Australian Government
Department of Defence
Defence Science and
Technology Organisation

Optimising the Parallelisation of OpenFOAM Simulations

Shannon Keough

Maritime Division
Defence Science and Technology Organisation

DSTO-TR-2987

ABSTRACT

The OpenFOAM computational fluid dynamics toolbox allows parallel computation of fluid dynamics simulations on a theoretically unlimited number of processor cores. The software utilises a message passing interface (MPI) library in order to facilitate communication between parallel processes as the computation is completed. In order to maximise the parallel efficiency of the computation, the way in which the MPI application distributes the problem among the processors needs to be optimised. This report examines the performance of a given high performance computing cluster with several OpenFOAM cases, running using a combination of MPI libraries and corresponding MPI flags. The effect of hardware variations on the speed of the computation is also briefly investigated. The results show a noticeable improvement in simulation time when certain flags are given to the MPI library in order to control the binding and distribution of processes amongst the cluster.

Published by

Maritime Division
DSTO Defence Science and Technology Organisation
506 Lorimer St
Fishermans Bend, Victoria 3207 Australia

Telephone: 1300 333 362
Fax: (03) 9626 7999

AR-015-993
June 2014

APPROVED FOR PUBLIC RELEASE

Optimising the Parallelisation of OpenFOAM Simulations

Executive Summary

Computation Fluid Dynamics (CFD) is used to simulate fluid flow in order to solve engineering problems for a variety of applications. The Electromagnetic Signature Management and Acoustics group (EMSMA) has recently begun using the OpenFOAM CFD toolbox in order to study various flow problems. The software is run on a high performance computing (HPC) cluster built from a series of workstations containing Intel Xeon processors, networked together to run computations simultaneously. Due to the computationally intensive nature of CFD calculations, even a large cluster can take significant periods of time to produce a solution. As such, even small percentage improvements in the performance efficiency of the system can result in savings of days or weeks in the computational time required to produce solutions for the problem being investigated.

In order to run a large problem across many different processors simultaneously, each case is divided into a number of smaller pieces each of which is handled by a single processor core. These pieces are distributed and assigned to each of the cores in the cluster by the software's message passing interface (MPI) library, which also handles the communication required between processes in order to assure consistent results across the case domain.

Multiple combinations of software utilisations are investigated in this report, including the comparison between Intel proprietary compilers and MPI library, with their open source equivalents. The various options provided in the MPI libraries for process distribution are investigated, as well as the effect of these options on different size and types of fluid dynamics problems.

The results show that for the HPC cluster used by the EMSMA group, the best performance is achieved by using an open source compiler and MPI library and running jobs using the "bind-to-core" and "bysocket" MPI flags. It was also seen that the case decomposition method could be tailored to the specific case in order to provide additional performance and that the size of the case was particularly relevant to the maximum number of parallel processes that could be run before no further performance gains would be achieved from the addition of further hardware.

This information will be used to ensure that all CFD simulations run on this hardware are optimised for maximum computational output and efficiency.

Computational Fluid Dynamics (CFD) is used to simulate fluid flow in order to solve a range of engineering problems. The Electromagnetic Signature Management and Anechoics (EMSMA) group has been using the OpenFOAM CFD toolbox with a high performance computing (HPC) cluster in order to solve fluid dynamics problems for signature management applications. Theoretically, OpenFOAM can be run in parallel on an infinite number of computer processors; however, due to the nature of CFD calculations, various bottlenecks exist that limit the scalability of parallel computations. There are many factors that contribute to the computational performance of the system including the specifications of the hardware, as well as the configuration of the software. In order to solve a CFD problem using parallel computation, the domain of the problem is split into a number of sub-domains equal to the number of parallel processes to be run on the available hardware. In order to ensure consistent results across the full domain, communication between processes is facilitated and managed by the message passing interface (MPI) library. The configuration of the MPI protocol can have a significant impact on the overall performance.

This report investigates a series of options available for optimising the configuration of the system including: different hardware specifications and capabilities; direct comparison between the Intel produced compiler and MPI library with an open source equivalent; comparison of different case decomposition methods available in OpenFOAM; as well as a comparison between many optimisation flags available in the MPI libraries for controlling the binding and distribution of parallel processes amongst the cluster. Similar benchmarking and optimisation investigations have been conducted for the SEA supercomputer, however the results of this work were not available for comparison at the time of writing.

The results show that for the given hardware, the best computational speed is achieved by using the open source compiler and MPI library, and using MPI flags to specify binding processes to individual cores and distributing in the “by socket” fashion. It is also shown that the best decomposition method is highly dependent on the case being run and that the greatest efficiency is achieved when the number of parallel processes run is tailored according to the size of the case. It is expected that further parallelisation efficiency can be achieved through the addition of fast networking equipment (e.g. Infiniband) to the cluster.

These outcomes will be used to ensure that the EMSMA HPC cluster is optimised correctly in order to maximise output efficiency for future simulations. The results may also be useful in optimising parallelisation of OpenFOAM on other HPC hardware.

Author

Shannon Joel Keough
Maritime Division

Shannon Keough joined DSTO in 2006 as a member of the Composite and Hybrid Structures group, before moving to the Electromagnetic Signature Management and Accoustics group at the beginning of 2013 to work on computational fluid dynamics for signature management applications. He has a Bachelor of Science in Nanotechnology (with honours) from Flinders University, a Master of Business Management from Monash University and is currently studying a Master of Engineering (Mechanical) at the University of Melbourne.

This page is intentionally blank

Contents

This page is intentionally blank

Abbreviations

CFD	Computational Fluid Dynamics
CPU	Central processing unit
EMSMA	Electromagnetic signature management and acoustics
GCC	GNU compiler collection
HPC	High performance computing
ICC	Intel C++ Compiler
LAN	Local Area Network
MPI	Message passing interface
NFS	Network File System
OpenFOAM	Open source Field Optimisation and Modulation
RAM	Random access memory

This page is intentionally blank

1. Introduction

Computational Fluid Dynamics (CFD) is used to simulate fluid flow in many industrial applications. As the simulations become more complex, the computational power required increases significantly. In cases where computation time can be measured in days or months, even small efficiency increases can result in significant computational cost savings.

The Electromagnetic Signature Management and Acoustics group (EMSMA) has recently begun using CFD simulations to study various flow problems. The main code used to perform the simulations is the open source CFD package OpenFOAM (version 2.2.2) [1]. This software is designed to run in parallel and can be configured to run on effectively any number of cores distributed across any number of networked machines. Ideally, software capable of running in parallel will divide a job into equal pieces and distribute them to as many processing units as are available. If each process is able to run independently, then the speedup gained will be linearly proportional to the increase in computational hardware. Due to the nature of CFD calculations, OpenFOAM requires a significant amount of inter-process communication to ensure consistent results across the case domain. This means that while the software is in theory infinitely parallelisable, every additional process increases communication overhead, which reduces the speedup below the linear optimum. Additionally, as the physical size of the hardware is increased, various system bottlenecks, such as network latency, can prevent the realisation of performance improvements from the addition of further hardware.

Optimising the system to run efficiently is extremely important and in many cases a maximum number of parallel processes will be reached at which point no further performance gains can be achieved through parallelisation. Several factors in both the hardware and software of the system can influence the performance achieved while running simulations. In order to determine the effect of these factors on the efficiency of a CFD simulation, a small High Performance Computing (HPC) cluster was used as the test platform. By using a cluster, effects due to inter-machine communication, as well as intra-machine effects, can be examined. Several tests were run to determine the most efficient methods of running simulations on the hardware available. This report outlines the results of these tests and details the best system configuration for minimising the run time of a CFD simulation on the HPC cluster. While many of the results will be specific to the hardware used in this test platform, other optimisations are expected to be transferable to other HPC clusters running CFD software. Similar benchmarking and optimisation studies have been completed as part of the commissioning of the SEA supercomputer operated by the Hydrodynamics group. At time of writing, the results of this investigation were not available for comparison.

2. Methodology

2.1 High Performance Cluster

The HPC cluster used in this study consists of eight workstations (nodes) with Intel Central Processing Units (CPUs), connected by a 1 Gbps Ethernet network. Each node has the following configuration:

Chassis: HP Z820 with factory liquid cooling
CPU: 2x Intel XEON-E5 2687W – 8 cores per CPU @ 3.1GHz
RAM: 8x 8GB DDR3-1600 Registered ECC memory
Storage: 1TB SATA HDD plus network file system (NFS) server.
OS: CentOS 6.5
CFD Software: OpenFOAM 2.2.2 with OpenMPI 1.5.4 accessed via NFS

The nodes are connected to each other via a local Ethernet switch, which is then connected via 1 Gbps local area network (LAN) to a separate workstation that acts as the network file system (NFS) server. The CFD software is located on the NFS server and loaded to RAM by each node at runtime. The configuration of the NFS server is as follows:

Chassis: HP Z820 with factory liquid cooling
CPU: 2x Intel XEON-E5 2687W – 8 cores per CPU @ 3.1GHz
RAM: 16x 32GB DDR3-1333 Registered ECC memory
Storage: 1TB SATA HDD plus 3x 3TB SATA HDD in RAID0
OS: Red Hat Enterprise Linux 6.5
CFD Software: OpenFOAM 2.2.2 with OpenMPI 1.5.4

2.2 Hardware factors

There are a variety of hardware factors that can affect the efficiency of a CFD calculation. This report will investigate those listed below.

2.2.1 CPU/RAM speed

Generally, the processing speed of the hardware is the main factor that determines the speed at which simulations will run. Since all workstations in the HPC cluster are currently fitted with the same CPU chips (Xeon E5-2687W), the effect of CPU speed is mostly irrelevant to this investigation. However, not all the workstations are fitted with the same speed RAM. For pre and post processing of large simulations, large amounts of RAM are often required. One workstation has been fitted with 32 GB RAM modules in order to maximise the available amount of memory for processing data. However, the larger 32 GB RAM modules are limited to a clock speed of 1333 MHz whereas the rest of the workstations in the cluster are fitted with 8 GB modules that run at 1600 MHz. Since OpenFOAM is a memory intensive application, it is expected that this variation should produce a notable difference in performance.

2.2.2 RAM Channels

As well as the speed of the RAM, the number of memory channels between the CPU and the shared memory can also affect the speed of CFD simulations on the system. The Xeon E5-2687W CPUs each contain eight cores but only four memory channels. Whenever more than four cores are active on a single CPU, there will be some sharing of the memory channels. Since the memory channels have a limited capacity for transferring data, usually referred to as bandwidth, it is possible for the bandwidth on a given memory channel to become saturated. In this situation the cores must idle until the data has finished transferring from the memory. When every core in a node is being utilised, the node contains 16 active cores sharing only 8 memory channels. Assuming that one core is able to saturate the memory bandwidth of one memory channel when accessing data stored in RAM, it could be expected that the performance gains from running a job on eight cores per CPU would be minimal over running on four cores per CPU. If this is the case, due to the increased communication overhead of running extra parallel processes, a simulation running on eight cores per CPU (with four memory channels) might perform worse than the same job running on four cores on the same CPU. Tests running the same CFD simulation on different numbers of cores will be used to probe this effect.

2.2.3 Turbo Boost

Modern Intel CPUs, such as those used in the HPC, contain a turbo boost function that is designed to increase the clock speed of the CPU according to a series of parameters (CPU load, temperature, etc.) in order to increase performance. Since computational performance is directly related to CPU speed, increasing the clock speed should have a noticeable impact on the performance.

Increasing the clock speed of a CPU above the specification (overclocking) is a procedure that has been utilised by enthusiasts for years in order to improve the performance of computer systems, generally for gaming applications [2]. However, overclocking can be difficult to perform successfully, can damage the CPU and voids the manufacturer's warranty. Due to these factors, the inbuilt turbo capability of the Intel chips is a preferable option for maximising performance. Nominally, the Intel turbo function is designed to operate when only a few cores on a CPU are being utilised and the extra heat produced by running at higher speeds can be distributed across the cooling mechanism of the entire chip. In theory, when all of the cores on a chip are running simultaneously, turbo does not activate and so no increase in clock speed is observed.

This raises an interesting point for investigation when considering the effect of limited memory channels as mentioned above. If the performance of four cores is similar to that of eight cores on a single CPU (due to memory channel saturation), the extra clock speed provided by the turbo function when only four cores are running could increase the performance over that of a simulation utilising all eight available cores.

2.2.4 Hyper-threading

In a similar manner to the turbo boost function, modern Intel chips offer a capability called hyper-threading in an attempt to further maximise performance. Hyper-threading allows each core on the CPU to present itself to the operating system as two cores: one real and one virtual. The operating system can then assign jobs to the virtual cores and these jobs are run when the real core would otherwise be idle (such as during memory read/write) theoretically maximising the utilisation of the CPU.

OpenFOAM is generally a memory intensive application, as opposed to strictly CPU intensive. In cases where the CPU is not fully utilised, hyper-threading allows processes assigned to virtual cores to run during the downtime and thus maximises utilisation. However, during CFD computations, CPU utilisation rarely drops below 100% and increasing the number of processes also increases the communication overhead. As a result, it is likely that this increased overhead will outweigh any benefit gained from maximising CPU utilisation through the use of virtual cores.

2.3 Software Factors

While changes to the specification and functionality of the hardware used to run CFD simulations will undoubtedly improve the performance, once hardware has been purchased it is often economically prohibitive to upgrade that hardware. To ensure that the current hardware is being utilised most efficiently, there are several optimisations that can be performed in the software, of which this report investigates the following.

2.3.1 Process Binding and Distribution

The method by which processes are allocated and bound to cores can have a significant impact on the overall speed of a parallel computation. The proximity of processes in the hardware architecture affects the speed and efficiency of inter-process communication. The basic topology of the EMSMA HPC cluster can be seen in Figure 1.

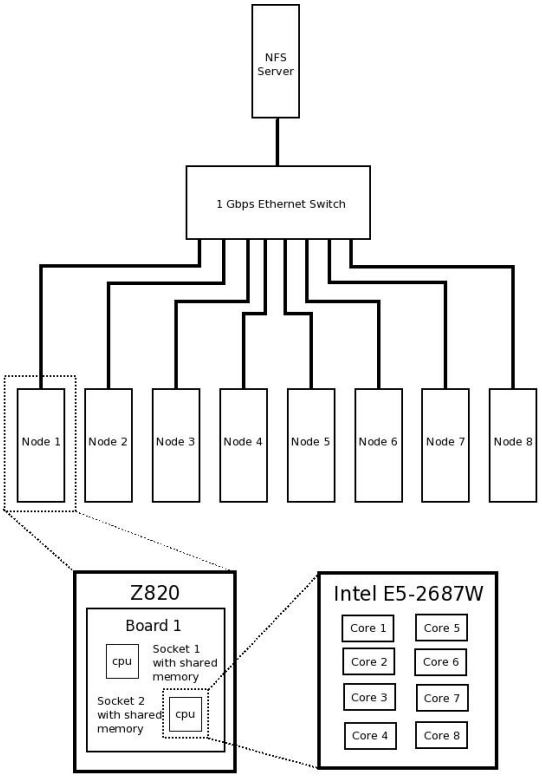

Figure 1 Hardware topology of the HPC cluster

Note: Since the Z820 workstations contain only one motherboard in each chassis, there is no difference between a node and a board for the purpose of this investigation. As such, boards will not be mentioned further in this report.

In most desktop computer systems, including those used for this investigation, process assignment is managed by the operating system so as to balance the load on the system. As part of this management, the operating system will move processes to different cores in an attempt to improve overall system performance. Typically, CFD applications assign discrete data to each parallel process that is stored in the memory attached to the core running that process. If the operating system moves a process to a core that is not directly attached to the same shared memory, the data relevant to that process needs to be rewritten in the memory before the job can continue. This is likely to have a detrimental effect on the performance of CFD simulations.

The need to rewrite data in memory and the subsequent degradation in performance can be avoided by binding (or pinning) a process to a particular core. This prevents the operating system from reassigning a process before the job is finished. OpenMPI (the default message passing interface (MPI) library provided with OpenFOAM) provides several options for process binding and distribution control, each of which can be invoked when initiating the job using the 'mpirun' executable.

The performance of jobs that rely on inter-process communication is also dependent on the distribution of processes across the cluster topology. Communication between cores on the

same CPU is faster than communication between sockets, which is faster than communication between nodes.

A perfectly optimised parallel computation would distribute processes so as to maximise communication between cores on the same CPU and minimise communication between nodes. This level of fine grain optimisation is difficult (perhaps impossible) to achieve given the complexity of most large CFD cases and the limitations in the software. However, some control over the distribution can be attained and, as with process binding, can be easily specified when initiating the job. The flags that can be used for process binding and distribution are outlined in the mpirun man page [3] and are reproduced in Table 1.

Table 1 Binding and Distribution options in OpenMPI

	Flag	**Description**
Binding	bind-to-none	Do not bind processes. (Default)
	bind-to-core	Bind processes to cores.
	bind-to-socket	Bind processes to processor sockets.
Distribution	bycore	Associate processes with successive cores if used with one of the *-bind-to-** options. (Default)
	bysocket	Associate processes with successive processor sockets if used with one of the *-bind-to-** options.
	bynode	Launch processes one per node, cycling by node in a round-robin fashion. This spreads processes evenly among nodes and assigns ranks in a round-robin, "by node" manner.

2.3.2 Case Size and Decomposition

When a case is prepared for running in parallel, the domain is broken into pieces of roughly the same size, each of which is assigned to a core. As the number of processes increases, each piece becomes smaller and the computations can be completed faster. This is the basic principle of parallel computing. As mentioned earlier, increasing parallelisation in OpenFOAM also increases the amount of inter-process communication required in order to keep the results consistent. As a result, the number of parallel processes that can be run on any given case reaches a maximum where the reduction in computation time per process is completely offset by the increase in communication time and no further speedup can be achieved. For any given case of a specified mesh size, this point of maximum efficiency is important to find to ensure that computational resources are not being wasted.

The method by which the domain is decomposed affects the size and shape of each piece and the order in which they are numbered, which in turn affects the relative locations within the domain (inter-processor distance). As a result, the decomposition method influences the time required for each process to complete its designated computations, the

amount of inter-process communication required and the topological distance between processes that need to communicate in the hardware architecture. OpenFOAM provides several built-in methods for decomposing cases to be run in parallel [4]. The description of each method is reproduced in Table 2.

Table 2 Decomposition methods available in OpenFOAM

Method	Description
Simple	Simple geometric decomposition in which the domain is split into pieces by direction, e.g. two pieces in the x-direction, one in the y-direction etc.
Hierarchical	Hierarchical geometric decomposition which is the same as simple except the user specifies the order in which the directional split is done, e.g. first in the y-direction, then the x-direction etc.
Scotch	Scotch decomposition which requires no geometric input from the user and attempts to minimise the number of processor boundaries. The user can specify a weighting for the decomposition between processors, through an optional *processorWeights* keyword which can be useful on machines with differing performance between processors.
Manual	Manual decomposition, where the user directly specifies the allocation of each cell to a particular processor.

Since the manual method requires specification of the binding for each cell, of which there are between several hundred thousand and several million for most cases, the manual method is not utilised at any point in this investigation.

The workstations in the HPC cluster all contain identical hardware, hence the processor weighting function of the Scotch algorithm is also not utilised in this investigation.

2.3.3 Proprietary vs Open Source Compilers/Libraries

Given the open source nature of the OpenFOAM CFD package, the software is most often compiled using an open source compiler (e.g. GCC) and utilises an open source MPI library (e.g. OpenMPI) for process communication. These packages are readily available and freely distributed with the OpenFOAM software and most Linux based operating systems.

Intel produce a proprietary compiler (ICC) and MPI library (IMPI) that purportedly provide performance increases due to optimisations specific to Intel based hardware. The HPC advisory council demonstrate performance increases using the Intel MPI library (compared to the Open MPI library) for clusters containing 8 nodes or more, with up to 44% performance increase for a 16 node cluster as seen in Figure 2.

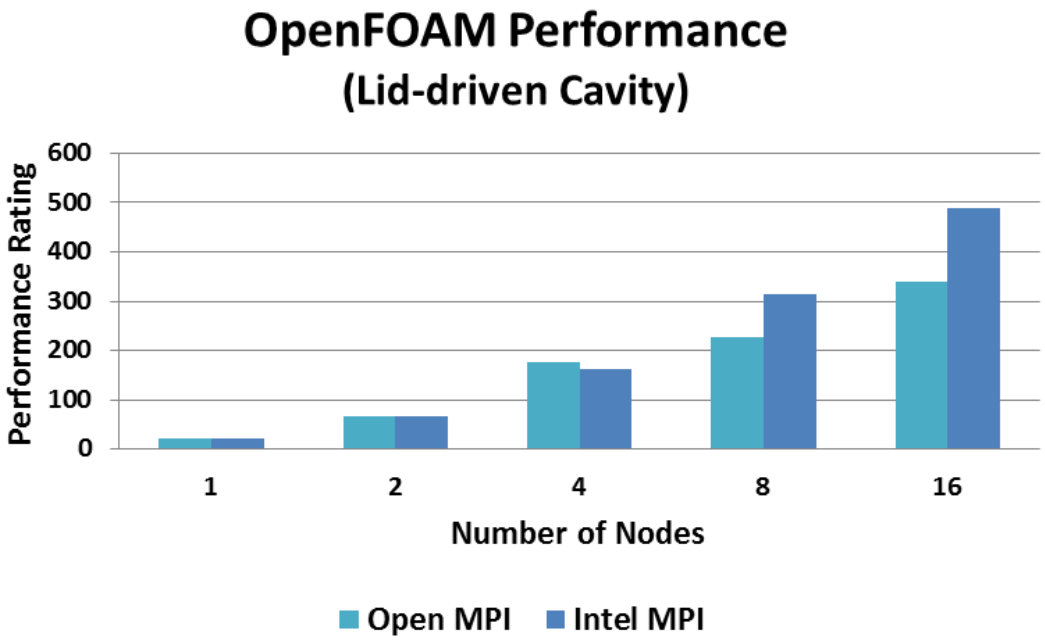

Figure 2 HPC advisory council advertised performance of Intel MPI vs OpenMPI [5]

2.4 CFD Test Cases

To investigate the effect on performance of each of the options outlined, many computations were performed using various combinations of the options to determine the optimum system configuration for running OpenFOAM on the given hardware. Two cases were tested to demonstrate any differences that may exist between the performance outcomes for different solvers. Different size meshes were used in each case to investigate the scalability of parallelisation for different size cases. Except in the case where the decomposition method was explicitly being investigated, the Scotch algorithm was used for parallelisation of all cases. The two cases used are:

(i) *Motorbike Tutorial*
The first case used for testing was a tutorial case provided with the OpenFOAM software that simulates the airflow around a 3D model of a motorbike and rider. This case uses the simpleFoam solver and is provided with a mesh of approximately 350,000 cells. (MB-350k)
A higher resolution mesh of this case was created by doubling the background mesh density in 3 dimensions, which increased the total cell count of the completed mesh to approximately 1.8 million cells. (MB-1.8M)

(ii) *Proprietary DSTO case*
The second case used for testing was a case that has been the subject of a recent DSTO research project. This case uses the interFoam multiphase solver to simulate the interface between water and air in a moving system. Three meshes were used to run performance tests with sizes of 2.7 million, 5.8 million and 64 million cells. (DSTO-2.7M, DSTO-5.8M, DSTO-64M)

2.5 Simulation details

Each computation was scripted to run using a variety of options in sequence. An example of the commands used to initiate jobs when comparing the effect of core binding is given here. For this test, a single node was used running 16 parallel processes.

```
mpirun –np 16 –bind-to-core simpleFoam –parallel > log   (with core binding)
mpirun –np 16 simpleFoam –parallel > log                 (without core binding)
```

After each computation had completed a predetermined number of iterations (time steps), the log files were examined in order to determine the performance of each configuration. The overall computation time was extracted to give an indication of which configuration completed the job the fastest. The total number of time steps completed was then used to determine a performance rating (average time steps per hour). For the purpose of visualising results, the performance rating for each test was normalised against the performance rating for a single unit (either single core or single node depending on the test) and plotted as a parallelisation speedup factor. When visualised in this manner, the slope of the curve can be used to determine the parallelisation efficiency as a measure of speedup per unit increase in hardware. It is important to recognise that even when normalised, the performance rating (or speedup factor) is highly dependent on the complexity of the case, the solver and the cell count of the mesh and so cannot be used to accurately compare the results of different cases directly. It can, however, be used to compare the performance of different system configurations on the same case, which is the purpose of this investigation.

To give a good visualisation of the level of speedup achieved through parallelisation, most tests were run multiple times using varying numbers of cores ranging from serial jobs run on a single core, up to highly parallelised jobs run on the maximum number of available cores. A manifest of computations completed can be found in Appendix A.

3. Results and Discussion

3.1 Process binding and distribution

3.1.1 Binding

It is expected that binding the processes will increase performance by preventing the operating system from moving processes as a part of load balancing operations.

Test Parameters	
Case(s)	Motorbike 1.8 million cells
Hardware	1 Node (1333 MHz RAM)
Processes	Up to 16
MPI flags	(no flags) -bind-to-core -bind-to-socket

The results shown in Figure 3 demonstrate that, as expected, the overall fastest performance was achieved when the processes were bound and all cores were utilised. For greater than twelve parallel processes, running with no binding flags was slower and in some cases the simulation crashed. For less than twelve processes, binding the processes actually caused the simulation to run slower; this is a result of the process distribution as is explained further in section 3.1.2.

In theory, the difference between binding jobs to a socket and binding to specific cores should be negligible, since the cores on each socket access the same shared memory. In practice, the results show that while the two binding options produce very similar performance outcomes, binding to a core was consistently a small margin faster than binding to a socket. It is possible that the operating system uses a small amount of CPU time monitoring the processes and potentially reassigning them within cores on the same socket. When all processes are bound to specific cores, this resource usage in process management would not need to take place.

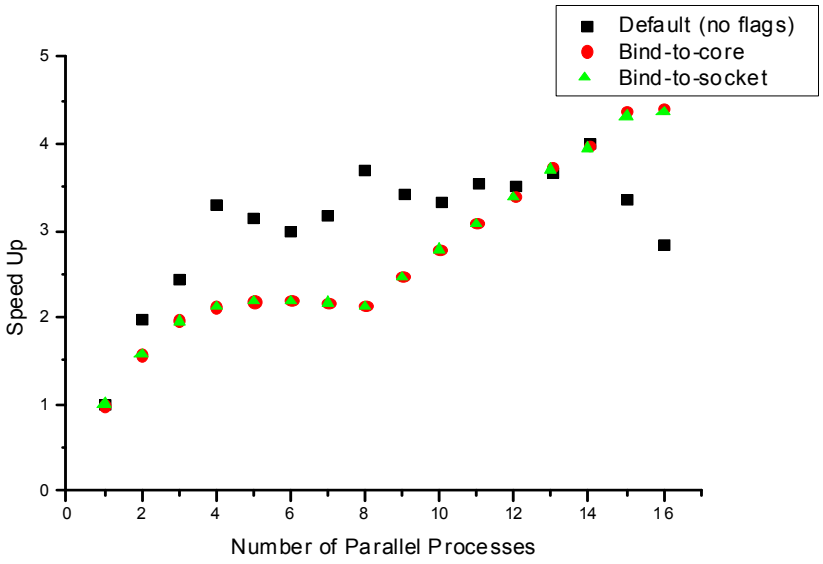

Figure 3 Comparison of core binding performance on a single workstation

3.1.2 Distribution

3.1.2.1 Single Node

The results shown in Figure 3 were obtained by running jobs using only the binding flags. By default, Open MPI distributes jobs consecutively by core number. This means that when sixteen parallel processes are run, the first eight are assigned to the first eight numbered cores - which are all on the first socket - and then the last eight are assigned to the second socket. When only eight processes are run, this default distribution assigns all eight processes to the first socket, while the CPU in the second socket remains idle.
The stepped curves in Figure 3 occur due to this process distribution in combination with the hardware architecture. While each Xeon CPU contains eight cores, they have only four memory channels that allow the cores to read and write from the shared memory. As a result, the expected parallelisation speedup is only fully realised up to four parallel processes per socket. Beyond that limit, each process is required to share memory bandwidth with at least one other process, which reduces the performance.

The job run without binding does not experience this effect due to the reallocation of processes. After the first eight processes are assigned to the first socket, they are then reassigned by the operating system in order to balance the load and so are able to access the additional memory bandwidth on the second socket. As a result, running without binding is faster than running with binding for up to twelve processes, despite the slowdown attributed to process reallocation.

By using the process distribution flags, it is possible to force the system to allocate the processes to each socket in turn such that for eight processes, four will be assigned to each socket and every available memory channel will be utilised. Figure 4 shows the same job run using the “bycore” (default) and “bysocket” distribution flags. Since this test was conducted using a single workstation, the “bynode” distribution flag would have produced the same result as the default option and was not tested.

Test Parameters	
Case(s)	Motorbike 1.8 million cells
Hardware	1 Node (1333 MHz RAM)
Processes	Up to 16
MPI flags	-bind-to-core -bycore -bind-to-core -bysocket

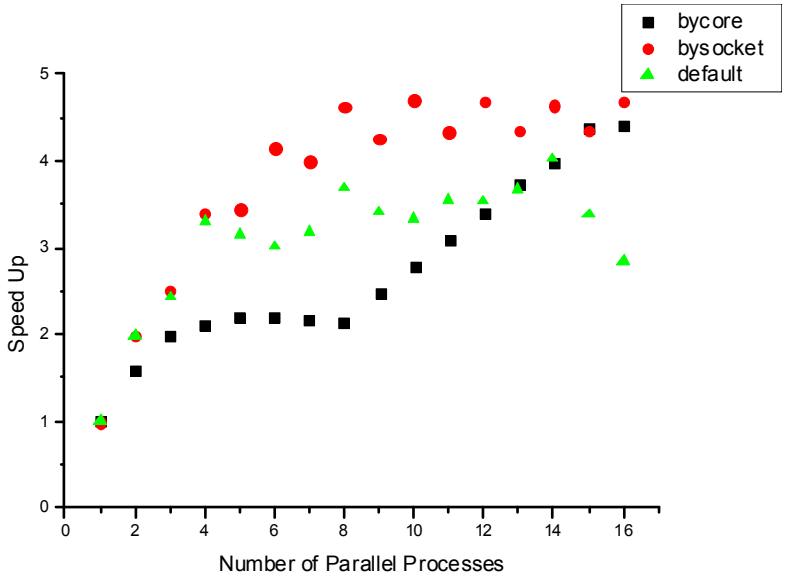

Figure 4 Comparison of process distribution options on a single workstation. The default (no flags) result from Figure 3 is included for comparison.

The results in Figure 4 show that:

- As expected, increases in parallelisation need to be accompanied by available memory bandwidth in order to produce significant speedup.
- Running sixteen parallel processes produces no realisable speedup over running eight evenly distributed processes on this hardware.
- Running odd numbers of parallel processes is slower than running even numbers when using the "bysocket" distribution option.

There is no immediately apparent reason why running with an odd number of processes would be slower than an even number. It is possible that the decomposition algorithm is not able to conveniently divide the domain into an odd number of pieces as easily as for an even number. Such differences could lead to increases in inter-process communication or unevenness in workload distribution that would cause a performance decrease, however this is beyond the scope of this investigation. The effect of case decomposition on performance is investigated in section 3.2.

Theoretically, the only difference between the "bycore" and "bysocket" distribution methods when all sixteen cores are utilised is the amount of communication required between processes on different sockets. It could be expected that processes numbered closely would require greater communication since the decomposition algorithm would number parts of the domain in sequence. This would in theory result in the "bycore" option providing marginally faster results since neighbouring processes would generally be located on the same socket. In practice, the "bysocket" flag consistently produced 1-2% faster run times. Since the decomposed domain in all of the cases presented in this report is 3-dimensional, it is reasonable to assume that the number of neighbouring processes

that are also numbered closely does not represent a significant portion of the overall inter-process communication.

An interesting conclusion from the result in Figure 4 is that it is not better to utilise all of the cores available in a node than to use only half. However, this scenario only occurs when using 1333 MHz RAM. The majority of the workstations used in this investigation contain faster 1600 MHz RAM, which partly overcomes the memory bandwidth issue. Figure 5 shows the same test run on a machine with the faster RAM installed. While the memory bandwidth bottleneck is still present, as evidenced by the change in the slope of the data above eight processes, the results show that on these systems, the fastest outcomes are achieved by utilising all sixteen available cores. The direct comparison between RAM speed is discussed in section 3.5.

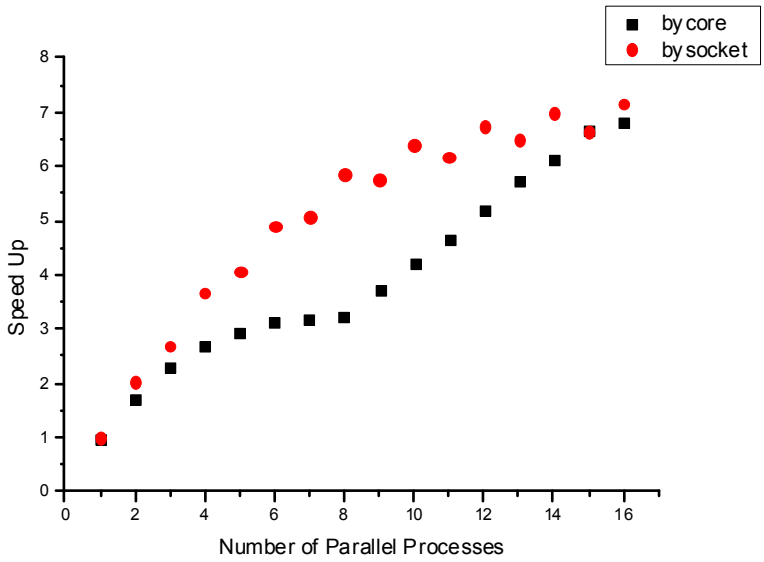

Figure 5 Comparison of process distribution options on a workstation with 1600 MHz RAM

3.1.2.2 Cluster

Once the level of parallelisation is expanded beyond the number of cores contained in a single workstation, processes are required to communicate across a network connection. This network connection can quickly become the most significant bottleneck in the system and, as such, distributing the processes so as to minimise the amount of network communication required can substantially impact the computational speed. The same test used to compare distribution options in a single workstation was used to make the same comparison in a larger networked cluster.

Test Parameters	
Case(s)	Motorbike 1.8 million cells
Hardware	8 Nodes (1600 MHz RAM) 1 Gbps Ethernet Networking
Processes	Up to 16 per node
MPI flags	-bind-to-core –bycore -bind-to-core –bysocket -bind-to-core –bynode

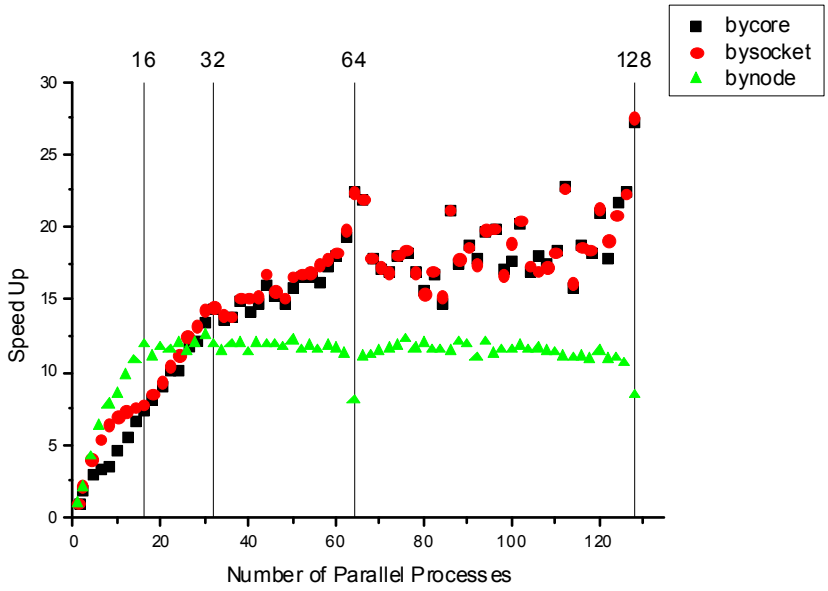

Figure 6 Comparison of process distribution options across a 128 core/8 node cluster

The results of this test, as seen in Figure 6, are ultimately the same as for a single workstation. Distributing "bycore" or "bysocket" gives very similar performance with the "bysocket" option providing slightly faster runtimes in most cases. The "bynode" option is clearly faster for low numbers of parallel processes since running 1 or 2 processes on each machine provides more memory bandwidth and faster CPU clock speeds (with turbo) than running on a single workstation. The "bynode" option very quickly reaches a point where network communication becomes the bottleneck in the system and adding further parallel processes provides no increase in performance.

The scatter in the data at higher numbers of parallel processes are most likely due to a similar effect as that which caused the performance difference between odd and even numbers of processes in a single node (see section 3.1.2.1). Figure 6 clearly shows the best performance being achieved at 64 and 128 processes, with diminished levels of performance between. It is possible that at much higher levels of parallelisation, optimum performance is only achieved when the number of processes is some even multiple of the amount of hardware being utilised.

3.2 Case decomposition method

The decomposition method used to parallelise the cases has a direct impact on the amount of inter-process communication that is required and the distance across the cluster topology that this communication has to travel. Several cases were parallelised using the following decomposition options:

1) Simple decomposition split evenly in the x dimension. (Simple A)
2) Simple decomposition split once in the y dimension and evenly in the x dimension. (Simple B)
3) Hierarchical decomposition split once in the y dimension and evenly in the x dimension. Dimension hierarchy specified as x-y-z. (Hierarchical A)
4) Hierarchical decomposition split once in the y dimension and evenly in the x dimension. Dimension hierarchy specified as z-y-x. (Hierarchical B)
5) Scotch decomposition as determined optimum by the Scotch algorithm.

The cases were decomposed both into 16 pieces for running on an individual workstation and into 128 pieces for running on the entire HPC cluster.

Each decomposition method produced the same average cell count per piece for each case. However the average number of processor faces (boundaries that require inter-process communication) varied significantly, as can be seen in Table 3.

Table 3 Average number of processor faces using various decomposition methods

16 Pieces

	MB-350k	MB-1.8M	DSTO-2.7M	DSTO-5.8M
Simple A	5147	13410	45050	47727
Simple B	3990	10907	24269	29592
Hierarchical A	4030	10949	24283	29641
Hierarchical B	4000	11140	24333	29622
Scotch	2969	10704	15675	37119

128 Pieces

	MB-350k	MB-1.8M	DSTO-2.7M	DSTO-5.8M	DSTO-64M
Simple A	4149	12539	42201	49954	201814
Simple B	2564	7123	23292	25846	104477
Hierarchical A	2572	7134	23299	25857	104477
Hierarchical B	2571	7185	23349	25889	104526
Scotch	981	3234	4933	10505	48829

For the Simple and Hierarchical methods, the cases were not split in the z dimension, as this would create a split at the multiphase interface in the DSTO case. Splitting across the multiphase interface is avoided so as to minimise the number of interface calculations that occur across the processor boundaries.

The performance of each decomposition method was measured and normalised against the best result for each case. The normalised data is presented in Figure 7 and Figure 8.

Test Parameters	
Case(s)	Motorbike 350 thousand cells Motorbike 1.8 million cells DSTO Case 2.7 million cells DSTO Case 5.8 million cells DSTO Case 64 million cells (128 cores only)
Hardware	1 Node (1600 MHz RAM) *or* 8 Nodes (1600 MHz RAM) 1 Gbps Ethernet Networking
Processes	16 per node
MPI flags	-bind-to-core –bysocket

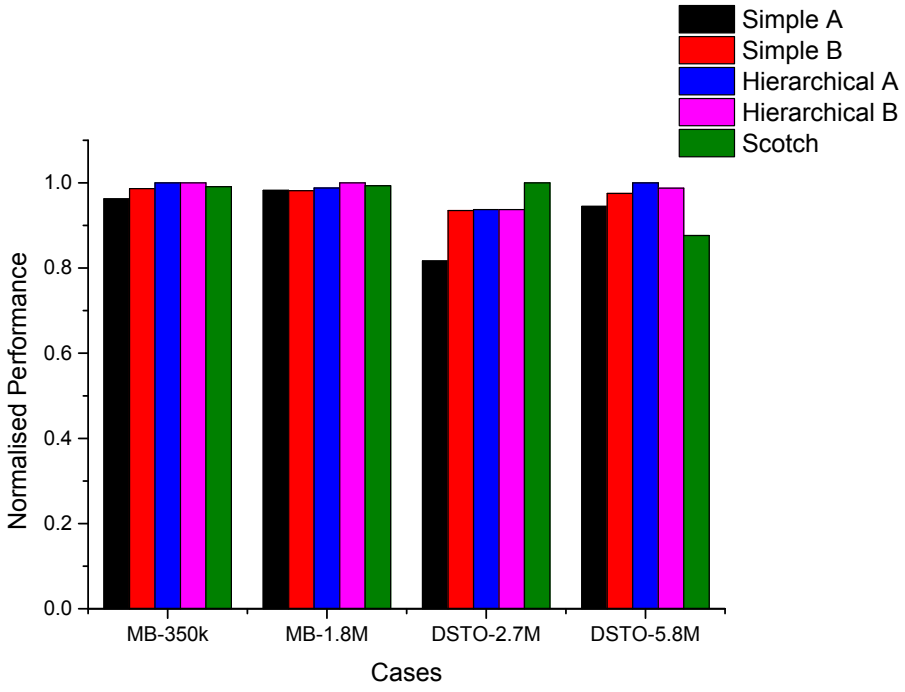

Figure 7 Performance comparison of decomposition methods on a single workstation

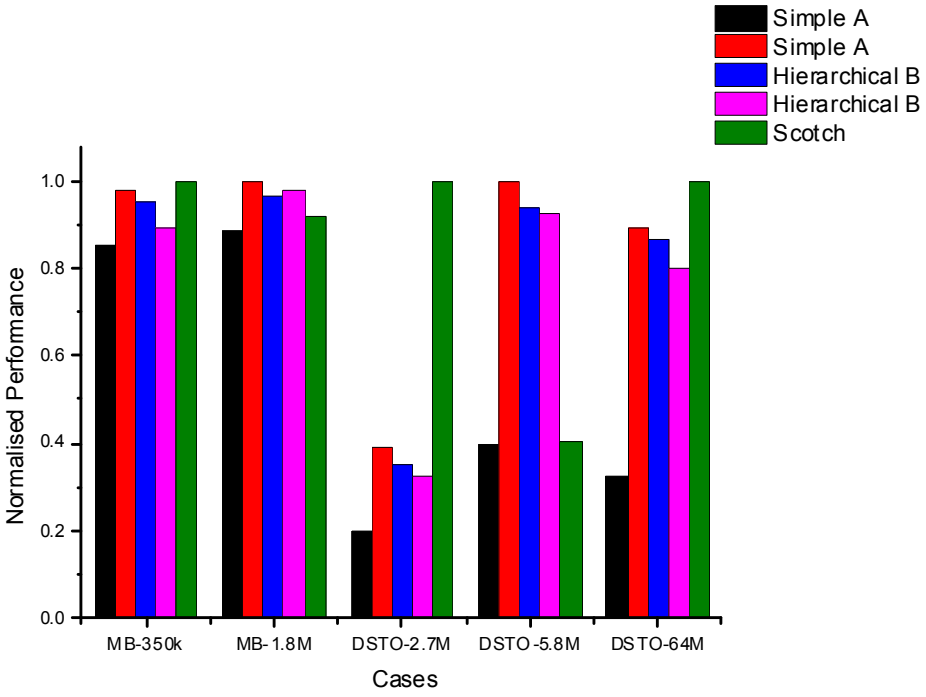

Figure 8 Performance comparison of decomposition methods on 128 core cluster

Since the Scotch decomposition method is optimised to reduce the number of processor faces, and thus reduce the amount of communication required, it could be expected that this would result in higher performance. With the exception of the 5.8 million cell case, the Scotch decomposition cases ran quickly, although not always significantly faster than the hierarchical method.

The variability in performance in these results, particularly for high numbers of parallel processes, suggests that the optimum decomposition method is very dependent on the case and that there may be other factors related to the mesh topology and/or solver that influence the performance. This suggests that for any given case, a worthwhile amount of performance improvement can be gained by testing a few different decomposition methods to determine which provides the best layout for that particular case.

3.3 Case Size Optimisation

It is expected that for any given case, a point will be reached at which the performance gains obtained by increasing the level of parallelisation will plateau. In order to determine where this point lies for different sized cases with different solvers, each of the cases were run at different parallelisation levels on the HPC cluster.

Figure 9 shows the performance of each case, run on 1-8 nodes of the cluster (16 – 128 cores). Since the larger cases show much lower performance as a measure of time steps per

hour, each result has been normalised as a speed-up factor over the performance on a single node.

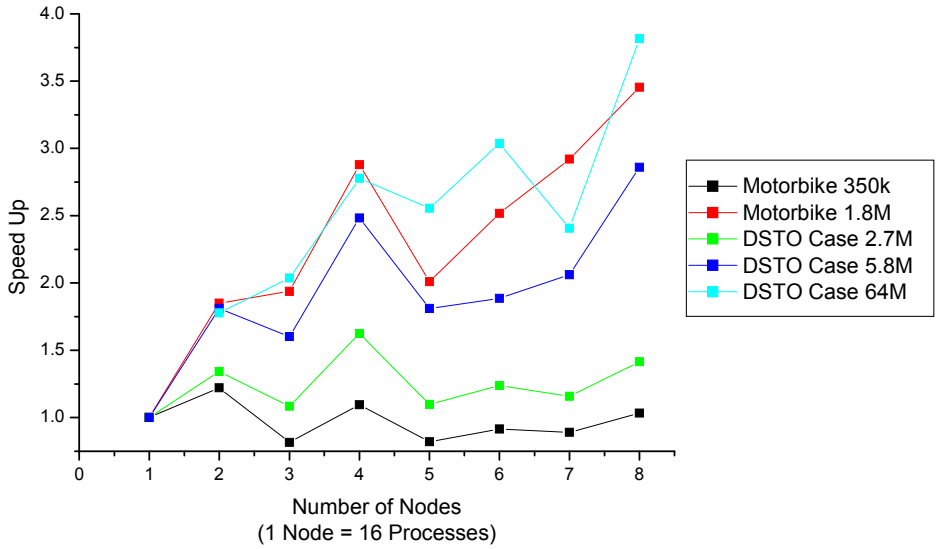

Figure 9 Speedup of each case when run on 1-8 nodes of the HPC cluster

These results show a large amount of variation with extra nodes often decreasing the performance of the computation. This fluctuation is reminiscent of the result seen in Figure 6, suggesting that optimum results are achieved through the parallelisation process when the number of processes is some even multiple of the hardware. If the data from Figure 9 is altered to only show results for numbers of nodes in powers of 2, the results show a trend much closer to the expected increase in performance.

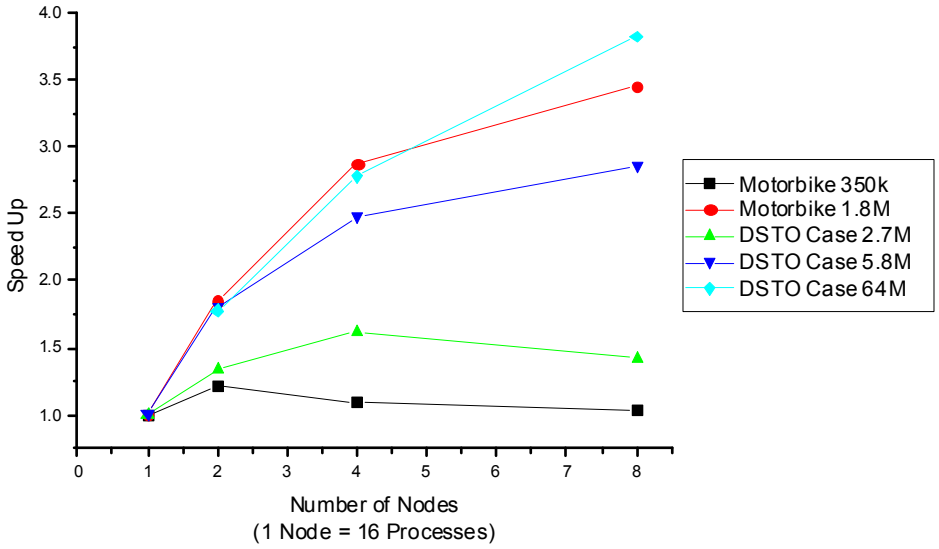

Figure 10 Speedup of each case when run on 1, 2, 4 or 8 nodes of the HPC cluster as per Figure 9

Figure 10 shows a clear performance improvement as the level of parallelisation is increased, with the larger cases scaling in performance up to large numbers of cores. The smaller cases show initial improvement, but then reach a point at which maximum parallelisation speedup has been achieved. Beyond this point, the performance can be seen to decrease with increasing parallelisation. This is a result of increased network communication requirements offsetting the gains achieved in computation speed by adding extra processing power.

Table 4 shows the cell count per core for each case with the best performance highlighted. It is clear that the optimum cell count per core is highly dependent on the solver being used, with the optimum point being approximately 11,000 cells for the simpleFoam solver used in the motorbike tutorial cases, and approximately 43,000 cells for the interFoam solver used for the DSTO cases. This solver dependency is visible in the results shown in Figure 10. Since the optimum cell count per core is much lower for the motorbike case than for the DSTO case, the 1.8 million cell motorbike case scales well up to 8 nodes, while the 2.7 million cell DSTO case reaches its maximum performance at 4 nodes. Due to the size limitations of the cluster, it is not possible to test whether these estimates of optimum cell count per core hold true for larger cases.

Table 4 Average cells per process for up to 128 processes. Best performance highlighted.

Nodes	MB-350k	MB-1.8M	DSTO-2.7M	DSTO-5.8M	DSTO-64M
1	21792	116481	173137	482104	3945990
2	**10896**	58241	86568	241052	1972995
3	7264	38827	57712	160701	1315330
4	5448	29120	**43284**	120526	986498
5	4358	23296	34627	96421	789198
6	3632	19414	28856	80351	657665
7	3113	16640	24734	68872	563713
8	2724	**14560**	21642	**60263**	**493249**

Table 5 shows the parallelisation efficiency for each doubling of the cluster size for each case. The parallelisation efficiency is determined as the slope of the line between the data points in Figure 10.

Table 5 Parallelisation efficiency with increases in cluster size from 1 node to 8 nodes

	Parallelisation Efficiency				
Increase in Nodes	Motorbike 350k	Motorbike 1.8M	DSTO 2.7M	DSTO 5.8M	DSTO 64M
1 - 2	22%	85%	34%	81%	78%
2 - 4	-6%	51%	14%	34%	50%
4 - 8	-2%	14%	-5%	9%	26%

While the larger cases are able to scale more efficiently than the smaller cases, all of the cases showed significant decreases in parallelisation efficiency as the number of nodes used was increased. This suggests that where possible, the smallest number of nodes should be used to run cases so as to maximise efficiency. This is practical when multiple cases are available to be run simultaneously and the cases are small enough that large parallelisation is required in order to complete the simulation in a reasonable timeframe. When only a single case is being run at a given time, the extent of the parallelisation should be tailored to the size of the case. For a large case, the fastest runtimes will generally be achieved by using the maximum available number of cores.

3.4 Proprietary Compiler and MPI Library

3.4.1 Intel MPI process binding and distribution

The Intel MPI library provides a different set of optimisation flags to those used by Open MPI. In order to be sure that the process distribution is optimal, a test was run comparing the available optimisation flags, similar to the test performed for Open MPI. A total of 19 different binding/distribution options were evaluated:

Test Parameters	
Case(s)	Motorbike 1.8 million cells
Hardware	1 Node (1600 MHz RAM)
Processes	16 per node
MPI flags	Intel MPI: Default (no flags) -binding pin=0 -binding pin=1 -binding cell=unit -binding cell=core -binding "cell=unit map=spread" -binding "cell=unit map=scatter" -binding "cell=unit map=bunch" -binding "cell=core map=spread" -binding "cell=core map=scatter" -binding "cell=core map=bunch" -binding map=spread -binding map=scatter -binding map=bunch -binding domain=cell -binding domain=core -binding domain=socket -binding order=compact -binding order=linear

The results in Figure 11 show that while there are many more distribution control options in the Intel MPI, with each providing different levels of performance as the parallelisation is increased, only the option that specifies to turn binding off (red line) causes significant reduction in performance when all cores are being utilised. For every other option, binding on is the default for the Intel MPI. The results also show that while some of the binding and distribution options for the Intel MPI library are sensitive to odd and even numbers of processes, others are less sensitive and some show a performance improvement with odd numbers rather than even. This suggests that the preference for even (or odd) numbers of processes is not a result of the case decomposition (which is identical for all of these options) but it is instead related to how the job distribution is handled by the MPI library.

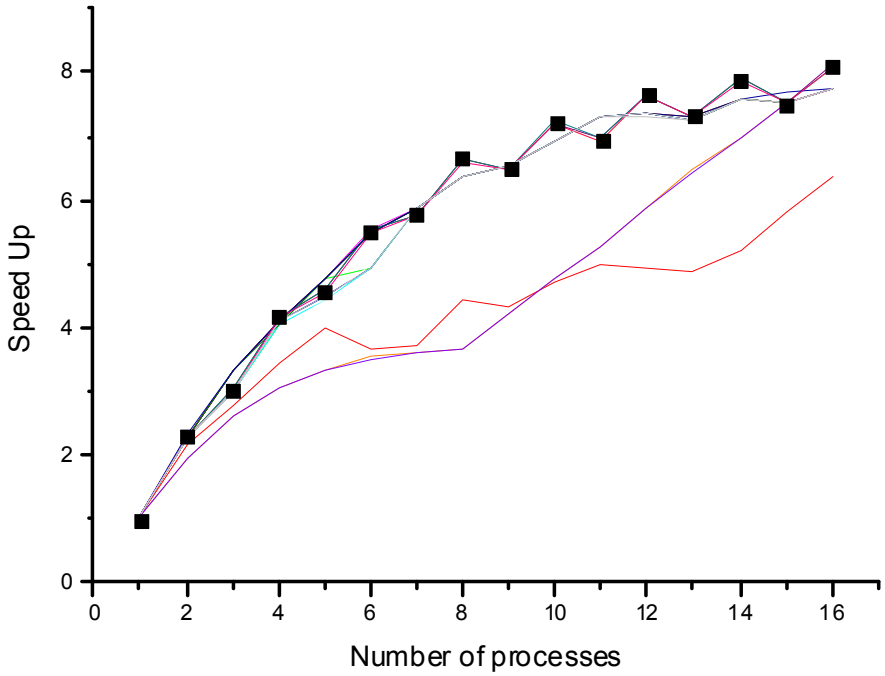

Figure 11 Performance of 19 Intel MPI binding flags in a single workstation. (Legend omitted) The marked data points show the best result, achieved using the flag "cell=unit map=scatter".

On a single workstation the best performance was obtained with the "cell=unit map=scatter" binding flag, represented by the marked data points in Figure 11. however many of the options tested produced very similar performance as can be seen by the close proximity of many of the lines in Figure 11. To test the optimal binding flags to be used on the whole cluster, a subset of the best performing options from the single-node test was run on all the available test cases.

Test Parameters	
Case(s)	Motorbike 350 thousand cells Motorbike 1.8 million cells DSTO Case 2.7 million cells DSTO Case 5.8 million cells DSTO Case 64 million cells
Hardware	8 Nodes (1600 MHz RAM) 1 Gbps Ethernet Networking
Processes	16 per node
MPI flags	Intel MPI: -binding "cell=unit map=spread" -binding "cell=unit map=scatter" -binding "cell=unit map=bunch" -binding "cell=core map=spread" -binding "cell=core map=scatter" -binding "cell=core map=bunch"

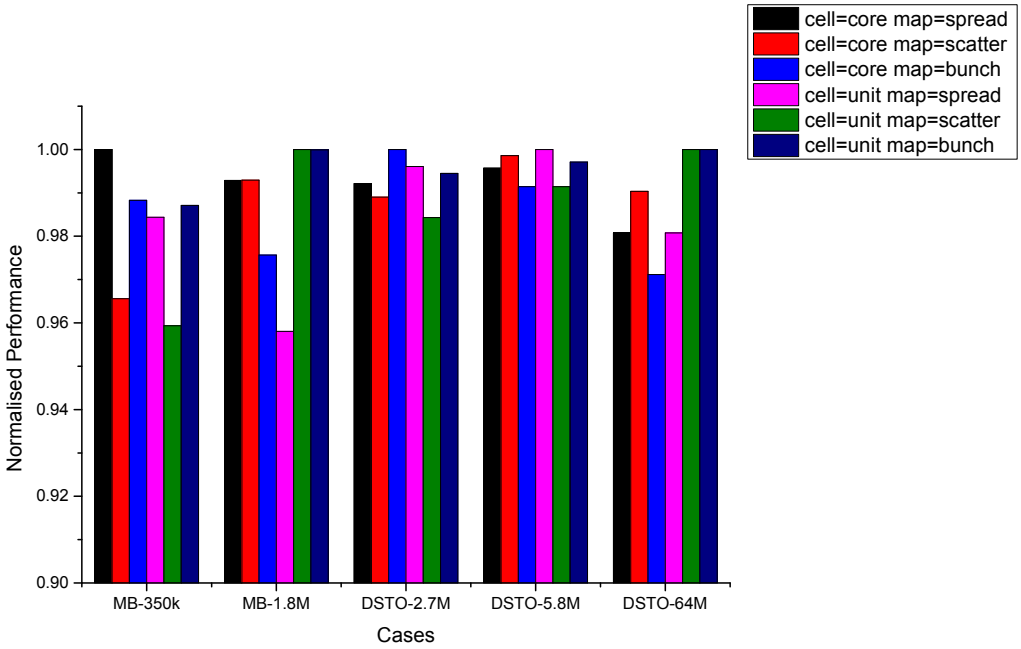

Figure 12 Normalised performance of Intel MPI binding flags run on 128 cores for all cases

Figure 12 shows that across the various test cases, there is no one binding flag that always produces the best performance. Across the results, each of the flags used in this test produced a performance measure within 2-4% of the best result for each case. It is then reasonable to expect that any of these flags could be used to achieve good performance. However, the results also show that across the cases, the "cell=unit map=bunch" option

produced the most consistent performance for all cases. As a result, the "cell=unit map=bunch" flag was used for all subsequent computations using the Intel MPI library.

3.4.2 Intel vs Open Source Performance

The Intel compiler and Intel MPI library can be used together, or in any combination, with the open source variations. As such there are four possible combinations of Intel/open source that can be used to run OpenFOAM.

- Intel compiler and Intel MPI library (ICC-IMPI)
- GCC compiler and Intel MPI library (GCC-IMPI)
- Intel compiler and Open MPI library (ICC-OMPI)
- GCC compiler and Open MPI library (GCC-OMPI)

Each combination was used to run the same five cases as the case size optimisation test. The results in Figure 13 show the performance of the 1.8 million cell motorbike case. The results of all other cases were comparable.

Test Parameters	
Case(s)	Motorbike 350 thousand cells Motorbike 1.8 million cells DSTO Case 2.7 million cells DSTO Case 5.8 million cells DSTO Case 64 million cells (128 cores only)
Hardware	8 Nodes (1600 MHz RAM) 1 Gbps Ethernet Networking
Processes	16 per node
MPI flags	Open MPI: -bind-to-core –bysocket Intel MPI: -binding "cell=unit;map=bunch"

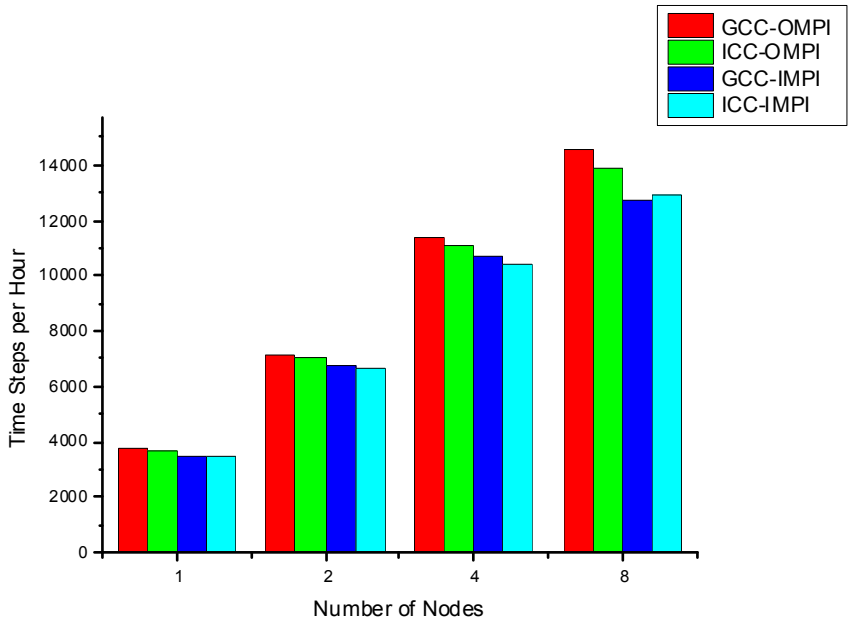

Figure 13 Performance of Open Source vs Intel compiler and MPI Library.

It is possible that the Intel compiler and MPI library may provide some advantages on different hardware or on larger scale clusters, however for the hardware currently being utilised for this study it is clear that maximum performance is achieved when using both the open source options (GCC-OMPI).

3.5 RAM speed

Figure 4 and Figure 5 show the results of the same simulation run on hardware configured with RAM at different clock speeds. It was shown that the memory channel bandwidth was a limiting factor in the speed of the simulation, and that faster memory is able to utilise this bandwidth more effectively to continue realising parallelisation gains once the number of processes on a socket exceeds the number of memory channels. Figure 14 shows an overlay of the results directly comparing the simulation performance with RAM speed. It is clear that increasing the speed of the RAM provides significant speed improvement in the calculation, especially at higher levels of parallelisation. For 8 parallel processes, the 1600 MHz RAM allows 42% faster simulations than the 1333 MHz RAM, while at 16 parallel processes the difference is 67%. This extra 25% performance increase is enabled by the higher memory bandwidth of the faster RAM.

Test Parameters	
Case(s)	Motorbike 1.8 million cells
Hardware	1 Node (512 GB - 1333 MHz RAM) 1 Node (64 GB - 1600 MHz RAM)
Processes	Up to 16
MPI flags	-bind-to-core -bysocket

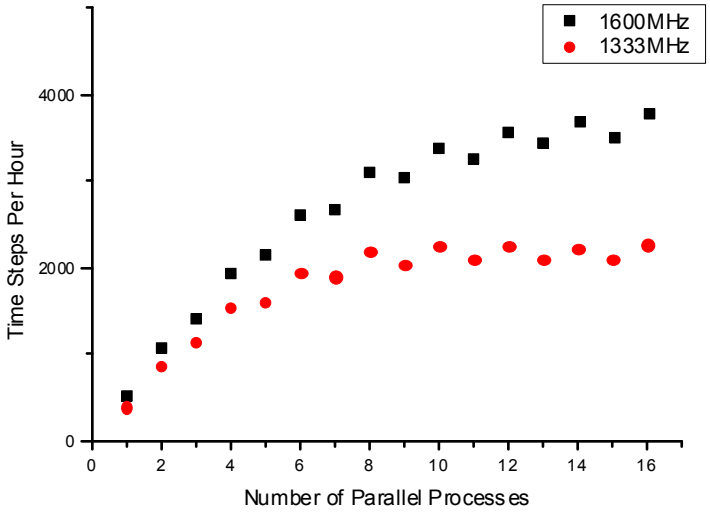

Figure 14 Comparison of performance with RAM speed

In the case where a job is being run on a single workstation, knowing that faster RAM produces faster simulations does not provide a significant advantage beyond being able to make informed purchasing decisions when procuring more hardware.

For large scale parallelisation across multiple workstations this information is very important. CFD calculations are run as a series of iterations/time steps each of which must be fully completed before the next can begin. In large scale parallelisation, having one node running at a slower speed would result in the faster nodes completing their calculations and then having to wait for the slower node to finish before proceeding to the next calculation. This ultimately means that a large cluster will only run at the speed of its slowest component. It is important then, to be aware of any differences between individual pieces of hardware in the cluster and ensure that where possible, only like hardware is used to run large scale parallel computations. OpenFOAM provides load balancing tools that can be used to ensure that mixed hardware is fully utilised, but the use and effectiveness of these tools is beyond the scope of this investigation.

3.6 Turbo Boost

Since the turbo function of the Xeon CPUs is nominally activated only when some of the cores on the CPU are idle, it is expected that the increased clock speed achieved by this function will only improve the simulation speed at low levels of parallelisation and so would not provide any advantage for the normal use case where all available cores are being utilised. Figure 15 shows the comparison between a job run with the turbo function enabled and disabled. Figure 16 shows the same results presented as the percentage improvement in performance with turbo enabled.

Test Parameters	
Case(s)	Motorbike 1.8 million cells
Hardware	1 Node (1600 MHz RAM) - Turbo Enabled - Turbo Disabled
Processes	Up to 16
MPI flags	-bind-to-core -bysocket

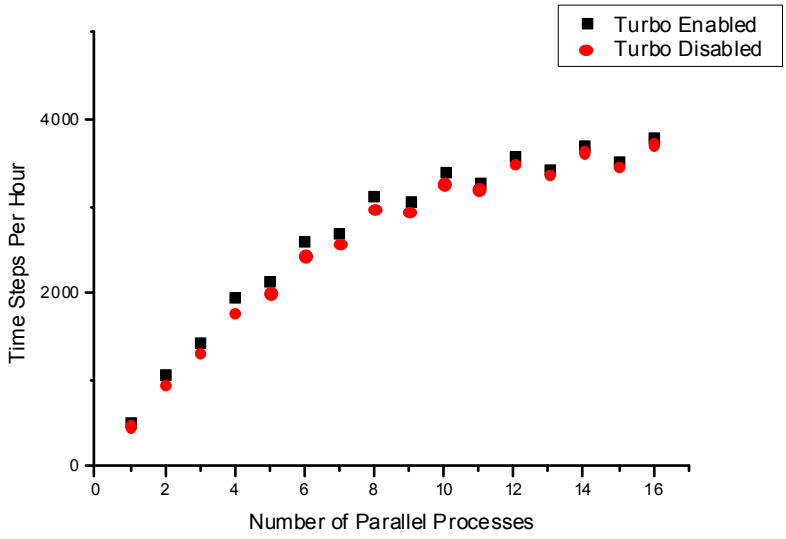

Figure 15 Comparison of performance with and without Intel Turbo Boost functionality

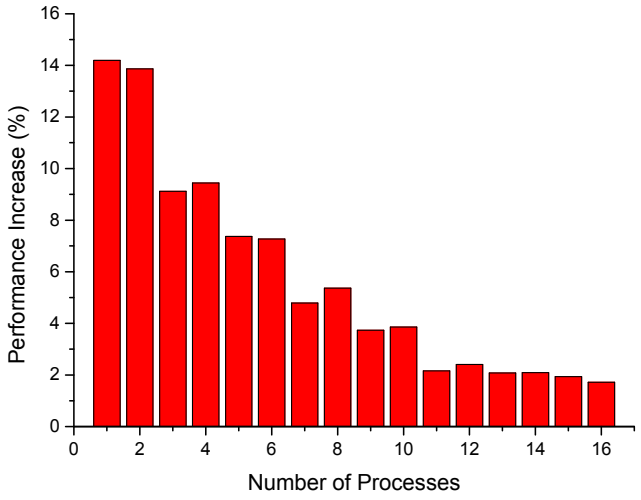

Figure 16 Percentage performance increase from enabling the turbo function on a single node

As expected, the greatest improvement from the turbo function was realised at low levels of parallelisation and so is not useful for practical purposes. However, the results showed that even at full utilisation, a small performance improvement (~2%) was seen when the function was enabled. Since there is no penalty for using the turbo boost, it is clearly best to always enable the capability, even when the expected improvements are small.

3.7 Hyper-threading

Unlike the turbo boost function, which can be utilised with no penalty, taking advantage of the hyper-threading capability of the Xeon CPUs requires increasing the level of parallelisation, which is known to increase overhead in CFD applications. Since the increase in parallelisation is not accompanied by an increase in memory bandwidth, it could reasonably be expected that little or no improvement would be achieved by assigning multiple processes to each physical core.

When attempting to analyse the performance of a parallel simulation run with 16 processes on one node, compared with 32 processes on the same node using hyper-threading, it was found that the system would not bind processes to a virtual core. If a case was run on greater than 16 cores with binding enabled it would fail to start. Figure 17 shows results of testing up to 32 processes, with hyper-threading enabled and binding disabled as well as the results up to 16 processes with binding enabled and hyper-threading both enabled and disabled.

Test Parameters	
Case(s)	Motorbike 1.8 million cells
Hardware	1 Node (1600 MHz RAM) - Hyper-threading enabled, binding enabled - Hyper-threading enabled, binding disabled - Hyper-threading disabled, binding enabled
Processes	Up to 16 with binding enabled Up to 32 with hyper-threading enabled, binding disabled
MPI flags	-bind-to-core –bysocket (binding enabled) -bysocket (binding disabled)

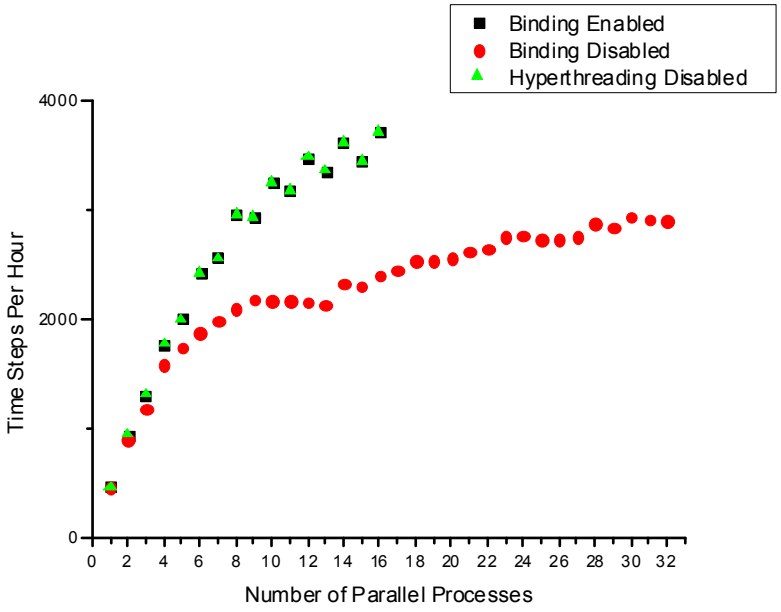

Figure 17 Comparison of standard parallelisation against 2x processes using hyper-threading

While the increase in CPU utilisation provided by the hyper-threading function does seem to improve performance of the system, it is apparent that the best performance achieved from using all the additional virtual cores (with binding disabled), is less than that of using only the real cores with binding. The results for up to 16 processes with binding enabled were identical, regardless of whether hyper-threading was enabled or disabled. This is to be expected, since running on 16 or less cores means that the virtual cores provided by the hyper-threading function are not utilised. As a result, it is unimportant whether hyper-threading is enabled or disabled for the speed of the simulation. In the interest of system robustness, it is worth disabling the function as this prevents the user from

mistakenly attempting to bind processes to virtual cores which would result in the simulation failing to run.

4. Conclusion

For optimum use of the EMSMA group's Intel Xeon based HPC cluster, the results of this investigation have shown that it is best to use a version of OpenFOAM that has been compiled with the open source compiler (GCC) and MPI library (Open MPI). Cases should be decomposed to run in parallel using either the hierarchical or Scotch decomposition methods depending on the case and level of parallelisation, then run using core binding and "by socket" process distribution on 16 processes per node and a maximum number of nodes dependent on the size of the case and the solver being utilised. Hardware containing the fastest available RAM should be used as a preference and clustering hardware with different performance levels should be avoided. Where possible, multiple cases should be run simultaneously using fewer nodes per case.

5. Future Work

The results comparing decomposition methods suggest that the optimum method varies on a case by case basis. Further investigation into which methods affect the performance in what way would provide a greater understanding of how the differences between the methods affects the performance and so which methods would likely be best to use in any given scenario.

The greatest bottleneck encountered in attempting to increase the computation speed through parallelisation is the limitation in the speed of the networking equipment. Upgrading the network fabric to a faster technology (such as 10 Gbps Ethernet or Infiniband) would allow the realisation of much higher speed-up using the number of CPUs currently available, as well as allowing further upscaling of the size of the cluster. At the time of writing, Infiniband hardware was being procured to allow for fast networking. After installing this hardware, the case size optimisation done in this report will need to be repeated in order to determine the optimum cluster size for any particular case. It is also possible that faster networking hardware and/or a larger cluster size may allow the proprietary Intel compiler and MPI libraries to realise some speed up over the open source alternatives.

In the case where upgrades to the cluster introduced a mismatch in hardware capability, it would be necessary to perform investigations into the load balancing tools available with the OpenFOAM package in order to ensure that large cases can be run on all the available hardware optimally.

6. References

1. *OpenFOAM - The Open Source Computational Fluid Dynamics (CFD) Toolbox*. 2014 [Accessed on 09/04/2014]; Available from: www.openfoam.com.
2. Colwell, B., *The Zen of Overclocking*. Computer, 2004. **37**(3): p. 9-12.
3. *mpirun(1) man page (version 1.5.5)*. 2012 [Accessed on 13/3/2014]; Available from: http://www.open-mpi.org/doc/v1.5/man1/mpirun.1.php.
4. *Running applications in parallel*. 2014 [Accessed on 13/3/2014]; Available from: http://www.openfoam.org/docs/user/running-applications-parallel.php.
5. *OpenFOAM Performance Benchmark and Profiling*. 2013 [Accessed on 13/3/2014]; Available from: http://www.hpcadvisorycouncil.com/pdf/OpenFOAM_Analysis_and_Profiling_Intel_2680_FCA.pdf.

This page is intentionally blank

Appendix A: Table of completed simulations

Hardware	Compiler	MPI	decomposition method	Cases	Processes	MPI flags
1 node, 1333 MHz RAM 1 node, 1600 MHz RAM	GNU	Open	Scotch	Motorbike 350k Motorbike 1.8M DSTO 2.7M	1-16	-bind-to-core (none) -bind-to-core -bysocket -bysocket
1 node, 1333 MHz RAM (turbo disabled) 1 node, 1600 MHz RAM (turbo disabled)	GNU	Open	Scotch	Motorbike 350k Motorbike 1.8M	1-16	-bind-to-core -bind-to-core -bysocket
1 node, 1333 MHz RAM (hyperthreading enabled) 1 node, 1600 MHz RAM (hyperthreading enabled)	GNU	Open	Scotch	Motorbike 1.8M	1-32	-bind-to-core -bysocket -bind-to-core
1 node, 1333 MHz RAM 1 node, 1600 MHz RAM	GNU	Open	Scotch	Motorbike 350k Motorbike 1.8M	1-16	-bind-to-core -bind-to-core -bysocket -bind-to-socket -bind-to-socket -bysocket (none)
1 node, 1333 MHz RAM	GNU	Open	Scotch	Motorbike 350k	1-16	-bind-to-core -bysocket -bind-to-core -npersocket 4

4 nodes, 1600 MHz RAM	GNU	Open	Scotch	Motorbike 1.8M	1-32	-bind-to-core -bysocket -npersocket 4 -bind-to-core -bynode -npersocket 4
8 nodes, 1600 MHz RAM	GNU	Open	Scotch	Motorbike 1.8M DSTO 2.7M	1-128	-bind-to-core -bycore -bind-to-core -bysocket -bind-to-core -bynode -bind-to-core -bynode -bysocket
8 nodes, 1600 MHz RAM	GNU	Open	Scotch	DSTO 64M	8 per node, 1-8 nodes	-bind-to-core -bysocket
8 nodes, 1600 MHz RAM	GNU	Open	Scotch	Motorbike 350k Motorbike 1.8M DSTO 2.7M DSTO 5.8M DSTO 64M	16 per node, 1-8 nodes	-bind-to-core -bysocket
4 nodes, 1600 MHz RAM	GNU	Open	Scotch	DSTO 2.7M	1-64	-bind-to-core -bysocket
4 nodes, 1600 MHz RAM	GNU	Open	Scotch	Motorbike 1.8M	1-64	-bind-to-core -bysocket **(**hostfile specifies 1 core per node**)** -bind-to-core -bysocket (hostfile specifies 8 cores per node) -bind-to-core -bysocket **(**hostfile specifies 16 cores per node**)**

1 node, 1333 MHz RAM 1 node, 1600 MHz RAM	Intel	Intel	Scotch	Motorbike 350k Motorbike 1.8M DSTO 2.7M	1-16	(none) -binding pin=0 -binding pin=1 -binding cell=unit -binding cell=core -binding "cell=unit;map=spread" -binding "cell=unit;map=scatter" -binding "cell=unit;map=bunch" -binding "cell=core;map=spread" -binding "cell=core;map=scatter" -binding "cell=core;map=bunch" -binding map=spread -binding map=scatter -binding map=bunch -binding domain=cell -binding domain=core -binding domain=socket -binding order=compact -binding order=linear

4 nodes, 1600 MHz RAM	Intel	Intel	Scotch	DSTO 2.7M	1-64	-binding "cell=unit;map=spread" -binding "cell=unit;map=scatter" -binding "cell=unit;map=bunch" -binding "cell=core;map=spread" -binding "cell=core;map=scatter" -binding "cell=core;map=bunch" -binding domain=socket -binding domain=cell -binding domain=core -binding order=compact -binding order=scatter -binding "domain=socket;order=scatter" -binding "domain=cell;order=scatter" -binding "cell=core;map=bunch;domain=core" -binding "cell=core;map=bunch;domain=core;order=compact" -binding "cell=core;map=bunch;domain=core;order=scatter" -binding "cell=core;map=bunch;domain=node" -binding "cell=core;map=bunch;domain=node;order=compact" -binding "cell=core;map=bunch;domain=node;order=scatter"
8 nodes, 1600 MHz RAM	GNU Intel	Open Intel	Scotch	Motorbike 350k Motorbike 1.8M DSTO 2.7M DSTO 5.8M DSTO 64M	16 per node, 1-8 nodes	-bind-to-core -bysocket -binding "cell=core;map=spread" -binding "cell=core;map=scatter" -binding "cell=core;map=bunch" -binding "cell=unit;map=spread" -binding "cell=unit;map=scatter" -binding "cell=unit;map=bunch"

1 node, 1600 MHz RAM	GNU	Open	Simple A	Motorbike 350k	16	-bind-to-core -bysocket
8 nodes, 1600 MHz RAM			Simple B	Motorbike 1.8M	128	
			Hierarchical A	DSTO 2.7M		
			Hierarchical B	DSTO 5.8M		
			Scotch	DSTO 64M		

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION
DOCUMENT CONTROL DATA

1. DLM/CAVEAT (OF DOCUMENT)

2. TITLE

Optimising the Parallelisation of OpenFOAM Simulations

3. SECURITY CLASSIFICATION (FOR UNCLASSIFIED REPORTS THAT ARE LIMITED RELEASE USE (L) NEXT TO DOCUMENT CLASSIFICATION)

Document	(U)
Title	(U)
Abstract	(U)

4. AUTHOR(S)

Shannon Keough

5. CORPORATE AUTHOR

DSTO Defence Science and Technology Organisation
506 Lorimer St
Fishermans Bend Victoria 3207 Australia

6a. DSTO NUMBER	6b. AR NUMBER	6c. TYPE OF REPORT	7. DOCUMENT DATE
DSTO-TR-2987	AR-015-993	Technical Report	June 2014

8. FILE NUMBER	9. TASK NUMBER	10. TASK SPONSOR	11. NO. OF PAGES	12. NO. OF REFERENCES
2014/1103169/1	CDG 07/298	Head Future Submarine Program	35	5

13. DSTO Publications Repository

http://dspace.dsto.defence.gov.au/dspace/

14. RELEASE AUTHORITY

Chief, Maritime Division

15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT

Approved for public release

OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SA 5111

16. DELIBERATE ANNOUNCEMENT

No Limitations

17. CITATION IN OTHER DOCUMENTS Yes

18. DSTO RESEARCH LIBRARY THESAURUS

CFD, OpenFOAM, parallel processing, Computational fluid dynamics

19. ABSTRACT

The OpenFOAM computational fluid dynamics toolbox allows parallel computation of fluid dynamics simulations on a theoretically unlimited number of processor cores. The software utilises a message passing interface (MPI) library in order to facilitate communication between parallel processes as the computation is completed. In order to maximise the parallel efficiency of the computation, the way in which the MPI application distributes the problem among the processors needs to be optimised. This report examines the performance of a given high performance computing cluster with several OpenFOAM cases, running using a combination of MPI libraries and corresponding MPI flags. The effect of hardware variations on the speed of the computation is also briefly investigated. The results show a noticeable improvement in simulation time when certain flags are given to the MPI library in order to control the binding and distribution of processes amongst the cluster.