How to make best use of the AMD Interlagos processor

Numerical Algorithms Group Ltd.

November 2011

Contents

Introduction

The Interlagos (IL) processor is AMD's new 16-core Opteron processor that will be used by HECToR during phase 3. It based on the 32nm Bulldozer/AMD Family 15h architecture in which cores share more resources than in previous 45nm AMD Family 10h designs such as the 12-core Magny Cours (MC) processor, but fewer than Intel's hyper-threading processors such as the 45nm 12-core Westmere (WM) design. This whitepaper gives a description of the architecture of an Interlagos processor, presents some benchmark performance results and analyses the results in the context of the architecture description, comparing results with the previous generation AMD and Intel processors Magny-Cours (MC) and Westmere (WM).

The Architecture of an Interlagos Processor

The most obvious difference in the design of IL compared with MC is that cores are paired into "modules" which share a significant proportion of resources. Figure 1 gives an overview of which resources are shared between the two cores of a module.

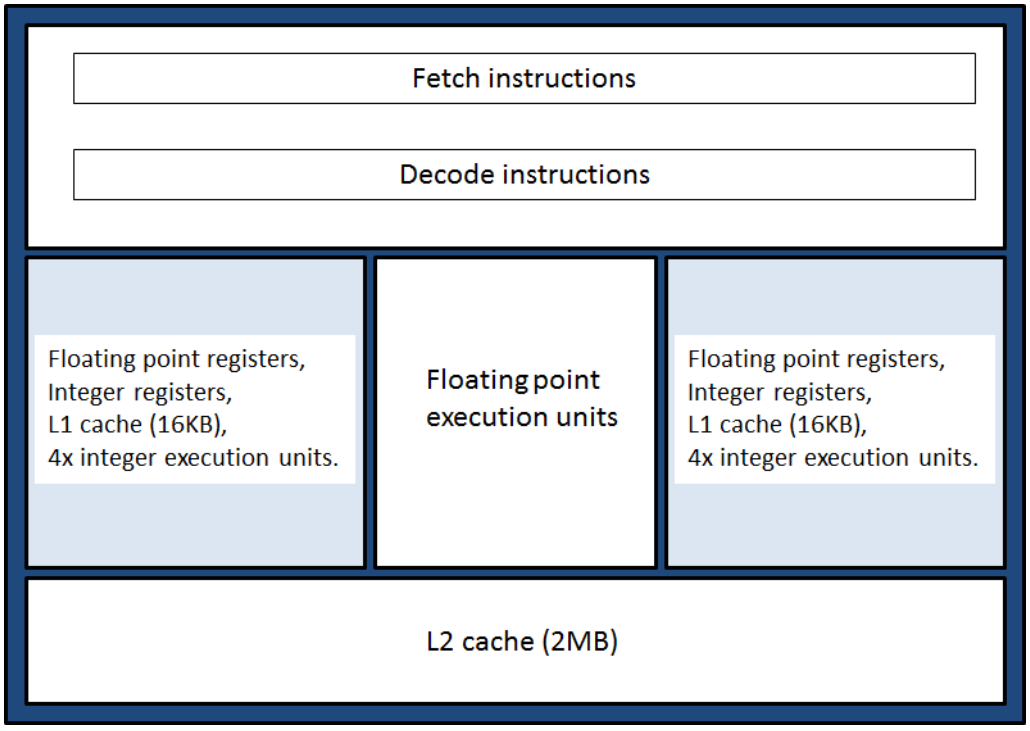


Figure 1 ***An Interlagos module. A module consists of two cores (shaded in light blue) which have independent L1 data cache, registers and execution units for integer operations. Shared resources include L2 data cache, floating point execution units, and the hardware used to fetch and decode instructions.***

This contrasts with adjacent cores in a MC processor which share resources from L2 cache, and virtual cores in a WM processor when hyper-threading, which share all resources shown in Figure 1 except control, general purpose and floating point registers. Thus the design of an IL module can be seen as halfway between the full separation of resources in a MC processor and the full sharing of resources in a WM processor. It can also be seen as a first step by AMD in the evolution of divorcing the FPU from the integer cores of a CPU, a step further towards their Fusion processor designs in which floating point calculations are performed by a co-processing unit that is separate to the CPU, but not as loosely coupled as a GPU.

Looking at a module in more detail, Figure 2 shows that the shared floating point unit consists of two 128-bit pipelines which are capable of grouping together as a single 256-bit pipeline. Thus each floating point unit is capable of executing two 128-bit SSE vector instructions or one 256-bit AVX add or multiply vector instruction per clock cycle. On top of SSE and AVX instructions there is a special 256-bit fused multiply-add instruction which effectively doubles the theoretical performance of the floating point units, and also has the benefit of improved accuracy compared with separate multiply followed by add instructions, because the multiplication is not rounded before being used as input to the addition operation. Compilers will try to use this instruction where possible when vectorizing loops. It is important to realise two things about the new AVX and fused multiply-add instructions:

- A single 256-bit AVX instruction has the same throughput as two 128-bit SSE instructions operating in parallel: 4 double precision FLOPS per cycle, or 8 single precision FLOPS per cycle, which is the same as in a Magny-Cours FPU. However, one of the advantages of AVX instructions is that they are non-destructive, i.e. the result does not overwrite the contents of one of the input registers, because AVX instructions take a third register argument for the result. This gives compilers the opportunity to improve performance by eliminating move operations that would have to happen with SSE instructions.
- The theoretical performance improvement brought about by the fused multiply-add instruction (8 double FLOPS per cycle or 16 single FLOPS per cycle) will not translate into double the performance of a MC processor for because there are 8 floating point units per IL processor compared with 12 per MC processor.

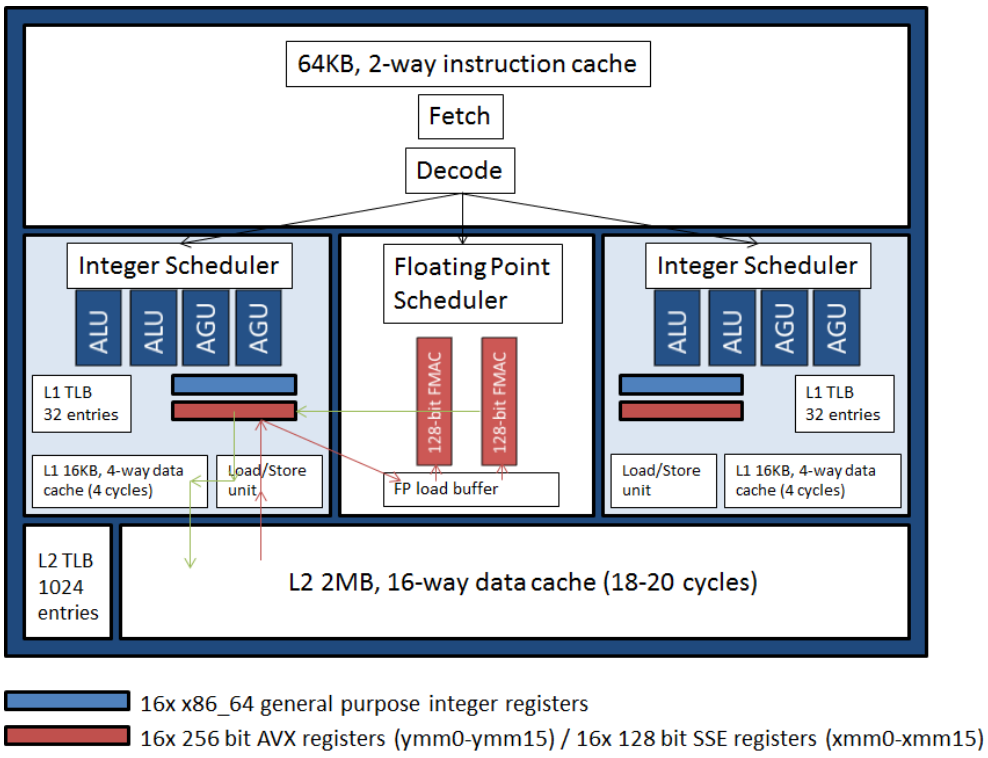


Figure 2 ***Detailed view of an Interlagos module. The red and green arrows give an example of the movement of data. Red arrows show some operands being loaded into registers from L2 cache before being dispatched into the FMAC (floating point multiply/accumulate) execution units with the result (green arrows) being written back into a register before being placed in L1 and L2 cache (L1 is a write-through cache).***

An Interlagos processor consists of two dies connected via HyperTransport interconnection links. Each die consists of 4 modules which share 6MB L3 data cache (plus 2MB which is reserved in hardware for maintaining cache coherency) plus local memory. This is illustrated in Figure 3. The two dies in a processor share memory address space, communicating via the HyperTransport links, in a Non-Uniform Memory Architecture (NUMA) where it is much quicker to access local memory than it is to access memory attached to the other die. This is illustrated in Figure 4.

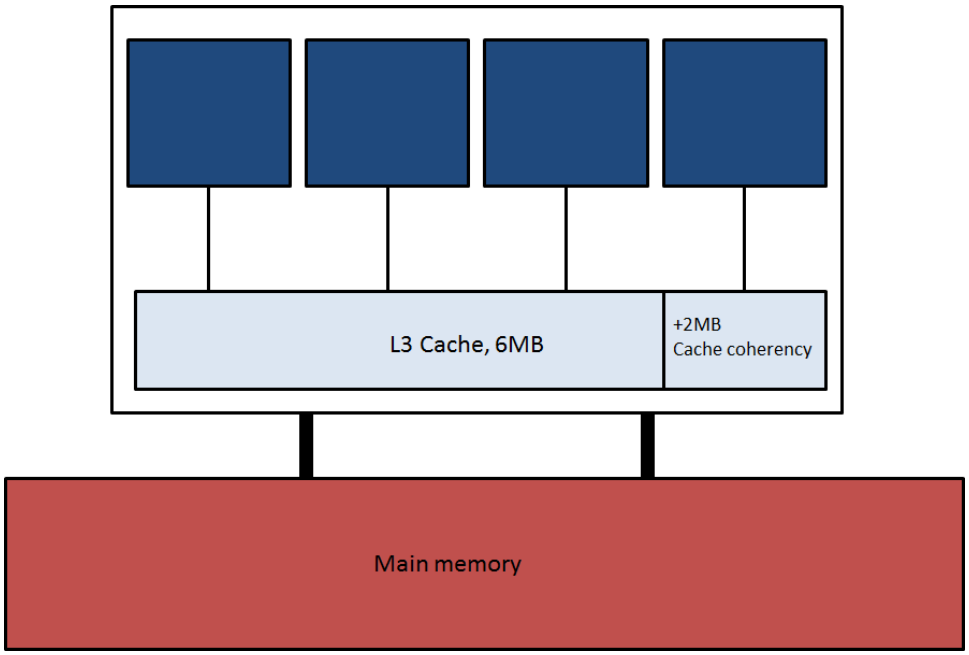


Figure 3 *An Interlagos die. Each blue square represents a module, containing 2 cores.*

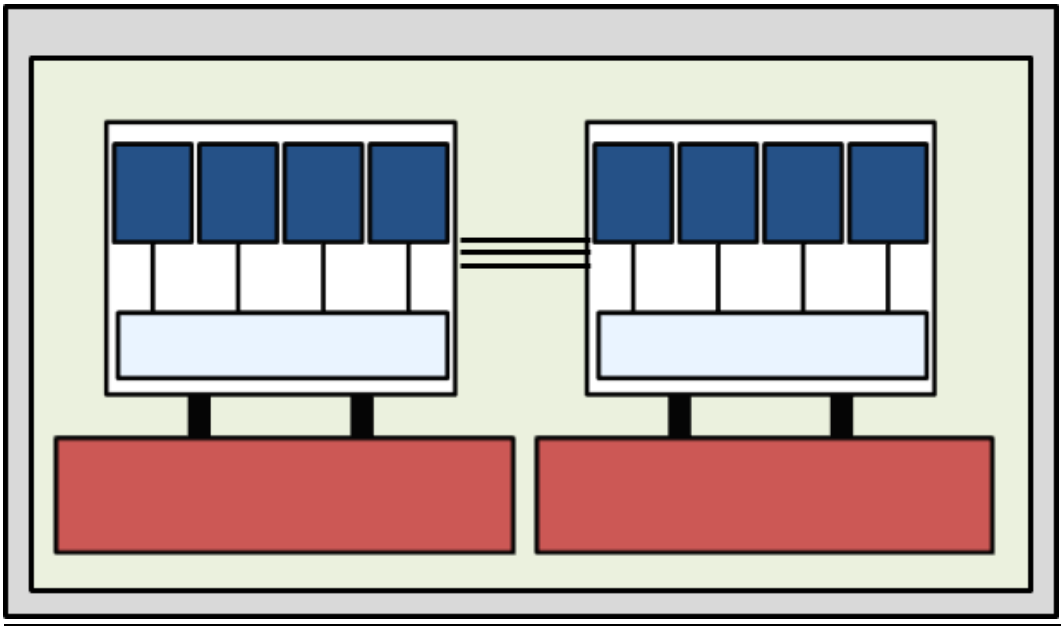


Figure 4 *An Interlagos processor consists of two dies connected via HyperTransport links in a NUMA.*

External HyperTransport links allow for the construction of NUMA *nodes* by connecting to a second processor, illustrated in Figure 5. For example this is the architecture of a HECToR node.

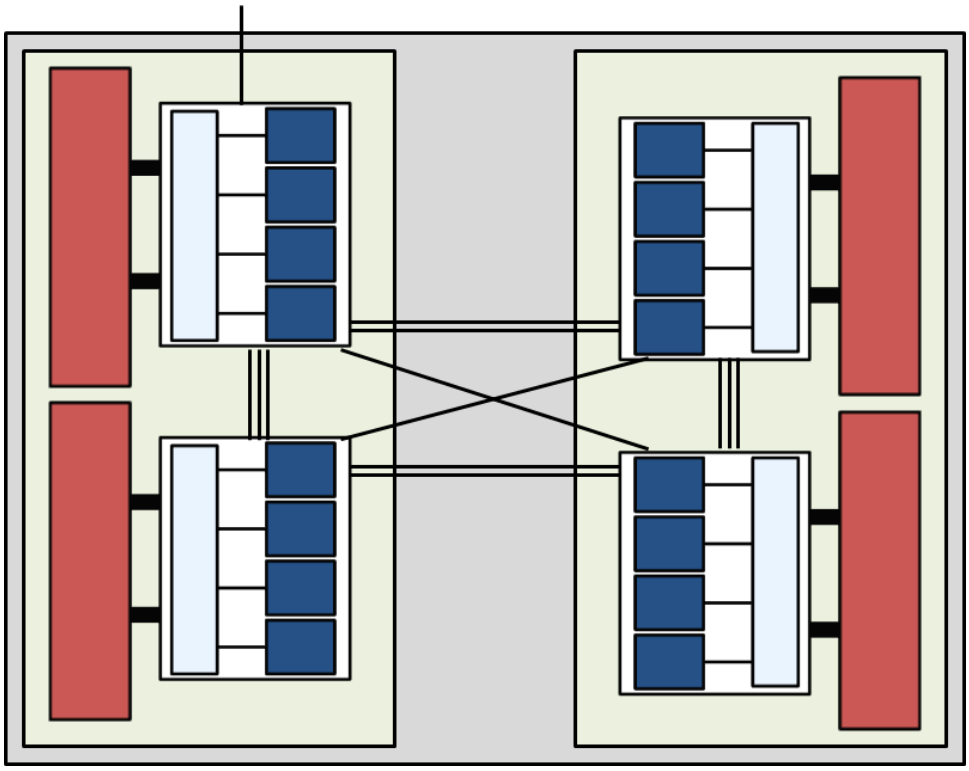


Figure 5 ***A NUMA node consisting two Interlagos processors, with bandwidth between the dies (white) being in the ratio of 3:2:1 depending on which die data is coming from/going to, as indicated by the number of inter-die connections.***

Test Systems

Interlagos (IL)

The Interlagos system used in the following tests is the HECToR Cray XE6 Test and Development System (TDS), which consists of 78 of the nodes described in Figure 5. The processors run at 2.3GHz, thus giving a peak performance of:

2.3 (GHz) * 8 (FLOPS) = **18.4 GFLOPS/s/core** * 16 (FPU) **= 294.4 GFLOPS/s/node**

Recall that 8 FLOPS per clock cycle is based on use of the new Fused Multiply-Add instruction (FMA) in double precision. Each of the 4 dies in a node has access to 8GB DDR3 @ 1333MHz local memory, thus providing 32GB shared memory across the node.

Magny-Cours (MC)

The Magny-Cours system used is the phase 2b main HECToR Cray XE6 system, which has 1856 nodes each containing 2 12-core processors in the same NUMA topology described in Figure 5 for the Interlagos processor. The processors run at 2.1 GHz and operate in superscalar mode, returning 2 double precision multiplication and addition results per clock cycle. The theoretical peak performance is therefore:

2.1 (GHz) * 4 (FLOPS) = **8.4 GFLOPS/s/core** * 24 (FPUs) = **201.6 GFLOPS/s/node**

Each of the 4 dies in a node has access to 8GB DDR3 @ 1333MHz local memory, thus providing 32GB shared memory across the node. Recall that each core in a Magny-Cours processor shares only L2 cache with its neighbours in the same die.

Westmere (WM)

The Westmere system comprises two 2.93GHz, 6-core Xeon processors which support hyper-threading, giving 24 virtual cores across the node. Each processor has access to 12GB DDR3 @ 1333MHz local memory, shared across the node via QuickPath interconnect links (Intel's equivalent of HyperTransport). The theoretical peak performance is:

2.93 (GHz) * 4 (FLOPS) = **11.7 GFLOPS/s/core** * 12 (FPUs) **= 140.6 GFLOPS/s/node**

Recall that each core in a Westmere processor supports in hardware 2 processes/threads, sharing all resources except registers, thus providing 2 virtual cores.

Benchmarks

From the architecture of an Interlagos processor outlined above a number of questions arise:

Q1. How does a tuned, computationally intensive kernel benefit from the introduction of the new AVX and FMA (Fused Multiply-Add) instructions? Is the benefit anywhere near 2x for a serial program compared with MC?
Q2. What are the effects on numerical routines of sharing resources within a module (floating point unit, instruction front-end, L2 cache and most of TLB)? How do the results compare with similar processors in which (a) fewer resources are shared – i.e. MC, and (b) more resources are shared – i.e. WM.
Q3. What is the effect of increased contention on memory bandwidth due to the extra 2 cores per die compared with MC?

To address these questions two benchmark tests have been chosen to look at the effects on (a) compute-bound floating point performance, and (b) memory-bound integer performance. DGEMM is an obvious choice for (a) since it is a compute-bound BLAS level 3 routine which is able to take advantage of new the Fused Multiply Add instruction of the Interlagos processor, and highly tuned OpenMP implementations exist from vendor libraries. Mersenne Twister is a good choice for (b) since the kernel of the problem performs non-vectorizable integer operations that will utilize the separate integer execution units in an Interlagos module, and the NAG library for SMP and multi-core contains an OpenMP implementation.

DGEMM Benchmarks

The following simple test program was used for the DGEMM tests:

```
program matmul_test
 use mpi
 use timers
 implicit none
 integer :: n
 double precision, allocatable, dimension(:,:) :: a,b,c
 integer :: ierr,myrank
 call mpi_init(ierr)
 call mpi_comm_rank(mpi_comm_world,myrank,ierr)
 if(myrank.eq.0) read(5,*) n
 call mpi_bcast(n,1,mpi_integer,0,mpi_comm_world,ierr)
 allocate(a(n,n),b(n,n),c(n,n))
 a(:,:) = 0.d0
 call random_number(b)
 call random_number(c)
 call start_timer(1)
 call dgemm('n','n',n,n,n,1.d0,a,n,b,n,0.d0,c,n)
 call end_timer(1)
 call print_timers()
 call mpi_finalize(ierr)
end program matmul_test
```

This program performs a number of independent calls to DGEMM in parallel. The timing module reports the maximum, minimum and mean times across all processors. On each test machine this program was compiled with and linked to the most appropriate compiler and vendor library, which were Cray 7.4.0, libsci 11.0 for IL and MC, and ifort 12.0, MKL 10.3 for WM. In both cases these libraries contain separate multi-threaded versions of DGEMM, controllable using the OMP_NUM_THREADS environment variable.

For each machine 3 types of performance scaling was investigated:

(a) The scalability of multithreading when the node is fully saturated.

(b) Scalability of multithreading using a single process.

(c) Scalability of multiprocessing.

For (a) and (b) a value of OMP_NUM_THREADS of 1, 2, 3, 4, 8, 16 and 32 was used for IL and 1, 2, 3, 4, 6, 12, 24 for MC and WM. In test (a) the number of MPI processes, NPROCS, was selected such that NPROCS*OMP_NUM_THREADS = 32[1] in order to saturate the node, and the mean time over NPROCS was taken as the result. The average time over NPROCS was also taken as the result in test (c). The results relevant to answering questions Q1-Q3 are presented in this paper.

The baseline results are taken from using a single process linked to the unthreaded version of the library. The baseline results for each processor are given in Table 1.

[1] For OMP_NUM_PROC=3, NPROC=10 was used for IL.

N	IL (Secs.,GFLOPS/s)	MC	WM
1,000	0.156, 12.9	0.262, 7.7	0.181, 11.1
2,000	1.211, 13.3	2.043, 7.9	1.433, 11.2
3,000	4.093, 13.3	6.868, 7.9	4.816, 11.3
4,000	9.630, 13.4	16.227, 7.9	11.395, 11.3

Table 1. *Baseline results for DGEMM. Note that IL achieves 73% theoretical peak performance (TPP) of a core, MC 94% and WM 97%. Even though the IL implementation achieves a smaller percentage of peak and has a slower clock speed than WM it is still the fastest due to the new fused multiply-add instruction.*

Table 1 answers question Q1. Raw performance is 1.7x that seen for MC, which although not quite 2x, is nevertheless impressive considering that compilers' abilities to generate optimal AVX instructions is probably immature compared with their abilities to generate SSE instructions. It is reasonable to expect the speedup to tend towards 2x for later compiler releases. The greater performance for WM compared to MC can be attributed to the faster clock speed of that processor, and that the MKL implementation appears better tuned to WM than Cray's libsci implementation is for MC/IL, achieving a higher percentage of TPP. Even so, IL outperforms WM.

Figures 6-9 show for N=4,000 scaling charts for runs performed on each test machine normalised against the respective baseline times given in Table 1.

In all scaling graphs the y-axis value is calculated as:

```
baseline_time/(OMP_NUM_THREADS*threaded_time),
```

thus a value of 1 indicates perfect scaling and a value < 1 indicates both the effects of overheads in the parallel algorithm (which is assumed to be roughly the same across both DGEMM implementations used) and, more importantly, contention on resources.

Figures 6 and 7 answer the first part of question Q2, and show that there is a significant effect on performance due to threads sharing module-level resources. In the case of DGEMM the loss of efficiency is around 40%.

For the second part of question Q2 the results shown in Figures 8 and 9 indicate what could perhaps have been expected from the architectures: scalability on MC does not suffer much compared with a single process due to the complete separation of cores down to L2 cache. On WM scalability suffers greatly[2]. There is a 70% loss of performance when using the whole virtual node compared with a single process due to the complete sharing by 2 processes or threads of each core up to registers when hyper-threading. The IL processor is somewhere in between these results, losing 40% performance due to the partial sharing of resources.

[2] It is also worth noting that the WM results showed by far the greatest variance between the max, min and mean process times.

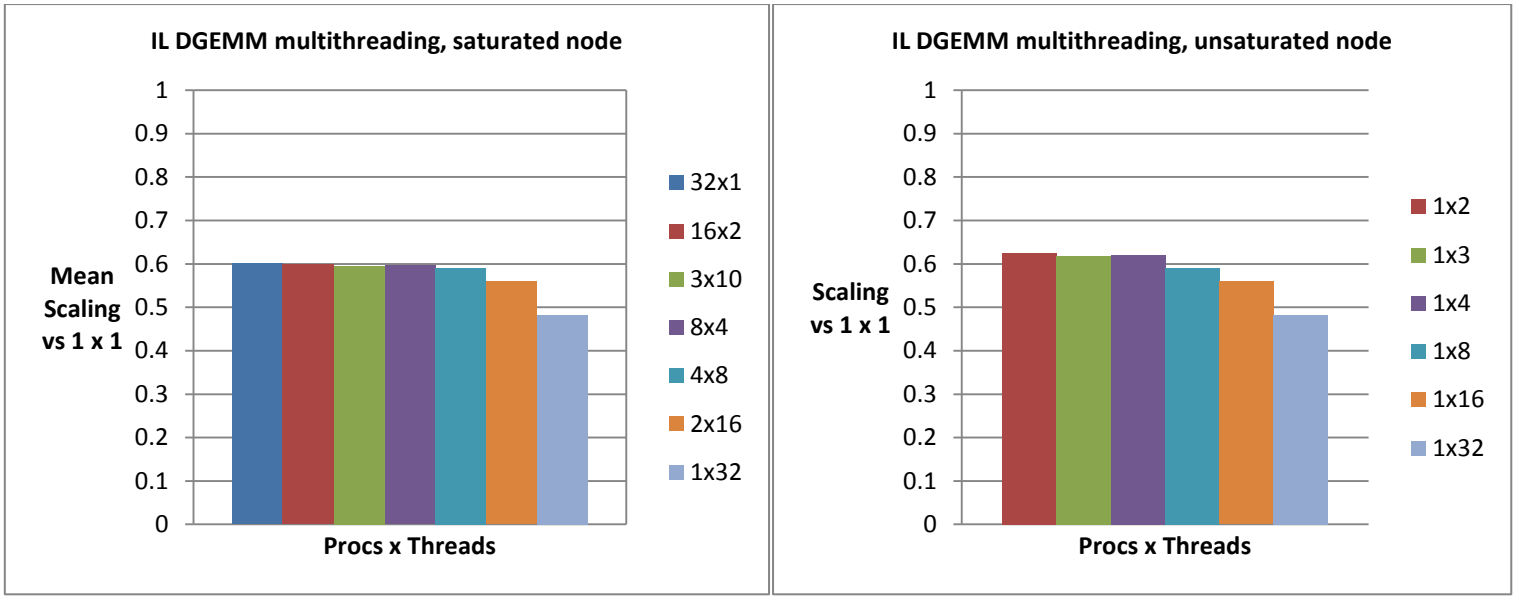


Figure 6 . *Multithreaded scaling of DGEMM test in a saturated IL node (left) and an unsaturated IL node (right). Compared with running a single process using a single thread, running 32 parallel DGEMM calls causes a 40% decrease in performance on IL (32x1, blue bar, left). This changes little up to 8 threads (turquoise bar, 4x8), in which threads occupy the same die, and therefore the small dip in performance can be attributed to parallel overheads rather than increased contention on resources. For 16 threads (orange bar, 2x16) communication over HyperTransport (HT) links within an IL processor occurs, causing a further dip, and for 32 threads (grey bar, 1x32) communication occurs over HT across the two IL processors, putting scaling below 50%. The chart on the right shows very little difference compared with that on the left, even when just 2 threads are used. This is because threads are placed sequentially by default, and e.g. in the case of 2 threads (red bar, 1x2) they share the same module. The similarity of the two charts suggests that module-level sharing is the dominant factor in the 40% loss of efficiency even with 2 threads. This is further evidenced by the results shown in Figure 7 below.*

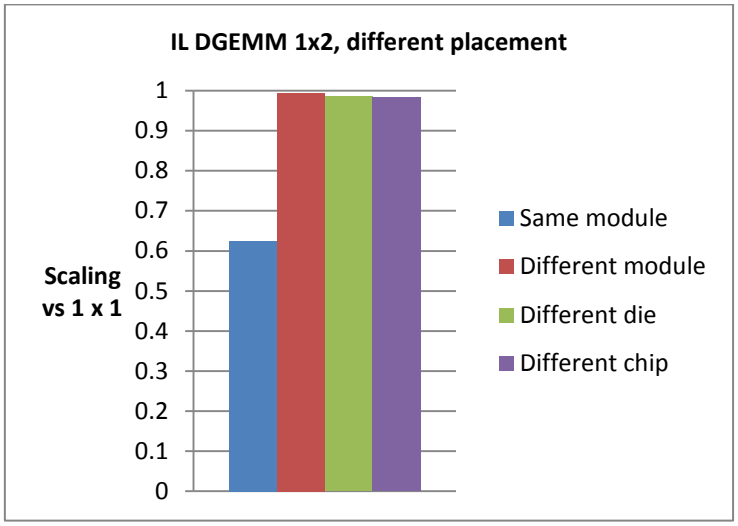


Figure 7 *The default placement (blue bar), to put threads in the neighbouring core, means both threads share the same module. Explicit placement of the 2nd thread in a different module within the same die (red bar), a different die in the same processor (green bar) and a different processor[3] (purple) yield significant performance gains and achieve near perfect scaling. This illustrates that the 40% loss of efficiency seen in Figure 6 can be attributed to threads sharing resources – most importantly the floating point unit in this case.*

[3] These placements can be controlled using the –cc flag to aprun on a Cray XE6, repectively here these are: aprun -n 1 -d 2 -cc 0,2, aprun -n 1 -d 2 -cc 0,8, aprun -n 1 -d 2 -cc 0,16.

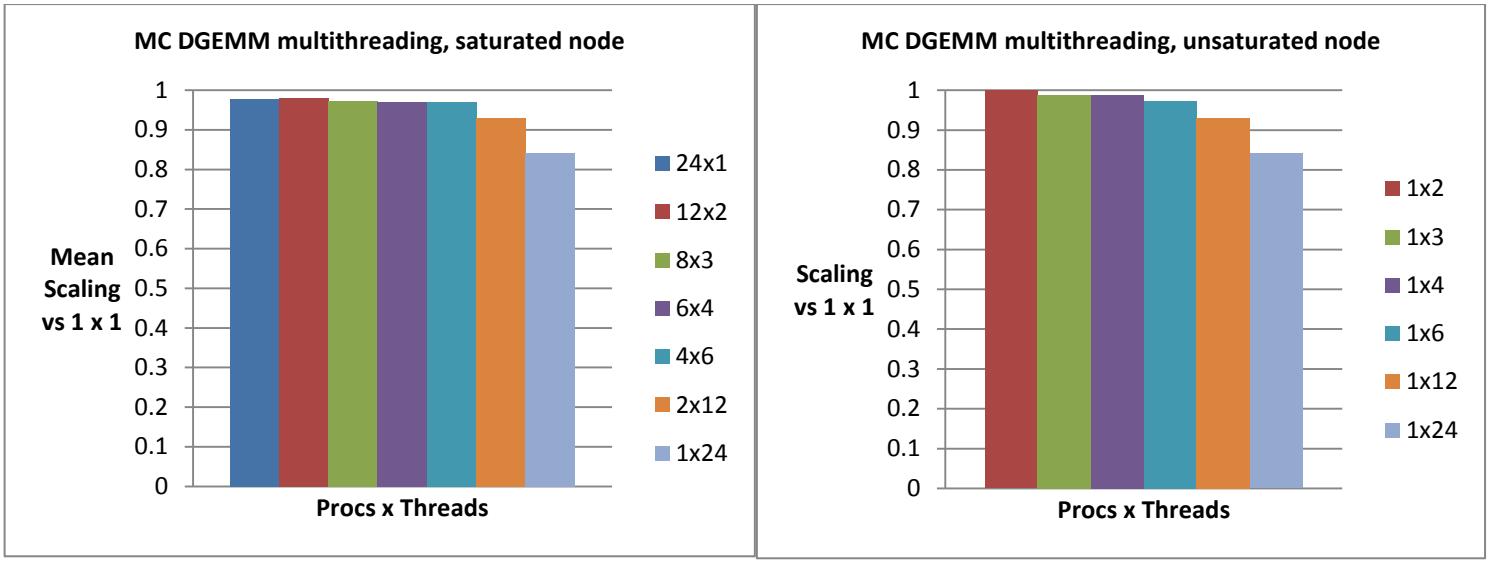


Figure 8 *Multithreaded scaling of DGEMM test in a saturated MC node (left) and an unsaturated MC node (right). As expected scalability is good because only L2 and below are shared by adjacent cores. In contrast to the data seen in Figure 6 multithreading suffers little compared with the baseline. Scalability only decreases significantly only when processes' threads cross die boundaries, as also seen in Figure 6 due to the NUMA.*

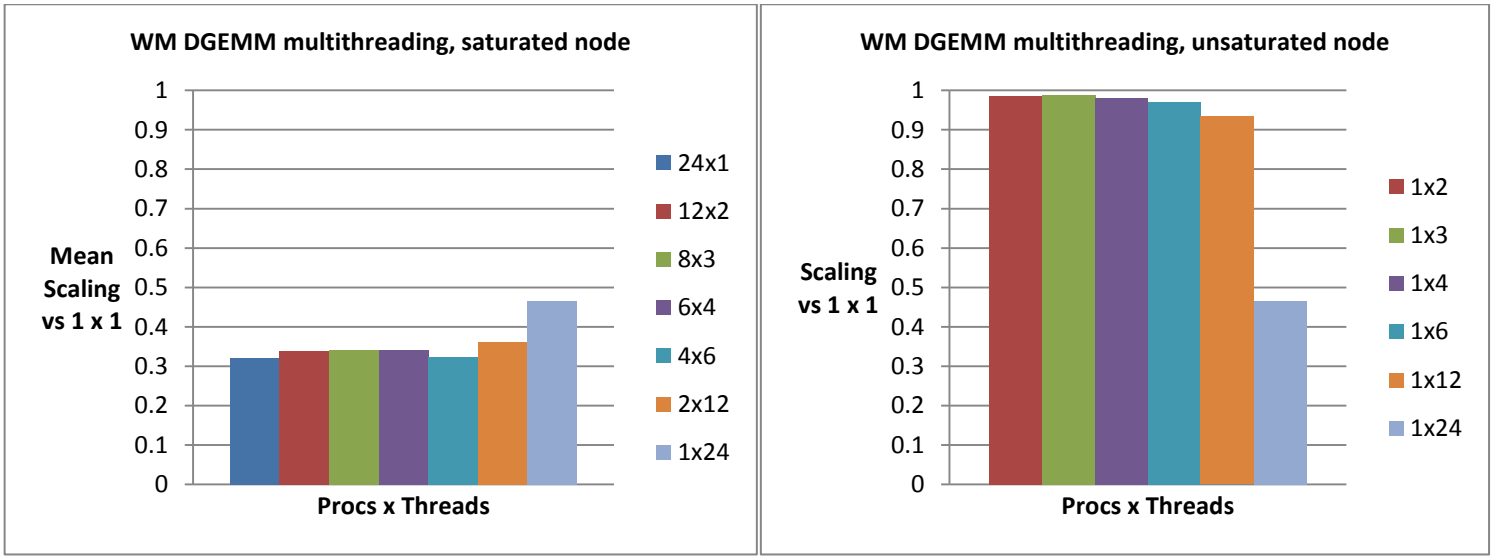


Figure 9 *Multithreaded scaling of DGEMM test in a saturated WM node(left) and an unsaturated WM node (right). Perhaps as expected due to the level of shared resources the saturated scalability of the DGEMM on WM is worst of all. Interestingly, when a single process occupies the whole node (grey bar, 1x24, left) scalability improves. A possible reason for this is that MKL may be optimised to take advantage of hyper-threading only when it proves profitable, utilizing fewer threads if it is preferable to do so, thereby decreasing contention on resources for compute-bound kernels such as DGEMM. Unsaturated scalability is as good as MC up to 12 threads. This is because, by default, threads are placed on separate physical cores until all physical cores have been used, and only then do threads share cores as hyper-threading occurs.*

Mersenne Twister Benchmarks

For this benchmark the NAG routine G05SAF (uniform pseudorandom number generator) example program was modified to perform NPROCS parallel calls.

```
  Program g05safe
!    G05SAF Example Program Text
!    Mark 24 Release. NAG Copyright 2011.
!    .. Use Statements ..
     Use nag_library, Only: g05kff, g05saf, nag_wp
     use timers
     use mpi
!    .. Implicit None Statement ..
     Implicit None
!    .. Parameters ..
     Integer, Parameter             :: lseed = 1, nin = 5, nout = 6
!    .. Local Scalars ..
     Integer                        :: genid, ifail, lstate, n, subid
!    .. Local Arrays ..
     Real (Kind=nag_wp), Allocatable  :: x(:)
     Integer                        :: seed(lseed)
     Integer, Allocatable           :: state(:)
     integer :: ierr, myrank
!    .. Executable Statements ..
     call mpi_init(ierr)
     call mpi_comm_rank(mpi_comm_world,myrank,ierr)
     if(myrank.eq.0)then
       Write (nout,*) 'G05SAF Example Program Results'
       Write (nout,*)
!      Skip heading in data file
       Read (nin,*)
!      Read in the base generator information and seed
       Read (nin,*) genid, subid, seed(1)
!      Read in sample size
       Read (nin,*) n
     end if
     call mpi_bcast(genid,1,mpi_integer,0,mpi_comm_world,ierr)
     call mpi_bcast(subid,1,mpi_integer,0,mpi_comm_world,ierr)
     call mpi_bcast(seed(1),1,mpi_integer,0,mpi_comm_world,ierr)
     call mpi_bcast(n,1,mpi_integer,0,mpi_comm_world,ierr)
!    Initial call to initialiser to get size of STATE array
     lstate = 0
     Allocate (state(lstate))
     ifail = 0
     Call g05kff(genid,subid,seed,lseed,state,lstate,ifail)
!    Reallocate STATE
     Deallocate (state)
     Allocate (state(lstate))
!    Initialize the generator to a repeatable sequence
     ifail = 0
     Call g05kff(genid,subid,seed,lseed,state,lstate,ifail)
     Allocate (x(n))
!    Generate the variates
     ifail = 0
     call init_counters
     call start_timer(1)
     Call g05saf(n,state,x,ifail)
     call end_timer(1)

!    Display the variates
     !Write (nout,99999) x(1:n)
     call print_timers()
     call mpi_finalize(ierr)
99999 Format (1X,F10.4)
   End Program g05safe
```

The OpenMP version of G05SAF from the NAG library for SMP and multi-core was used, thus allowing a similar set of multiprocessing and multithreading performance tests as performed in the DGEMM benchmark tests. PGI compiler version 11.9 was used on IL, PGI 11.6 on MC, and ifort 12.0 on WM. In each case the "–fast" compiler flag was used. The following input file was used in all tests, requesting 125,000,000 random numbers be generated for each process:

```
G05SAF Example Program Data
3  1  1762543    :: GENID,SUBID,SEED(1)
125000000      :: N
```

Aside from the fact that the main computational kernel in the Mersenne Twister algorithm predominantly operates on integers, only casting an integer into a floating point number as the last step, this routine differs from DGEMM in that its performance is bound by memory accesses rather than computational throughput. Thus contention on memory accesses as well as core-level resources is likely to be an important factor. Another difference compared with the DGEMM test is that for small N (125,000,000 is relatively small N) the overhead of parallelizing the generation sequence (ensuring the same sequence is generated as in serial) is not insignificant. Table 2 gives the baseline results for a single threaded process running exclusively on a clean node.

	IL	MC	WM
Time (secs)	1.573	1.501	0.656

Table 2. *Baseline results for Mersenne Twister. WM appears to way outperform IL and MC.*

The raw performance of WM far outstrips the two AMD processors. This is likely to be due to a combination of a superior memory subsystem[4], a higher clock speed and possibly a better compiler.

As with the DGEMM tests the remainder of this chapter will focus on how performance within a given processor changes depending on processor load and job placement using the respective baseline figures from Table 2.

The three types of tests performed for the DGEMM benchmark have been repeated for Mersenne Twister:

(a) The scalability of multithreading when the node is fully saturated.

(b) Scalability of multithreading using a single process.

(c) Scalability of multiprocessing.

The most interesting results are presented here.

Figure 10 shows that in the most extreme case where no data is shared between 32 independent processes the overheads compared with a single process, single threaded run amount to a 50% loss of performance. These overheads include contention within a single module, which is shown to be around 10% in Figure 11, and contention for L3 cache and main memory bandwidth, but not algorithmic parallel overheads. When running 2 cooperating threads in each module Figure 10 shows

[4] WM can have 20-30% more memory operations in flight than IL: http://www.realworldtech.com/page.cfm?ArticleID=RWT082610181333&p=8

that these same overheads amount to a 30% loss of performance as more data is shared in L2 and L3 cache. The same figures for MC are 20% for 24 independent processes and 10% for 12 processes with 2 threads. These results indicate that, in answer to Q3, even when contention for resources within an IL module is deducted from the cost, the loss of efficiency due to contention for L3 and memory bandwidth is significant.

Compared with WM the 10% loss of efficiency due to contention in the module makes no difference when the node is fully saturated. However, WM again has the advantage that the OS fills physical cores before it relies on hyper-threading, whereas for IL the OS fills a module first, leading to the 10% penalty being seen straight away in scaling graphs. E.g. in the right-hand charts of Figure 10 efficiency loss is at 25% for 2 threads, whereas in Figures 12 and 13 it is more like 15%.

So although the 10% cost of contention within a module seen in this test is much less than the 40% cost seen for DGEMM (as would be expected since the scalar integer execution units are not shared) it is still significant when making comparisons with MC and WM.

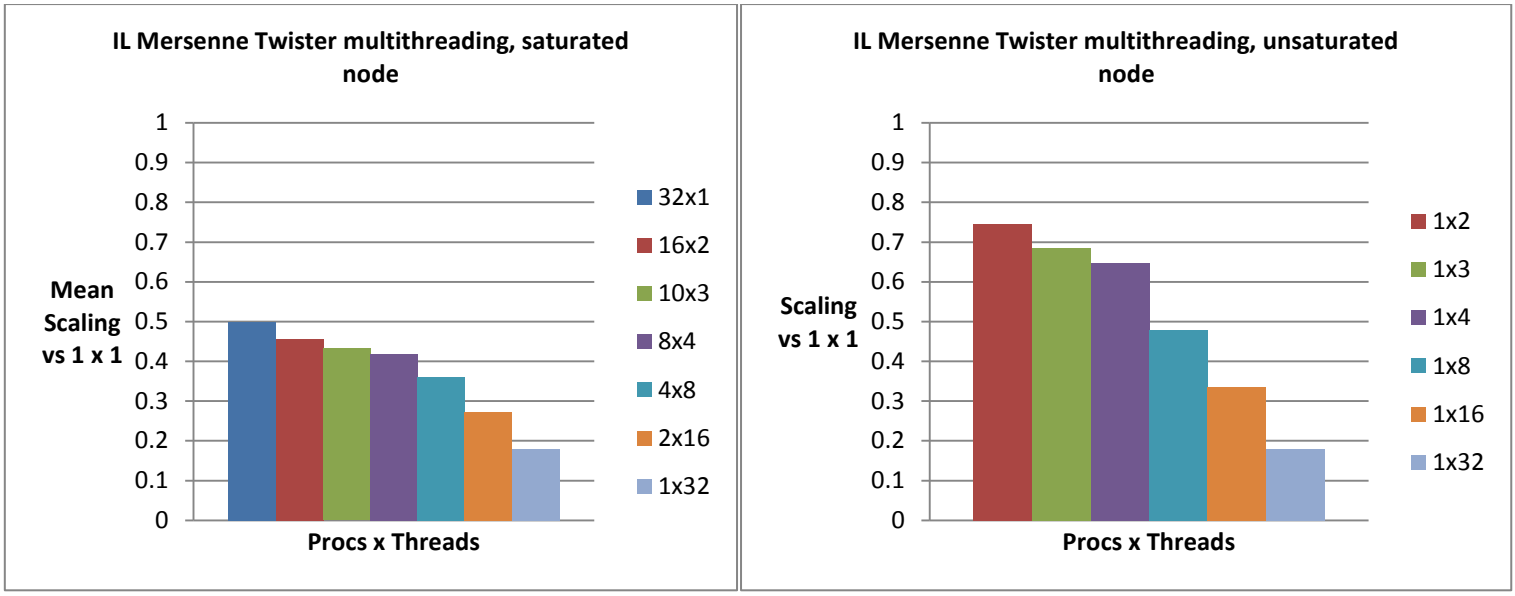


Figure 10 *Multithreaded scaling of Mersenne Twister in a saturated IL node (left) and an unsaturated IL node (right). Compared with a single process using a single thread around 55% efficiency is lost when performing 16 independent G05SAF calls with 2 threads each (red bar, left). When the node is unsaturated for 2 threads (red bar, right) the loss of efficiency drops to around 25%. This indicates that around 30% of the 55% loss of efficiency when the node is fully saturated is due to contention with other modules for L3 cache and memory bandwidth, since the parallel overheads and module-level contention are the same in the red bar on the right. The blue bar on the left shows a 50% loss of efficiency with no parallel overheads. However, module level contention for L2 and contention for L3 and main memory will be greater than for 2 threads because no data is shared.*

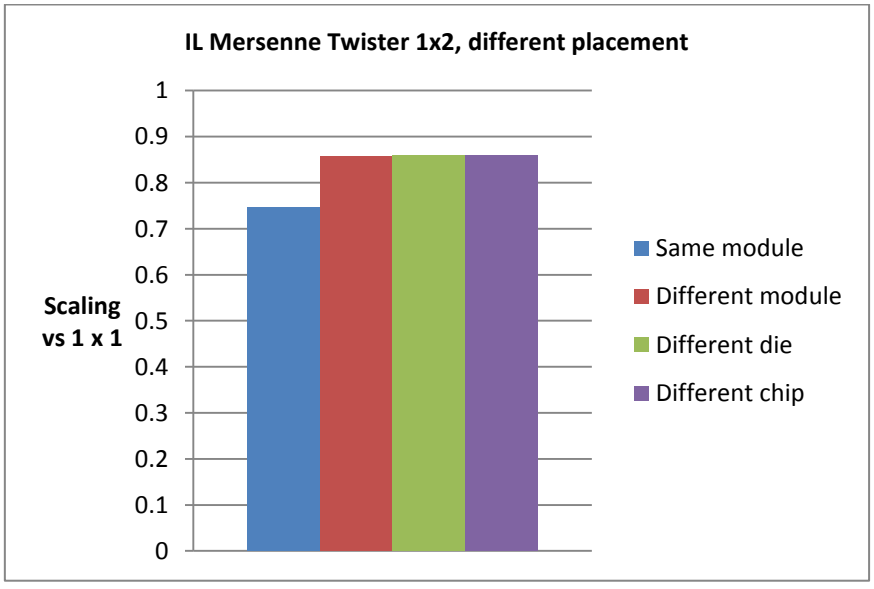
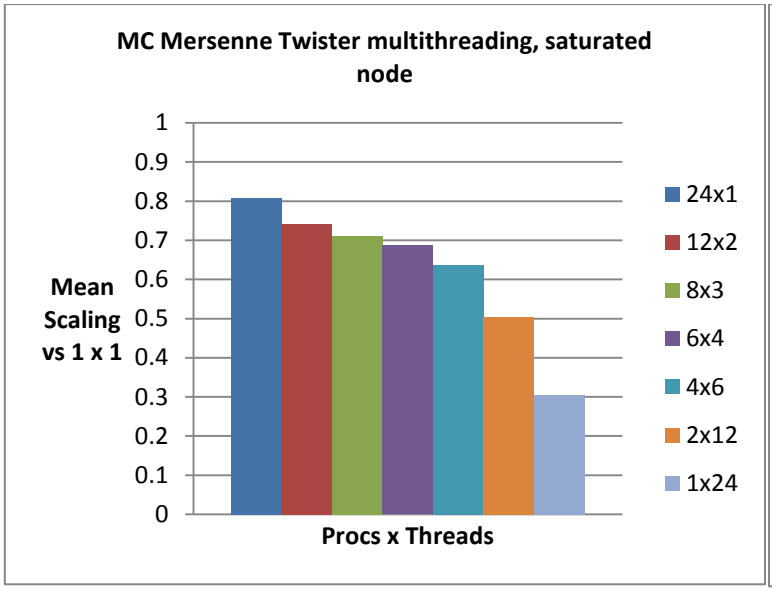


Figure 11 *Altering placement of the second thread eliminates module-level contention. There is a 10% loss of efficiency when the two threads are placed in the same module, compared to placing them in neighbouring modules, separate dies or separate processors. This is a much smaller loss of efficiency due to module-level sharing than seen in the DGEMM tests (Figure 7), which was 40%. The remaining 15% loss of efficiency can be attributed to parallel overheads and L3/main memory contention.*



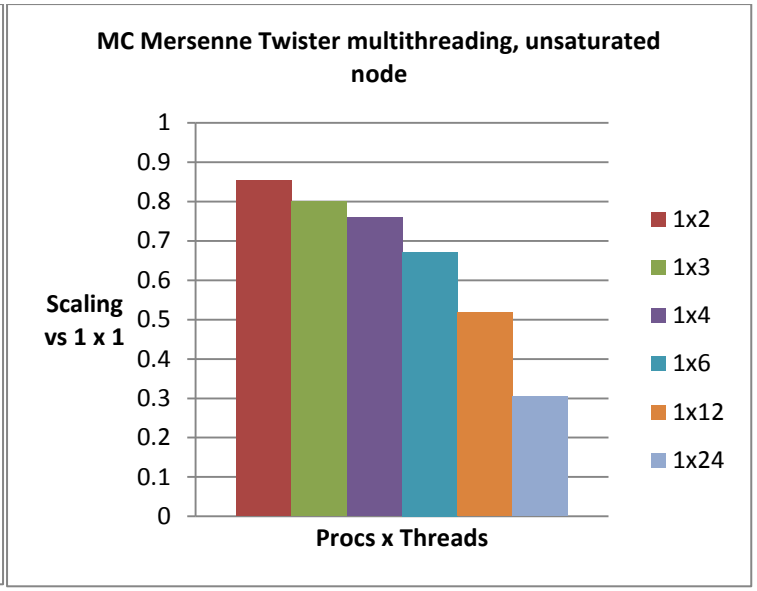
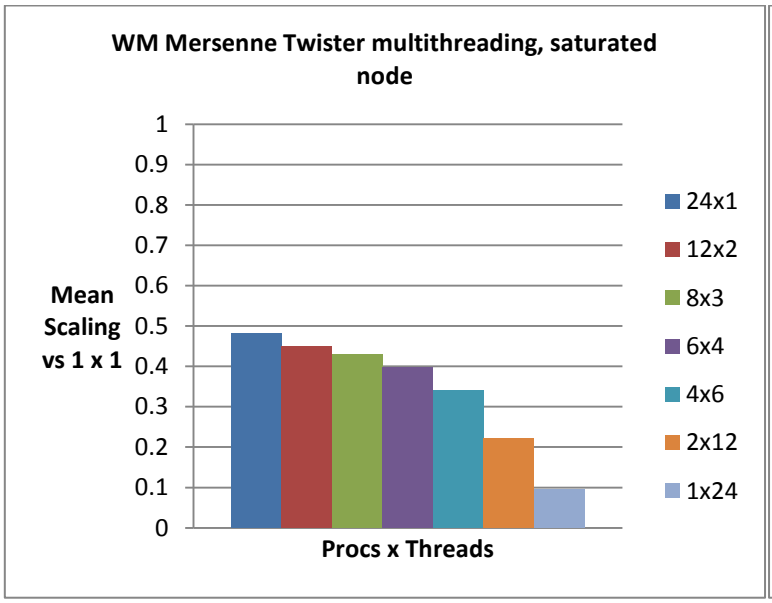


Figure 12 *Multithreaded scaling of Mersenne Twister in a saturated MC node (left) and an unsaturated MC node (right). There is a 25% loss of efficiency due to L3 cache and memory bandwidth contention when performing 12 independent G05SAF calls with 2 threads each (red bar, left), which is the same as the unsaturated results for IL in Figure 10. The unsaturated result for 2 threads (red bar, right) is the same as IL when threads are placed in separate modules as seen in Figure 11, a 15% efficiency loss. The difference between the red bar on the right and the red bar on the left indicates the amount of L3 and memory bandwidth contention, which is around 10% here, compared with 30% for IL. In the case with no parallel overheads (blue bar, left) the loss of efficiency is less than 20% compared with 50% for IL. This indicates that the effective decrease in available memory bandwidth per core for IL is significant.*



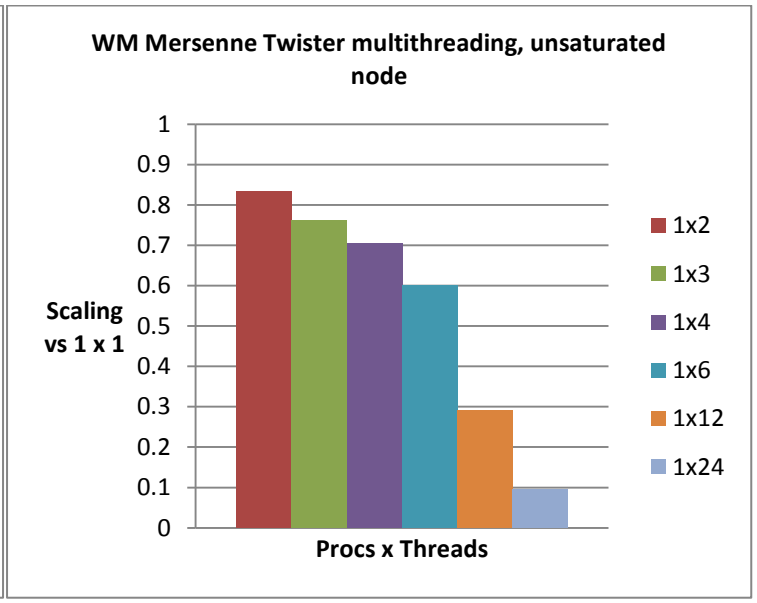


Figure 13 *Multithreaded scaling of Mersenne Twister in a saturated WM node (left) and an unsaturated WM node (right). The chart on the left shows an almost identical pattern to Figure 10 for IL. However, in the chart on the right there is only a 15% loss of efficiency at 2 threads, the same as MC and IL when threads do not share a module. This is because threads are not sharing core-level resources in WM until more than 12 processes or threads are executing.*

Conclusions

The results clearly show that module-level resource sharing in IL has a very significant impact on performance for a compute-bound HPC kernel, DGEMM. *Under full load t*he impact is relatively less than in a hyper-threading processor (WM) and greater than a more traditional multi-core processor (MC). The penalties are roughly a few percent for MC, 70% for WM and IL is somewhere in between at 40%. When executing a memory-bound code performance is limited less by contention within modules, but more due of the increased number of cores having a smaller slice of memory bandwidth.

When considering processors not at full load the placement policy used by the respective operating systems has a key impact on performance. For IL the OS sees each core in a module as a separate physical core, even though for floating point intensive applications these results suggest a module should be considered a single core. Default placement of threads/processes is to fill modules, dies, processors in that order. Thus using just 2 threads or processes in a job means that those tasks will share the same module by default. In the case of DGEMM this has the same 40% impact on performance as seen under full load, suggesting module level sharing of resources dominates. This contrasts with how the OS sees the WM processor. On WM it sees 12 physical cores and default placement of tasks is to fill physical cores in the same processor, then across both processors and finally to use “virtual cores” or hyper-threading. This means that although the core-level sharing of resources has a higher impact on performance (70%) than IL, this is not seen until more than 12 tasks are active. Thus for compute bound floating point codes a module should be seen as a single core until there is the need to “overpopulate” with 2 processes/threads, as is the OS’s policy for WM. For memory bound codes it can be seen as a dual core unit with a much smaller penalty, especially if working on integer data. However, the penalty is seen immediately if e.g. only 2 threads are being used.

These tests demonstrates how it is becoming increasingly desirable and important for applications to have control over how processes/threads are placed, but the complexity of doing so from a job launcher such as Cray’s aprun command or MPICH’s mpirun command often means that it is only ever investigated by the most expert users.

Experiences on HECToR

The trend of inferior performance compared with MC seen in the benchmarks has been broadly echoed by real-world HPC applications running on HECToR. This section gives some performance graphs for the quantum chemistry code CASTEP, which makes heavy use of L3 BLAS and LAPACK routines, and is the second most heavily used application on HECToR.

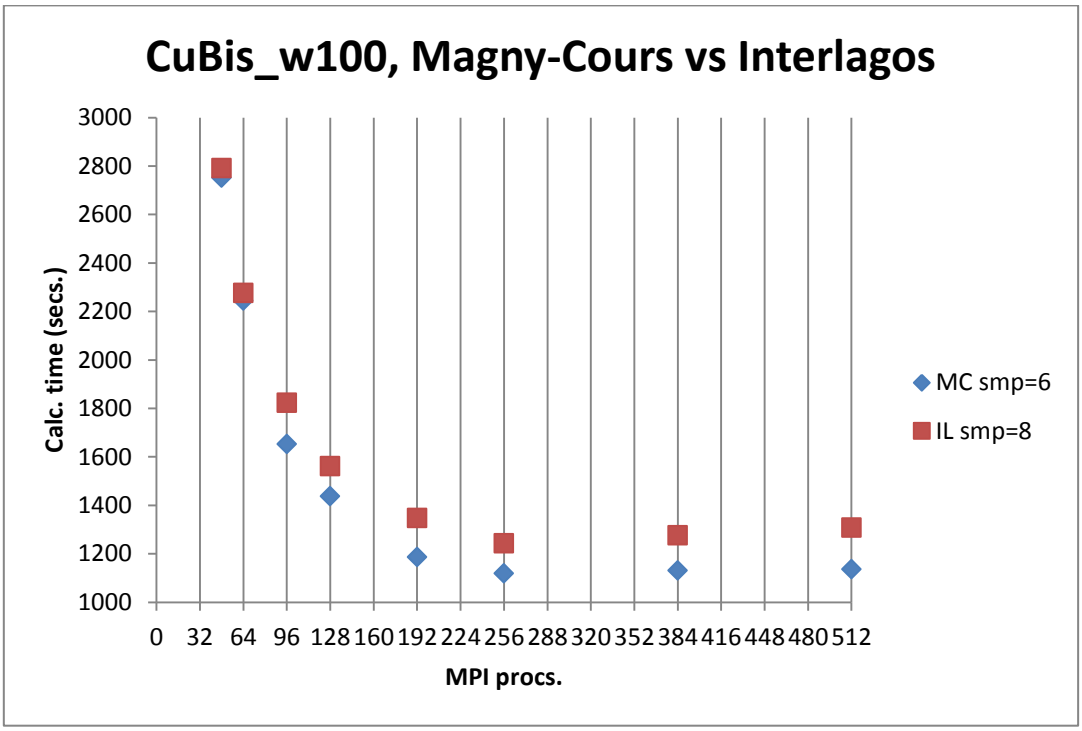


Figure 14 *CASTEP, Benchmark 1, comparing IL with MC using the same number of processes.*

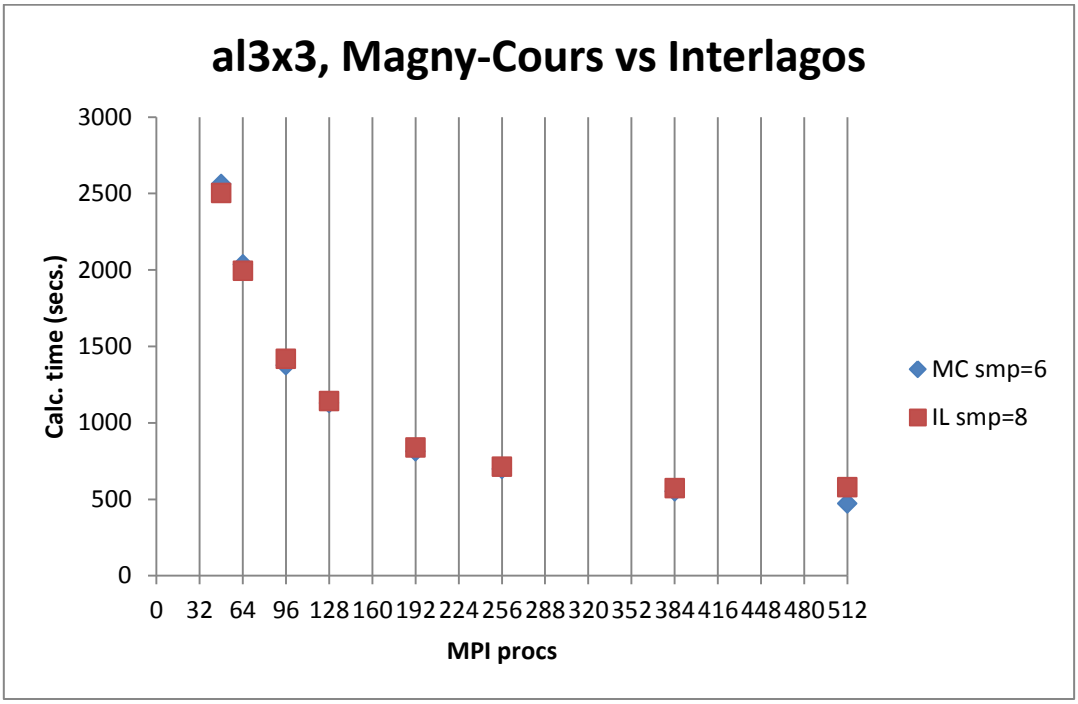


Figure 15 *CASTEP, Benchmark 2, comparing IL with MC using the same number of processes.*

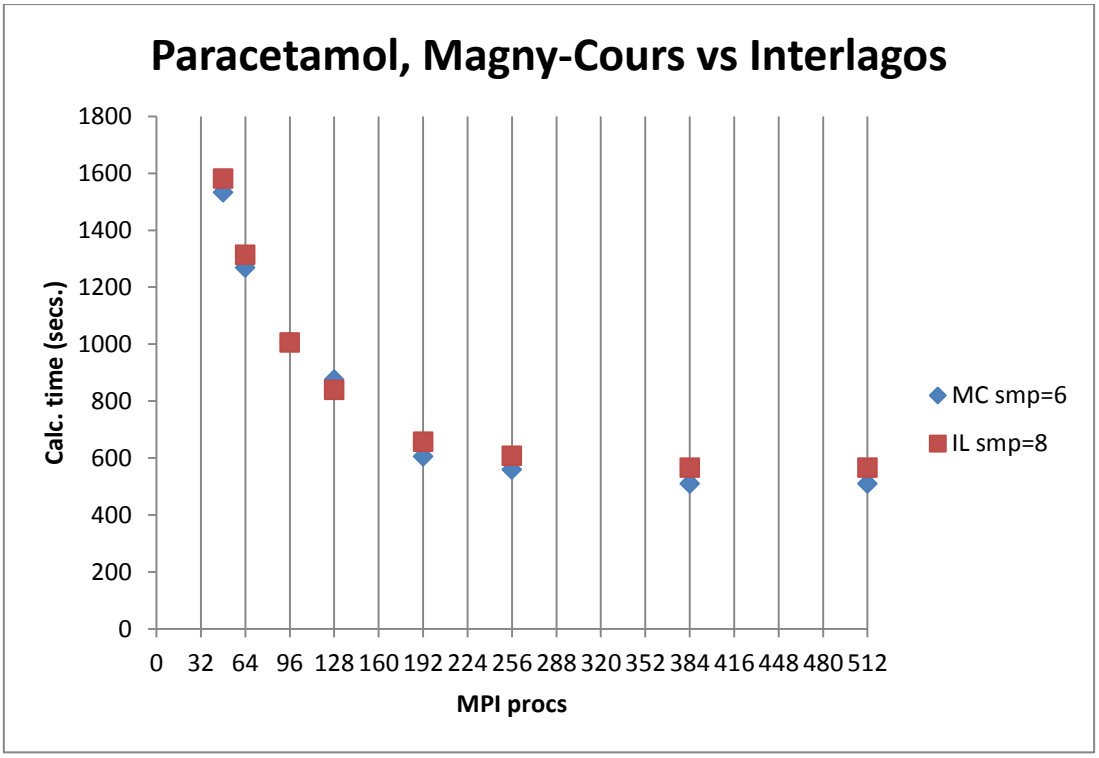


Figure 16 *CASTEP, Benchmark 3, comparing IL with MC using the same number of processes.*

In Figures 14-16 smp=8 for IL and smp=6 for MC indicates that a System V shared memory optimization has been enabled in CASTEP, utilizing a separate segment for each die in a node, which should have a neutral effect in these comparisons.

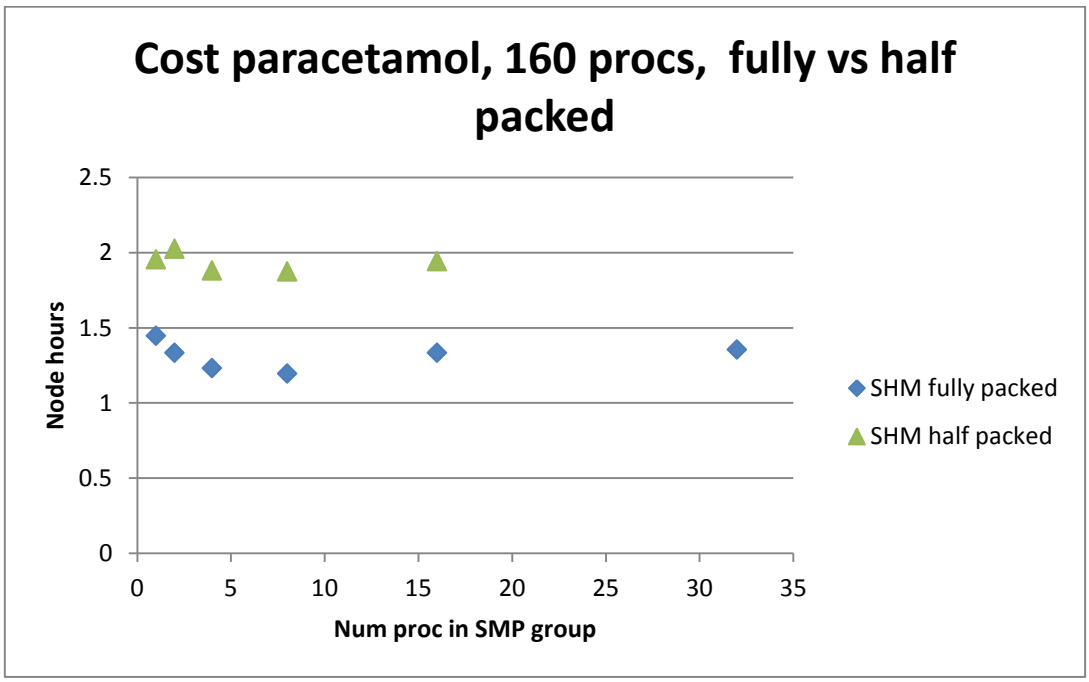


Figure 17 The cost of running half packed, i.e. one process per module, is not justified. For the most appropriate SMP group size, 8, it is approximately 1.6x more expensive to use double the number of nodes for the same sized job.

In 2 out of the 3 benchmarks shown in Figures 14-16 IL is significantly slower than MC when running fully packed (32 MPI processes per node IL, 24 MPI processes per node MC). It seems that the performance gain enjoyed due to the new AVX/FMA instruction set is less than the performance loss due to the issues of saturating a node discussed in the benchmarking section. Figure 17 shows that the performance gain seen by running the same job using a single process per module (i.e. doubling the number of nodes in a job) is not cost-effective.

Overall Conclusions

AMD's Bulldozer architecture, represented by the Interlagos processor in this paper, appears to be a half-way design between those of previous AMD generations, which have a traditional multi-core design where cores only start sharing resources at the level of L2 or L3 cache, represented by the Magny-Cours processor here, and Intel's hyper-threading designs where virtual cores share all resources except registers that maintain state, represented by a Westmere processor in this paper.

When these processors are fully saturated with work these expectations pan out: Magny-Cours gives the best parallel efficiency followed by Interlagos and then Westmere. However, from looking at how programs scale up from a serial benchmark the Interlagos processor is immediately penalised by the operating system's view of the processor as 16 physical cores. It is clear from the benchmark results presented in this paper that, especially for compute-intensive floating point applications, the processor should be viewed as an 8-core chip, and filled with tasks as such before sharing modules only when more than 8 processors or threads are active. This policy pays dividends in the case of Westmere.

In the specific case of the phase 3, 32-core HECToR Interlagos upgrade there is the added penalty of less memory bandwidth per core in comparison to the previous phase of 24-core Magny-Cours nodes. This has been shown to be significant for the memory-bound RNG Mersenne Twister benchmark.

The greatest challenge posed by Interlagos is therefore one of job placement. On HECToR users will have to be educated in how to place processes and tasks using the aprun launcher, which is not trivial, and to benchmark their codes in order to select the best strategy for running their jobs. More generally, e.g. in the context of the NAG SMP library, it is no longer the case that routines can rely on OS placement in order to perform optimally. Some control over thread placement at run-time based on system load, affinity, and the type of routine being used would be ideal in order to make the best use of this kind of processor appear transparent to users. Fortunately these issues are already being addressed in OpenMP 4.0, via e.g. a new form for specifying OMP_NUM_THREADS as a list for nested parallelism, controlling processor affinity with OMP_PROC_BIND and restricting execution to a nominated set of cores with OMP_PROCSET.