2 Core counts

mpitune produces configuration strings as below:
I_MPI_ADJUST_XXX='2:0-123;3:123-4194304'

Let's assume the above output has been generated for 24 processes: 2 nodes, ppn of 12. If a given application uses the same node count and ppn values and the -tune switch to mpirun is used, the MPI implementation will pick the configuration file with the above configuration string.

However, there is a problem in that a given application running on 24 cores may internally have "islands" of communication that use, for example, 8 cores. The above configuration string will not necessarily be optimal for MPI calls within a 24-core application that use core counts smaller than 24. Furthermore, an island of 8 cores may have varying topology: for example, it may be spread equally over both nodes or collocated on one node. The topology probably also has an influence on the optimal choice of algorithm.

For the above reasons, we have decided that yampitune should leave the default heuristics for core counts smaller than the one declared by the application. In order to do that, the create the configuration string as follows:
I_MPI_ADJUST_XXX='2:0-123@24;3:123-4194304@24'

C.3 Verbose logging

mpitune does not produce verbose logs and does not allow for detailed analysis of the impact of tuning on performance. yampitune allows for comparing the original and tuned values by producing a file with the original timings and tuned timings. The timings can be compared using the following:

```
diff -u 10 algo_0_nn_NN_np_NP_ppn_PPN.imb algo_X_nn_NN_np_NP_ppn_PPN.imb
```
NN is the node count, NP the process count and PPN the number of process per node.

5.0 References

[1 Asimina Maniopoulou, Chris Armstrong, CASTEP Quad Core Benchmarking and Optimisation, June 2009, CSE Report

[2] S. J. Clark, M. D. Segall, C. J. Pickard, P. J. Hasnip, M. J. Probert, K. Refson, M. C. Payne, First principles methods using CASTEP, Zeitschrift fuer Kristallographie, 220(5-6) pp. 567-570 (2005)

[3] http://www.hpcadvisorycouncil.com/best_practices.php