Best Practice Guide - Knights Landing

Vali Codreanu, SURFsara

Jorge Rodríguez, BSC

Ole Widar Saastad (Editor), University of Oslo

31-01-2017

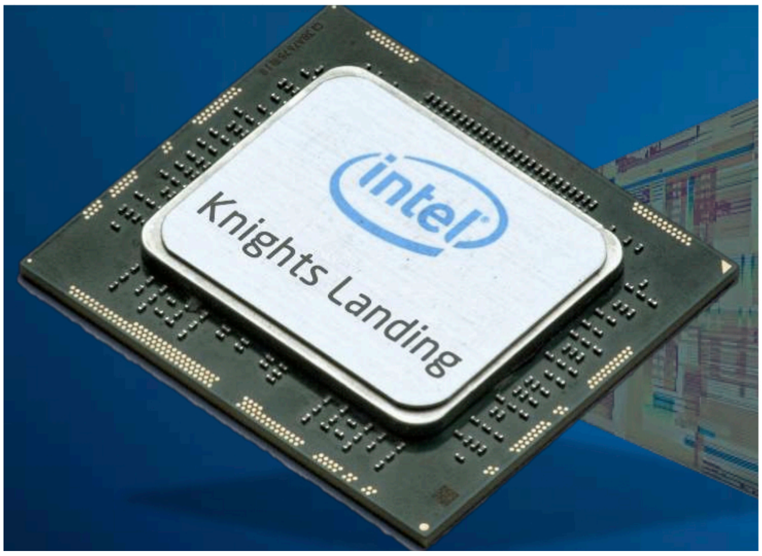
Table of Contents

1. Introduction

This best practice guide provides information about Intel's MIC architecture and programming models for the Intel Xeon Phi co-processor in order to enable programmers to achieve good performance of their applications. The guide covers a wide range of topics from the description of the hardware of the Intel Xeon Phi co-processor through information about the basic programming models as well as information about porting programs up to tools and strategies how to analyze and improve the performance of applications.

The Knights Landing (KNL) processor differ from the usual Intel processor due to its very high core count and the hardware threading architecture. It represent an approach where a large number of simples cores are employed in large number as opposed to larger more sophisticated cores in smaller number. The idea is that a higher fraction of the transistors could be used for arithmetric operations. Over the decades the flops per transistor have declined. The KNL represent the second generation of this approach, the Knights Corner (KNC) being the first. There is also a best practice guide for the KNC [http://www.prace-ri.eu/best-practice-guide-intel-xeon-phi-html].

Figure 1. Knights Landing processor.



The KNL processor is a x86-64 compatible processor architecture and hence most of the content of the Best Practice Guide for the x86 architecture is still valid. It's a good idea to have the x86 guide handy as this guide cover the more KNL specific topics. The x86 BPG can be found at the PRACE web site [http://www.prace-ri.eu/Best-Practice-Guide-Generic-x86-HTML].

2. System Architecture / Configuration

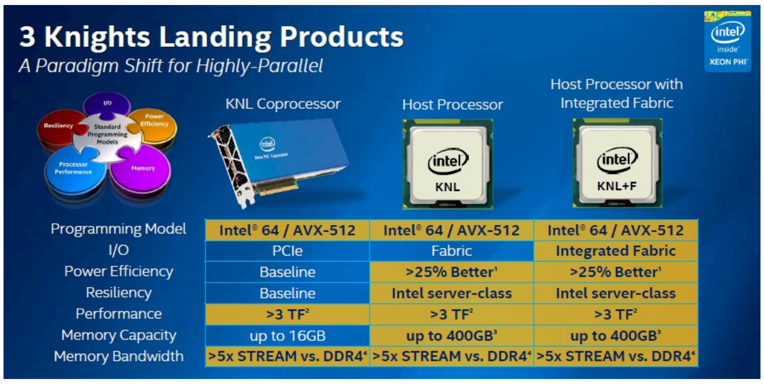
From a system point-of-view, the KNL platform comes in three configurations. The three options are outlined in Figure 2.

- The first, similar to the previous incarnation of the Xeon Phi architecture (Knights Corner), is a co-processor card. This card has access to 16GB of high-bandwidth memory (HBM) and PCIe access to the host memory.

- The second option is a standalone host processor. This option has the main advantage that it can boot and run a full-fledged OS. Another advantage that it has both access to the HBM and to the much-larger system DDR4, without being slowed by PCIe access. For communication with other KNL nodes, the system I/O fabric is used.

- The third option, and perhaps most appropriate for HPC is the standalone host processor that also has the Intel OmniPath communication fabric integrated on the package. It provides a 100Gb/s link across the KNL computing nodes, with arguably lower latencies than InfiniBand EDR.

Figure 2. Knights Landing variants.


I/O
Power Efficiency
Memory
Processor Performance
Resiliency
Standard Programming Models
intel
KNL
intel
KNL+F
intel inside XEON PHI

3 Knights Landing Products

A Paradigm Shift for Highly-Parallel

	KNL Coprocessor	Host Processor	Host Processor with Integrated Fabric
Programming Model	Intel® 64 / AVX-512	Intel® 64 / AVX-512	Intel® 64 / AVX-512
I/O	PCIe	Fabric	Integrated Fabric
Power Efficiency	Baseline	>25% Better[1]	>25% Better[1]
Resiliency	Baseline	Intel server-class	Intel server-class
Performance	>3 TF[2]	>3 TF[2]	>3 TF[2]
Memory Capacity	up to 16GB	up to 400GB[3]	up to 400GB[3]
Memory Bandwidth	>5x STREAM vs. DDR4[4]	>5x STREAM vs. DDR4[4]	>5x STREAM vs. DDR4[4]

Thus, at a high level, the KNL system is composed of:

- The computing subsystem: composed of up to 36 compute tiles, connected in a 2D mesh fashion, each tile being composed of 2 cores.
- The memory subsystem: composed of 16GB of high-speed stacked memory accessed by 8 high-speed memory controllers, as well as up to 384GB of DDR4 accessed by 2 3-channel memory controllers.
- The I/O subsystem: composed of 36 PCIe lanes and an optional Omnipath controller.

One of the big paradigm shifts between the Knights Corner and Knights Landing capabilities is that the latter allows running a full-fledged operating system. This is because the KNL cores are fully Intel Xeon ISA-compatible through AVX2.

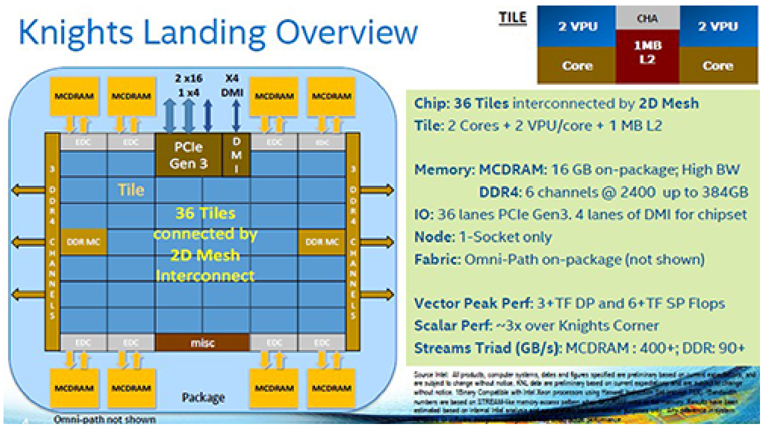
Another important detail is that the socketed KNL systems are single-socket systems. There is no QPI link connecting more KNL dies together. Hence, the performance per node is limited to the performance of a single KNL chip. In order to offer scalable performance, the communication fabric has to be very efficiently used, as large KNL installations will feature a very large number of nodes.

2.1. Processor Architecture / MCM Architecture

The Knights Landing processor architecture is composed of up to 72 Silvermont-based cores running at ~1.3-1.4 GHz. The cores are organized into tiles, each tile comprising two cores, and each core having two AVX-512 vector processing units. Each tile has 1MB of L2 cache, shared by the two cores, for a total of 36MB of L2 cache across the chip. The tiles are connected in a 2D mesh topology. The cores are 14nm versions of Silvermont, rather than 22nm P54C used in Knights Corner, with claims by Intel that the out-of-order performance is vastly improved, the KNL cores delivering up to 3 times the single-core performance of the Knights Corner cores. The architecture is depicted in Figure 3.

Each of the 72 cores is out-of-order and is multithreaded, supporting 4 SMT threads, similar to the Knights Corner. However, in order to reach peak performance for KNC, one needed to use at least 2 threads/core. In the case of KNL, it is claimed that peak performance can be achieved by using 1 thread/core for certain application. Another advantage of the KNL cores is that they are ISA-compatible with the regular Xeon cores, and can thus run any Xeon application without recompilation.

Figure 3. Knights Landing architecture.



Chip: 36 Tiles interconnected by 2D Mesh
Tile: 2 Cores + 2 VPU/core + 1 MB L2

Memory: MCDRAM: 16 GB on-package; High BW
DDR4: 6 channels @ 2400 up to 384GB
IO: 36 lanes PCIe Gen3. 4 lanes of DMI for chipset
Node: 1-Socket only
Fabric: Omni-Path on-package (not shown)

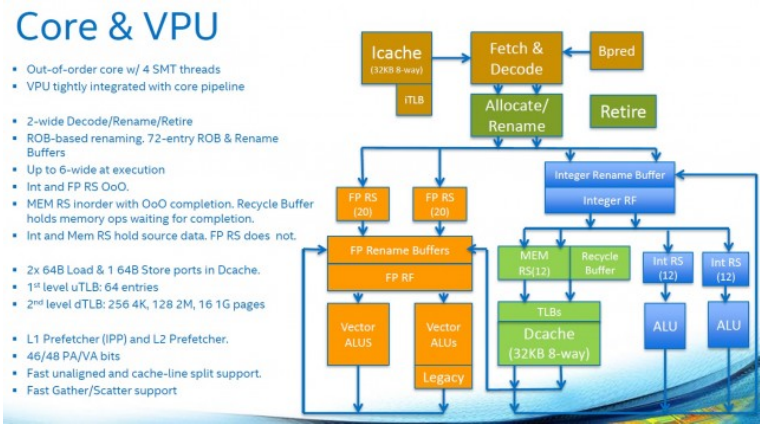
Vector Peak Perf: 3+TF DP and 6+TF SP Flops
Scalar Perf: ~3x over Knights Corner
Streams Triad (GB/s): MCDRAM : 400+; DDR: 90+

Going back to the core architecture, each KNL core features:

- 32KB Instruction and Data caches, 64 bytes cache line length.
- 2 AVX-512 units, each supporting up to 8DP/16SP operations per cycles. The units are tightly integrated with the core pipeline.
- L1/L2 prefetchers.
- Fast unaligned and cache-line split support. Fast Gather/Scatter support.
- Dynamic resource sharing between active threads.
- 4 SMT threads (HyperThreading).

Besides the two cores and the 1MB L2 cache, each tile includes a Cache/Home Agent (CHA) that sits on the 2D mesh interconnect and helps keep the 1 MB L2 caches from all tiles coherent.

Figure 4. Knights Landing core.



- Out-of-order core w/ 4 SMT threads
- VPU tightly integrated with core pipeline

- 2-wide Decode/Rename/Retire
- ROB-based renaming. 72-entry ROB & Rename Buffers
- Up to 6-wide at execution
- Int and FP RS OoO.
- MEM RS inorder with OoO completion. Recycle Buffer holds memory ops waiting for completion.
- Int and Mem RS hold source data. FP RS does not.

- 2x 64B Load & 1 64B Store ports in Dcache.
- 1st level uTLB: 64 entries
- 2nd level dTLB: 256 4K, 128 2M, 16 1G pages

- L1 Prefetcher (IPP) and L2 Prefetcher.
- 46/48 PA/VA bits
- Fast unaligned and cache-line split support.
- Fast Gather/Scatter support

The KNL core microarchitecture has changed tremendously from the KNC one. The Silvermont-based core is modified to better accomodate HPC applications, and it offers 3x the single-thread performance of a KNC core. As can be seen in Figure 4, in order to achieve this Intel uses an out-of-order execution engine, and has increased the buffering structures that help with Instruction Level Parallelism. Each KNL core features 2-wide decode/rename/retire stages, 72 inflight µops/core out-of-order buffers, 72-entry ROB and rename buffers, up to 6-wide at execution. The branch prediction logic was also improved, less cycles being wasted due to branch misprediction.

2.1.1. Instruction Set Architecture

Being backward compatible with Intel Xeon products, KNL supports all previous ISA extensions: SSE, AVX, AVX2. On top of them, KNL adds support for the AVX-512 ISA, that will also be supported by upcoming CPU architectures such as Intel Skylake. In this guide, we will cover more extensively the additions from AVX2 to AVX-512, and we advise the reader to check out the Haswell and generic x86 guides for descriptions of the previous ISA extensions.

Thus, the 512-bit vector functionality from KNL is comprised of:

- AVX512F: 512b vector extensions with mask support.
- AVX512PFI: Introduces new prefetch instructions
- AVX512ERI: Introduces new exponential and reciprocal instructions
- AVX512CDI: Introduces conflict detection instructions that help with vectorization

Intel AVX-512 Foundation instruction set is a natural extension to AVX and AVX2. It introduces the following architectural enhancements:

- Support for 512-bit wide vectors and SIMD register set. 512-bit register state is managed by the operating system using XSAVE/XRSTOR instructions previously introduced.
- Support for 16 new, 512-bit SIMD registers (for a total of 32 SIMD registers, ZMM0 through ZMM31) in 64-bit mode. The extra 16 registers state is managed by the operating system using XSAVE/XRSTOR/XSAVEOPT.
- Support for 8 new opmask registers (k0 through k7) used for conditional execution and efficient merging of destination operands. These opmask registers are also managed by the operating system using the XSAVE/XRSTOR/XSAVEOPT instructions.
- A new encoding prefix (referred to as EVEX) to support additional vector length encoding up to 512 bits. The EVEX prefix builds upon the foundations of VEX prefix, to provide compact, efficient encoding for functionality available to VEX encoding plus the following enhanced vector capabilities:

  (1) opmasks

  (2) embedded broadcasts

  (3) instruction prefix-embedded rounding control

  (4) compressed address displacements.

2.1.1.1. Masking

Most AVX-512 instructions may indicate one of 8 opmask registers (k0–k7). The k0 register is a hardcoded constant used to indicate unmasked operations. These registers are mostly used to control which values are written to the destination. A flag controls the opmask behavior, which can either be "zero" or “merge”. “Zero” zeroes out everything not selected by the mask, while "merge" leaves everything not selected unmodified. The merge behavior is identical to the blend instructions.

2.1.1.2. Embedded broadcasts

One addition to the EVEX encoding is a bit-field that encodes data broadcast for some instructions that load data from memory and perform some computational or data movement operation. An element from memory can be broadcasted (repeated) across all the elements of the effective source operand (up to 16 times for 32-bit data element, up to 8 times for 64-bit data element using 512-bit vector registers). This functionality reduces the overhead of reusing the same scalar operand for all the operations in a vector instruction. The broadcasting functionality is only enabled for elements of either 32 bits or 64 bits in size. Byte and word instructions do not support embedded broadcast.

2.1.1.3. Embedded rounding control

In previous SIMD extensions rounding control was usually specified in MXCSR, with some instructions providing per-instruction rounding override via encoding fields within the imm8 operand. AVX-512 offers a more flexible encoding attribute to override MXCSR-based rounding control for floating-pointing instruction with rounding semantic. This rounding attribute embedded in the EVEX prefix is called Static (per instruction) Rounding Mode or Rounding Mode override. This attribute allows programmers to statically apply a specific arithmetic-rounding mode irrespective of the value of RM bits in MXCSR. It is available only to register-to-register flavors of EVEX-encoded floating-point instructions with rounding semantic. Four rounding modes are supported by direct encoding within the EVEX prefix overriding MXCSR settings. This embedded rounding is implemented both because saving, modifying and restoring MXCSR is usually slow and cumbersome, and because being able to avoid suppressions and set the rounding-mode on a per instruction basis simplifies development of high performance math codes.

2.1.1.4. Other AVX-512 extensions in KNL

Besides the foundation instructions, KNL features three additional extensions:

- AVX512PFI: Introduces instructions to:

  (1) Prefetch a cache line into the L2 cache with intent to write (RFO ring request). This reduces ring traffic in core-to-core data communication.

  (2) Prefetch vector of D/Qword indexes into the L1/L2 cache.

  (3) Prefetch vector of D/Qword indexes into the L1/L2 cache with intent to write. Reduce overhead of software prefetching: dedicate side engine to prefetch sparse structures while devoting the main CPU to pure raw flop.

- AVX512ERI: Introduces instructions to compute approximation of

  (1) exponential functions

  (2) reciprocal

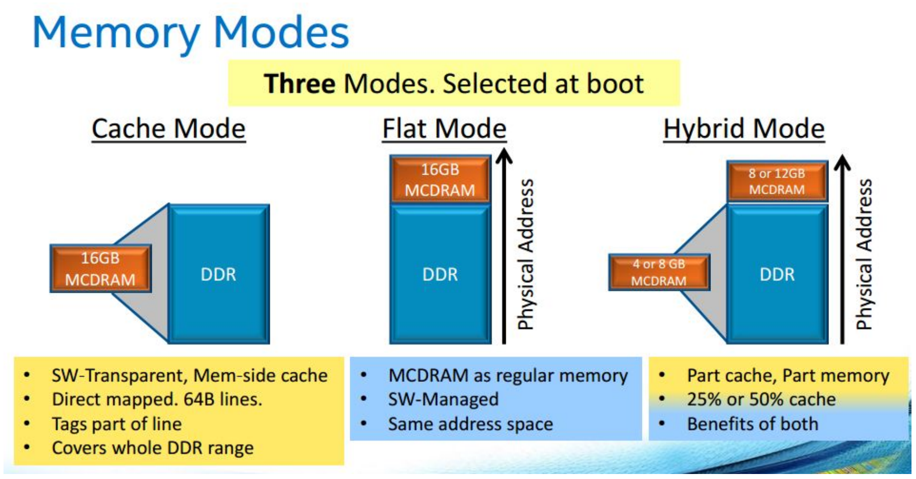
  (3) reciprocal square root.

- AVX512CDI: Introduces conflict detection instructions that help with vectorization (a CDI example can be found on page 28 : http://www.alcf.anl.gov/files/Sewall_ANL-ESPKnightsLanding.pdf). The VPCONFLICT instruction detects elements with previous conflicts in a vector of indexes. It allows to generate a mask with a subset of elements that are guaranteed to be conflict free, and afterwards the computation loop can be re-executed with the remaining elements until all the indexes have been operated upon.

2.2. Memory Architecture

The memory subsystem: composed of 16GB of high-speed stacked memory accessed by 8 high-speed memory controllers, as well as up to 384GB of DDR4 accessed by 2 3-channel memory controllers. While raw floating point and integer performance is an important aspect of the Knights Landing design, the achievable memory bandwidth is perhaps of the same or even greater importance. It is expected that the KNL chip can get more than 400 GB/sec of bandwidth out of the 16 GB of MCDRAM and more than 90 GB/sec out of the regular DRAM attached to the chip running the STREAM Triad memory bandwidth benchmark.

Figure 5. Different memory modes"



As can be seen in Figure 5, the KNL memory can work in three different modes. These are determined by the BIOS at POST time and thus require a reboot to switch between them.

2.2.1. Flat mode

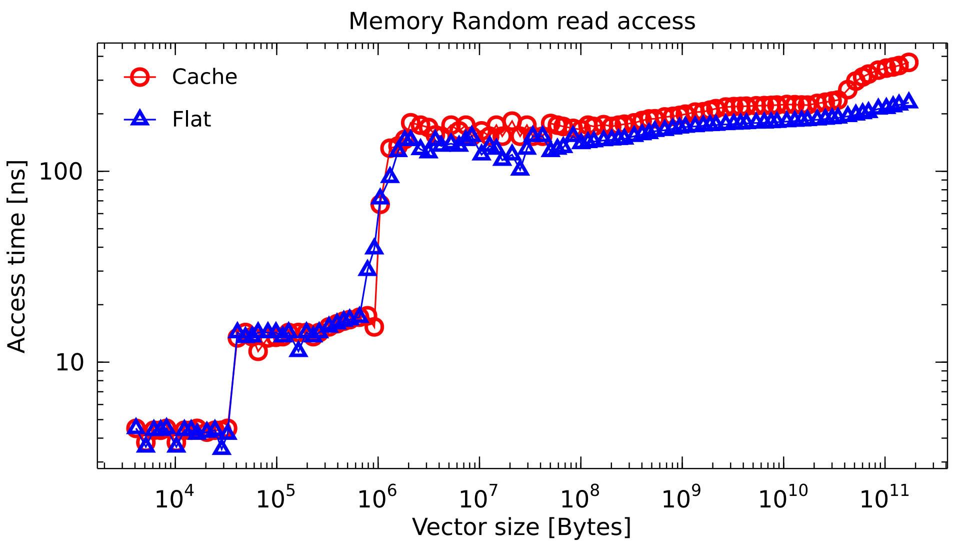
The first mode is called "Flat Mode". This mode allows the MCDRAM to have a physical addressable space. This in turn allows the programmer to migrate data structures in and out of the MCDRAM. This can be useful to keep large structures in DDR4 and smaller structures in MCDRAM. This mode offers the highest bandwidth and lowest latency. Another advantage is that the 16GB of MCDRAM are seen as addressable, hence increasing the total addresable memory in a KNL system. The downside of the flat mode is that software modifications are required in order to use both the DDR and the MCDRAM in the same application. The application has to maintain and keep track of what data goes where, increasing software design and maintenance costs. One consequence of the "Flat mode" is that the KNL system is seen as a two-node NUMA domain, similar to a dual-socket Xeon system. The memory is allocated by default in the DDR memory in order to keep non-critical data out of the MCDRAM. Applications that explicitly want to allocate data in MCDRAM have two ways to achieve this: (1) the "fast malloc" functions from the Memkind library: http://memkind.github.io/memkind/, and (2) the "FASTMEM" compiler annotation for Intel Fortran.

2.2.2. Cache mode

The second mode is called "Cache Mode". When using this mode, the application code can remain unchanged. The OS will organize the data to use the MCDRAM similar to an L3 cache, caching data from the DDR4 level of memory. The obvious benefit is increased bandwidth, as after careful application design most memory request will hit in this huge 16GB L3 cache. The downside here is when the MCDRAM experiences cache misses, the missed requests have to be communicated back into the die and then another request is issued out into DDR for the relevant memory, leading to increased latency. This means that an MCDRAM cache miss is more expensive than a simple read out to DDR. Another downside of this mode is that all memory needs to be transferred, from DDR, to MCDRAM, and finally to the L2 cache, see Figure 6 with access for large vector sizes. Serving as a cache,the 16GB MCDRAM are not addressable, leading to less system-addressable memory.

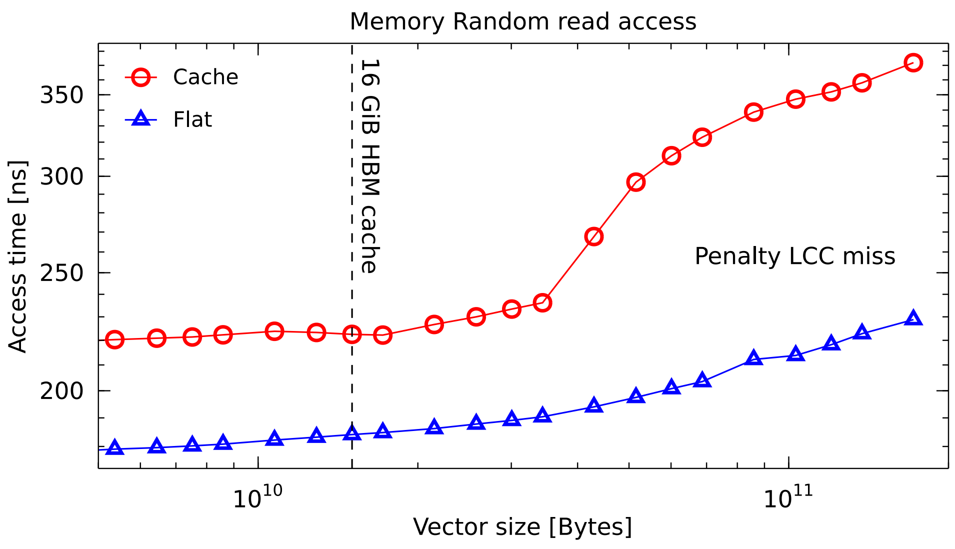
The figure below show the random access time when reading some random located data from memory using two different HBM layouts. As exptected the access time is lower in the Flat mode case as there is no cache memory management to pass through. The cache sizes are clearly visible. The jump at 16 GiB where the size of the High Bandwidth memory is exhausted is rather small, implying that the latency of the HBM is just slightly less of the DDR4 main memory. It is also evident that direct access to memory is faster than cached access for large data vectors. We see that for allocated vectors in excess of 16 GiB the random access time to memory increases when HBM is used as last level cache.

Figure 6. Random read memory access



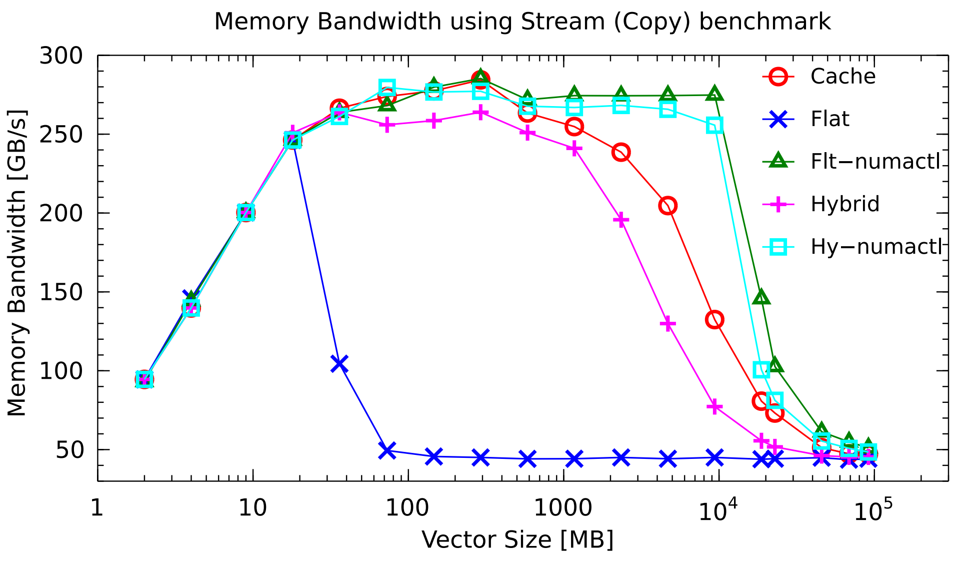
From the figure above we can observe that the cache coherency has a significant cost with respect to memory random access times. All memory transactions need to be handled by the cache cohenrency machinery and incure increased access times. In the figure below the random access times for larger vector sizes is shown. It is evident that the penalty for a cache miss in the Last Level Cache (the High Bandwidth Memory) can be quite significant when accessing memory randomly outside the cached part. In selected cases when the memory access pattern is random with unknown stride the Flat memory profile might yield better performance then using the HBM as a cache. The same is of course also true when the programmer is in full control over memory allocation. Some users are in full control using numactl and related tools other are unaware of such tools. Hence the system wide settings must be set to suite the users of the system. Administrators should be prepared to change the setting upon request.

Figure 7. Random read memory access for large vectors



The figure below show the effect of High Bandwidth memory settings using the Stream memory bandwidth benchmark[1]. Here the benefit of the HBM is evident. Bypassing the cache coherency machinery seems to provide higher bandwidth for larger vector sizes, but require the users to be aware of the numa memory and allocate memory accordingly. It's also interesting to note that the DDR4 bandwidth is rather limited. This is a property of the chipset and motherboard. This system is a Ninja dvelopment system and one might expect HPC-compute nodes to have a better motherboard design.

Figure 8. Memory Bandwidth using the Stream benchamrk (Copy test)



[1] Run without and with numactl, *numactl --preferred=1 ./stream.x* , where numanode 1 is HBM.

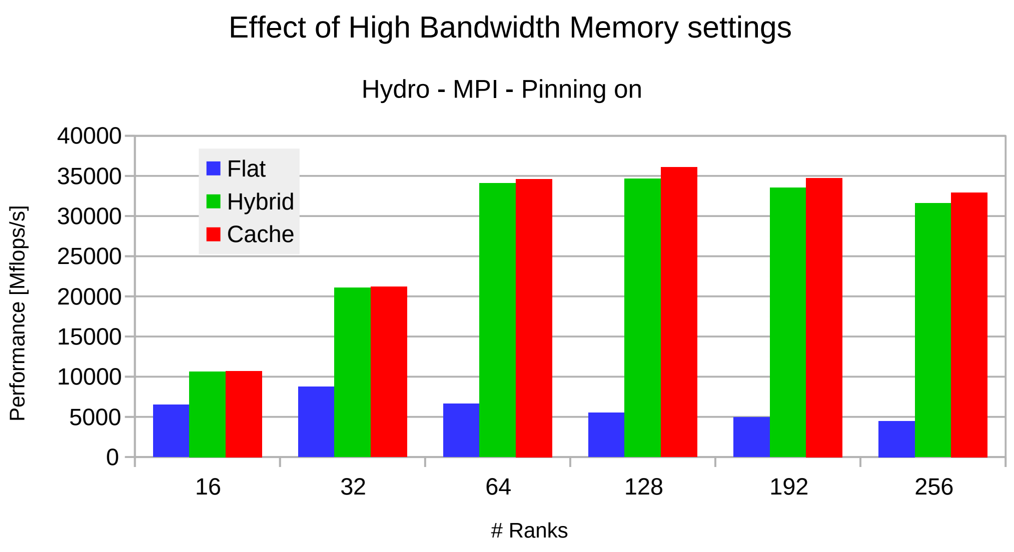
2.2.3. Hybrid mode

The third mode is called "Hybrid Mode". This mode allows the MCDRAM to be split in two, part of it to be used in "Cache Mode", and the other part in "Flat mode", using the High bandwidth Memkind library: http://memkind.github.io/memkind/. There is also an interposer library over Memkind available called AutoHBW which simplifies some of the commands at the expense of fine control. Under Memkind/AutoHBW, data structures aimed at MCDRAM have their own commands in order to be generated in MCDRAM.

2.2.4. Optimal Settings

Of course, the best setting is application dependent. However, based on the characteristics of the application, there are three alternatives. If the whole application fits in the 16GB MCDRAM, it is advised to run the whole program in MCDRAM, by mapping it to the appropriate NUMA domain with numactl. This will give the highest possible bandwidth. If the application does not fit in MCDRAM, but the data can be easily split between critical/non-critical, the proper solution is to use the the Memkind library and manually allocate the bandwidth-critical memory to MCDRAM. Finally, if the data cannot be easily split, or if it is desired that the codebase is left unmodified, the "Cache mode" should be employed. An example using the HYDRO benchmark in MPI implementation show the effect of High Bandwidth Memory settings.

Figure 9. Effect of High Bandwidth Memory settings



Switching between these modes require change in the BIOS settings and a subsequent reboot. The command to alter the BIOS setting from command line is described in Section 2.3.6.

2.2.5. Example: Programming with the memkind library

An example with Fortran code can be found in Section 7.8.2.

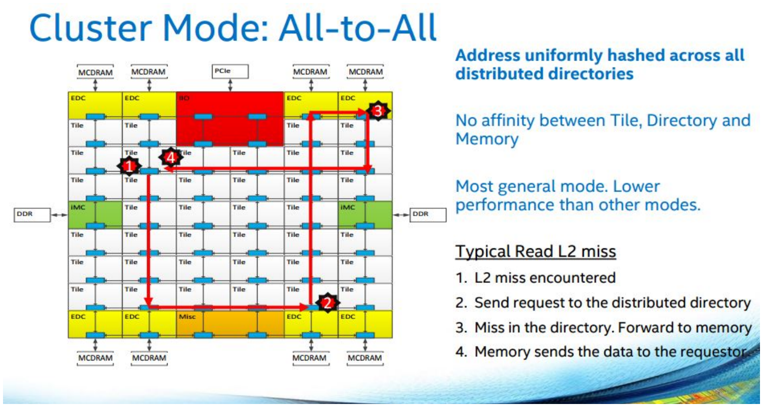
2.3. Clustering modes

As noted before, in KNL each of its cores has an L1 cache, pairs of cores are organized into tiles with a slice of the L2 cache symmetrically shared between the two cores, and the L2 caches are connected to each other with a mesh. All caches are kept coherent by the mesh with the MESIF protocol. In the mesh, each vertical and horizontal link is a bidirectional ring. To maintain cache coherency, KNL has a distributed tag directory, organized as a set of per-tile tag directories (TDs), which identify the state and the location on the chip of any cache line. For any memory address, the hardware can identify with a hash function the TD responsible for that address. To manage this complexity and set the optimal mode of operation for any given computational application, the programmer has access to cache clustering modes. The Knights Landing interconnecting mesh operates in one of three clustering modes: all-to-all, quadrant, and sub-NUMA. These modes are selected at boot-time, Intel providing no way to modify this setting without restarting the system.

2.3.1. All to all mode

When using the all-to-all clustering mode, depicted in Figure 10 the memory addresses are uniformly distributed across all tag directories in the chip. This is the most general mode with the easiest programming model, but it will offer lower performance than the other modes.

Figure 10. All-to-all clustering mode.



Cluster Mode: All-to-All

Address uniformly hashed across all distributed directories

No affinity between Tile, Directory and Memory

Most general mode. Lower performance than other modes.

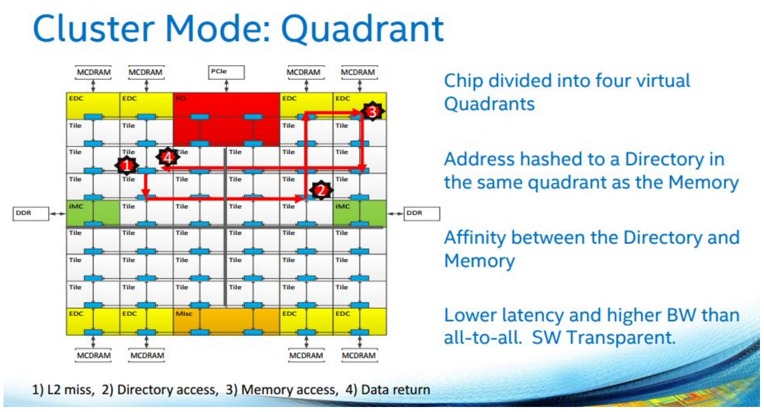
Typical Read L2 miss

1. L2 miss encountered
2. Send request to the distributed directory
3. Miss in the directory. Forward to memory
4. Memory sends the data to the requestor

2.3.2. Quadrant mode

In quadrant/hemisphere modes, depicted in Figure 11, the Knights Landing chip is divided into two (hemispheres) or four parts (quadrants), and addresses are hashed to directories in the same hemisphere or quadrant, the quadrants being spatially local to the four groups of memory controllers. The operating system is not aware that this is going on underneath the covers, and it provides lower latency and higher bandwidth for the cores running in each hemisphere/quadrant. In the quadrant and hemisphere modes, the latency of L2 cache misses is reduced compared to the all-to-all mode.

Figure 11. Quadrant clustering mode.



Cluster Mode: Quadrant

Chip divided into four virtual Quadrants

Address hashed to a Directory in the same quadrant as the Memory

Affinity between the Directory and Memory

Lower latency and higher BW than all-to-all. SW Transparent.

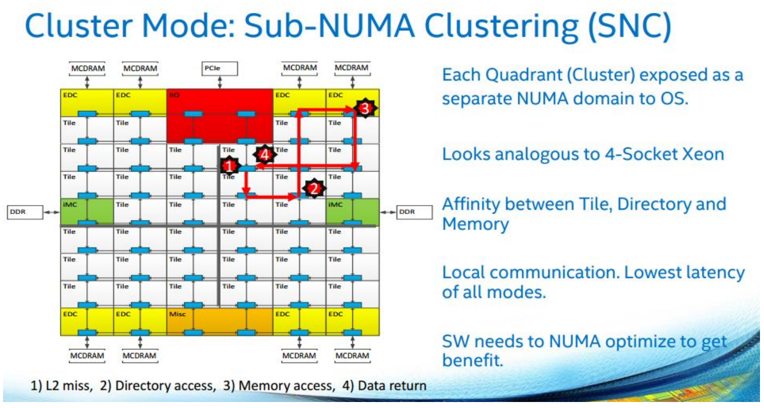
1) L2 miss, 2) Directory access, 3) Memory access, 4) Data return

2.3.3. Sub NUMA mode

In the sub-NUMA cluster modes, depicted in Figure 12, the operating system exposes all four quadrants as virtual NUMA clusters. This makes a Knights Landing chip look like a four-socket Xeon server. This mode provides the lowest latency, provided that applications are NUMA-aware and hence thread and memory pinning are employed.

One important detail is that if cache traffic crosses the NUMA boundaries, using sub-NUMA clustering is less efficient than using the quadrant mode.

Figure 12. Sub-NUMA clustering mode.



Cluster Mode: Sub-NUMA Clustering (SNC)

Each Quadrant (Cluster) exposed as a separate NUMA domain to OS.

Looks analogous to 4-Socket Xeon

Affinity between Tile, Directory and Memory

Local communication. Lowest latency of all modes.

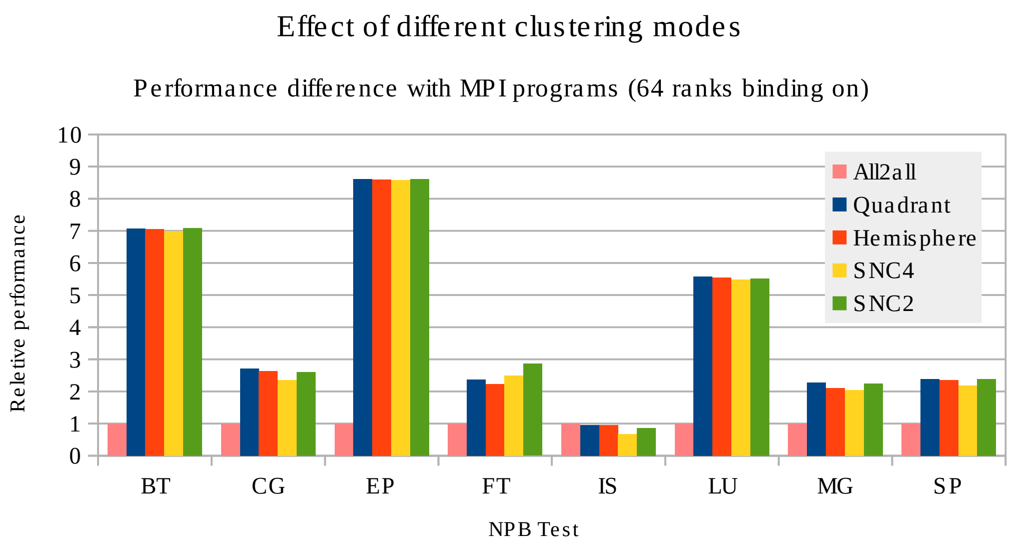
SW needs to NUMA optimize to get benefit.

1) L2 miss, 2) Directory access, 3) Memory access, 4) Data return

2.3.4. Performance using the different modes

Performance varies with the different modes, the following figure give an example of the differences one might expect when running MPI applications. The suggested setting from Intel for daily work is Quadrant.

Figure 13. Example MPI Performance with different modes



2.3.5. Example: Programming with sub-NUMA clusters

This is very much like programming a dual socket or four socket Intel system. Each of the sockets has its own local memory e.g. NUMA node. A four socket system will present four NUMA banks, when listed using the *numactl -H*. There a numerous example of NUMA programming available and programming a KNL set up with sub NUMA clusters are almost identical.

2.3.6. BIOS settings for clustering, memory etc

As with the memory modes switching betwene these core layout modes require BIOS setting changes and subsequent reboot. This can be done remote via a IPMI redirected serial console, a web based console or using a command line tool.

2.3.6.1. CLI tools for BIOS settings

The tool *Syscfg* can be used to update the BIOS settings from the command line. It require the OpenIPMI tool to be installed as a prerequisite. The Syscfg tool can be found at Intel's web site [https://downloadcenter.intel.com/download/25439/Save-and-Restore-System-Configuration-utility-syscfg]. Once installed it provides a range of useful functions.

To save all the BIOS (and BMC) settings to a text file use the following command:

```
syscfg /s BIOSBMC.ini /b /f
```

From the saved file names of the relevant BIOS settings can be extracted and used as inputs to update the BIOS. After editing this config file the updated config file can be used to set the BIOS parameters.

```
syscfg /r BIOSBMC.ini /b /f
```

A power off is generally adviced. This is done in the normal way using IPMI.

To verify the settings the utility *hwloc* can be used. Just issue the command:

```
hwloc-dump-hwdata | egrep 'Cluster|Mode'
```

and the output should look something like :

```
Cluster mode: Quadrant
Memory Mode: Cache
```

2.4. (Node) Interconnect

2.5. I/O Subsystem Architecture

From an I/O perspective, the three Knights Landing variants have different capabilities. The basic KNL chip has 36 PCIe 3.0 lanes available. When using the KNL co-processor, next to a Xeon host, one has the advantage that it can include more than one KNL chip inside a compute node. However, a major disadvantage when using the KNL cards is that only the 16GB MCDRAM are available as fast storage. The system memory needs to be accessed through PCIe, at a much lower rate. Perhaps the most interesting KNL variant from an I/O perspective is the KNL version that features two on-package integrated OmniPath fabric ports. This will drive data among KNL nodes at 100Gb/s.

3. Programming Environment / Basic Porting

3.1. Default System settings

The default settings are as suggested by Intel for every day general use, Chip layout set to "Quadrant" and High Bandwidth Memory to "Cache". This yield a system with a single memory image e.g. no NUMA and a rather large Last Level Cache LLC.

To change the settings please refer to sections in Section 2.3.6 and Section 7.9.

3.2. Available Compilers

A set of different compilers are available on the system. All support C, C++ and FORTRAN (90 and 2003) all with OpenMP threading support. For information about OpenMP support please refer to Section 3.5

Compilers installed:

- Intel compiler suite, *icc, ifortran, icpc*

- GNU compiler suite, *gcc*, *gfortran*, *g++*

3.2.1. Compiler Flags

We assume the users are familiar with the common flags for output, source code format, preprocessor etc. A nice overview of general compiler usage is found in the Best Practice Guide for x86 [http://www.prace-ri.eu/Best-Practice-Guide-Generic-x86-HTML]. The C compiler flags are also covered in this BPG.

Only the default flags will be listed and only for the FORTRAN compilers. Only Intel and GNU are covered as they make up the vast majority of usage. For C some flags may differ. For a discussion of the common flags for optimization please refer to Section 7.1

Only the default flags for the Intel FORTRAN compiler is given below. The C compiler will have a set of default compiler flags quite similar. As for the GNU gfortran and gcc the defaults are even more elaborate then Intel's and cannot be covered in detail here.

Table 1. Default Intel FORTRAN compiler flags

Default flag	Description
-O2	Optimize for maximum speed
-f[no-]protect-parens	enable/disable(DEFAULT) a reassociation optimization for REAL and COMPLEX expression evaluations by not honoring parenthesis.
-f[no-]omit-frame-pointer	enable(DEFAULT)/disable use of EBP as general purpose register. -fno-omit-frame-pointer replaces -fp
-f[no-]exceptions	enable(DEFAULT)/disable exception handling
-[no-]ip	enable(DEFAULT)/disable single-file IP optimization within files
-[no-]scalar-rep	enable(DEFAULT)/disable scalar replacement (requires -O3)
-[no]pad	enable/disable(DEFAULT) changing variable and array memory layout
-[no-]ansi-alias	enable(DEFAULT)/disable use of ANSI aliasing rules optimizations; user asserts that the program adheres to these rules
-[no-]complex-limited-range	enable/disable(DEFAULT) the use of the basic algebraic expansions of some complex arithmetic operations. This can allow for some performance improvement in programs which use a lot of complex arithmetic at the loss of some exponent range.
-[no-]ansi-alias	enable(DEFAULT)/disable use of ANSI aliasing rules optimizations; user asserts that the program adheres to these rules
-[no-]complex-limited-range	enable/disable(DEFAULT) the use of the basic algebraic expansions of some complex arithmetic operations. This can allow for some performance improvement in programs which use a lot of complex arithmetic at the loss of some exponent range.
-no-heap-arrays	temporary arrays are allocated on the stack (DEFAULT)
-[no-]vec	enables(DEFAULT)/disables vectorization
-coarray	enable/disable(DEFAULT) coarray syntax for data parallel programming
-q[no-]opt-matmul	replace matrix multiplication with calls to intrinsics and threading libraries for improved performance (DEFAULT at -O3 -parallel)
-[no-]simd	enables(DEFAULT)/disables vectorization using SIMD directive
-qno-opt-prefetch	disable(DEFAULT) prefetch insertion. Equivalent to -qopt-prefetch=0
-qopt-dynamic-align	enable(DEFAULT) dynamic data alignment optimizations. Specify -qno-opt-dynamic-align to disable
-[no-]prof-data-order	enable/disable(DEFAULT) static data ordering with profiling
-pc80	set internal FPU precision to 64 bit significand (DEFAULT)

Default flag	Description
-auto-scalar	make scalar local variables AUTOMATIC (DEFAULT)
-[no]zero	enable/disable(DEFAULT) implicit initialization to zero of local scalar variables of intrinsic type INTEGER, REAL, COMPLEX, or LOGICAL that are saved and not initialized
-init=<keyword>	enable/disable(DEFAULT) implicit initialization of local variables of intrinsic type INTEGER, REAL, COMPLEX, or LOGICAL that are saved and not initialized The <keyword> specifies the initial value keywords: zero (same as -zero), snan (valid only for floating point variables), arrays
-Zp[n]	specify alignment constraint for structures (n=1,2,4,8,16 -Zp16 DEFAULT)
-fstack-security-check	enable overflow security checks. -fno-stack-security-check disables (DEFAULT)
-fstack-protector	enable stack overflow security checks. -fno-stack-protector disables (DEFAULT)
-fstack-protector-strong	enable stack overflow security checks for routines with any buffer. -fno-stack-protector-strong disables (DEFAULT)
-fstack-protector-all	enable stack overflow security checks including functions. -fno-stack-protector-all disables (DEFAULT)
-fpic, -fPIC	generate position independent code (-fno-pic/-fno-PIC is DEFAULT)
-fpie, -fPIE	generate position independent code that will be linked into an executable (-fno-pie/-fno-PIE is DEFAULT)
[no-]global-hoist	enable(DEFAULT)/disable external globals are load safe
-f[no-]keep-static-consts	enable/disable(DEFAULT) emission of static const variables even when not referenced
-mcmodel=<size>	use a specific memory model to generate code and store data. • small - Restricts code and data to the first 2GB of address space (DEFAULT) • medium - Restricts code to the first 2GB; it places no memory restriction on data • large - Places no memory restriction on code or data
-falign-functions=[2|16]	align the start of functions on a 2 (DEFAULT) or 16 byte boundary

The performance impact of the different flags differ, and there are not really an optimal set of flags that suits all.

For the Intel compiler experience have shown that some flags are generally a good starting option. Intel have done a good job to ensure that the default flags are adapted to a range of applications. The vectorization is enabled per default, but we can force the compiler to generate code for our target architecture, in this case the KNL and the corresponding flag MIC-AVX512 (-xMIC-AVX512). The fused multiply add -fma is almost always yielding better performance. Another important flag is -O3, while not always yield better performance than -O2, it usually does. Aligning is another important flag to use, even though the penalty for a misaligned load is not as high as it use to be. With a 64 byte cache line the flag -align array64byte is useful. Inlining of functions can enable vectorization of some functions so -finline-functions might provide a higher degree of vectorization.

Flags like -ipo and loop unrolling are discussed in the tuning section as they they are more in the application tuning section.

A suggested set of flags for the Intel compilers are given in the table below.

Table 2. Suggested compiler flags for Intel compilers

Compiler	Suggested flags
Intel C compiler	-O3 -xMIC-AVX512 -fma -align -finline-functions
Intel C++ compiler	-std=c11 -O3 -xMIC-AVX512 -fma -align -finline-functions
Intel Fortran compiler	-O3 -xMIC-AVX512 -fma -align array64byte -finline-functions

With a system with the High Bandwidth Memory set to cache the streaming stores should be disabled, *-qopt-streaming-stores=never*. The HBM is a memory side cache and cannot be bypassed. The cache line have to be loaded into the HBM cache in any case. See Figure 16 and associated text in Section 4.1 for a discussion about this.

Below is a table showing the most relevant default flags set by the GNU compilers.

Table 3. Default GNU Fortran compiler flags

Default flag	Description
-O0	Reduce compilation time and make debugging produce the expected results.
-fno-inline	Do not expand any functions inline apart from those marked with the "always_inline" attribute.
-mcmodel=small	Generate code for the small code model. The program and its statically defined symbols must be within 4 GiB of each other. Pointers are 64 bits. Programs can be statically or dynamically linked.
-funderscoring	By default, GNUFortran appends an underscore to external names.
-fno-protect-parens	By default the parentheses in expression are honored for all optimization levels such that the compiler does not do any re- association. Using -fno-protect-parens allows the compiler to reorder "REAL" and "COMPLEX" expressions to produce faster code.

The GNU compiler set of default flags are not aggressively optimized for performance as is the default flags set by Intel. Some more investigation is generally needed with the GNU compilers.

A suggested set of flags for the GNU set of compilers are given in the table below.

Table 4. Suggested compiler flags for GNU compilers

Compiler	Suggested flags
gcc compiler	-march=knl -O3 -mavx512f -mavx512pf -mavx512er -mavx512cd -mfma -malign-data=cacheline -finline-functions
g++ compiler	-std=c11 -march=knl -O3 -mavx512f -mavx512pf -mavx512er -mavx512cd -mfma -malign-data=cacheline -finline-functions
gfortran compiler	-O3 -march=knl -mavx512f -mavx512pf -mavx512er -mavx512cd -mfma -malign-data=cacheline -finline-functions

The flags setting the vector capabilities (-mavx512) let the compiler generate code for the 512 bits wide vector registers, it is assumed that any former instruction subset is included, e.g. that also SSE, AVX instructions are generated.

3.3. Available (Vendor Optimized) Numerical Libraries

The most common for Intel based systems are Intel Math Kernel Library, MKL and the Intel Performance Primitives. MKL contain a large range of high level functions of like Basic Linear Algebra, Fourier Transforms etc, while IPP contains a large number of of more low level functions like converting, scaling etc. For even lower level functions like scalar and vector versions of simple functions like square root, logarithmic and trigonometric there are libraries like libimf and libsvml.

3.3.1. Math Kernel Library, MKL

Intel MKL is installed with the compiler suite and are an integrated part of the compiler suite. The simplest way of enabling the MKL is to just issue the flag *-mkl*, this will link the default version of MKL, for a multicore system this will be the threaded version (*-mkl=parallel*). The actual link line is hidden and not presented when using this shortcut. On a KNL system using a parallel MKL will effectively turn a pure MPI application into a hybrid application.

There are both a single threaded sequential version and a threaded multicore version. Most pure MPI jobs just require *-mkl=sequential*. To link with Intel MKL cluster components (sequential) that use Intel MPI use the *cluster* option. If ScaLapack routines are needed, use the *-mkl=cluster* option. The *-mkl=cluster* flag links to the sequential routines of the libraries, if the ScaLapack with threaded libraries are needed, you need to provide the correct combinations of the libraries to the link line. Please refer to the examples below or the Intel MKL link line advisor (see 3.3.1.2).

The sequential and cluster version are mostly used with none hybrid MPI programs. The parallel version can be controlled using the environment variable OMP_NUM_THREADS.

In some cases the user want to control the amount of High Bandwidth Memory that MKL allocate. Setting *MKL_FAST_MEMORY_LIMIT=0* will prevent MKL from allocating memory in the High Bandwidth Memory.

Table 5. Invoking different versions of MKL

MKL Version	Link flag
Single thread, sequential	-mkl=sequential
Single thread MPI, Sequential	-mkl=cluster
Multi threaded	-mkl=parallel or -mkl

MKL contains a range of functions and other libraries. Several of the common widely used libraries and functions have been incorporated into MKL. Intel provide a large set of documentations etc about MKL, see Intel MKL documentation [https://software.intel.com/en-us/articles/intel-math-kernel-library-documentation].

Libraries contained in MKL:

- BLAS, BLAS95 and Sparse BLAS
- FFT and FFTW (wrapper)
- LAPACK
- Direct and Iterative Sparse Solvers, including PARADISO
- ScaLAPACK with BLACS
- Vector Math
- Vector Statistics Functions

A large fraction of the common functions that are needed for development are part of the MKL, Basic linear algebra (BLAS 1,2,3), FFT and the wrappers for FFTW. Software often require the FFTW package, with the wrappers there is no need to install FFTW which is outperformed by MKL in most cases.

3.3.1.1. MKL examples

MKL is very simple to use, the functions have simple names and the parameter lists are well documented. An example of calling a matrix matrix multiplications is show below:

```
write(*,*)"MKL dgemm
call dgemm('n', 'n', N, N, N, alpha, a, N, b, N, beta, c,N)
```

or with fftw syntax:

```
call dfftw_plan_dft_r2c_2d(plan,M,N,in,out,FFTW_ESTIMATE)
call dfftw_execute_dft_r2c(plan, in, out)
```

The calling syntax is just as for dgemm from reference Netlib BLAS implementation and the widely used FFTW. The same apply to the other functions, calling and parameter list is kept as close to the reference implementation as practically possible.

The usage and linking sequence of some of the libraries can be somewhat tricky. Please consult the Intel compiler and MKL documentation for details. An example from building the application VASP is given below (list of object files object contracted to *.o):

```
mpiifort -mkl -lstdc++ -o vasp *.o -Llib -ldmy\
-lmkl_scalapack_lp64 -lmkl_blacs_intelmpi_lp64 -lfftw3xf_intel_lp64
```

or even simpler for a fftw example:

```
ifort fftw-2d.f90 -o fftw-2d.f90.x -mkl
```

3.3.1.2. Intel MKL link line advisor

Since the link line can be a bit complicated Intel has produced a nice tool to help making the correct link line, Intel MKL linking advisor [https://software.intel.com/en-us/articles/intel-mkl-link-line-advisor].

3.3.2. Intel Performance Primitives, IPP

Performance primitives library contain functions that do simple operations on scalars, vectors and smaller matrices. It is best suited for image and signal processing, data compression and cryptography.

Intel IPP offers thousands of optimized functions covering frequently used fundamental algorithms including those for creating digital media, enterprise data, embedded, communications, and scientific/technical applications.

The library contains a huge number for functions with a range of data types. Please refer to the documentation for a detailed description. Intel provide online documentation for IPP [https://software.intel.com/en-us/intel-ipp].

Examples provided with the library includes a high performance compress, gzip and bzip program. Another example of usage is shown below:

```
#include "ipp.h"
#include "ippcore.h"
#include "ipps.h"
#include <stdio.h>
#include <math.h>
#include <stdint.h>

void libinfo(void) {
  const IppLibraryVersion* lib = ippsGetLibVersion();
  printf("%s %s %d.%d.%d.%d\n",
  lib->Name, lib->Version,
  lib->major,
  lib->minor, lib->majorBuild, lib->build);
}

#define N 207374182

main() {
  int16_t x[N];
  __declspec(align(64))  float a[N],b[N],c[N];
  float sum;
  int i,j;
  double t0,t;
  extern double mysecond();

  libinfo();

  printf("Vector sizes int-vec %ld float vec %ld\n",(long)N*2, (long)N*4);
  for (j=0; j<N; j++) x[j]=1;

  t0=mysecond();
  ippsConvert_16s32f(x, a, N);
  t=mysecond()-t0;
  printf("Convert time ipp %lf sec.\n",t);

  t0=mysecond();
  ippsCopy_32f(a, b, N);
  ippsAdd_32f_I(b, c , N);
  ippsMul_32f_I(b, c , N);
  ippsAddProduct_32f(a, b, c, N);
  ippsSqrt_32f(c, c, N);
  ippsSum_32f(c, N, &sum, 1);

  for (j=0; j<10; j++) printf("%d : %f\n",j,c[j]);
  printf("IPP: Sum %10.0f time %lf sec Flops %ld %lf Gflops/s\n",
  sum, t, (long)N*2, ((double)N*10/t)/1e9);
}
```

3.3.3. Math library, libimf

Intel Math Libraries, in addition to libm.{a,so}, the math library provided with gcc and Linux.

Both of these libraries are linked in by default because certain math functions supported by the GNU math library are not available in the Intel Math Library. This linking arrangement allows the GNU users to have all functions available when using ifort, with Intel optimized versions available when supported.

This is why libm always shows up when using the Intel compilers, even when you linked with libimf (even before libm) when you check the library dependencies using ldd.

Performance of the libimf is generally better that using libm from gcc. The functions is optimized for use with Intel processors and seems to be well optimized. An example of performance gain can be found at Intel's web site, optimizing-without-breaking-a-sweat [https://software.intel.com/en-us/articles/optimizing-without-breaking-a-sweat], where they claim very high gains when making heavy use of the *pow* function. The simple test below:

```
for(j=0; j<N; j++) c[j]=pow(a[j],b[j]);
```

show a speedup of 3x when compiled with gcc 5.2.0 and linked *-limf* instead of the more common *-lm*.

3.3.4. Short vector math library, libsvml

The short vector math library functions take advantage of the vector unit of the processor and provide an easy access to well optimized routines that map nicely on the vector units.

The svml is linked by default and the functions are not easily available directly from source code. They are, however accessible through intrinsics. Intel provide a nice overview of the intrinsic functions available [https://software.intel.com/en-us/node/524288]. Usage of intrinsics can yield quite good performance gain. In addition intrinsics are compatible with newer versions of processors as the compilers and libraries and updated while the names stay the same. Usage of inline assembly might not be forward compatible. Intel strongly suggest using intrinsics instead of inline assembly.

A simple example of intrinsic usage is show below:

```
for(j=0; j<N; j+=8){
  __m512d vecA = _mm512_load_pd(&a[j]);
  __m512d vecB = _mm512_load_pd(&b[j]);
  __m512d vecC = _mm512_pow_pd(vecA,vecB);
  _mm512_store_pd(&c[j],vecC);
}
```

The performance can be quite good compared to the serial code below.

```
for(j=0; j<N; j++) c[j]=pow(a[j],b[j]);
```

Mostly due to the difference in calls to *svml_d_pow8* vector function and the libimf serial *pow* function. However, at high optimization the compiler will recognize the simple expression above and vectorize it and performance gain will be lesser. The usage of these intrinsics is at its best when the compiler totally fails to vectorize the code.

If you want to play with this there is a blog entry by Kyle Hegeman [http://kylehegeman.com/blog/2013/12/27/using-intrinsics/] that will be helpful.

3.4. Available MPI Implementations

There are two different MPI implementations installed.

MPIs installed:

- Intel compiler MPI
- OpenMPI

The two has quite common syntax during compiling and simple runs. The mpirun have a range of options, which is quite different. Please consult the help files and documentation for details about these options. For most runs the queue system will set up the run in a sensible way.

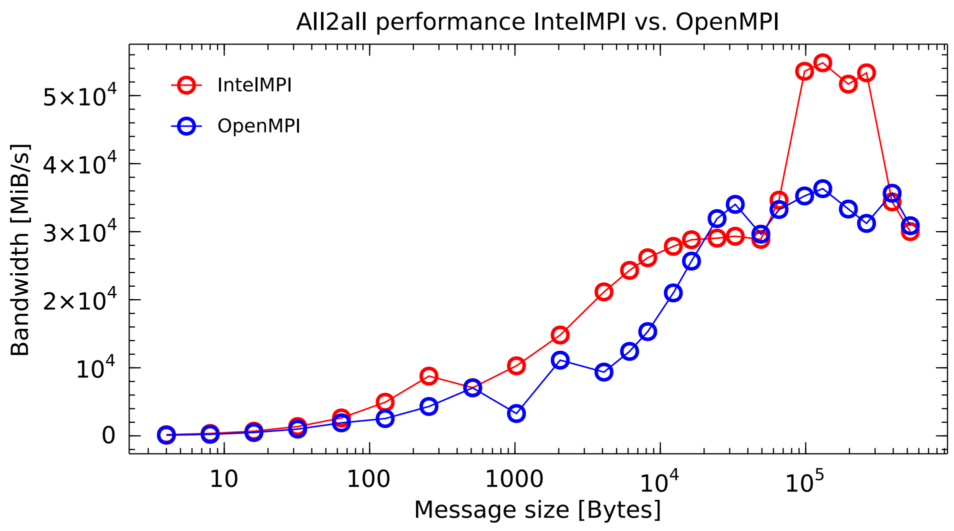
Table 6. Implementations of MPI

MPI library	MPI CC	MPI CXX	MPI F90
Intel MPI	mpiicc	mpiicpc	mpiifort
OpenMPI	mpicc	mpicxx	mpifort

OpenMPI is built with a specific compiler suite so the mpicc, mpicxx and mpifort wrappers will invoke the compilers used during the build. This can be overrun with environment variables, but FORTRAN modules (expressions like "use mpi") might not work as expected.

The two common MPI implementations Intel MPI and OpenMP are both fully supported with Knights Landing. Both perform good with some minor differences which is evident in the figure below. The all2all test is just one of a large range of MPI functions, but as this guide is not a benchmark report we show just an example that they both perform on KNL.

Figure 14. Comparing two MPI implementations



3.5. OpenMP

OpenMP is are supported with all the above compilers, Intel, GNU.

Table 7. Versions of OpenMP supported

Compiler suite	Compiler version	Version supported
Intel	2017.beta	OpenMP 4.0
GNU	6.1.0	OpenMP 4.0

3.5.1. Compiler Flags

Table 8. OpenMP enabling flags

Compiler	Flag to enable OpenMP
Intel	-qopenmp
GNU	-fopenmp

A common set of flags with the Intel Fortran compiler ifort is:

-O3 -xMIC-AVX512 -qopenmp -align array64byte -fma -ftz -finline-functions

and for the C/C++ icc and icpc

-O3 -xMIC-AVX512 -qopenmp -align -fma -ftz -finline-functions.

4. Benchmark performance

Benchmark performance is used to show the different performance aspect of a processor.

4.1. STREAM benchmark

Stream is a well known benchmark for measuring memory bandwidth written by John D. Calpin of TACC. TACC also happen to host the large supercomputer system called “Stampede” which is an accelerated system using a large array of Intel Xeon Phis.

The system is set up in quadrant mode and High Bandwidth Memory is used as a last level cache.

Stream can be built in several ways, it turned out that static allocation of the three vectors of which to operate on provided the best results. The source code illustrate how the data is allocated:

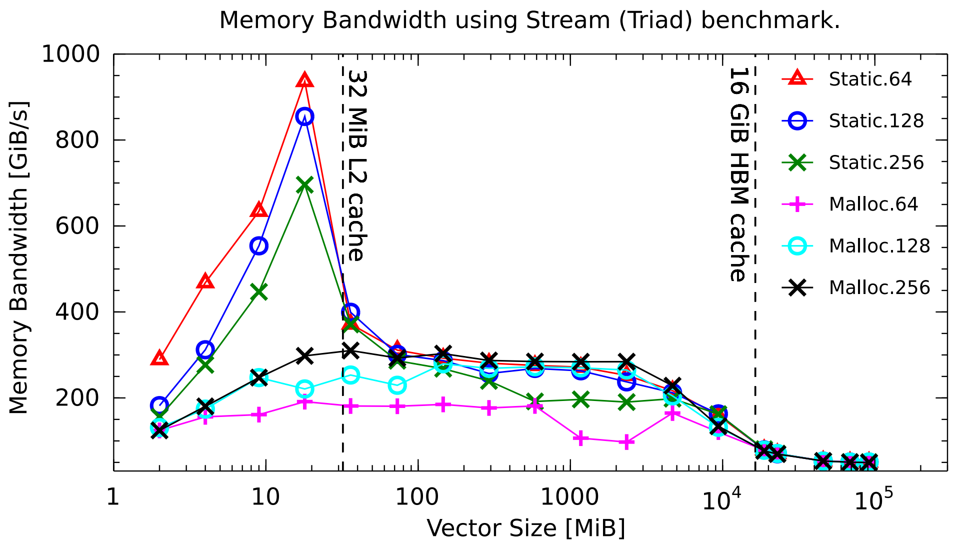
```
#ifndef USE_MALLOC
static double a[N+OFFSET], b[N+OFFSET],c[N+OFFSET];
#else
static volatile double *a, *b, *c;
#endif
#ifdef USE_MALLOC
a = malloc(sizeof(double)*(N+OFFSET));
b = malloc(sizeof(double)*(N+OFFSET));
c = malloc(sizeof(double)*(N+OFFSET));
#endif
```

The C version of benchmark was compiled using Intel icc with the following flags used: *-O3 -xMIC-AVX512 -fomit-frame-pointer -qopenmp -fma*.

The figure below show the difference between using malloc and static allocation, when running the Stream benchmark with a range of threads, ranging 64 to 256, e.g. 1 to 4 threads per core. In order to use all Level 2 cache in all cores the threads have been scattered over most possible cores using the environment variable *KMP_AFFINITY* set to *scatter*.

It's evident that L2 cache has a very high bandwidth as does the Last Level Cache (High Bandwidth Memory). All subsequent runs using stream were done using static allocation.

Figure 15. Memory bandwidth using Static allocation versus Malloc allocation



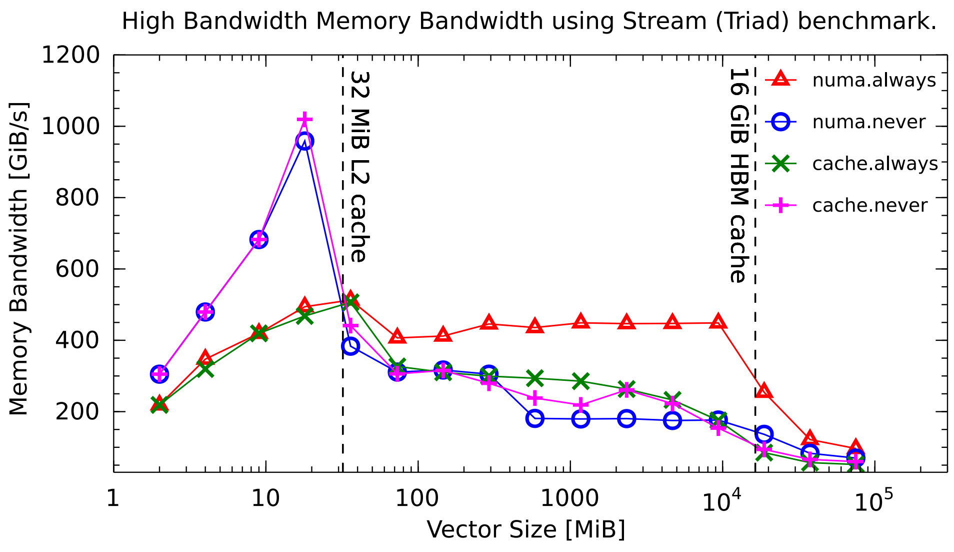
When running memory bandwidth benchmarks like stream the option to bypass the cache by using store instruction (*-qopt-streaming-stores=always* that store directly into memory can have an impact on performance. The figure below show the impact of using streaming store instructions that effectively bypass the cache. For small vector sizes there is a penalty when bypassing the cache. MCDRAM as cache is strictly inclusive. Every cache line has to be allocated in MCDRAM cache which basically mean that even in the case of nontemporal stores. The system need to do an extra read to MCDRAM thus wasting bandwidth for unnecessary reads (essentially like a regular store from the point-of-view of MCDRAM). Hence the bandwidth difference between the NT stores to flat MCDRAM and MCDRAM as cache. Since, from the point-of-view of the CPU, streaming stores are more complicated and require more complex micro ops to function, they really do not give that much benefit as their main benefit is negated by the inclusivity of the MCDRAM cache. It can look like the last level cache (the High Bandwidth Memory) is not bypassed by the streaming store instructions.

The figure below show the obtained bandwidth numbers when using the stream benchmark with 64 scattered threads. In the NUMA cases the HBM memory is set as a separate NUMA bank and numactl is used to preferably allocate from this bank.

```
numactl --preferred=1 ./stream.large.x
```

The benchmark is compiled using streaming stores always or never (*-qopt-streaming-stores=always or never*).

Figure 16. High Bandwidth Memory bandwidth NUMA vs. Cache / streaming store instructions or not



From the figure above it's beneficial to not use streaming stores for small size accesses, obvious for the cases where the chunk fit in the L2 cache. However in the case of flat memory and chunks larger than the L2 cache it's yielding significantly higher bandwidth numbers.

4.2. NAS kernel benchmark, NPB

The NAS benchmarks are well known. They are mostly known as MPI benchmarks, but they have been rewritten to OpenMP version and other parallel implementations.

As the KNL is a general processor it can run both OpenMP threaded shared memory code or distributed memory code like MPI. Hence both implementations have been tested. Most attention to the threaded version as this is more interesting with a cache coherent shared memory system.

For this kind of benchmarks based on real application all optimization comes into play, prefetch, placement and affinity, threads per core etc. The effect of different placements is shown in figure below where the three models compact, scatter and balanced are shown. The best result for each test is compared, the actual number of cores might change as behavior changes with most parameters. The performance difference effect of placement is significant and care must always be taken to select optimal affinity. Which placement model yield best performance is not obvious. For small selected problems where all data for two or more threads can be kept in the L2 cache a compact model might be the best option. However, if those threads are competing for the execution units the core might be starved for execution units. The scheduler can fill the vector unit a thread, this is an improvement from Knights Corner where two threads were needed. In addition memory bandwidth are often a limiting factor. One core has a certain bandwidth and by spreading the threads onto many cores the total aggregated bandwidth is far larger than from a smaller set of cores.

4.2.1. Scaling and Speedup test

The scaling of performance increase as a function of applying more mores are always a good technique to start assessing the performance of an application or a benchmark. The figures below show the performance speedup when increasing the core count. Perfect scaling up to four cores have been assumed.

Figure 17. Scaling test using MPI version of NPB

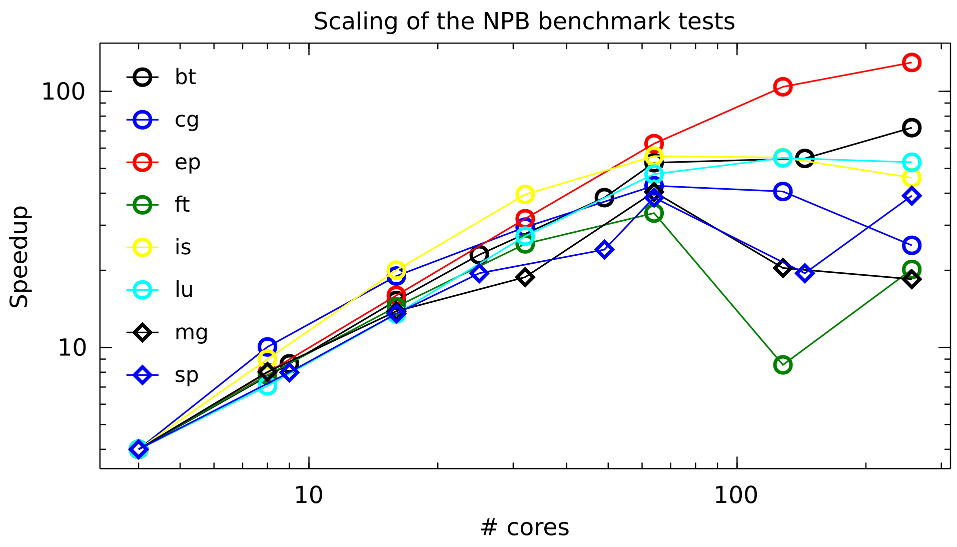
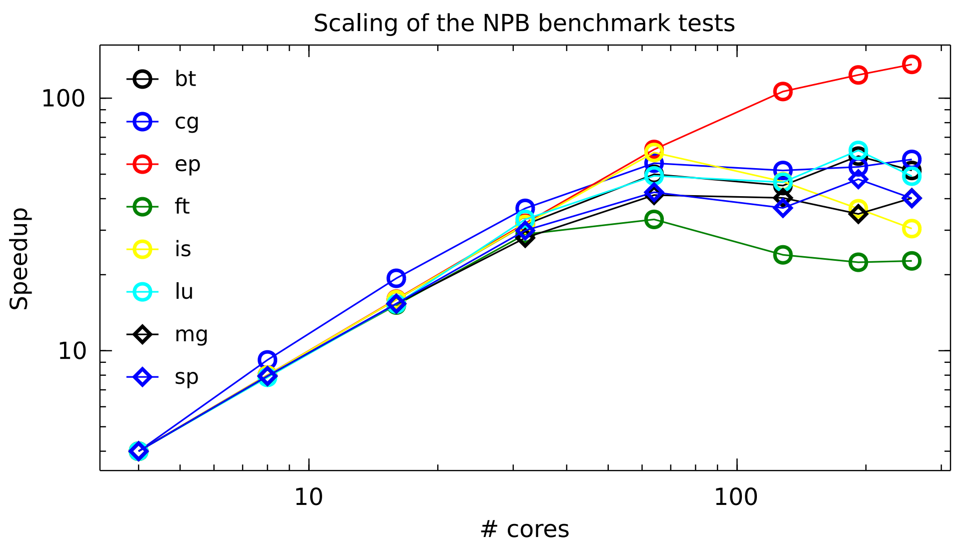


Figure 18. Scaling test using OpenMP version of NPB



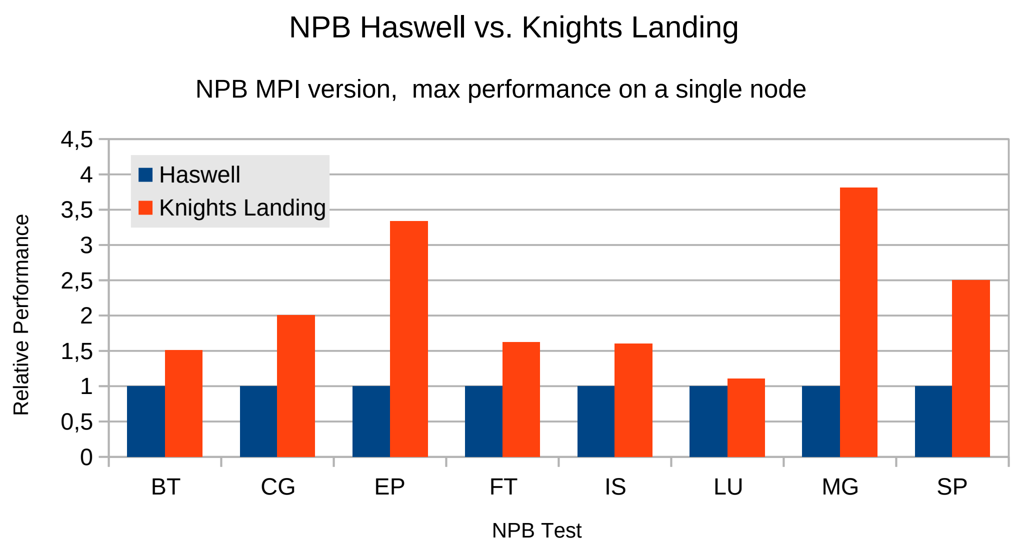
From the figure we see that applying more physical cores and hence more execution/vector units increase performance nicely. The behavior start to change when more than one MPI rank or OpenMP thread per physical core is scheduled. Some benchmarks cores benefit others not. The NPB benchmarks are well selected to represent different kind of application kernels. From the figure it is clear that for most applications using up to 64 cores are ok, but thorough testing is required if using more than one rank per core.

4.2.2. Performance

The NPB set of benchmarks is a nice set of tests to assess the processor and system performance and compare with other processors and systems. The figure below show the performance obtained from a single systems with Dual Haswell[2] processors and the KNL system. Optimal core count is used in each case. The figure below thus compare compute node versus compute node, even though the Haswell node is a dual socket node. It is usually better to compare compute node versus compute node then processor versus processor.

[2]Supermicro, SYS-6028TR-HTFR, E5-2660v3 2.6GHz, 64GiB, 8GB DDR4-2133 1Rx4 LP ECC RAM.

Figure 19. Performance Dual socket Haswell node versus KNL node



The figure show that a single socket KNL node outperform a dual socket Haswell node. Again we see the different NPB kernels show exercises different type of computation and communication.

4.3. HYDRO benchmark

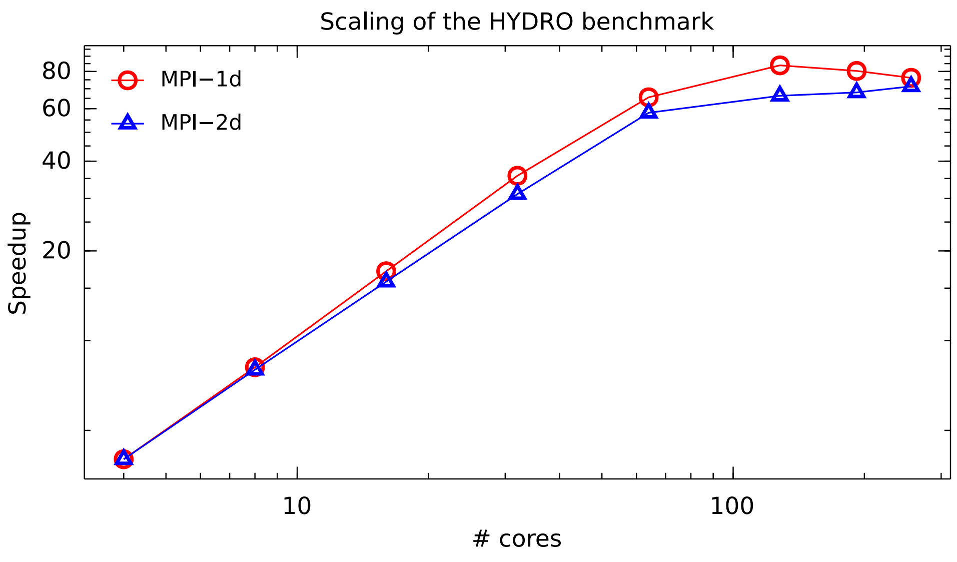
HYDRO is a much used benchmark in the PRACE community, it is extracted from a real code (RAMSES, which is a computational Fluid Dynamics code). Being widely used it has been ported to a number of platforms. The code exist in many versions, FORTRAN 90, C, CUDA, OpenCL as well as serial, OpenMP and MPI versions of these. Some versions have been instrumented with performance counters to calculate the performance in Mflops/s.

The instrumented version is a FORTRAN 90 versions and this version in both OpenMP and MPI versions have been used for evaluation. The OpenMP implementations exhibit far inferior performance than the MPI versions. Hence performance testing and comparison is only done using the MPI version.

4.3.1. Scaling and Speedup test

Assessing the increased speedup with increasing core count is important with any benchmark or application. In some fortunate cases like well used benchmarks there are both MPI and OpenMP implementations.

Figure 20. Scaling test using MPI versions of HYDRO



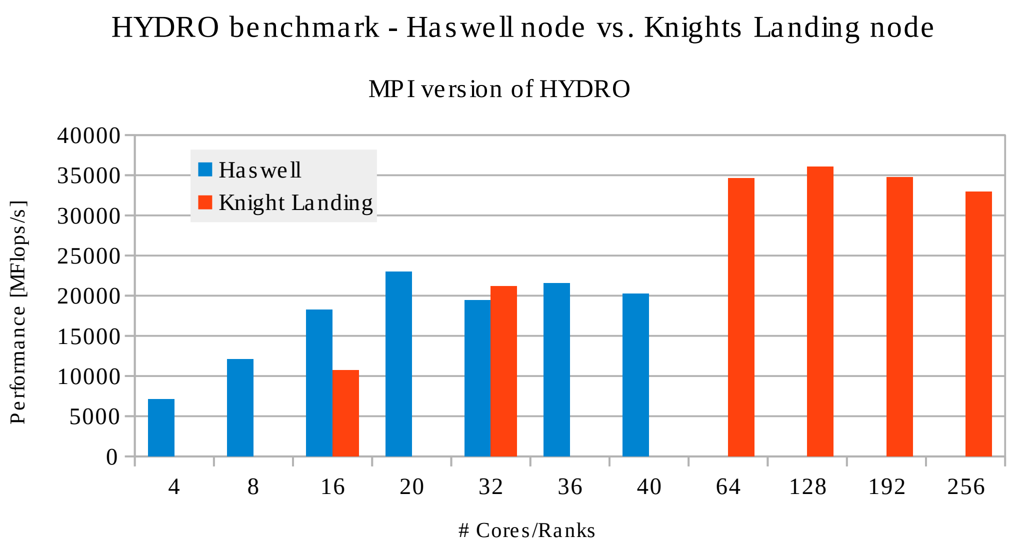
From the figure above we see that adding more executions units e.g. more physical cores improves performance linearly. Issuing more than one rank per core onto the already busy executing units will not deliver any more performance. The 1-dimensional and 2-dimensional implementations exhibit just minor performance differences, both in terms of absolute Mflops/s numbers and in scaling.

4.3.2. Performance

The HYDRO benchmark being a kernel of a real scientific applications is a nice example to compare compute nodes based on Haswell by KNL based ones. The following performance evaluations is done using two single compute nodes one dual socket Haswell based compute node[3] and one single socket KNL node. The figure below show the performance recorded for the two compute nodes.

[3]Supermicro, SYS-6028TR-HTFR, E5-2660v3 2.6GHz, 64GiB, 8GB DDR4-2133 1Rx4 LP ECC RAM.

Figure 21. Performance of a Dual socket Haswell node versus a KNL node



Using the MPI version of HYDRO which scales to a higher core count the Knights Landing processor can utilize all the cores and outperform a dual socket Haswell based system.

5. Application Performance

A range of different application has been tested. Both threaded single node applications and application based on Message Passing Interface (MPI) and threading model OpenMP.

5.1. ALYA

ALYA is a Computational Mechanics code capable of solving different physics, each one with its own modelization characteristics, in a coupled way. Among the problems it solves are: convection-diffusion reactions, incompressible flows, compressible flows, turbulence, bi-phasic flows and free surface, excitable media, acoustics, thermal flow, quantum mechanics (DFT) and solid mechanics (large strain).

ALYA is part of the Unified European Applications Benchmark Suite (UEABS), the codes can be found here [http://www.prace-ri.eu/ueabs/].

5.1.1. Building and compiling

The ALYA version used was downloaded from SURFSARA web drive [https://surfdrive.surf.nl/files/index.php/s/cbo1up9bZCP1iTT?path=%2FAlya] in October 2016.

The building of ALYA was straightforward, changing the configure files for the makefiles to use Intel MPI and compilers was well documented. Building of third libraries could be done with GNU or Intel compilers. Both versions link with the Intel build. Compilation flags suggested in this guide were used, see table below.

Table 9. Compiler and library versions

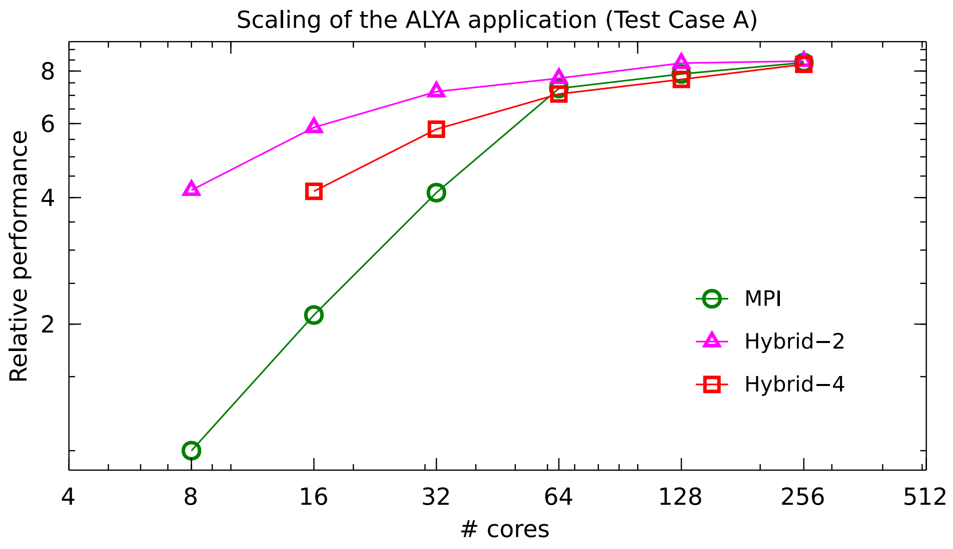
Compiler / Library	Versions and information
C/Fortran	Intel 2017 (2017.0.098)
MPI	Intel MPI Version 2017 Build 20160721
Compiler flags	-O3 -xMIC-AVX512 -align array64byte -fma -ftz -fomit-frame-pointer -finline-functions -qopenmp (hybrid model)

No major issues were experienced during the building process. The usual warnings from both C and Fortran were ignored and a workable executable was generated. The make system also support parallel compilation, a must on the slow cores on KNL.

5.1.2. Scaling and speedup

ALYA used a lot of IO during a run and might be sensitive to local storage, hence the total CPU time can vary slightly. CPU times for the NASTRAN module were used for the timings. The smaller test case A was used to assess the scaling and performance. The figure below show the scaling for pure MPI and Hybrid models.

Figure 22. Scaling of ALYA, MPI and Hybrid (MPI/OpenMP) runs



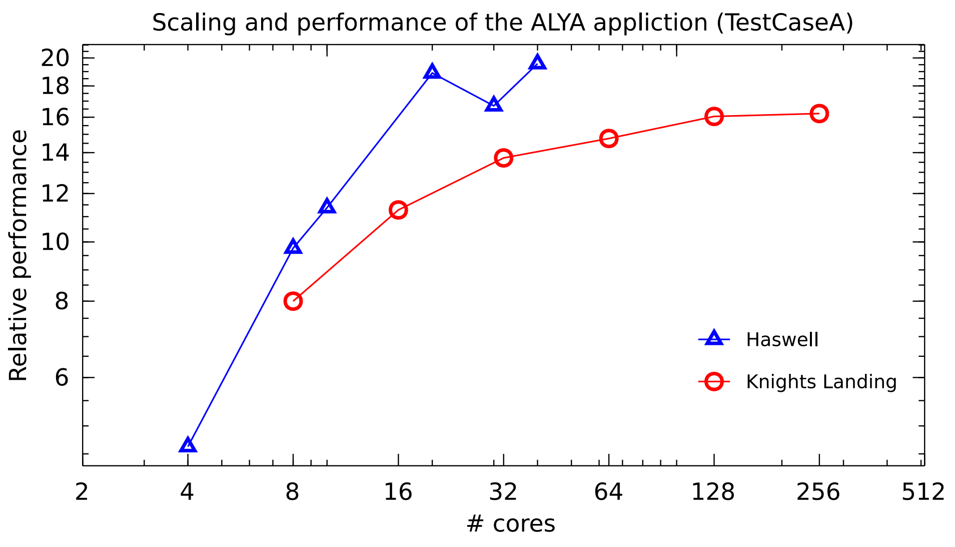
The hybrid model were run using 2 and 4 threads per MPI rank, named Hybrid-2 and Hybrid-4. Both hybrid models yield higher performance at low core core count but the MPI and hybrid implementations converge at higher core counts. At lower core count the scaling of the hybrid model are weaker, but startes at a higher performance. One possible reason for this is that the calculation becomes limited by a common resource, like memory bandwidth.

5.1.3. Performance

The ALYA is a well know benchmark and comparing it to the Haswell[4] processor is important to compare absolute performance, not only scaling. In both cases a node versus node performance is done. All cores in each nodes are used. This means two sockets in the Haswell node one socket in the KNL node. The best performance numbers for a given core count are reported.

[4]Supermicro, SYS-6028TR-HTFR, E5-2660v3 2.6GHz, 64GiB, 8GB DDR4-2133 1Rx4 LP ECC RAM.

Figure 23. Scaling of ALYA, MPI and Hybrid (MPI/OpenMP) runs



From the figure it's evident that the performance when running on a KNL is about 20% less than when running a on a Haswell node. The scaling is weaker on the KNL and roll off a around 128 cores. The memory requirements are relatively high for this application and it might be an effect of somewhat lower memory bandwidth. The aggregated memory bandwidth of two Haswell processors (over 100 GiB/s) are higher that the KNL's DDR4 memory bandwidth (about 50-60 GiB/s). Higher performance with twice the bandwidth for a memory intensive benchmark is no surprise.

5.2. GROMACS

GROMACS is a versatile package to perform molecular dynamics, i.e. simulate the Newtonian equations of motion for systems with hundreds to millions of particles. It is primarily designed for biochemical molecules such as proteins, lipids and nucleic acids that have a lot of complicated bonded interactions, but since GROMACS is extremely fast at calculating the non-bonded interactions (that usually dominate simulations) many groups also use it for research on non-biological systems, e.g. polymers.

GROMACS is part of the Unified European Applications Benchmark Suite (UEABS), the codes can be found here [http://www.prace-ri.eu/ueabs/].

5.2.1. Building and compiling

The Gromacs version used is Gromacs-2016-beta3 , downloaded from the Gromacs web site. The beta version is currently the only version supporting AVX-512 vectors.

The building process of Gromacs uses cmake and it mostly a question of setting up the compiler environment and selecting the right options to cmake. Experience using Haswell have shown that often GNU gcc/g++ outperform the Intel compilers. For the following tests Gromacs was compiled with the Intel compilers and Intel MPI libraries.

Table 10. Compiler and library versions

Compiler / Library	Version
C/C++/Fortran	Intel 2017.beta (042-17.0.0-042)

Compiler / Library	Version
MPI	Intel MPI Version 2017 Beta Build 20160302
FFTW	3.3.5 prerelease
BLAS/LAPACK	MKL (042-2017.0-042)
Compiler flags	-std=c11 -O3 -xMIC-AVX512 -align -fma -ftz -fomit-frame-pointer -finline-functions

Gromacs in the latest beta is sensitive to the C11 standard. Even though the Intel compiler support both C11 and C14 the underlying gcc libraries must also support C11 (gcc 5.x is ok, but not 6.1 which Intel 2017.beta does not support).

5.2.2. Scaling and speedup

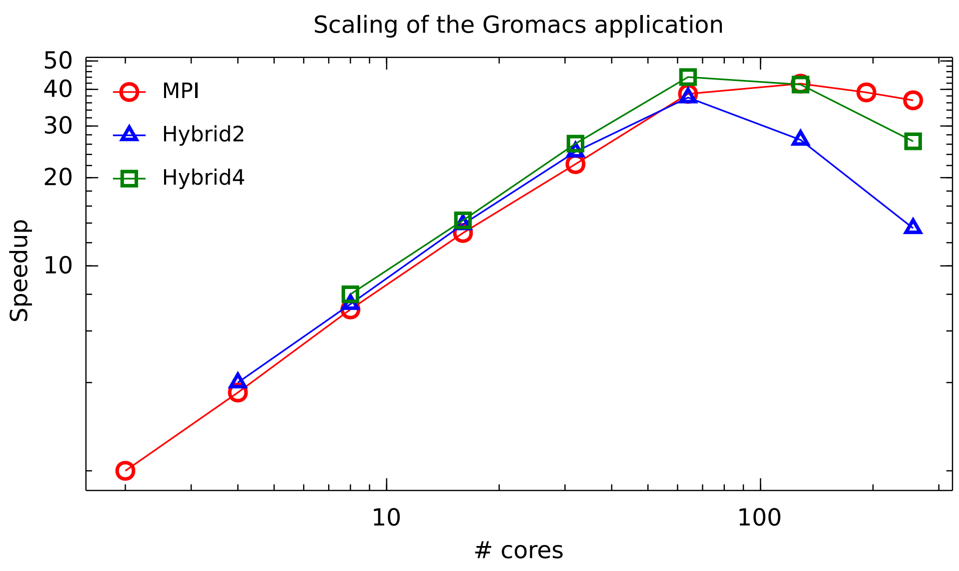
Testing of speedup or reduction in compute or wall time is essential for any application. Gromacs can be run in several modes, pure MPI and in Hybrid models combining OpenMP and MPI. The results in the figure below was obtained using both. Hybrid models employing two and fours threads per MPI ranks (names Hybrid2 and Hybrid4).

Table 11. Run time environment

Environment	Variable	Value
Hardware	Processor layout	Quadrant
Hardware	High Bandwidth Memory usage	Cache
MPI pinning	I_MPI_PIN	on
MPI placement	I_MPI_PIN_PROCESSOR_LIST	"processor list" (0,1,2..etc)
Thread placement	KMP_AFFINITY	"unset"
OpenMP processor binding	OMP_PROC_BIND	close

The benchmark set is the "adh_cubic" from the ADH_bench_systems set of Gromacs benchmarks. This set is freely available from the Gromacs ftp site.

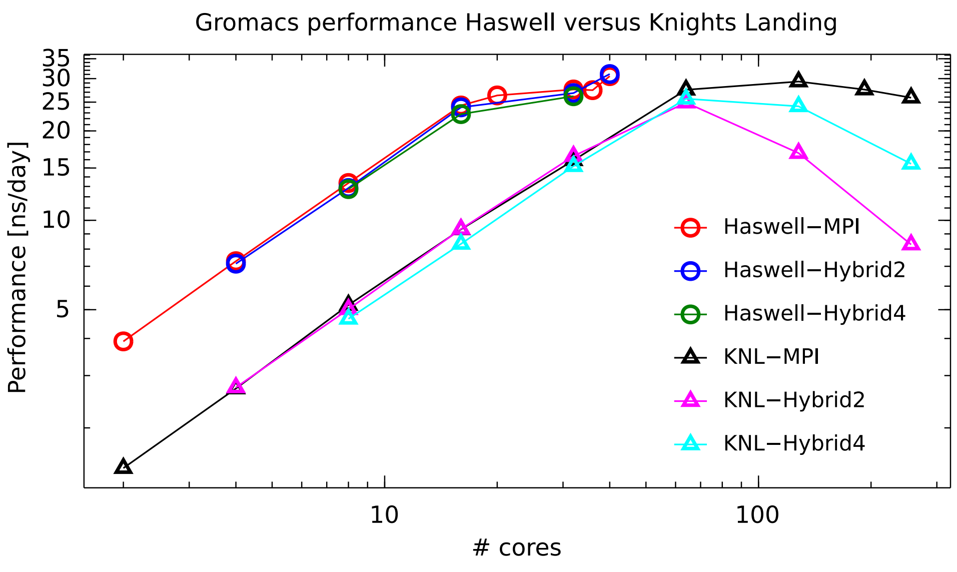
Figure 24. Scaling of Gromacs, MPI and Hybrid (MPI/OpenMP) runs



5.2.3. Performance

Gromacs is a widely used application and due to it's scaleability to high core counts a good candidate for KNL. The figure below show the performance and scaling combined when comparing KNL with Haswell. In each case single node performance is compared, the Haswell node is a dual socket[5] node while the KNL is single socket node, all cores are used. The benchmark input is the same a the scaling test above (adh_cubic).

Figure 25. Gromacs performance comparing Haswell and Knights Landing



From the figure it's evident that Knights Landing struggle to beat even the aging Haswell processor on performance. The performance different per core is clearly seen when only 4 cores are used. The KNL scale well as long as there are more execution units (physical cores) to add. However, the scaling is not perfect as it starts at slightly above 1 using four cores and reach about 30 when using 64 cores, while linear, the slope is not 45°. Scheduling more than one rank or thread per physical core does not really yield any more performance. Further tuning using chip layout, HBM memory settings, rank and thread placement might yield better performance, this is discussed in Section 7.

The default settings with Intel MPI and OpenMP thread placement yield good performance. The gain using settings to control the pinning, placements etc is limited.

5.3. Bifrost (Stellar atmosphere simulation code)

Numerical simulations of stellar convection and photospheres have been developed to the point where detailed shapes of observed spectral lines can be explained. Stellar atmospheres are very complex, and very different physical regimes are present in the convection zone, photosphere, chromosphere, transition region and corona. To understand the details of the atmosphere it is necessary to simulate the whole atmosphere since the different layers interact strongly. These physical regimes are very diverse and it takes a highly efficient massively parallel numerical code to solve the associated equations.

Bifrost is a code using a large portion of nested loops stencil type code. It scales well using MPI but is demanding in terms of memory bandwidth.

Bifrost is routineously run at large systems with NASA, it has also run on PRACE Tier 0 systems.

[5]Supermicro, SYS-6028TR-HTFR, E5-2660v3 2.6GHz, 64GiB, 8GB DDR4-2133 1Rx4 LP ECC RAM.

5.3.1. Building and compiling

Intel compiler, Intel MPI and math libraries have been used to build the applications.

Some performance issues was discovered during the build and performance anaylis process. In some cases the Fortran compiler reallocate the left hand side of an expression when it struggle to be sure of correctness and hence waste memory and bandwidth. This unneccerry step can be avoided by using *-assume noreallocate_lhs*. In the Bifrost case the effect were small, but were visible when the codes was analysed using Intel VTune Amplifier. The Fortran function *spread* which expand indxed variables were replaced with SIMD friendly loops with good results. Some experiment using *-fp-model fast=n*, (n=2) yielded acceptable results, faster execution than the original *-fp-model source*.

Table 12. Compiler and library versions

Compiler / Library	Version
ifort / icc	Intel 2017, 17.0.0 (2017.0-098)
Intel mpi	2017 Build (2017.0-098)
Math Kernel Library (MKL)	2017 Build (2017.0-098)
Fortran flags	-O3 -xMIC-AVX512 -fp-model source

5.3.1.1. Run setup

MPI ranks and placement. The number of ranks is restricted due to model constaints so it's not really possible to run using power of two set of ranks or all possible ranks.

5.3.1.2. Pure MPI model

Bifrost is currently a pure MPI application, hence the performance is assessed only for MPI parallelization.

5.3.2. Vectorization and scaling

5.3.2.1. Effect of vectorization

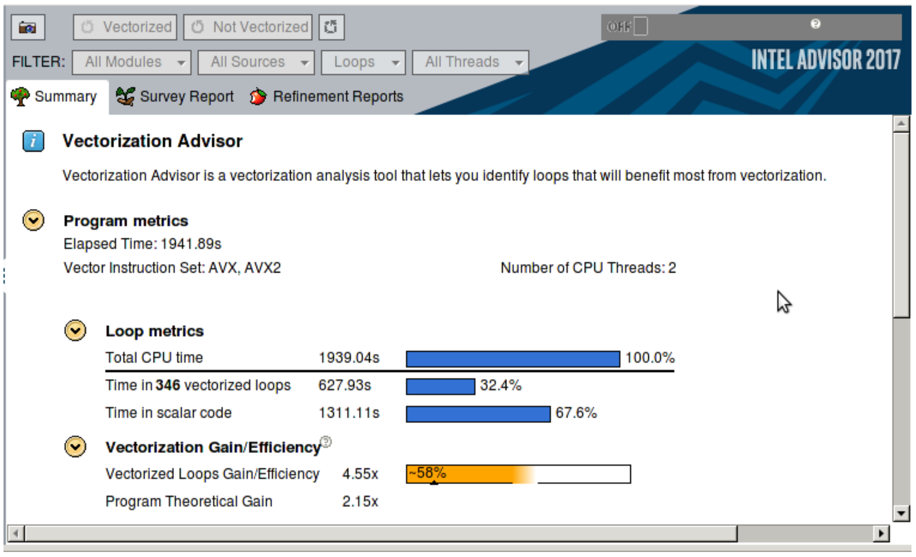
The effect of vectorization of the code can be tested by compiling with the vectorization turned on and off (using the *-no-vec* flag). This one run using 64 cores, the effect differ somewhat with varying number of cores.

Table 13. Effect of vectorization

Vectorization	Wall time [seconds]	Speedup
OFF (*-no-vec -no-simd -no-openmp-simd*)	452.20	1
ON (*-xMIC-AVX512 -qopenmp-simd*)	242.72	1.86

The advisor show that a significant fraction of the code is vectorized, with room for improvement both on the amount of vectorization and the efficiency of the vectorized part.

Figure 26. Intel vector Advisor screendump



Vectorization Advisor

Vectorization Advisor is a vectorization analysis tool that lets you identify loops that will benefit most from vectorization.

Program metrics

Elapsed Time: 1941.89s

Vector Instruction Set: AVX, AVX2

Number of CPU Threads: 2

Loop metrics

Total CPU time	1939.04s	100.0%
Time in **346** vectorized loops	627.93s	32.4%
Time in scalar code	1311.11s	67.6%

Vectorization Gain/Efficiency

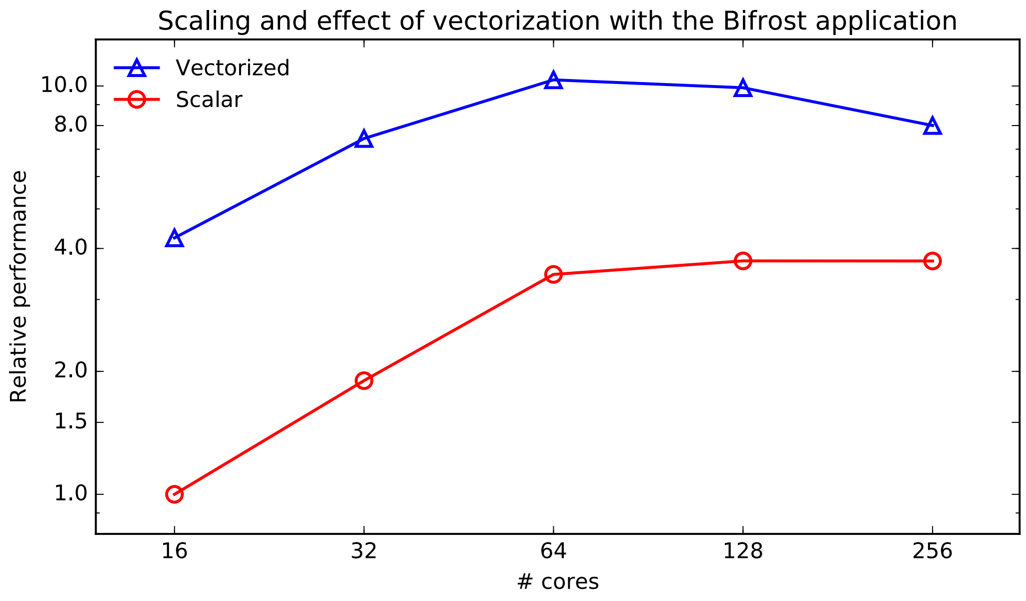
Vectorized Loops Gain/Efficiency	4.55x	~58%
Program Theoretical Gain	2.15x	

For more information about the Advisor see Section 7.2.

The theoretical speedup of vectorization range from 8 to 16 depending on data type, 64 bit or 32 bit floats. In this example we see a speedup of 1.86 which is nice, but there are still a huge potential for improvement. The code contains parts that at typical stencil code that usually are memory bandwidth dependent and might not benefit strongly from vectorization.

The example illustrate the importance of focus on vectorization.

Figure 27. Bifrost scaling and vectorization benefit

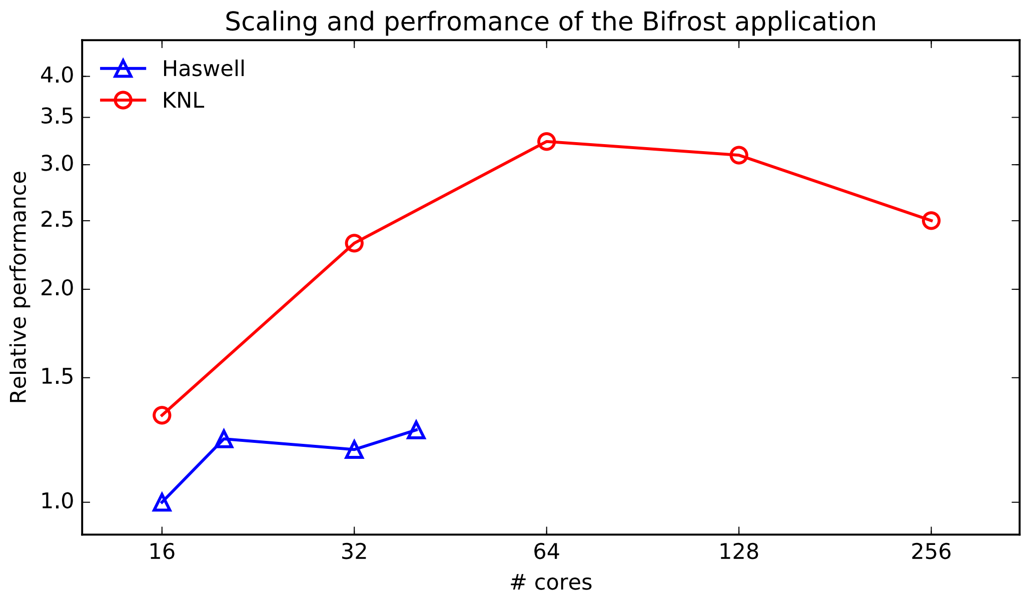


The figure above show that scaling seems dependent of vectorization, could be to the fact that some other limiting factors limit the performance at the higher yields. The striking difference is that vectorization yield far better absolute performance. A performace gain close to 4 is very good.

5.3.3. Performance

The figure below show the performance and scaling combined when comparing the Bifrost performance on KNL with Haswell. In each case single node performance is compared, the Haswell node is a dual socket[6] node while the KNL is single socket node, all cores are used.

Figure 28. Bifrost Performance



From the figure it's evident that Knights Landing outperform the Hawsell almost 3 to 1. The KNL scale well with the number of physical cores. The performance drop when issuing more than one rank per core, this is commonly observed when running a pure MPI model. This is the only time in this study that KNL outperform Haswell to such an extent.

5.4. Dalton and LS-Dalton (molecular electronic structure program)

The kernel of the Dalton-2016 suite is the two powerful molecular electronic structure programs, Dalton and LS-Dalton. Together, the two programs provide an extensive functionality for the calculations of molecular properties at the HF, DFT, MCSCF, and CC levels of theory. Many of these properties are only available in the Dalton-2016 suite. The tests focus on the Linear Scaling Dalton as this is currently the most relevant for KNL.

5.4.1. Building and compiling

An example of the compability of the KNL with standard Intel x86-64 systems the Dalton application build nicely from the downloaded source without any changes. Not all applications have run times that justifies the effort needed to dig into the makefiles etc to set the flags needed to fully utilize the KNL architecture.

The Dalton application can be build with threading parallelism using OpenMP or by message passing using the MPI version. Hybrid model combining the two is also supported.

Table 14. Building environment, Compiler, flags and library versions

Compiler / Flags / Library	Version and Flags
C/C++/Fortran	Intel 2017, 17.0.0 (2017.0.098)

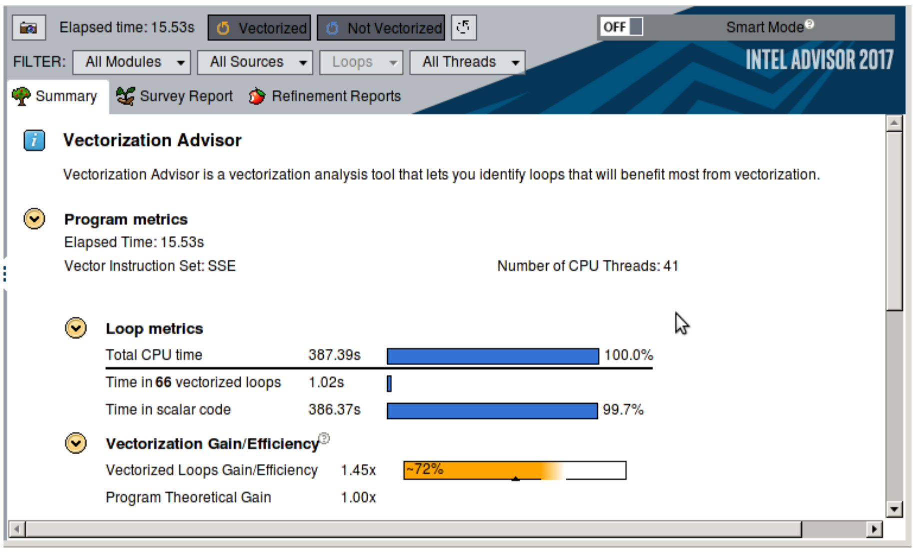
[6]Supermicro, SYS-6028TR-HTFR, E5-2660v3 2.6GHz, 64GiB, 8GB DDR4-2133 1Rx4 LP ECC RAM.

Compiler / Flags / Library	Version and Flags
MPI	Intel MPI Version 2017 Build (2017.0-098)
FFT/BLAS/LAPACK/SCALAPACK	MKL (2017.0-098)
C flags	-std=c11 -O3 -xMIC-AVX512 -align -fma -ftz -fomit-frame-pointer -finline-functions
C++ flags	-std=c++11 -O3 -xMIC-AVX512 -align -fma -ftz -fomit-frame-pointer -finline-functions
Fortran flags	-O3 -xMIC-AVX512 -align array64byte -fma -ftz -fomit-frame-pointer -finline-functions

5.4.2. Effect of vectorization

The vector Advisor produce a quick overview of the amount of vectorization, for LS-Dalton the nature of the problems is such that it's quite hare to write the code in such a way that the compiler can easily vectorize the code. This is evident from the figure below where the Advisor have analysed the code.

Figure 29. Intel vector Advisor screendump



Vectorization Advisor

Vectorization Advisor is a vectorization analysis tool that lets you identify loops that will benefit most from vectorization.

Program metrics
Elapsed Time: 15.53s
Vector Instruction Set: SSE
Number of CPU Threads: 41

Loop metrics

Total CPU time	387.39s	100.0%
Time in **66** vectorized loops	1.02s	
Time in scalar code	386.37s	99.7%

Vectorization Gain/Efficiency

Vectorized Loops Gain/Efficiency	1.45x	~72%
Program Theoretical Gain	1.00x	

For more information about the Advisor see Section 7.2.

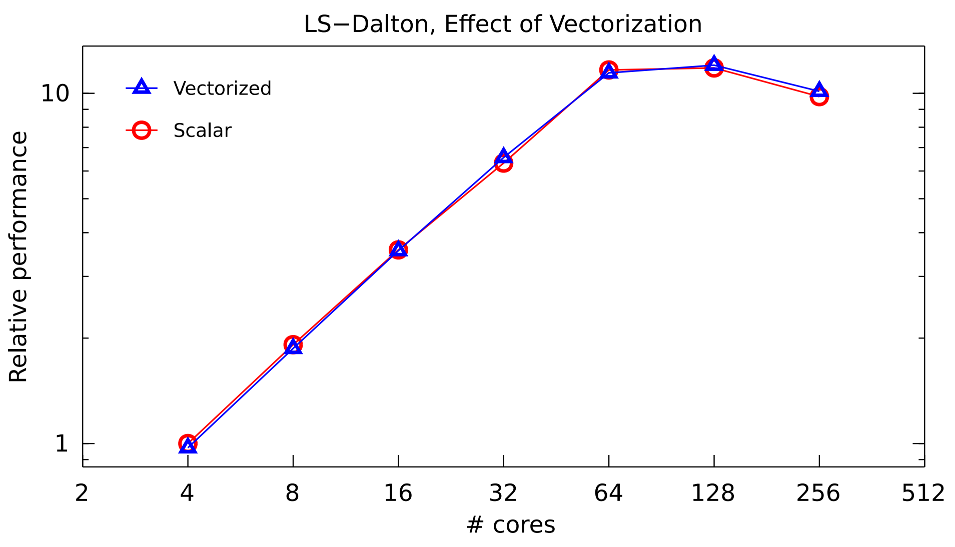
The effect of vectorization of the code can be tested by compiling with the vectorization turned on and off (using the *-no-vec* flag). This one run using 128 threads (64 ranks with 3 threads per rank),the effect differ somewhat with varying number of cores.

Table 15. Effect of vectorization

Vectorization	Wall time [seconds]	Speedup
OFF (*-no-vec -no-simd -no-openmp-simd*)	2142	1
ON (*-xMIC-AVX512 -qopenmp-simd*)	2134	1.004

The theoretical speedup of vectorization range from 8 to 16 depending on data type, 64 bit or 32 bit floats. In this example we see no significant difference in the scalar code and the vectorized code. It should be noted that the libraries are identical for the two runs. The MKL are highly optimized and vectorized and time spent in these libraries are the same in both tests.

Figure 30. LS-Dalton vectorization benefit



The very limited effect of vectorization hints that there are other factors than just floating point operations that are limiting performance. Either the time consuming parts do not vectorize or that processing speed is irrelevant when the processor is starved of data to process.

This kind of behavior can be observed in certain types of applications, making the adaptation of vector based processors like KNL more complicated. Longer vector units might even be counterproductive as the theoretical number increase while the measured performance stay constant effectively lowering the efficiency of the processor.

The figure Advisor-LSDalton.png from the Vector Advisor tool show that almost no time is spent in vectorized loops.

5.4.3. Scaling and speedup

Testing of speedup or reduction in compute or wall time is essential for any application. Dalton can be run in several modes, a pure shared memory threaded model, a pure MPI model or a Hybrid which is a combination of the two.

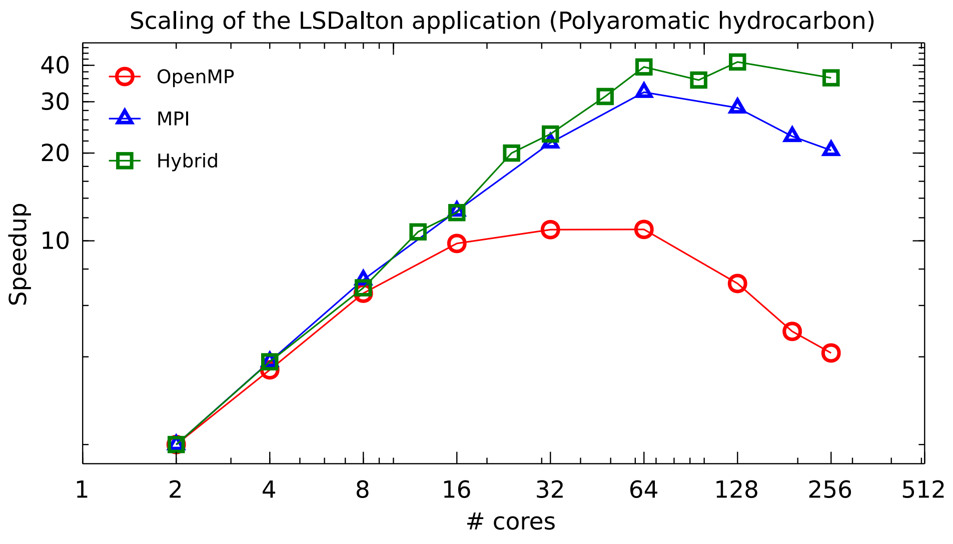
Table 16. Run time environment

Environment	Variable	Value
Hardware	Processor layout	Quadrant
Hardware	High Bandwidth Memory usage	Cache
MPI pinning	I_MPI_PIN	on
MPI placement	I_MPI_PIN_PROCESSOR_LIST	*unset*
OpenMP processor binding	OMP_PROC_BIND	TRUE
Thread placement	KMP_AFFINITY	granularity=core,scatter

There are a range of different values to use, a nice value for KMP_AFFINITY verbose,granularity=core/fine,compact/scatter, where a printout of the thread placements is printed and granularity and placement can be selected. The usage of both OMP_PROC_BIND and KMP_AFFINITY is not possible as a warning is issued: "OMP_PROC_BIND: ignored because KMP_AFFINITY has been defined".

The figure below show the speedup observed using the threaded/OpenMP, the MPI and the Hybrid model. It's assumed that the performance for the very low core counts are almost identical so that the performance at 2 cores are the same for all three models making scaling performance comparison relevant. The speedup is far from ideal, 41x speedup at 64 cores are not perfect, but the speedup is close to linear with increase in physical core count.

Figure 31. LSDalton scaling



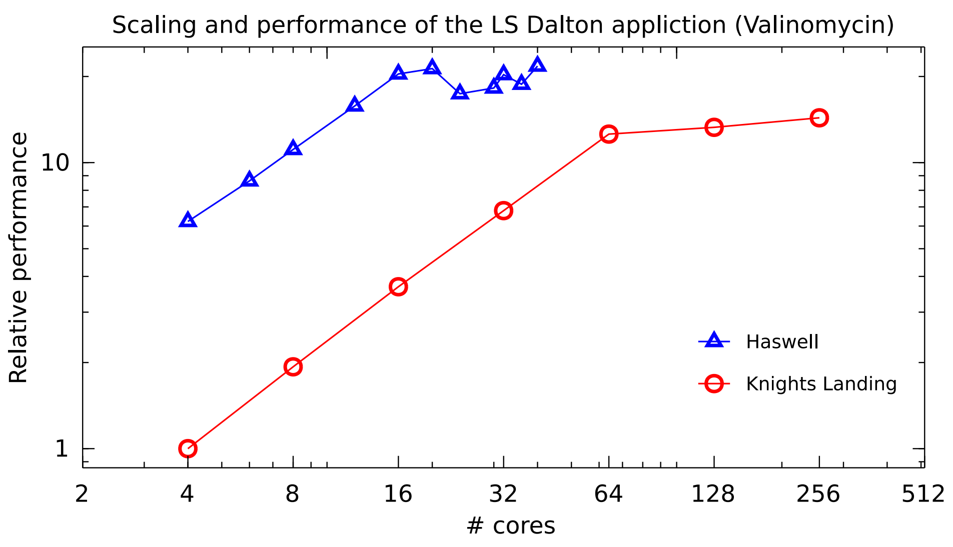
As expected by the developers the suggested run setup is using a Hybrid model. Hybrid models reduce the amount of MPI ranks, with its associated buffer memory requirements and hence make more memory available for the application. For this input using 4 threads per rank yield the highest performance.

5.4.4. Performance

LS-Dalton is widely user for theoretical chemistry work and performance evaluation is a neverending task. The figure below show the performance and scaling combined when comparing KNL with Haswell. In each case single node performance is compared, the Haswell node is a dual socket[7] node while the KNL is single socket node, all cores are used. The benchmark input is a Valinomycin molecule with the basis set 6-31G**, a larger and more relevant case than the former figure.

[7]Supermicro, SYS-6028TR-HTFR, E5-2660v3 2.6GHz, 64GiB, 8GB DDR4-2133 1Rx4 LP ECC RAM.

Figure 32. LS-Dalton Performance



From the figure it's evident that Knights Landing struggle to beat even the aging Haswell processor on performance. The performance different per core is clearly seen when only 4 cores are used. The KNL scale well with the number of physical cores. Further tuning using chip layout, HBM memory settings, rank and thread placement might yield better performance. The guidelines from Intel suggest using quadrant mode and using the HBM as cache which is the default settings in all runs in this guide (unless clearly stated). More on tuning in Section 7.

6. Performance Analysis

6.1. Performance Monitoring in the Knights Landing Tile

A Knights Landing tile is a core-pair similar to those found in Intel Atom processors based on the Silvermont (SLM) microarchitecture, but added HyperThreading Technology and additional enhancements described in Section 2. The Knights Landing Processor provides several new capabilities on top of the SLM performance monitoring facilities. For more information on performance monitoring capabilities in SLM, refer to the Section 18.6 in the Intel® 64 and IA-32 Architectures Software Developer Manuals (SDM). The Knights Landing Processor supports architectural performance monitoring capability with version ID 3 (see Section 18.2.3 in the SDM) and a host of nonarchitectural performance monitoring capabilities. The Knights Landing Processor provides two general-purpose performance counters (IA32_PMC0, IA32_PMC1) and three fixed-function performance counters (IA32_FIXED_CTR0, IA32_FIXED_CTR1, IA32_FIXED_CTR2). Non-architectural performance monitoring in the Knights Landing Processor uses the IA32_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter. The list of non-architectural performance monitoring events will be listed in a future revision of this document. The bit fields within each IA32_PERFEVTSELx MSR are defined in Figure 18-6 and described in Section 18.2.1.1 and Section 18.2.3 in the SDM. The Knights Landing Processor supports AnyThread counting in three architectural performance monitoring events. For a full description of the Knights Landing processor tile performance monitoring registers, refer to the section 5.1 Core MSRs. The notable enhancements in Knights Landing tile over SLM core include:

- AnyThread support. This facility is limited to following three architectural events - Instructions Retired, Unhalted Reference Cycles, Unhalted Core Cycles. Both fixed counter (IA32_FIXED_CTR0-3) and programmable counter (IA32_PMCx) are supported.

- PEBS-DLA (Precise Event-Based Sampling-Data Linear Address) support. Knights Landing Processor provides memory address in addition to the SLM PEBS record support on select events. Knights Landing Processor PEBS recording format as reported by IA32_PERF_CAPABILITIES [11:8] is 2.

- Off-core response counting facility. This facility in the processor core allows software to count certain transaction responses between the Knights Landing Processor tile to sub-systems outside the tile (untile). Counting off-core response requires additional event qualification configuration facility in conjunction with IA32_PERFEVTSELx. Two off-core response MSRs are provided to use in conjunction with specific event codes that must be specified with IA32_PERFEVTSELx. Knights Landing expands off-core response capability to match Knights Landing Processor untile changes.

- Average request latency measurement. The off-core response counting facility can be combined to use two performance counters to count the occurrences and weighted cycles of transaction requests.

- Last Branch Record (LBR) support changes. LBR is not officially part of the performance monitoring architecture and is often documented separately. It is beneficial, however, to note the LBR changes in Knights Landing Processor is in conjunction with performance monitoring changes. Knights Landing Processor supports 8 LBR pairs per thread. The LBR MSR addresses were changed to 0x680-687 (MSR_LASTBRANCH_x_FROM_IP) and 0x6C0-6C7 (MSR_LASTBRANCH_x_TO_IP) to match the Intel® Xeon® processor based servers (they have 16 LBR pairs).

6.2. Available Performance Analysis Tools

6.2.1. Intel Application Performance Snapshot

Intel Application Performance Snapshot is a tool for a quick and easy way of providing an overview of the application performance. This is a relatively new tool introduced with the 2017 version of the parallel studio. A simple to use tool which relies on Vtune Amplifier as the base sampling tool.

The APS analysis is easy to run, just prepend aps.sh to your command line :

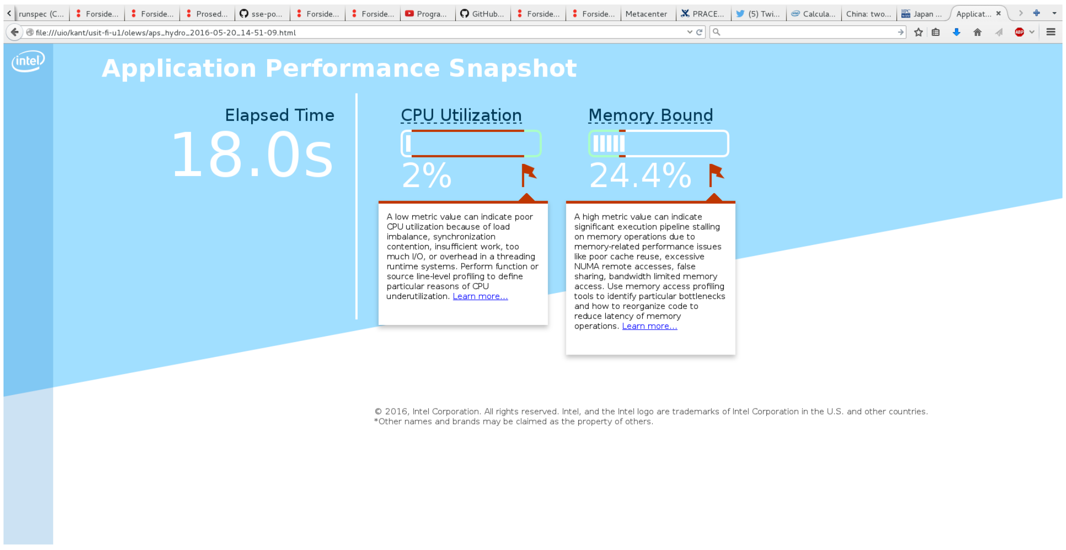
```
 aps.sh Bin/hydro  Input/input_sedov_250x250.nml
```

After the execution has finished a summary of the analysis is displayed.

```
 --------------------------------------------
 Application Performance Snapshot 2017 Beta
--------------------------------------------

Elapsed Time:    18.0s

--------------------------------------------

CPU Utilization: 2.0%
> A low metric value can indicate poor CPU utilization because of load imbalance,
synchronization contention, insufficient work, too much I/O, or overhead in a
threading runtime systems. Perform function or source line-level profiling to
define particular reasons of CPU underutilization. Learn more at
https://software.intel.com/en-us/concurrency-analysis-win

Memory Bound:    24.4%
> A high metric value can indicate significant execution pipeline stalling on
memory operations due to memory-related performance issues like poor cache reuse,
excessive NUMA remote accesses, false sharing, bandwidth limited memory access.
```

```
Use memory access profiling tools to identify particular bottlenecks and how to
reorganize code to reduce latency of memory operations. Learn more at
https://software.intel.com/en-us/memory-access-analysis-win

--------------------------------------------

Generated HTML report:
HYDRO/F90_instrumented/OpenMP/FineGrain/aps_hydro_2016-05-20_14-51-09.html
```

Figure 33. Application Performance Snapshot



Application Performance Snapshot

Elapsed Time
18.0s

CPU Utilization
2%

A low metric value can indicate poor CPU utilization because of load imbalance, synchronization contention, insufficient work, too much I/O, or overhead in a threading runtime systems. Perform function or source line-level profiling to define particular reasons of CPU underutilization. Learn more...

Memory Bound
24.4%

A high metric value can indicate significant execution pipeline stalling on memory operations due to memory-related performance issues like poor cache reuse, excessive NUMA remote accesses, false sharing, bandwidth limited memory access. Use memory access profiling tools to identify particular bottlenecks and how to reorganize code to reduce latency of memory operations. Learn more...

© 2016, Intel Corporation. All rights reserved. Intel, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.
*Other names and brands may be claimed as the property of others.

Using APS with MPI is also possible, just start the ranks to analysed prefixed with aps.sh.

```
mpirun -np 19 ./xhpl : -np 1 aps.sh ./xhpl
```

More than one rank can be analysed.

```
mpirun -np 20 aps.sh ./xhpl
```

One output file per rank is emitted, can be messy with high core count. Alternatively one can analyse the whole mpi job if running on a shared memory single node.

```
aps.sh mpirun -np 20 ./xhpl
```

The tool is not really well suited for MPI applications, but it runs well also under the MPI environment. The CPU percentage is measured against all cores and for a single rank it will be very low. Memory bandwidth is possibly a better metric, but in general other tools are better suited for MPI application analysis.

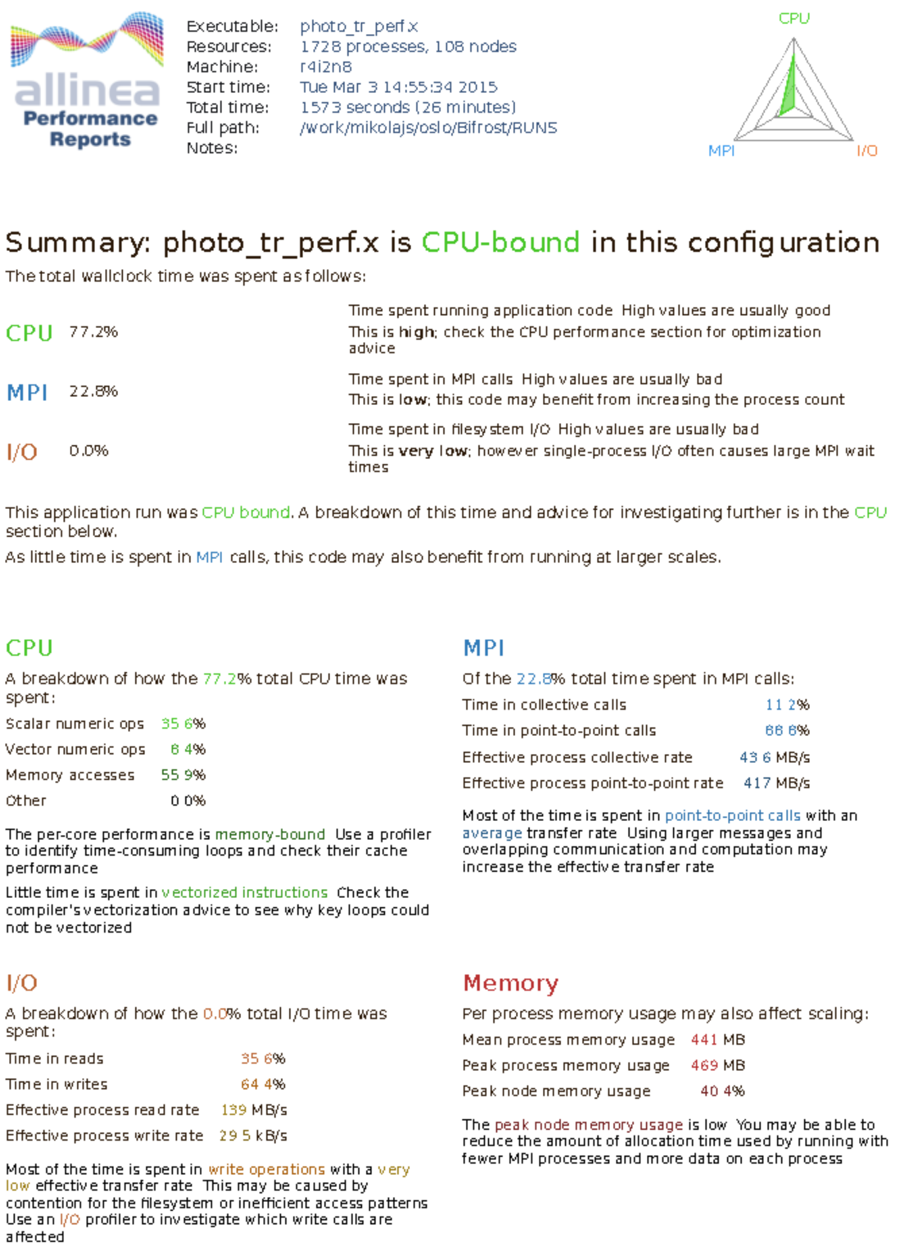
6.2.2. Allienea Performance Reports

Allinea Performance Reports is a tool for a quick and easy way of providing an overview of the application performance.

It is a very easy to use tool, just insert perf-report before mpirun and a performance report is generated in an ASCII format and a HTML format.

```
perf-report  mpirun  -np 20 ../photo_tr.x
```

Figure 34. Allinea Performance Reports



Executable:	photo_tr_perf.x
Resources:	1728 processes, 108 nodes
Machine:	r4i2n8
Start time:	Tue Mar 3 14:55:34 2015
Total time:	1573 seconds (26 minutes)
Full path:	/work/mikolajs/oslo/Bifrost/RUNS
Notes:	

Summary: photo_tr_perf.x is CPU-bound in this configuration

The total wallclock time was spent as follows:

CPU	77.2%	Time spent running application code High values are usually good This is **high**; check the CPU performance section for optimization advice
MPI	22.8%	Time spent in MPI calls High values are usually bad This is **low**; this code may benefit from increasing the process count
I/O	0.0%	Time spent in filesystem I/O High values are usually bad This is **very low**; however single-process I/O often causes large MPI wait times

This application run was CPU bound. A breakdown of this time and advice for investigating further is in the CPU section below.

As little time is spent in MPI calls, this code may also benefit from running at larger scales.

CPU

A breakdown of how the 77.2% total CPU time was spent:

Scalar numeric ops	35 6%
Vector numeric ops	8 4%
Memory accesses	55 9%
Other	0 0%

The per-core performance is memory-bound Use a profiler to identify time-consuming loops and check their cache performance

Little time is spent in vectorized instructions Check the compiler's vectorization advice to see why key loops could not be vectorized

MPI

Of the 22.8% total time spent in MPI calls:

Time in collective calls	11 2%
Time in point-to-point calls	88 8%
Effective process collective rate	43 6 MB/s
Effective process point-to-point rate	417 MB/s

Most of the time is spent in point-to-point calls with an average transfer rate Using larger messages and overlapping communication and computation may increase the effective transfer rate

I/O

A breakdown of how the 0.0% total I/O time was spent:

Time in reads	35 6%
Time in writes	64 4%
Effective process read rate	139 MB/s
Effective process write rate	29 5 kB/s

Most of the time is spent in write operations with a very low effective transfer rate This may be caused by contention for the filesystem or inefficient access patterns Use an I/O profiler to investigate which write calls are affected

Memory

Per process memory usage may also affect scaling:

Mean process memory usage	441 MB
Peak process memory usage	469 MB
Peak node memory usage	40 4%

The peak node memory usage is low You may be able to reduce the amount of allocation time used by running with fewer MPI processes and more data on each process

The alternative ASCII text file report contains the same information in text form:

```
Executable: photo_tr_perf.x
Resources: 1728 processes, 108 nodes
Machine: r4i2n8
Started on: Tue Mar 3 14:55:34 2015
Total time: 1573 seconds (26 minutes)
Full path: /work/mikolajs/oslo/Bifrost/RUNS
Notes:

Summary: photo_tr_perf.x is CPU-bound in this configuration
CPU:       77.2% |=======|
MPI:       22.8% |=|
I/O:        0.0% |
This application run was CPU-bound. A breakdown of this time and advice
for investigating further is found in the CPU section below.
As little time is spent in MPI calls, this code may also benefit from
running at larger scales.

CPU:
A breakdown of how the 77.2% total CPU time was spent:
Scalar numeric ops:     35.6% |===|
Vector numeric ops:      8.4% ||
Memory accesses:     55.9% |=====|
Other:        0.0% |
The per-core performance is memory-bound. Use a profiler to identify
time-consuming loops and check their cache performance.
Little time is spent in vectorized instructions. Check the compiler's
vectorization advice to see why key loops could not be vectorized.

MPI:
A breakdown of how the 22.8% total MPI time was spent:
Time in collective calls:    11.2% ||
Time in point-to-point calls:    88.8% |========|
Effective collective rate: 4.36e+07 bytes/s
Effective point-to-point rate: 4.17e+08 bytes/s
Most of the time is spent in point-to-point calls with an average
transfer rate. Using larger messages and overlapping communication and
computation may increase the effective transfer rate.

I/O:
A breakdown of how the 0.0% total I/O time was spent:
Time in reads:      35.6% |===|
Time in writes:      64.4% |=====|
Effective read rate:  1.39e+08 bytes/s
Effective write rate:  2.95e+04 bytes/s
Most of the time is spent in write operations with a very low effective
transfer rate. This may be caused by contention for the file system or
inefficient access patterns. Use an I/O profiler to investigate which
write calls are affected.


Memory:
Per-process memory usage may also affect scaling:
Mean process memory usage: 4.41e+08 bytes
Peak process memory usage: 4.69e+08 bytes
Peak node memory usage:     40.4% |===|
The peak node memory usage is low. You may be able to reduce the amount of
allocation time used by running with fewer MPI processes and more data on
each process.
```

6.2.3. Intel Software Development Emulator

Intel Software Development Emulator is a tool that can do a lot of interesting stuff. A very useful feature is it's ability to count the different instructions executed. With this one might extract the number of floating point instructions executed. Armed with this number it's possible to calculate the ratio between theoretical number of instructions and instructions actually executed. A typical application will normally have an efficiency ratio of less than 50%. A highly tuned benchmark like HPL will show a far higher ratio.

The command line for running the instruction counts look like this:

```
sde -knl -iform 1 -omix myapp_mix.out -top_blocks 5000 -- ./myapp
```

The output file "myapp_knl_mix.out" contains the raw data to be analysed. The last section with the header "EMIT_GLOBAL_DYNAMIC_STATS" contains sums of each counter. The interesting labels we are interesting in looks like "*elements_fp_(single/double)_(1/2/4/8/16)" and "*elements_fp_(single/double)_(8/16) _masked". An example of how to extract these is given here:

```
cat myapp_mix.out | awk '/EMIT_GLOBAL_DYNAMIC_STATS/,\
      /END_GLOBAL_DYNAMIC_STATS/ {print $0}'| grep
      "elements_fp_single_8"
```

A simple script[8] can be written to parse this file to get an estimate of the total number of floating point instructions executed during a run. An example of my simple script output is given below. This script does not deal with masks etc. Only calculates the floating point instructions with variable vector lenghts and operands[9].

```
$ flops.lua myapp_mix.out
File to be processed: myapp_mix.out
Scalar  vector (single) 1
Scalar  vector (double) 2225180366
4 entry vector (double) 5573099588
Total Mflops without FMA and mask corrections
Total Mflops single  0
Total Mflops double  7798
Total Mflops including FMA instructions
Total Mflops single  0
Total Mflops double  8782
Total Mlops :  8782
```

Using the instrumented version of HYDRO for comparison the numbers do differ somewhat. HYDRO F90_Instrumented arrive at 2330.26 Mflops/s at a cpu time of 4.4323 seconds giving 10328 Mflops in total. The numbers differ slightly, but within a reasonable margin. For tests using matrix matrix multiplication the numbers match up far better, suggesting some minor glitches in the counting of flops with HYDRO.

Some more documentations are available at Intel's web server, calculating flops using SDE [https://software.intel.com/en-us/articles/calculating-flop-using-intel-software-development-emulator-intel-sde].

6.3. Hints for Interpreting Results

7. Tuning

7.1. Guidelines for tuning

There are many ways to approach the tuning of an application. Based on experience the following guidelines provide a starting point.

The tests have been done with the suggested settings from Intel for every day general use, Chip layout set to "Quadrant" and High Bandwidth Memory to "Cache". Unless specified for the individual test this is the default settings. These defaults settings might not be optimal for many applications and a range of setting will be tested in the sections below.

7.1.1. Top down

7.1.1.1. Measure fraction of theoretical performance

If possible, try establishing an estimate of the number of floating point operations that is done in total, if not possible to calculate, try to make an educated guess. Then compare the run time of the application with the theoretical

[8]Script can be found at http://folk.uio.no/olews/flops.lua
[9] The numbers are taken from a Haswell run

performance. Intel claim a theoretical performance of about 3 Tflops/s. This is a number arrived to when using all the cores with full vector registers performing only FMA. This number is to be regarded as a marketing number, but the top 500 benchmark HPL might come quite close.

Comparing flops/s numbers for the application in question is a top down approach and a good place to start to establish a performance baseline. Remember that if full vectorization is not achieved the performance might drop to 1/8 (64 bit floats) or 1/16 (32 bit floats) as is the case when using only scalars. For other data types numbers might be even worse. The performance might drop from excellent (3 Tflops/s) to only 375 Gflops/s.

7.1.1.1.1. Calculating or Measuring the total number of flops

Using Gromacs as an example we can do some simple estimates. Gromacs' developers fortunately enough provides us with an operations count in the form of Mega flops. Consider a run where the counter clocks in at 471661 Gigaflops and a wall time of 1010 seconds. For a dual socket Haswell processor performing single precision floats system with a total of 20 cores which provides 20 times 16 vector flops times 2 for FMA times 2.6 GHz we arrive at 1664 Gigaflops/s theoretical performance. In 1010 seconds it could have performed 1680640 Gigaflops. Gromacs utilized 471661 and since 471661/1680640 is 0.28, the utilization is estimated to 28%. Numbers like this (and possibly much lower) are quite common and demonstrate that processor performance seldom can be fully exploited in real applications. It's only the HPL Top 500 test that comes close to the theoretical number.

If calculating the number of floating point operations is impossible or very time consuming it can be measured using the Intel Software Emulation tool. See Section 6.2.3 about this tool.

7.1.1.2. Measure the effect of vectorization

Establishing a rough estimate how well the code vectorize is done by building and running the code with full vectorization and recording the wall clock time and then building it without vectorization (flags *-no-vec -no-simd -no-qopenmp-simd*) effectively turn off any vectorization by the compiler. It does not inhibit libraries like MKL from being vectorized.

If vectorization is perfect the fraction between the two should be 8 (64 bit floats) or 16 (32 bit floats). If the gain in speedup is small there is huge potential for performance increase by facilitate vectorization, see Section 7.2.3 about the Vector adviser. If vectorization is very low you need to go back to the source and look deeper into it and possibly rewrite to vectorize better. Examples can be found in Section 5.3 and Section 5.4.

There are a lot of tutorials about how to vectorize your code. It's a too large topic to be covered in this guide, many guides and tutorials are available. Intel provide several, this is a nice introduction: vectorization essentials [https://software.intel.com/en-us/articles/vectorization-essential].

7.1.1.3. Is the application compute or memory bound

If the application is memory bound the performance measures might be very low compared to the numbers above. A quick way of checking this is to look at the source and try to establish how many floating point operations you need per byte of data. If this number is very low the application is memory bound and a different set of tuning is needed as opposed to a compute bound application. You might find the Amplifier tool useful for this kind of challenges, see Section 7.6.3 about the VTune Amplifier.

7.1.2. Bottom up

The following is a bottom up approach, starting with the source code at the implementation and single core level.

- Select proper algorithm for problem, maybe there is a published package that solves your problem or a vendor library containing a better algorithm.
- Optimize single core performance, e.g. Vectorization, memory alignment, code generation etc.
- Optimize thread/OpenMP performance, does it scale with the number of threads?
- Optimize MPI performance, does it scale with the number of MPI ranks ?

- Optimize Scaling, thread placement, hybrid placement, processor placement.
- Optimize IO performance, serial or parallel IO, POSIX, MPI-IO etc.

Remember the the only tool that actually read your source code is the compiler. The compiler messages from the optimizer and other diagnostics presented are output from the only time the source code is analyzed. Any other tool will analyze the code generated by the compiler.

There are different tool to help you gain insight in your special application. Some of them are covered in the sections below. However, the important things like checking if the application's performance increase with an increased number of cores does not require any tools.

7.2. Intel tuning tools

Intel provide a set of tuning tools that can be quite useful. The following sections will give an overview of the tools and provide a quick guide of the scope of the different tools and for which type of tuning and what kind of programming model that are applicable for, vectorization, threading, MPI.

The tools will not be covered in depth as extensive documentations and tutorials are provided by Intel. Intel also provide training covering these tools during workshops at some places.

Below is a list of tools that can be used to analyses the application, starting from text analysis of the source code to massive parallel MPI runs.

- Intel compiler, analysis of source code.
- Intel XE-Advisor, analysis of Vectorization.
- Intel XE-Inspector, analysis of threads and memory.
- Intel VTune-Amplifier, analysis and profiling of complete program performance.
- Intel MPI, profile the MPI calls.
- Intel Trace Analyzer, analysis of MPI communication.

7.2.1. Intel compiler

This section will provide some hints and tricks with the Intel compiler tools with some additional reference to the Intel tuning tools. It is not a tutorial for using neither the compiler nor the tuning tools. It will however, give an overview of what these tools do and provide a guide where to start.

All the compiler flags given below are taken from the Intel compiler documentation. This documentation comes with the compiler installation and should be available locally or it can be found on the web pages published by Intel. Intel compiler documentation [https://software.intel.com/en-us/intel-parallel-studio-xe-support/documentation]

7.2.1.1. Compilers optimization

Compiler flags provide a mean of controlling the optimization done by the compiler. There are a rich set of compiler flags and directives that will guide the compiler's optimization process. The details of all these switches and flags can be found in the documentation, in this guide we'll provide a set of flags that normally gives acceptable performance. It must be said that the defaults are set to request a quite high level of optimization, and the default might not always be the optimal set. Not all the aggressive optimizations are numerically accurate, computer evaluation of an expression is as we all know quite different from paper and pencil evaluation.

7.2.1.1.1. Optimization flags

The following table gives an overview of optimization flags that will be useful for a simple start of the tuning process.

Table 17. Common optimization flags

Compiler flag	Information
-O1	Optimize for maximum speed, but disable some optimizations which increase code size for a small speed benefit
-O2	Optimize for maximum speed (default)
-O3	Optimize for maximum speed and enable more aggressive optimizations that may not improve performance on some programs. *For some memory bound programs this has proven to be true.*
-Ofast	Enable -O3 -no-prec-div -fp-model fast=2 optimizations. *This might not be safe for all programs.*
-fast	enable -xHOST -O3 -ipo -no-prec-div -static -fp-model fast=2 *This might not be safe for all programs.*
-xAVX	AVX and CORE-AVX2 May generate Intel Advanced Vector Extensions. This is supported with KNL, but vector length is only 256 bits. Using -x generated code that runs exclusively on the processor indicated.
-xMIC-AVX512	May generate Intel(R) Advanced Vector Extensions 512. Using -x generated code that runs exclusively on the processor indicated.
-funroll-loops	Unroll loops based on default heuristics. *With the Loop Stream Detector hardware this unrolling might not always yield better performance.*
-align array64byte	Align data at 64 byte boundaries, this is the cache line width and also 512 bits matching the vector width.
-qopt-malloc-options	Specify malloc configuration parameters. *This will provide different ways of allocation memory with the extended memory hierarchy on the KNL.*
-lmemkind	*When using FASTMEM directive in a program, you must specify this flag to the compiler.* See Section 7.8.2 for more information about compiler directives for memory allocation.
-ipo	Enable multi-file IP optimization between files. This option perform whole program optimization. This combined with optimization reports can yield significant insight and performance gain.

7.2.1.1.2. Optimization reporting flags

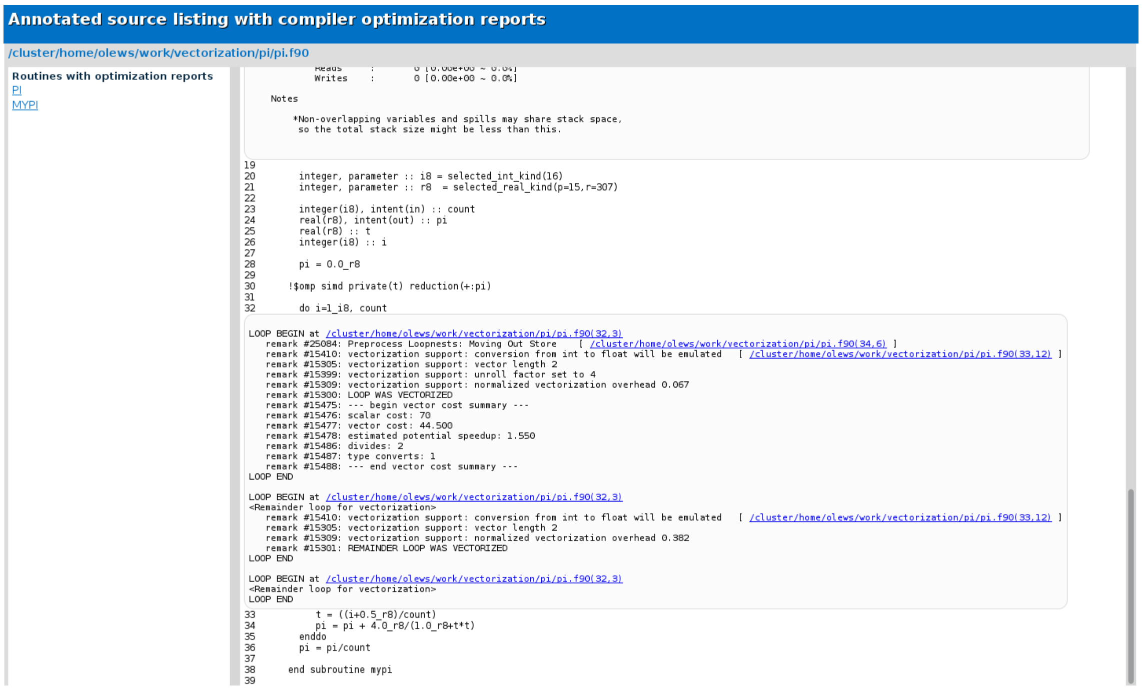
The following table gives an overview of compiler flags that is helpful to get reports from the compiler. Only a few flags are shown, please refer to the documentation for more information.

Table 18. Information request flags

Compiler flag	Information
-vec-report[=n]	Control amount of vectorizer diagnostic information. Any number from 0 to 7. 3 is most common as it provide information about both vectorized loops and none vectorized loops and data dependence.
-qopt-report[=n]	Generate an optimization report. Any level from 0 to 5 is valid. Higher number yield more information.
-qopenmp-report[n]	Control the OpenMP parallelizer diagnostic level, valid numbers are 0 to 2.
-qopt-report-annotate	Emit reports with source code and compiler diagnotics.

The reports generated by the compiler can be emitted in HTML format which makes them easier to read and to navigate.

Figure 35. Compiler optimization report in HTML format



Annotated source listing with compiler optimization reports

/cluster/home/olews/work/vectorization/pi/pi.f90

Routines with optimization reports
PI
MYPI

```
          Reads    :        0 [0.00e+00 ~ 0.0%]
          Writes   :        0 [0.00e+00 ~ 0.0%]

    Notes

        *Non-overlapping variables and spills may share stack space,
         so the total stack size might be less than this.


         integer, parameter :: i8 = selected_int_kind(16)
         integer, parameter :: r8  = selected_real_kind(p=15,r=307)

         integer(i8), intent(in) :: count
         real(r8), intent(out) :: pi
         real(r8) :: t
         integer(i8) :: i

         pi = 0.0_r8

       !$omp simd private(t) reduction(+:pi)

         do i=1_i8, count

LOOP BEGIN at /cluster/home/olews/work/vectorization/pi/pi.f90(32,3)
   remark #25084: Preprocess Loopnests: Moving Out Store    [ /cluster/home/olews/work/vectorization/pi/pi.f90(34,6) ]
   remark #15410: vectorization support: conversion from int to float will be emulated   [ /cluster/home/olews/work/vectorization/pi/pi.f90(33,12) ]
   remark #15305: vectorization support: vector length 2
   remark #15399: vectorization support: unroll factor set to 4
   remark #15309: vectorization support: normalized vectorization overhead 0.067
   remark #15300: LOOP WAS VECTORIZED
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 70
   remark #15477: vector cost: 44.500
   remark #15478: estimated potential speedup: 1.550
   remark #15486: divides: 2
   remark #15487: type converts: 1
   remark #15488: --- end vector cost summary ---
LOOP END

LOOP BEGIN at /cluster/home/olews/work/vectorization/pi/pi.f90(32,3)
<Remainder loop for vectorization>
   remark #15410: vectorization support: conversion from int to float will be emulated   [ /cluster/home/olews/work/vectorization/pi/pi.f90(33,12) ]
   remark #15305: vectorization support: vector length 2
   remark #15309: vectorization support: normalized vectorization overhead 0.382
   remark #15301: REMAINDER LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at /cluster/home/olews/work/vectorization/pi/pi.f90(32,3)
<Remainder loop for vectorization>
LOOP END
            t = ((i+0.5_r8)/count)
            pi = pi + 4.0_r8/(1.0_r8+t*t)
         enddo
         pi = pi/count

       end subroutine mypi

```

The HTML format contain routine and function indexing that makes seaching and navigating a more streamlined process. To instruct the compiler to generate such an output use the flags *-qopt-report-annotate=html*.

7.2.1.1.3. Floating point Optimization

Some flags affect the way the compiler optimize with floating point operations.

Table 19. Floating point flags

Compiler flag	Information
-fp-model fast[=n]	enables more aggressive floating point optimizations, a value of 1 or 2 can be added top the fast keyword.
-fp-model strict	Sets precise and enable exceptions.
-fp-speculation=fast	speculate floating point operations (default)
-fp-speculation=strict	Turn floating point speculations off.
-mieee-fp	maintain floating point precision (disables some optimizations).
-[no-]ftz	Enable/disable flush denormal results to zero.
-[no-]fma	Enable/disable the combining of floating point multiplies and add/subtract operations. *While not always numerically stable it is vital for getting maximum performance. Published performance is always with FMA enabled.*

7.2.1.2. Code generation

The code generated by the compiler can be studied with an object dump tool. This will provide a list of the instructions generated by the compiler. One might want to look for FMA or AVX instructions to see if the compiler have generated the right instructions for the processor in question.

The following example is taken from the well known benchmark Streams, it is generated with the following command:

```
ifort -c -g  -O0 -no-vec stream.f
```

```
objdump -S -D stream.o
```

Giving output like shown below, where parts of a loop is shown:

```
          DO 60 j = 1,n
     edb:       c7 85 48 fc ff ff 01    movl   $0x1,-0x3b8(%rbp)
              a(j) = b(j) + scalar*c(j)
     ee5:       8b 85 48 fc ff ff       mov    -0x3b8(%rbp),%eax
     eeb:       48 63 c0                movslq %eax,%rax
     eee:       48 6b c0 08             imul   $0x8,%rax,%rax
     ef2:       ba 00 00 00 00          mov    $0x0,%edx
     ef7:       48 03 d0                add    %rax,%rdx
     efa:       48 83 c2 f8             add    $0xfffffffffffffff8,%rdx
     efe:       f2 0f 10 85 18 fe ff    movsd  -0x1e8(%rbp),%xmm0
     f06:       8b 85 48 fc ff ff       mov    -0x3b8(%rbp),%eax
     f0c:       48 63 c0                movslq %eax,%rax
     f0f:       48 6b c0 08             imul   $0x8,%rax,%rax
     f13:       b9 00 00 00 00          mov    $0x0,%ecx
     f18:       48 03 c8                add    %rax,%rcx
     f1b:       48 83 c1 f8             add    $0xfffffffffffffff8,%rcx
     f1f:       f2 0f 10 09             movsd  (%rcx),%xmm1
     f23:       f2 0f 59 c1             mulsd  %xmm1,%xmm0
     f27:       f2 0f 10 0a             movsd  (%rdx),%xmm1
     f2b:       f2 0f 58 c8             addsd  %xmm0,%xmm1
     f2f:       8b 85 48 fc ff ff       mov    -0x3b8(%rbp),%eax
     f35:       48 63 c0                movslq %eax,%rax
     f38:       48 6b c0 08             imul   $0x8,%rax,%rax
     f3c:       ba 00 00 00 00          mov    $0x0,%edx
     f41:       48 03 d0                add    %rax,%rdx
     f44:       48 83 c2 f8             add    $0xfffffffffffffff8,%rdx
     f48:       f2 0f 11 0a             movsd  %xmm1,(%rdx)
   60     CONTINUE
```

This is the code produced with optimization turned off. It even generates an imul instruction which is known to be very time consuming. Compare this to the code generated with vectorization and optimization turned on. Compiled with :

```
ifort -c -g -xMIC-AVX512 -O3 -fma -align array64byte stream.f
```

and again using objdump:

```
          DO 60 j = 1,n
     f45: 62 f1 7c 48 10 4c 24  vmovups 0xc0(%rsp),%zmm1
              a(j) = b(j) + scalar*c(j)
     f4d: 62 b1 7c 48 10 04 c5  vmovups 0x0(,%r8,8),%zmm0
     f58: 62 b2 f5 48 a8 04 c5  vfmadd213pd 0x0(,%r8,8),%zmm1,%zmm0
     f63: 62 b1 fd 48 2b 04 c5  vmovntpd %zmm0,0x0(,%r8,8)
   60     CONTINUE
```

However, this is just parts of the label 60 loop. The compilers optimizes substantionally and make several versions of the loop. This one looks like the most efficient one for normal runs. We see that the b+s*c expression is translated nicely into a vectorized fused multiply add instruction. Users need to review the generated code carefully and probably compare with output from the Intel tuning tools.

7.2.2. Intel MPI library

The Intel MPI library has some nice features built into it.

The MPI Perf Snapshot is a built in lightweight tool that will provide some helpful information with little effort from the user. Link your program with -profile=vt and simply issue the -mps flag to the mpirun command (some environment need to be set up first, but this is quite simple).

```
   mpiifort -o ./photo_tr.x -profile=vt *.o
   mpirun  -mps  -np 16 ./photo_tr.x
```

Disk IO, MPI and memory usage are all displayed in simple text format, in addition to some simple statistics about the run.

Below is an example of this output :

```
 ==================== GENERAL STATISTICS ====================
Total time:   145.233 sec (All ranks)
       MPI:     27.63%
   NON_MPI:     72.37%

WallClock :
      MIN :            9.068 sec (rank 10)
      MAX :            9.101 sec (rank 0)

=================== DISK USAGE STATISTICS ==================
                 Read                  Written            I/O Wait time (sec)
All ranks:      302.6 MB                 9.6 MB             0.000000
      MIN:       18.4 MB (rank 6)        0.3 KB (rank 1)     0.000000 (rank 0)
      MAX:       26.1 MB (rank 0)        9.6 MB (rank 0)     0.000000 (rank 0)

================== MEMORY USAGE STATISTICS =================
All ranks:   1391.074 MB
      MIN:     64.797 MB (rank 15)
      MAX:     99.438 MB (rank 0)

================= MPI IMBALANCE STATISTICS =================
MPI Imbalance:           30.809 sec          21.214% (All ranks)
          MIN:            0.841 sec           9.242% (rank 0)
          MAX:            4.625 sec          51.004% (rank 15)
```

The log files contain a detailed log of the MPI communication, one section for each rank. Data transfers which rank communicated with which, which MPI calls and message size, which Collectives with corresponding message sizes, and Collectives in context. This is a very good starting point, but the sheer amount of data call for the trace analyzer application. Below is given a small sample of the log file, only the rank 0 communication is summary:

```
   Data Transfers
Src Dst Amount(MB) Transfers
-----------------------------------------
000 --> 000 0.000000e+00 0
000 --> 001 2.598721e+02 8515
000 --> 002 5.228577e-01 1427
000 --> 003 4.708290e-01 135
000 --> 004 3.904312e+02 19407
000 --> 005 1.042298e+01 825
000 --> 006 4.864540e-01 143
000 --> 007 4.561615e-01 135
```

```
000 --> 008 3.485235e+02 19407
000 --> 009 1.042298e+01 825
000 --> 010 4.864540e-01 143
000 --> 011 4.561615e-01 135
000 --> 012 6.762256e+01 8239
000 --> 013 3.765289e+00 1903
000 --> 014 4.712982e-01 143
000 --> 015 4.419518e-01 135
=========================================
Totals  1.094853e+03 61517
```

It is easy to understand that this does not scale to a large number of ranks, but for development using a small rank count it is quite helpful.

7.2.3. Intel XE-Advisor

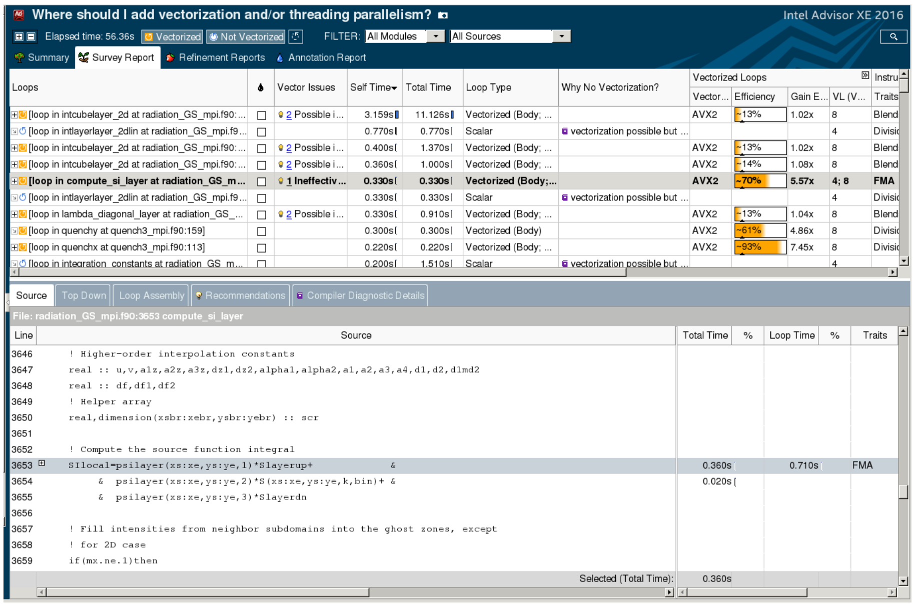
Vectorization Advisor is an analysis tool that lets you identify if loops utilize modern SIMD instructions or not, what prevents vectorization, what is performance efficiency and how to increase it. Vectorization Advisor shows compiler optimization reports in user-friendly way, and extends them with multiple other metrics, like loop trip counts, CPU time, memory access patterns and recommendations for optimization.

Vectorization is very important. The vector units in KNL are 512 bits wide. They can hence operate on 16 single precision (32 bits) or 8 double precision (64 bits) numbers simultaneously, often in one clock cycle. A possible speedup of 8 or 16 compared to none vectorized code is possible.

The advisor can be used as a graphic X11 based tool (GUI) or it can display results as text using a command line tool. The GUI version provide a large range of different analysis tabs and windows. Intel provide a range of tutorials and videos on how to use this tool.

Below is an illustration of the GUI of the Advisor tool.

Figure 36. Advisor GUI example



Loops		Vector Issues	Self Time	Total Time	Loop Type	Why No Vectorization?	Vector...	Efficiency	Gain E...	VL (V...	Traits
[loop in intcubelayer_2d at radiation_GS_mpi.f90:...	☐	2 Possible i...	3.159s	11.126s	Vectorized (Body; ...		AVX2	~13%	1.02x	8	Blend
[loop in intlayerlayer_2dlin at radiation_GS_mpi.f9...	☐		0.770s	0.770s	Scalar	vectorization possible but ...				4	Divisi
[loop in intcubelayer_2d at radiation_GS_mpi.f90:...	☐	2 Possible i...	0.400s	1.370s	Vectorized (Body; ...		AVX2	~13%	1.02x	8	Blend
[loop in intcubelayer_2d at radiation_GS_mpi.f90:...	☐	2 Possible i...	0.360s	1.000s	Vectorized (Body; ...		AVX2	~14%	1.08x	8	Blend
[loop in compute_si_layer at radiation_GS_m...	☐	**1 Ineffectiv...**	**0.330s**	**0.330s**	**Vectorized (Body;...**		**AVX2**	**~70%**	**5.57x**	**4; 8**	**FMA**
[loop in intlayerlayer_2dlin at radiation_GS_mpi.f9...	☐		0.330s	0.330s	Scalar	vectorization possible but ...				4	Divisi
[loop in lambda_diagonal_layer at radiation_GS_...	☐	2 Possible i...	0.330s	0.910s	Vectorized (Body; ...		AVX2	~13%	1.04x	8	Blend
[loop in quenchy at quench3_mpi.f90:159]	☐		0.300s	0.300s	Vectorized (Body)		AVX2	~61%	4.86x	8	Divisi
[loop in quenchx at quench3_mpi.f90:113]	☐		0.220s	0.220s	Vectorized (Body; ...		AVX2	~93%	7.45x	8	Divisi
[loop in integration_constants at radiation_GS_m...	☐		0.200s	1.510s	Scalar	vectorization possible but ...				4	

File: radiation_GS_mpi.f90:3653 compute_si_layer

Line	Source	Total Time	%	Loop Time	%	Traits
3646	`! Higher-order interpolation constants`					
3647	`real :: u,v,a1z,a2z,a3z,dz1,dz2,alpha1,alpha2,a1,a2,a3,a4,d1,d2,d1md2`					
3648	`real :: df,df1,df2`					
3649	`! Helper array`					
3650	`real,dimension(xsbr:xebr,ysbr:yebr) :: scr`					
3651						
3652	`! Compute the source function integral`					
3653	`SIlocal=psilayer(xs:xe,ys:ye,1)*Slayerup+ &`	0.360s		0.710s		FMA
3654	`& psilayer(xs:xe,ys:ye,2)*S(xs:xe,ys:ye,k,bin)+ &`	0.020s				
3655	`& psilayer(xs:xe,ys:ye,3)*Slayerdn`					
3656						
3657	`! Fill intensities from neighbor subdomains into the ghost zones, except`					
3658	`! for 2D case`					
3659	`if(mx.ne.1)then`					
	Selected (Total Time):	0.360s				

Please refer to the Advisor documentation for more information on how to use this tool.

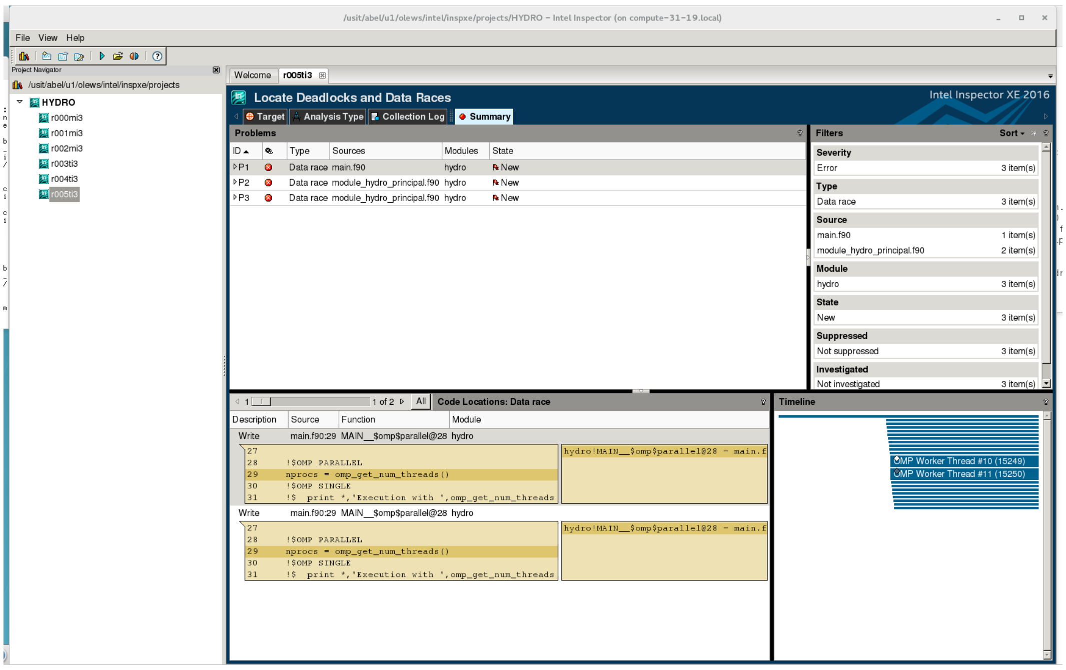
An official Intel FAQ about vectorization is found here : vectorization-advisor-faq [https://software.intel.com/en-us/articles/vectorization-advisor-faq]

7.2.4. Intel XE-Inspector

Intel Inspector is a dynamic memory and threading error checking tool for users developing serial and multithreaded applications.

The tuning tool XE-advisor as tool tailored for threaded shared memory applications (it can also collect performance data for hybrid MPI jobs using a command line interface). It provide analysis of memory and threading that might prove useful for tuning of any application. Usage of the application is not covered here, please refer to the documentation or the tutorial.

Figure 37. Inspector GUI example


/usit/abel/u1/olews/intel/inspxe/projects/HYDRO – Intel Inspector (on compute-31-19.local)
File View Help
Project Navigator
/usit/abel/u1/olews/intel/inspxe/projects
HYDRO
r000mi3
r001mi3
r002mi3
r003ti3
r004ti3
r005ti3
Welcome
r005ti3
Locate Deadlocks and Data Races
Intel Inspector XE 2016
Target
Analysis Type
Collection Log
Summary
Problems
Filters
Sort
Code Locations: Data race
Timeline
1 of 2
All
OMP Worker Thread #10 (15249)
OMP Worker Thread #11 (15250)

ID		Type	Sources	Modules	State
P1	●	Data race	main.f90	hydro	New
P2	●	Data race	module_hydro_principal.f90	hydro	New
P3	●	Data race	module_hydro_principal.f90	hydro	New

Severity	
Error	3 item(s)
Type	
Data race	3 item(s)
Source	
main.f90	1 item(s)
module_hydro_principal.f90	2 item(s)
Module	
hydro	3 item(s)
State	
New	3 item(s)
Suppressed	
Not suppressed	3 item(s)
Investigated	
Not investigated	3 item(s)

Description	Source	Function	Module
Write	main.f90:29	MAIN__ompparallel@28	hydro

```
27
28      !$OMP PARALLEL
29      nprocs = omp_get_num_threads()
30      !$OMP SINGLE
31      !$  print *,'Execution with ',omp_get_num_threads
```

```
hydro!MAIN__$omp$parallel@28 - main.f
```

Description	Source	Function	Module
Write	main.f90:29	MAIN__ompparallel@28	hydro

```
27
28      !$OMP PARALLEL
29      nprocs = omp_get_num_threads()
30      !$OMP SINGLE
31      !$  print *,'Execution with ',omp_get_num_threads
```

```
hydro!MAIN__$omp$parallel@28 - main.f
```

This tool is published by Intel and its web page is: Intel inspector [https://software.intel.com/en-us/intel-inspector-xe]

Intel provide an official tutorials for the Inspector tool: [https://software.intel.com/en-us/articles/inspectorxe-tutorials]

7.2.5. Intel VTune Amplifier

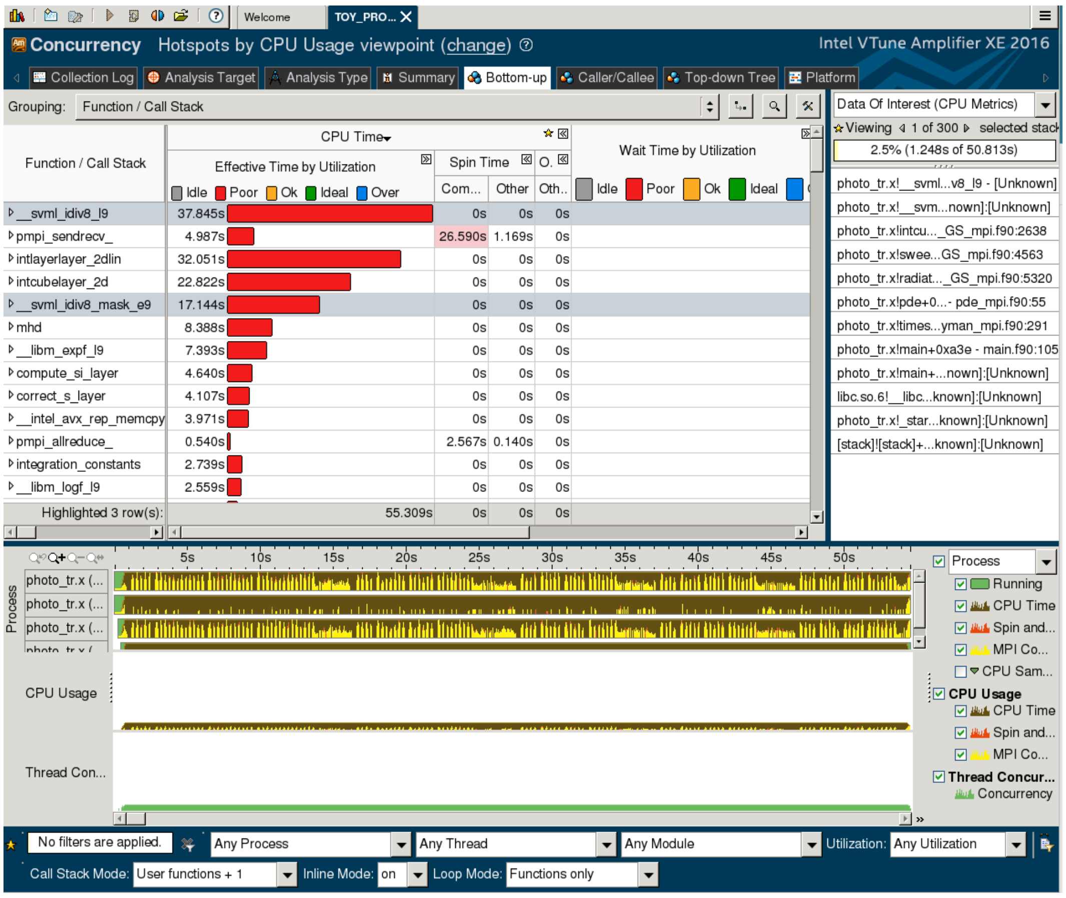
Intel VTune Amplifier provides a rich set of performance insight into CPU performance, threading performance and scaleability, bandwidth, caching and much more. Originally a tool for processor development, hence the strong focus on hardware counters. The usage of hardware counters require a kernel module to be installed which again require root access. Once installed the module can be accessed by any user.

Analysis is fast and easy because VTune Amplifier understands common threading models and presents information at a higher level that is easier to interpret. Use its powerful analysis to sort, filter and visualize results on the timeline and on your source. However, the sheer amount of information is sometimes overwhelming and in some cases intimidating. To really start using VTune Amplifier some basic training is suggested.

This tool is published by Intel and its web page is: VTune Amplifier [https://software.intel.com/en-us/intel-vtune-amplifier-xe]

VTune can operate as a command line tool and a X11 graphical version with an extensive GUI.

Figure 38. Amplifier GUI example



Grouping: Function / Call Stack

Function / Call Stack	CPU Time: Effective Time by Utilization	Spin Time: Com...	Spin Time: Other	O.: Oth..
__svml_idiv8_l9	37.845s	0s	0s	0s
pmpi_sendrecv_	4.987s	26.590s	1.169s	0s
intlayerlayer_2dlin	32.051s	0s	0s	0s
intcubelayer_2d	22.822s	0s	0s	0s
__svml_idiv8_mask_e9	17.144s	0s	0s	0s
mhd	8.388s	0s	0s	0s
__libm_expf_l9	7.393s	0s	0s	0s
compute_si_layer	4.640s	0s	0s	0s
correct_s_layer	4.107s	0s	0s	0s
__intel_avx_rep_memcpy	3.971s	0s	0s	0s
pmpi_allreduce_	0.540s	2.567s	0.140s	0s
integration_constants	2.739s	0s	0s	0s
__libm_logf_l9	2.559s	0s	0s	0s
Highlighted 3 row(s):	55.309s	0s	0s	0s

Data Of Interest (CPU Metrics)
Viewing 1 of 300 selected stack
2.5% (1.248s of 50.813s)

photo_tr.x!__svml...v8_l9 - [Unknown]
photo_tr.x!__svm...nown]:[Unknown]
photo_tr.x!intcu..._GS_mpi.f90:2638
photo_tr.x!swee...GS_mpi.f90:4563
photo_tr.x!radiat..._GS_mpi.f90:5320
photo_tr.x!pde+0...- pde_mpi.f90:55
photo_tr.x!times...yman_mpi.f90:291
photo_tr.x!main+0xa3e - main.f90:105
photo_tr.x!main+...nown]:[Unknown]
libc.so.6!__libc...known]:[Unknown]
photo_tr.x!_star...known]:[Unknown]
[stack]![stack]+...known]:[Unknown]

No filters are applied. Any Process | Any Thread | Any Module | Utilization: Any Utilization
Call Stack Mode: User functions + 1 | Inline Mode: on | Loop Mode: Functions only

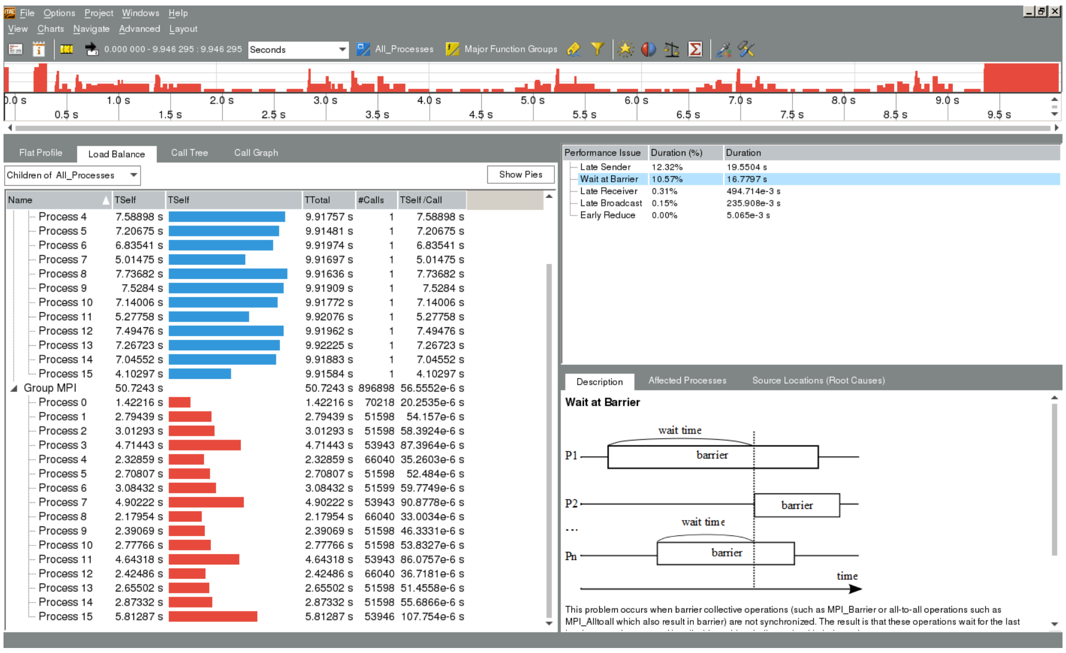
As for the other Intel performance tools this guide is not a guide about these tools. More information about the tools, how to obtain, install and use them must be found in the documentation. Official tutorials are found here: VTune tutorials [https://software.intel.com/en-us/articles/intel-vtune-amplifier-tutorials] and a FAQ is found here : VTune FAQ [https://software.intel.com/en-us/intel-vtune-amplifier-xe-support/faq]

7.2.6. Intel Trace Analyzer

Intel® Trace Analyzer and Collector is a graphical tool for understanding MPI application behavior, quickly finding bottlenecks, improving correctness, and achieving high performance for parallel cluster applications based on Intel architecture. Improve weak and strong scaling for small and large applications with Intel Trace Analyzer and Collector.

The collector tool is closely linked to the MPI library and the profiler library must be compiled and linked with the application. There is an option to run using a preloaded library, but the optimal way is to link in the collector libraries at build time.

Figure 39. Trace Analyzer GUI example



Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Process 4	7.58898 s	🟦	9.91757 s	1	7.58898 s
Process 5	7.20675 s	🟦	9.91481 s	1	7.20675 s
Process 6	6.83541 s	🟦	9.91974 s	1	6.83541 s
Process 7	5.01475 s	🟦	9.91697 s	1	5.01475 s
Process 8	7.73682 s	🟦	9.91636 s	1	7.73682 s
Process 9	7.5284 s	🟦	9.91909 s	1	7.5284 s
Process 10	7.14006 s	🟦	9.91772 s	1	7.14006 s
Process 11	5.27758 s	🟦	9.92076 s	1	5.27758 s
Process 12	7.49476 s	🟦	9.91962 s	1	7.49476 s
Process 13	7.26723 s	🟦	9.92225 s	1	7.26723 s
Process 14	7.04552 s	🟦	9.91883 s	1	7.04552 s
Process 15	4.10297 s	🟦	9.91584 s	1	4.10297 s
Group MPI	50.7243 s		50.7243 s	896898	56.5552e-6 s
Process 0	1.42216 s	🟥	1.42216 s	70218	20.2535e-6 s
Process 1	2.79439 s	🟥	2.79439 s	51598	54.157e-6 s
Process 2	3.01293 s	🟥	3.01293 s	51598	58.3924e-6 s
Process 3	4.71443 s	🟥	4.71443 s	53943	87.3964e-6 s
Process 4	2.32859 s	🟥	2.32859 s	66040	35.2603e-6 s
Process 5	2.70807 s	🟥	2.70807 s	51598	52.484e-6 s
Process 6	3.08432 s	🟥	3.08432 s	51599	59.7749e-6 s
Process 7	4.90222 s	🟥	4.90222 s	53943	90.8778e-6 s
Process 8	2.17954 s	🟥	2.17954 s	66040	33.0034e-6 s
Process 9	2.39069 s	🟥	2.39069 s	51598	46.3331e-6 s
Process 10	2.77766 s	🟥	2.77766 s	51598	53.8327e-6 s
Process 11	4.64318 s	🟥	4.64318 s	53943	86.0757e-6 s
Process 12	2.42486 s	🟥	2.42486 s	66040	36.7181e-6 s
Process 13	2.65502 s	🟥	2.65502 s	51598	51.4558e-6 s
Process 14	2.87332 s	🟥	2.87332 s	51598	55.6866e-6 s
Process 15	5.81287 s	🟥	5.81287 s	53946	107.754e-6 s

Performance Issue	Duration (%)	Duration
Late Sender	12.32%	19.5504 s
Wait at Barrier	10.57%	16.7797 s
Late Receiver	0.31%	494.714e-3 s
Late Broadcast	0.15%	235.908e-3 s
Early Reduce	0.00%	5.065e-3 s

This problem occurs when barrier collective operations (such as MPI_Barrier or all-to-all operations such as MPI_Alltoall which also result in barrier) are not synchronized. The result is that these operations wait for the last

Information about the tool: Intel Trace Analyzer [https://software.intel.com/en-us/intel-trace-analyzer] and a user guide Intel Trace Analyzer user guide. [https://software.intel.com/en-us/node/561463]

7.3. Single Core Optimization

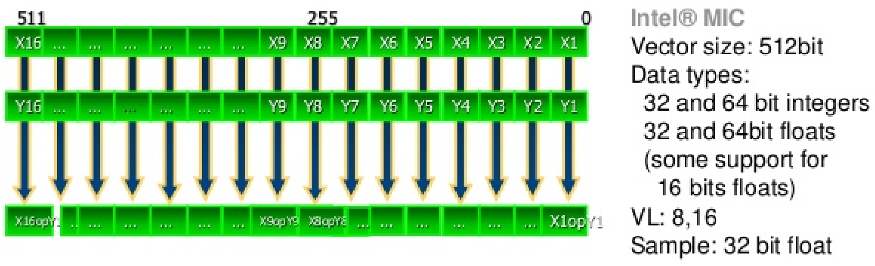
Single core optimization also cover each individual MPI rank as the individual ranks are often single core executables. In this section things that typically optimizes the sequential part of the code, e.g. each rank for pure MPI application. Vectorization is an important element of this section as the performance gain using the vector units are potentially quite high.

7.3.1. Vectorization

A vector instruction is an SIMD instruction, Single Instruction Multiple Data. A vector instruction refers to vector registers where multiple data resides. For example, a Cray-1's vector register contained up to 64 64-bit double-precision floating point numbers. The Cray-1 had eight of these registers. Many operations, for example: add and multiply can be issued to add or multiply two vector registers and place the result in a third vector register, it could even do fused multiply add (with a performance at 140 Mflops/s).

Vectorization is still very important. The vector units in KNL are 512 bits wide. They can hence operate on 16 single precision (32 bits) or 8 double precision (64 bits) numbers simultaneously, often in one clock cycle. A possible speedup of 8 or 16 compared to none vectorized code is possible. The details about how the processor schedule the four hardware threads onto this vector unit is found in Section 2.1.1. Recall that one thread can only schedule a vector instruction every second clock cycle, hence two threads per core is needed to schedule a vector instruction every clock cycle.

Figure 40. Vector unit on KNL


511
255
0
X16 ... ... X9 X8 X7 X6 X5 X4 X3 X2 X1
Y16 ... ... Y9 Y8 Y7 Y6 Y5 Y4 Y3 Y2 Y1
X16opY16 ... ... X9opY9 X8opY8 ... ... X1opY1
Intel® MIC
Vector size: 512bit
Data types:
32 and 64 bit integers
32 and 64bit floats
(some support for
16 bits floats)
VL: 8,16
Sample: 32 bit float

A scalar code will only use one if these entries in the vector and performance will only be a fraction (1/16 th or 1/8 th) of what is possible. The very high performance of the modern vector enabled processors is only possible to achieve when the vector are fully utilized. While the compiler does a good job at automatically vectorize the code it is often that an analysis will provide insight and open up for vectorizing more loops. For this purpose there are different tool that can collect and display the analysis. Some tools will be discussed later in this section.

Use:

- Straight-line code (a single basic block).
- Vector data only; that is, arrays and invariant expressions on the right hand side of assignments. Array references can appear on the left hand side of assignments.
- Only assignment statements.

Avoid:

- function calls (other than math library calls, it's very hard for the compiler to figure out if this is a vector function or not, using inlining can enable vectorization of functions).
- non-vectorizable operations (either because the loop cannot be vectorized, or because an operation is emulated through a number of instructions).
- mixing vectorizable types in the same loop (leads to lower resource utilization).
- data-dependent loop exit conditions (leads to loss of vectorization).

To make your code vectorizable, you will often need to make some changes to your loops. You should only make changes needed to enable vectorization, and avoid these common changes:

- loop unrolling, which the compiler performs automatically.
- decomposing one loop with several statements in the body into several single-statement loops.

7.3.2. Compiler auto vectorization

The compiler tries to vectorize loops automatically when the appropriate compiler flags are set. An analysis report of this process are provided by the compiler. This report will also give hints about why some loops fail to vectorize. Following the guidelines given above the compiler stand a good change of auto vectorizing your code.

There are however several ways of helping the compiler to vectorize the code, the modern way of doing this in a portable way is to use OpenMP 4.0 SIMD directives. The older Intel specific directives work well on Intel compilers, but are not portable. See Table 18 for more information about reporting compiler options. For hints on using OpenMP SIMD instructions please refer to Section 7.5.

Intel provide an array of presentations and videos about vectorizations. An interesting webinar about this subject found in this link [https://software.intel.com/en-us/videos/from-serial-to-awesome-part-2-advanced-code-vectorization-and-optimization], and this [https://software.intel.com/en-us/videos/vectorizing-fortran-using-openmp-4x-filling-the-simd-lanes]. These presentations contain pointers to a lot of relevant information and represent a good starting point.

7.3.3. Interprocedural Optimization

Interprocedural Optimization (IPO) is an automatic, multi-step process that allows the compiler to analyze your code to determine where you can benefit from specific optimizations. As each source file is compiled with IPO, the compiler stores an intermediate representation (IR) of the source code in a mock object file. The mock object files contain the IR instead of the normal object code. Mock object files can be ten times or more larger than the size of normal object files. During the IPO compilation phase only the mock object files are visible.

Before embarking on the rather tedious job to read and understand the IPO optimization report one must first do a profiling of the application to gather information about which part of the code that uses most CPU-time. After that one might do a deeper analysis of the relevant part of the source code. The optimization reports are quite detailed and analysis of the output might take considerable time.

7.3.3.1. IPO link time reporting

When you link with the -ipo compiler option the compiler is invoked a final time. The compiler performs IPO across all mock object files. The mock objects must be linked with the Intel compiler or by using the Intel linking tools. While linking with IPO, the Intel compilers and other linking tools compile mock object files as well as invoke the real/true object files linkers provided on the user's platform.

Using the Intel linker, xild, with its options one will get the IPO optimization report. This report will reveal the work done using the intermediate representation of the object files.

An example of a command line might look like this:

```
xild -qopt-report=5 -qopt-report-file=ipo-opt-report.txt
```

The optimization report will contain valuable information about what the IOP machinery did. An example is given below, where few lines are shown:

```
LOOP BEGIN at square_gas_mpi.f90(666,8)
   remark #15541: outer loop was not auto-vectorized: consider
   using SIMD directive
```

with furthermore inside the current loop:

```
LOOP BEGIN at square_gas_mpi.f90(666,8)
     remark #25084: Preprocess Loopnests: Moving Out Store
     [square_gas_mpi.f90(666,8)]

     remark #15331: loop was not vectorized: precise FP model implied
     by the command line or a directive prevents vectorization. Consider using
     fast FP model

     remark #25439: unrolled with remainder by 2
LOOP END
```

or like this example:

```
LOOP BEGIN at square_gas_mpi.f90(682,5)
   remark #15388: vectorization support: reference var has aligned access
   remark #15388: vectorization support: reference var has aligned access
   remark #15388: vectorization support: reference var has aligned access
   remark #15305: vectorization support: vector length 2
   remark #15300: LOOP WAS VECTORIZED
   remark #15450: unmasked unaligned unit stride loads: 2
   remark #15451: unmasked unaligned unit stride stores: 1
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 6
   remark #15477: vector loop cost: 2.500
   remark #15478: estimated potential speedup: 1.630
   remark #15488: --- end vector loop cost summary ---
   remark #25015: Estimate of max trip count of loop=1
LOOP END
```

The finer details of the meaning of this output must be found in the compiler and linker documentation provided with the Intel compiler suite.

7.3.4. Intel Advisor tool

7.3.4.1. Collecting performance data

Using the command line version of the advisor tool for single threaded applications is quite straightforward. The tool can launch the application on the command line and run just as normal.

```
advixe-cl -collect survey myprog.x
```

Collecting the survey is just the start, there are many more possible collections. The built in help functions in addition to suggestions presented by the GUI a deep analysis can be collected.

7.3.4.2. Displaying performance data

Displaying the collect performance data can be done using the GUI or displayed as text in the terminal using the command line version of the tool.

```
advixe-cl -report survey
```

This will produce a report of what the tool collected during the run. There are several possible reports, please refer to the Intel Advisor tool to get more information.

It should be pointed out that compiler reports should be at hand when working with the Advisor GUI. For vectorization analysis the following flags can be useful when compiling.

```
-qopt-report5 -qopt-report-phase=vec
```

These reports are vital as this is the only time the actual source code is analyzed. Messages like:

```
LOOP BEGIN at EXTRAS/spitzer_conductivity_mpi.f90(2181,5)
  remark #15541: outer loop was not auto-vectorized: consider using SIMD directive
LOOP BEGIN at EXTRAS/spitzer_conductivity_mpi.f90(2185,7)
  remark #15541: outer loop was not auto-vectorized: consider using SIMD directive

LOOP BEGIN at EXTRAS/spitzer_conductivity_mpi.f90(2191,9)
  remark #15344: loop was not vectorized: vector dependence prevents vectorization
  remark #15346: vector dependence: assumed ANTI dependence between tg line 2220
  and tg line 2310
  remark #15346: vector dependence: assumed FLOW dependence between tg line 2310
  and tg line 2220
```

provide valuable insight to what the compiler understand of the source code. It is playing it safe and assume dependencies where there might not be one, only the programmer can know this. Informing the compiler about this can in some cases suddenly flip a scalar code to a vectorized one with a potential gain in performance.

The combination of compiler reports and the advisor tool a large fraction of single core optimization challenges can be addressed. For the more advanced performance issues like memory access the more CPU related performance tool VTune-Amplifier can be employed.

7.3.5. Intel VTune Amplifier tool

7.3.5.1. Collecting performance data

The Amplifier GUI can be used to launch and run the application, but it's often necessary to collect the analysis on another node. The command line tool provide a range of functions for both collections and analysis. Syntax is quite simple :

```
amplxe-cl -collect hotspots -r <directory>  Bin/hydro  Input/2500x2500.nml
amplxe-cl -collect advanced-hotspots  -r <directory> Bin/hydro  Input/2500x2500.nml
amplxe-cl -collect memory-access  -r <directory>  Bin/hydro  Input/2500x2500.nml
```

There is a comprehensive online help with the command line tool. There is a range of different collection options, the most common is shown above. The Amplifier collection phase make use of hard links and not all file systems support this.

7.3.5.2. Displaying performance data

The GUI would normally be used to display and analyze the collected performance data as this provide a nice overview of the data and a powerful navigation space.

However, many times there is a need to get the results in a text report format. The collection phase provide a summary report. To display the full result in text format the command line tool can be used to prepare full reports.

```
amplxe-cl -report hotspots -r <directory>
```

The report contain a range of collected metrics of which many does not make sense for a non trained user. For the non experts the GUI provide a far better tool to view and analyze the collected data.

7.4. Threaded performance tuning

7.4.1. Shared memory / single node

Shared memory application make up a significant part of the total application suite. Their performance rely on effective vectorization and threading usage. Tuning of OpenMP is covered in Section 7.5.

7.4.2. Core count and scaling

As always one of the fist steps is to assess the performance increase as the number of cores and threads increase. Plotting speedup versus core count is always a good start. Not all threaded applications scale well to a core count of 64 which can be found on some of today's systems. Hence assessment of scaling is important. Please see Section 4.2.1 and figure Figure 18 for more on examples with scaling results.

One simple way of overcome poor scaling scaling problem is to increase the problem size. Review Gustavsson's law to understand more about this.

7.4.3. False sharing

When programming for cache coherent shared memory the memory is shared, but the caches are local. Multiple threads might operate on separate memory locations, but close enough to be held in one cache line. Any write operation on any of the data contained in the cache will immediately invalidate the cache lines for the other threads. Forcing a writeback and a reload of all the other caches. This is a relatively costly process and will impede the performance.

A relatively simple example might illustrate this:

```
!$omp parallel private(i,j) shared(a,s)
       do i=1,m
          s(i)=0.0
          do j=1,n
             s(i)=s(i)+a(i,j)
          enddo
       enddo
!$omp end parallel
```

This code will run at nicely using one or any number of threads, but the performance will be far higher using a single thread than any multiple thread run. A simple test with the above code show that it took twice as long time using two thread compared to a single thread.

7.4.4. Intel XE-Inspector tool

7.4.4.1. Collecting performance data

Collecting data using the command line tool is quite straightforward.

```
inspxe-cl -collect=mi2  myprog.x
```

7.4.4.2. Analyzing performance data

Analysis of the collect data can be done using the GUI or using tools to display it as ASCII text on the terminal.

The following command can be issued to output a short summary of problems detected in text format:

```
inspxe-cl -report problems
```

The output can look like this:

```
P1: Error: Invalid memory access: New
P1.117: Error: Invalid memory access: New
P2: Warning: Memory not deallocated: New
P2.140: Warning: Memory not deallocated: 23 Bytes: New
```

The command line tool is quite powerful and will provide all the data need so that they can be analyzed using user scripts.

The graphical interface provide a quick overview of the collected data and is an interactive tool for exploring the applications thread and memory performance.

7.5. Advanced OpenMP Usage

OpenMP version 4.0 specify both thread parallelism and SIMD vector parallelism. There is also the concept of workshare which is mostly another form of threading. See the OpenMP web site [http://openmp.org] for more information. SIMD OpenMP tutorial is beyond the scope of this guide. Only examples of how us use the SIMD OpenMP and the tools to do so will be covered in this guide. As vectorization is very important it's vital to use the techniques listed in Section 7.3.1 to tune the vectorization.

Usage of the threading Intel tools are covered in the chapter above and the same analysis tools will be just as applicable for applications in this section.

Intel provides more information on this topic in this article [https://software.intel.com/en-us/articles/threading-fortran-applications-for-parallel-performance-on-multi-core-systems].

7.5.1. SIMD vectorization

The SIMD parallelization of OpenMP is relatively new, introduced with the OpenMP 4.0 standard in July 2013, see OpenMP 4.0 Specifications [http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf], chapter "SIMD Constructs" for details. It was introduced to provide portable mechanisms to describe when multiple iterations of the loop can be executed concurrently using SIMD instructions and to describe how to create versions of functions that can be invoked across SIMD lanes.

Quite often the compiler cannot auto vectorize the code and the programmer need to help or instruct the compiler to vectorize the code. Here is where the OpenMP SIMD directives comes in handy.

Table 20. OpenMP SIMD flags, Intel compilers

Default flag	Description
-qopenmp	Enable full OpenMP support, including SIMD support

Default flag	Description
-qopenmp-simd	Only interpret the OpenMP SIMD directives.
-no-openmp-simd	Disable OpenMP SIMD directives.

OpenMP SIMD directives are used to instruct the compiler to use the SIMD vector unit as the programmer want. This will override the default vectorization where the compiler might back off as it cannot know that it's safe to vectorize. The Intel specific directives like *$IVDEP* (Ignore Vector Dependencies) is not portable and it's suggested to use OpenMP SIMD directives instead. The OpenMP SIMD is far more powerful as it's part of the OpenMP standard, only that is uses the vector unit for parallel processing as opposed to the threading model more commonly associated with OpenMP.

Below is an example of how powerful the OpenMP SIMD directives can be, including reductions, far more powerful that the Intel specific directives.

```
!$omp simd private(t) reduction(+:pi)
  do i=1, count
     t = ((i+0.5)/count)
     pi = pi + 4.0/(1.0+t*t)
  enddo
  pi = pi/count
```

It's just like the syntax used for threading except that operations are run in parallel on a vector unit and not as parallel threads.

A lot of teaching materials have been produced. An Intel presentation about Vectorization using SIMD directives can be found here [https://software.intel.com/en-us/videos/vectorizing-fortran-using-openmp-4x-filling-the-simd-lanes]. The presentation contain links to more information from Intel. As the vectorization becomes more and more important it well spend effort to spend time learning how OpenMP SIMD can be beneficial.

7.5.2. OpenMP Thread parallel

The OpenMP thread parallel programming model will not be covered here. There are a lot of tutorials and books covering this topic. This section gives an overview of the tools that can be used for tuning and using the compiler's built in diagnostic and ways of using the OpenMP implementation.

Table 21. OpenMP flags

Default flag	Description
-qopenmp	enable the compiler to generate multi-threaded code based on the OpenMP* directives.
-fopenmp	Same as -qopenmp.
-qopenmp-stubs	Enables the user to compile OpenMP programs in sequential mode.
-qopenmp-report{0|1|2}	Control the OpenMP parallelizer diagnostic level
-qopenmp-lib=<ver>	Choose which OpenMP library version to link with • compat - use the GNU compatible OpenMP run-time libraries (DEFAULT)
-qopenmp-task=<arg>	Choose which OpenMP tasking model to support: • omp - support OpenMP 3.0 tasking (DEFAULT) • intel - support Intel taskqueuing
-qopenmp-threadprivate=<ver>	Choose which threadprivate implementation to use • compat - use the GNU compatible thread local storage • legacy - use the Intel compatible implementation (DEFAULT)

The KNL architecture is a cache coherent none uniform memory architecture that facilitates shared memory programming like OpenMP. However, the sheer number of cores and threads can represent a scaling problem. Not all OpenMP applications scale to this high core count and adding to that the number of transactions placed on the cache coherency machinery when running with this high core count might also limit the scaling. While the KNL is a well founded architecture some threaded applications might still overwhelm the cc-NUMA transaction machinery.

7.5.3. Tuning / Environment Variables

7.5.3.1. Thread placements

The environment variables :

- KMP_PLACE_THREADS
- KMP_AFFINITY
- OMP_PROC_BIND

allow you to control how the OpenMP runtime uses the hardware threads on the processors. The variables starting with KMP are specific to the Intel compilers while the variables starting with OMP are GNU variables which are also honored by the Intel run time system. The KMP_AFFINITY variable controls how the OpenMP threads are bound to the hardware resources allocated by the KMP_PLACE_THREADS variable. The OMP_ and the KMP_ have slightly different syntax and meaning, OMP_PROC_BIND=true is the same as KMP_AFFINITY=scatter. Intel have a web page about environment variables [https://software.intel.com/en-us/node/522775].

7.5.3.2. Thread Affinity

Controlling thread allocation is well documented here. [https://software.intel.com/en-us/node/581389] The following are the recommended affinity types to use to run your OpenMP threads on the processor:

- compact: sequentially distribute the threads among the cores that share the same cache (Level 1-2).
- scatter: distribute the threads among the cores without regard to the cache.

The following table illustrate how the threads are bound to the cores when you want to use four threads per core on two cores by specifying KMP_PLACE_THREADS=2c,4t :

Table 22. Affinity settings

Affinity	OpenMP* Threads on Core 0	OpenMP* Threads on Core 1
KMP_AFFINITY=compact	0, 1, 2, 3	4, 5, 6, 7
KMP_AFFINITY=scatter	0, 2, 4, 6	1, 3, 5, 7

The above table is for just a very simple example. The KNL has up to 72 cores. For a chip with 64 cores the numbers will be more like 64c,1t through 64c,4t. These settings can have major performance impact. It is however also depending of the chip clustering model selected. For Quadrant and Hemisphere layouts and high bandwidth memory set to last level cache there is only one NUMA and the effects expected to be small. It's a rather large space to explore and not all this space can be covered.

To see in real time the layout and load on the various cores use the utility htop, described in figure Figure 50 and described in 7.6.7: “Mapping Tasks on Node Topology”.

7.5.3.3. Excecution control

The Intel OpenMP library contain several execution modes. These modes can be selected by environment flags.

- KMP_LIBRARY
- KMP_BLOCKTIME

The library select different run time libraries, see the table below.

Table 23. Run time library selection

Library selected	Information
serial	Run in serial mode with one thread only
turnaround	Idle threads do not yield to other processes
throughput (default)	Idle threads sleep and yield after KMP_BLOCKTIME msec

The KMP_BLOCKTIME variable set thread wait time before going to sleep. It's the amount of time a thread will be active and not yield to other threads that are waiting to run. There is a cost of waking up a sleeping thread so keeping it awake might be beneficial if new work is imminent. This is a variable that can be tuned and its impact can be significant, as the table below with some numbers from the testing of the application LS-Dalton hope to illustrate.

Table 24. Effect of KMP_BLOCKTIME settings

KMP_BLOCKTIME [ms]	Wall time [secs]
2	1721
10	1720
50	1734
100	1741
200	1750

7.5.4. Usage of numactl

The utility numactl which is commonly availble on current Linux systems is an excellent tool for placing threads into both memory nodes (NUMA nodes) and cores. Basic information about numactl if found on the man page, more information is commonly available on the internet.

To list the NUMA nodes use the flag "-H" for hardware or "-s" for policy settings. Both will show the current layout of the processor. An example with the chip layout is set to SNC-4 is given below.

```
available: 4 nodes (0-3)
node 0 cpus: 9 10 19 20 29 30 39 40 41 81 82 83 84 85 86 87 88 89 90 91 92 93 94
135 136 137 138 139 140 141 142 143 144 145 146 147 148 189 190 191 192 193 194
195 196 197 198 199 200 201 202 243 244 245 246 247 248 249 250 251 252 253 254 255
node 0 size: 8192 MB
node 0 free: 7823 MB
node 1 cpus: 0 1 2 3 4 11 12 13 14 21 22 23 24 31 32 33 34 42 43 44 45 46 47 48
49 50 51 52 95 96 97 98 99 100 101 102 103 104 105 106 149 150 151 152 153 154
155 156 157 158 159 160 203 204 205 206 207 208 209 210 211 212 213 214
node 1 size: 8088 MB
node 1 free: 7278 MB
node 2 cpus: 5 6 15 16 25 26 35 36 53 54 55 56 57 58 59 60 61 62 63 64 65 66
107 108 109 110 111 112 113 114 115 116 117 118 119 120 161 162 163 164 165
166 167 168 169 170 171 172 173 174 215 216 217 218 219 220 221 222 223 224
225 226 227 228
node 2 size: 8192 MB
node 2 free: 7816 MB
node 3 cpus: 7 8 17 18 27 28 37 38 67 68 69 70 71 72 73 74 75 76 77 78 79 80
121 122 123 124 125 126 127 128 129 130 131 132 133 134 175 176 177 178 179
180 181 182 183 184 185 186 187 188 229 230 231 232 233 234 235 236 237 238
239 240 241 242
node 3 size: 8192 MB
node 3 free: 7845 MB
node distances:
node   0   1   2   3
```

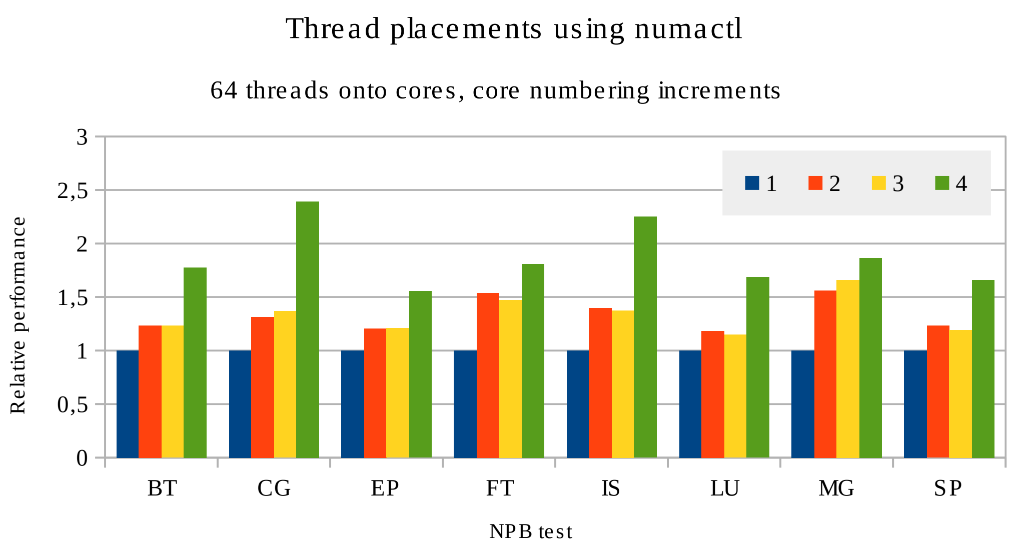
```
  0:  10  21  21  21
  1:  21  10  21  21
  2:  21  21  10  21
  3:  21  21  21  10
```

The cores numbering is not regular making simple lists of cores harder to make.

Common usage of numacatl are local allocation of memory by the flag "-l" or interleaved allocation with the flag "-i <nodes>" where some NUMA (could be all or a subset) nodes are used.

Placing threads on cores is not trivial, putting them close with several threads on a single core give good cache effects while the possible starvation of execution units. Placing the threads using one thread per physical core the opposite effects are in force. The following figure show this effect where 64 allocated cores are numbered from 0 to in with increments of 1, 2, 3 and 4 effectively placing anything from one four threads to a single thread per core.

Figure 41. Thread placement on differently scheduled cores



For these tests it seems that the starvation of executional units is more important than data locality.

One might also use *numactl* to control which numa node the memory should be allocated from fist. This is relevant for the case where the HBM is set to "flat" and the NHM is exposed as numa nodes, show here using *numactl -H*:

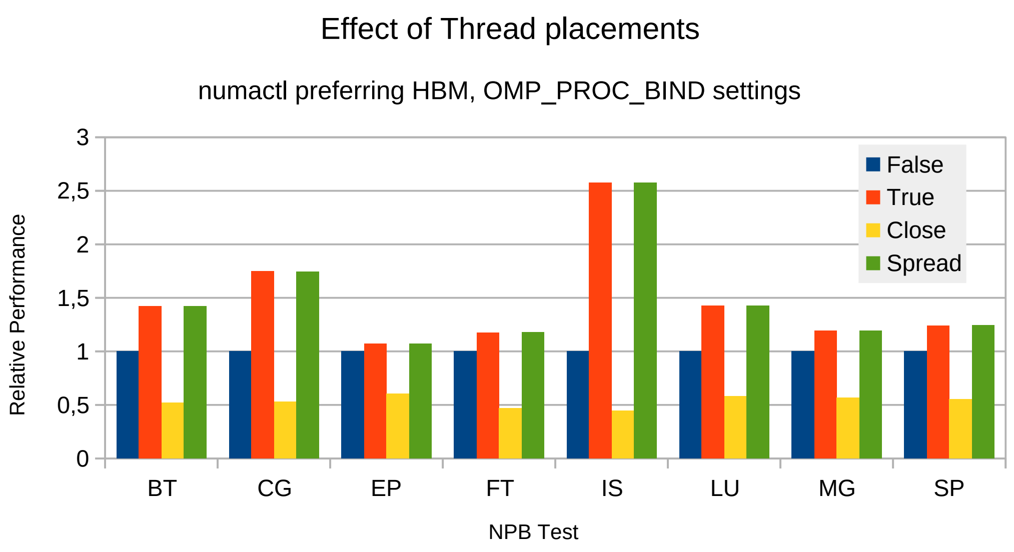
```
node 0 size: 32664 MB
node 0 free: 30720 MB
node 1 cpus:
node 1 size: 8192 MB
node 1 free: 7905 MB
node distances:
node   0   1
  0:  10  31
  1:  31  10
Cluster and HBM settings
 Cluster mode: Quadrant
 Memory Mode: Flat
```

In this case the *numactl* can be used to set memory allocation to prefer the HBM nodes using *--preferred=1* (we see from the size, 8 GiB that this is the HBM) argument. The command issued look like:

```
numactl --preferred=1  bin/prog.x
```

The effect of processor binding using the flag OMP_PROC_BIND when numactl is used to set used the preferred NUMA memory bank (*numactl --preferred=N*) is shown in the figure below:

Figure 42. Processor binding, with allocation set to use HBM.



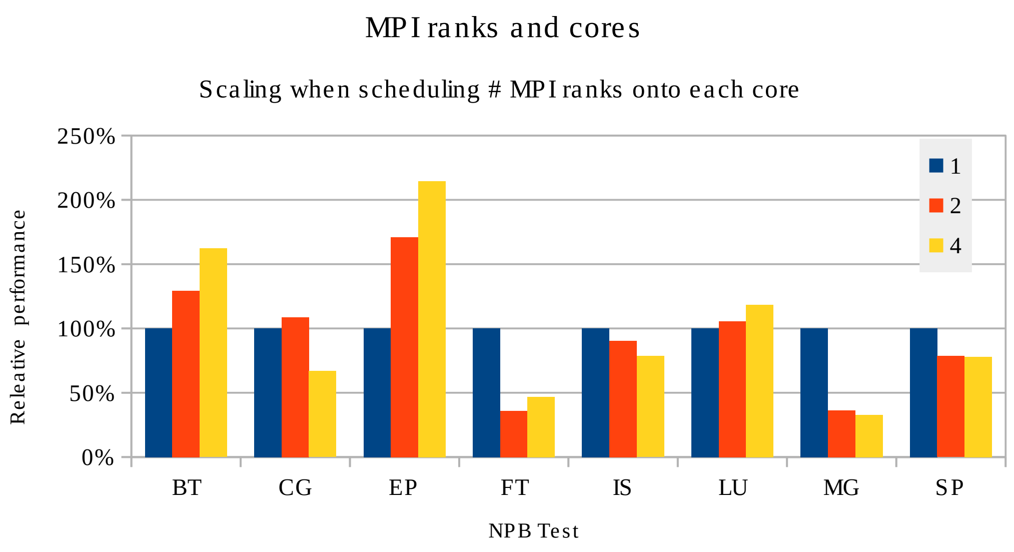
7.6. Advanced MPI Usage

7.6.1. Effect of multiple ranks per core

Each core has four hardware threads, and hence might run four ranks per core. Scheduling more than one rank per core might starve the application for executions units, but on the other hand hide memory latency and keeping the execution units fully occupied. It's not easy to tell a priori which direction this will pull. Testing is clearly needed for most applications.

By using the common benchmark set NPB in MPI version a range of different application cores can be tested. NPB exercises a range of memory access types and MPI communication patterns.

Figure 43. Ranks per core scheduling



From the figure we see the effect on the NPB applications when scheduling 1,2 and 4 cores per physical core.

The placement of ranks onto cores are controlled by environment variables. Environment variables are discussed in Section 7.6.6.

7.6.2. Intel Advisor tool

7.6.2.1. Collecting performance data

Collecting data with a MPI job is done using some extra options to mpirun, numbering follow the MPI ranks starting at 0 to np-1 :

```
mpirun  -np 8 -gtool "advixe-cl -collect survey:0" ./photo_tr.x
```

or collecting from all ranks :

```
mpirun  -np 4 -gtool "advixe-cl -collect survey:0,1,2,3" ./photo_tr.x
```

Data is collected for each individual rank, the advisor tool is not MPI aware, it collects performance data for each individual rank.

Another syntax where rank 2 and 3 are run with collection and rank 0,1 and 4 through 27 are run without collection.

```
mpirun -machinefile ./nodes -np 2 ./photo_tr.x : \
-np 2 advixe-cl -project-dir ./REAL_SMALL_27CPU -collect survey \
-search-dir src:r=./src/ -- ./photo_tr.x : -np 23 ./photo_tr.x
```

7.6.2.2. Displaying performance data

To display the data on the terminal the following command can be used:

```
advixe-cl -report survey –project-dir <DIR>
```

This will provide a report containg a lot of information about vectorisation of loops and their timing. It will tell if the loops were vectorised or not and if only parts of the loop was vectorised. A text file containing all the information is also provided.

```
stagger_mesh_mpi.f90:4483   photo_tr.x
518   -[loop in ddzdn_sts at stagger_mesh_mpi.f90:4483] 0s      0.1301s
Scalar outer loop was not auto-vectorized: consider using SIMD directive
```

To get help to interpreting the advisor output please consult the inspector documentation that comes with the software.

7.6.3. Intel VTune Amplifier tool

Intel Vtune Amplifier is a rather complex tool that normally require some training to use. There are several youtube and Intel videos in addition to training sessions hosted by Intel. Specially the hardware counters require knowledge to exploit. Only sampling using MPI will be touched upon here.

7.6.3.1. Collecting performance data

Collecting data with a MPI job is done using some extra options to mpirun, numbering follow the MPI ranks 0 and 2, placing the result into the current directory :

```
mpirun  -gtool "amplxe-cl -r <DIR> -collect hotspots:0,2" -np 4 ../photo_tr.x
```

or collecting from all ranks :

```
mpirun  -gtool "amplxe-cl -r <DIR>  -collect hotspots:all" -np 4 ../photo_tr.x
```

There are a range of different metrics to collect, the example show hotspots which is one of the more common. There are however, a range of different ones like advanced-hotspots, memory-access etc (to use these hardware based samples a special module must be loaded). Below is given two examples to collect using the hardware based sampling. One that sample all launched from mpirun the other for collecting only a single or many MPI rank.

```
amplxe-cl -r /tmp/general-expl -collect general-exploration\
-- mpirun -np 27 -machinefile ./nodes ./photo_tr.x

mpirun -np 27 -machinefile ./nodes -gtool "amplxe-cl -r /tmp/general-rank-0\
 -collect general-exploration:0" ./photo_tr.x

mpirun -np 27 -machinefile ./nodes -gtool "amplxe-cl -r /tmp/general-rank-all\
 -collect general-exploration:all" ./photo_tr.x
```

Please refer to the Intel VTune-Amplifier documentation for more information.

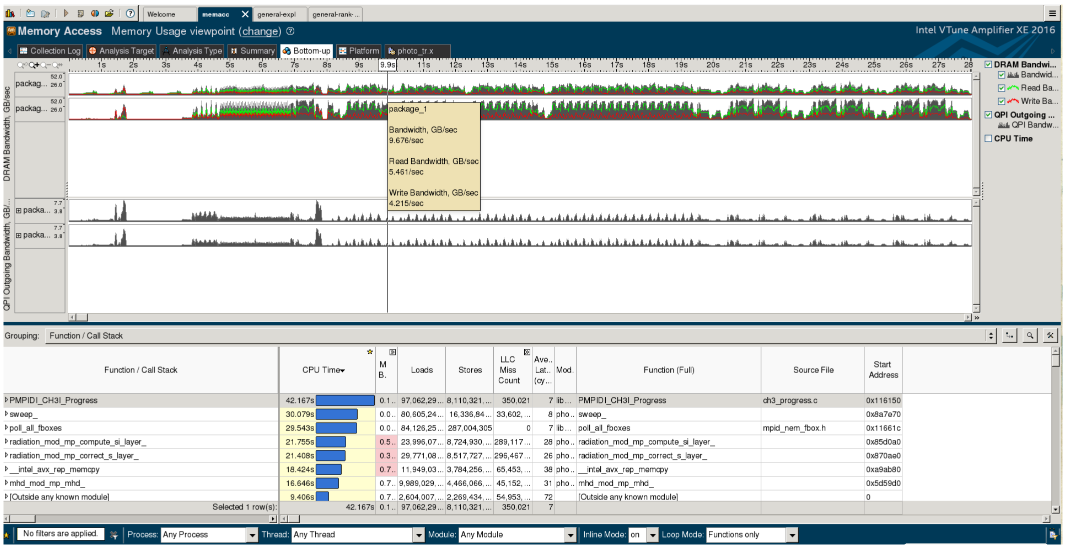
7.6.3.2. Displaying performance data

To display the data on the terminal the following command can be used:

```
amplxe-cl -report  hotspots -r <sample directory>
```

This will provide an extensive text report about all the samples collected related to the hotspots. More information about the content of this report can be found in the Amplifier documentation.

Figure 44. Example of Amplifier GUI with HW based sampling MPI collections



Grouping: Function / Call Stack

Function / Call Stack	CPU Time	M B.	Loads	Stores	LLC Miss Count	Ave.. Lat.. (cy...	Mod.	Function (Full)	Source File	Start Address
PMPIDI_CH3I_Progress	42.167s	0.1..	97,062,29...	8,110,321,...	350,021	7	lib...	PMPIDI_CH3I_Progress	ch3_progress.c	0x116150
sweep_	30.079s	0.0..	80,605,24...	16,336,84...	33,602,...	8	pho..	sweep_		0x8a7e70
poll_all_fboxes	29.543s	0.0..	84,126,25...	287,004,305	0	7	lib...	poll_all_fboxes	mpid_nem_fbox.h	0x11661c
radiation_mod_mp_compute_si_layer_	21.755s	0.5..	23,996,07...	8,724,930,...	289,117...	28	pho..	radiation_mod_mp_compute_si_layer_		0x85d0a0
radiation_mod_mp_correct_s_layer_	21.408s	0.3..	29,771,08...	8,517,727,...	296,467...	26	pho..	radiation_mod_mp_correct_s_layer_		0x870ae0
__intel_avx_rep_memcpy	18.424s	0.7..	11,949,03...	3,784,256,...	65,453,...	38	pho..	__intel_avx_rep_memcpy		0xa9ab80
mhd_mod_mp_mhd_	16.646s	0.7..	9,989,029,...	4,466,066,...	45,152,...	31	pho..	mhd_mod_mp_mhd_		0x5d59d0
[Outside any known module]	9.406s	0.7..	2,604,007,...	2,269,434,...	54,953,...	72		[Outside any known module]		0
Selected 1 row(s):	42.167s	0.1..	97,062,29...	8,110,321,...	350,021	7				

No filters are applied. Process: Any Process Thread: Any Thread Module: Any Module Inline Mode: on Loop Mode: Functions only

7.6.4. Intel Trace Analyser tool

7.6.4.1. Collecting MPI trace data

Collections is straightforward, just add the trace option to mpirun and it will load a trace enabled dynamic MPI library. It is also possible to link static during the link stage.

```
mpirun -trace -np 20 ../photo_tr.x
```

After completion there will be a range of files (two for each rank plus some others) with the application name with .stf and other suffixes appended.

It is possible to apply filters during the collection. The most common are point to point and collectives.

- -trace-pt2pt – to collect information only about point-to-point operations
- -trace-collectives – to collect information only about collective operations

An example taken from a real test is given below:

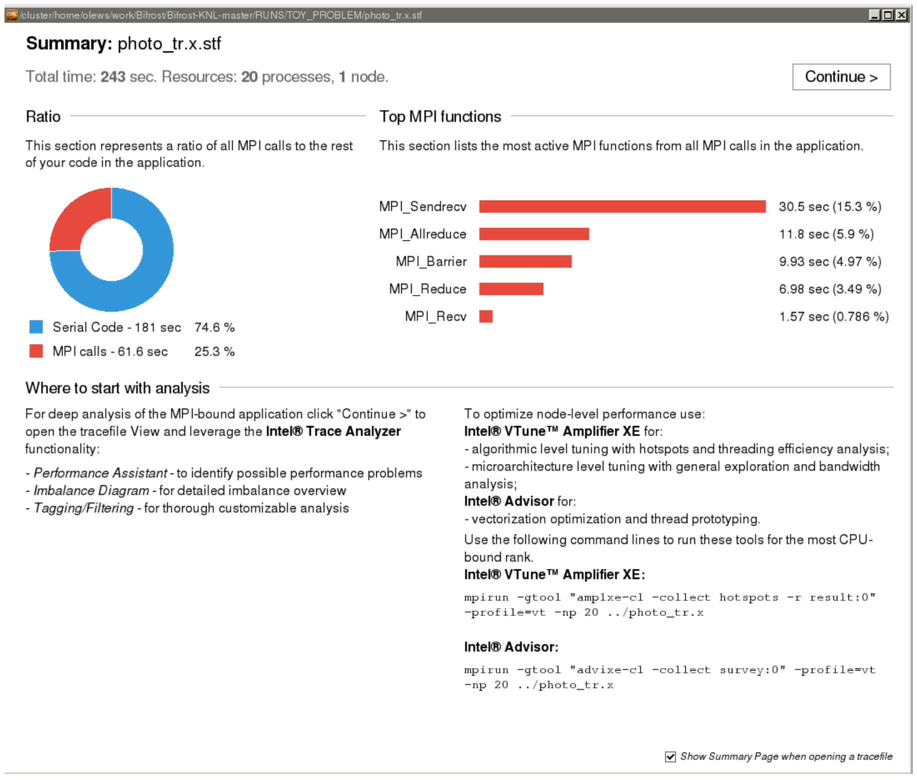
```
mpirun -np 27 -machinefile ./nodes  -trace -trace-collectives ./photo_tr.x
```

As there are a huge range of options and variants to the collection tool the documentation and manual should be consulted. You can prepare your own configiration file to the tool. Using this OS parameters can also be collected during the run. The guide for this trace collection rools can be found here [https://software.intel.com/en-us/itc-user-and-reference-guide].

7.6.4.2. Displaying and analysing MPI trace data

The collected data can be displayed with a command line tool or the graphical interface. The usage of the GUI is higly reccomended for this kind of complex analyses.

Figure 45. Trace Analyser GUI summary example



/cluster/home/olews/work/Bifrost/Bifrost-KNL-master/RUNS/TOY_PROBLEM/photo_tr.x.stf

Summary: photo_tr.x.stf

Total time: **243** sec. Resources: **20** processes, **1** node.

Ratio

This section represents a ratio of all MPI calls to the rest of your code in the application.

🟦	Serial Code - 181 sec	74.6 %
🟥	MPI calls - 61.6 sec	25.3 %

Top MPI functions

This section lists the most active MPI functions from all MPI calls in the application.

MPI_Sendrecv	30.5 sec (15.3 %)
MPI_Allreduce	11.8 sec (5.9 %)
MPI_Barrier	9.93 sec (4.97 %)
MPI_Reduce	6.98 sec (3.49 %)
MPI_Recv	1.57 sec (0.786 %)

Where to start with analysis

For deep analysis of the MPI-bound application click "Continue >" to open the tracefile View and leverage the **Intel® Trace Analyzer** functionality:

- *Performance Assistant* - to identify possible performance problems
- *Imbalance Diagram* - for detailed imbalance overview
- *Tagging/Filtering* - for thorough customizable analysis

To optimize node-level performance use:
Intel® VTune™ Amplifier XE for:
- algorithmic level tuning with hotspots and threading efficiency analysis;
- microarchitecture level tuning with general exploration and bandwidth analysis;

Intel® Advisor for:
- vectorization optimization and thread prototyping.

Use the following command lines to run these tools for the most CPU-bound rank.

Intel® VTune™ Amplifier XE:

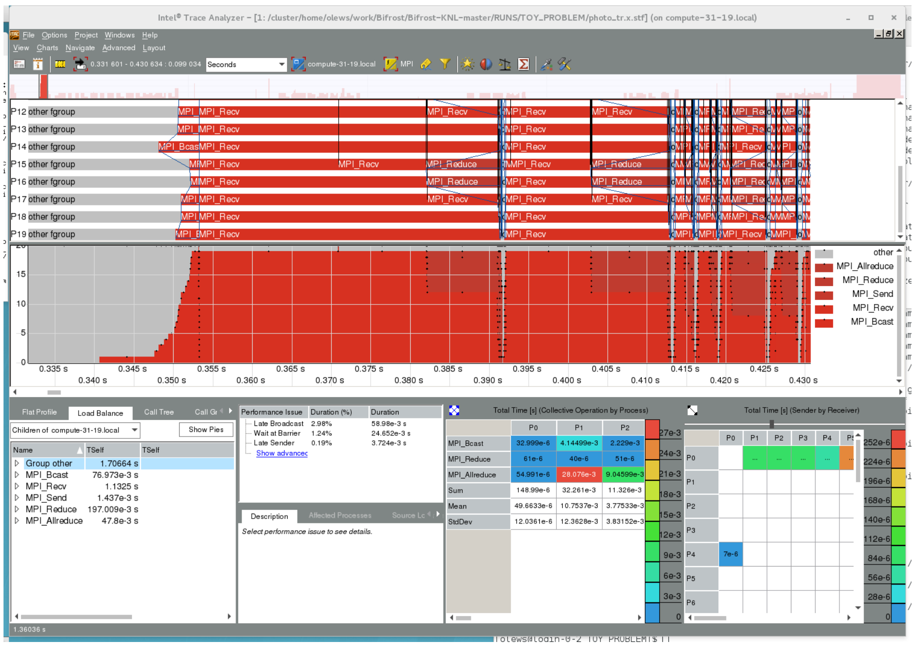
```
mpirun -gtool "amplxe-cl -collect hotspots -r result:0"
-profile=vt -np 20 ../photo_tr.x
```

Intel® Advisor:

```
mpirun -gtool "advixe-cl -collect survey:0" -profile=vt
-np 20 ../photo_tr.x
```

☑ *Show Summary Page when opening a tracefile*

Figure 46. Trace Analyser GUI complex example



Name	TSelf	TSelf
Group other	1.70664 s	
MPI_Bcast	76.973e-3 s	
MPI_Recv	1.1325 s	
MPI_Send	1.437e-3 s	
MPI_Reduce	197.009e-3 s	
MPI_Allreduce	47.8e-3 s	

Performance Issue	Duration (%)	Duration
Late Broadcast	2.98%	58.98e-3 s
Wait at Barrier	1.24%	24.652e-3 s
Late Sender	0.19%	3.724e-3 s
Show advancec		

	P0	P1	P2
MPI_Bcast	32.999e-6	4.14499e-3	2.229e-3
MPI_Reduce	61e-6	40e-6	51e-6
MPI_Allreduce	54.991e-6	28.076e-3	9.04599e-3
Sum	148.99e-6	32.261e-3	11.326e-3
Mean	49.6633e-6	10.7537e-3	3.77533e-3
StdDev	12.0361e-6	12.3628e-3	3.83152e-3

	P0	P1	P2	P3	P4	P5
P0		...	...	...	...	...
P1						
P2						
P3						
P4	7e-6					
P5						
P6						

Another option is to analyse the traces with scripts using the ascii data provided with the tools.

The following command is useful to display the trace on the terminal or redirecting to a text file.

```
stftool photo_tr.x.stf --print-statistics
```

This command yield an ascii file with a lot of data about the MPI function calls. A tiny part of an example output is given below:

```
# Timing <process ()>:<:runtime (seconds)>
INFO TIMING "1":2.882300e-02
INFO TIMING "2":2.746900e-02

 # ONE2ONE_PROFILE  <sender>:<receiver>:<func_sender>:<func_receiver>:<tag>:
<communicator>:<size>:<count>:<min_time>:<max_time>:<total_time>
INFO ONE2ONE_PROFILE 5:0:212:212:3:1:10920:6866:24576000:2359296000:839892992000
INFO ONE2ONE_PROFILE 5:0:212:212:4:1:10920:6866:24576000:614400000:267157504000

# COLLOP_PROFILE  <type>:<root>:<communicator>:<index of process>:<count>:<min_byte_sen
um_byte_sent>:<min_byte_recv>:<max_byte_recv>:<sum_byte_recv>:<min_rate>:<max_rate>:<av
>:<max_duration>:<sum_duration>
INFO COLLOP_PROFILE 1:0:1:0:690:0:0:0:0:0:0:0.000000:0.000000:0.000000:8192000:13926400
INFO COLLOP_PROFILE 1:0:1:1:690:0:0:0:0:0:0:0.000000:0.000000:0.000000:40960000:33997619

# FUNCTION_PROFILE <willy>:<func>:<count>:<min_self>:<max_self>:<total_self>:<min_inclu
:<total_inclusive>
INFO FUNCTION_PROFILE 6:17:6:0:16384000:24576000:0:16384000:24576000
INFO FUNCTION_PROFILE 8:22:3:0:24576000:24576000:0:24576000:24576000
```

This output is complex and need scripts to process. More information is found at Intel web pages using the following link: STF Manipulation with stftool [https://software.intel.com/en-us/node/561433]

7.6.5. Auto Tuning tool

Intel provide an auto tuning tool. This tool runs an application t through a range of different MPI library settings in order to find an optimal setting of MPI library environment variables. The tool can perform automatic tuning of a range of parameters, communications parameters and rank placement parameters. The documentation is found at Intel's web site [https://software.intel.com/en-us/node/528811].

7.6.5.1. Application auto tuning

Application tuning is of wide relevance for all application consuming significant amount of cpu time. To initiate the auto tuning use the following command:

```
mpitune --application \"mpirun -np 64 bin/$PROG\" -of ./${PROG}.tuning.out
```

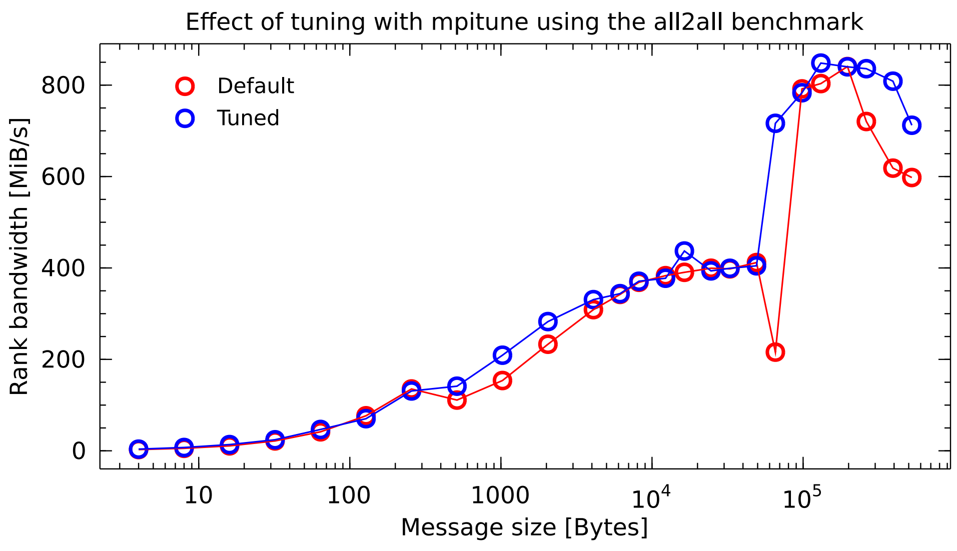
where PROG have been set to any relevant application. The autotuning utility will run through a set of parameters which can be quite time consuming and produce a file with a set of hopefully optimal parameters for a production run. To start a run using the newly found parameters, the syntax is as follows:

```
mpirun -tune ./${PROG}.tuning.out -np 64 bin/$PROG
```

The optimal settings might not be optimal for a different input to the same application so the process might be repeated with each new set of inputs.

For a shared memory system like a single processor KNL system the impract of tuning the MPI parameters are expected to be relatively small as the defaults are well tuned to a shared memory system. The figure below show the effect on autotuning the all2all benchmark.

Figure 47. Effect of MPI autotuning



Tests have confirmed the rather small effect on shared memory systems with a range of benchmarks and applications. For a interconnected cluster the effects can be rather significant.

7.6.6. Tuning / Environment Variables

7.6.6.1. Shared memory MPI communication

Several environment variables control the MPI shared memory communication.

- export I_MPI_EAGER_THRESHOLD=xMB
- export I_MPI_SHM_BYPASS=0 or 1
- export I_MPI_INTRANODE_EAGER_THRESHOLD=xMB

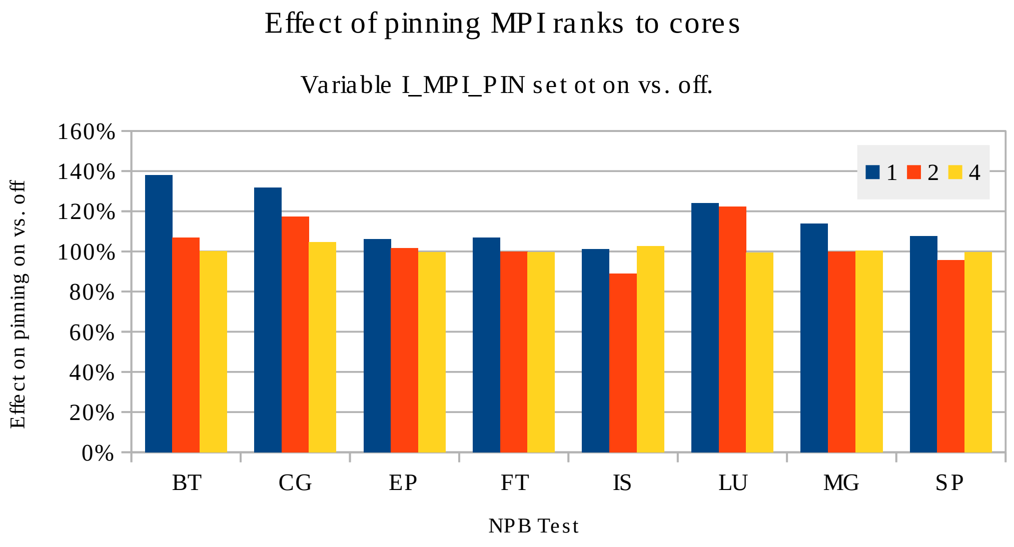
Intel MPI shared memory documentation pages [https://software.intel.com/en-us/node/5288227]

7.6.6.2. Pinning of MPI ranks on cores

First of all it's quite clear that Intel has done a good job with the defaults in the mpi run time environment. It is not a trivial task to get better performance compared to the defaults. All the commonly applied tricks are already built into the mpi run time envirnonment. Keeping the ranks on the allocated cores seems to yield some performance gain in the majority of cases.

The environment variable I_MPI_PIN control pinning to cores and can be set to "on" or "off". The figure below show how the effect on performance with the different NPB tests when selctiong pinning off and on when scheduling 1, 2 or 4 threads per core. The effect of pinning is expected to be very small when all cores are used, this is evident from the figure.

Figure 48. Pinning of ranks



Intel provide documentaion on the web site, it can be found using the link below. Intel MPI pinning placement documentation [https://software.intel.com/en-us/node/561772].

7.6.7. Mapping Tasks on Node Topology

Core placement are as important when running MPI jobs as it is with threaded jobs. In order to control the placements of the individual MPI ranks placement different environment variable can be used.

The environment variable I_MPI_PIN_PROCESSOR_LIST is used to control the MPI processor placement. It can take simple argument and more complicates explicit lists of cores.

Table 25. Processor placement control

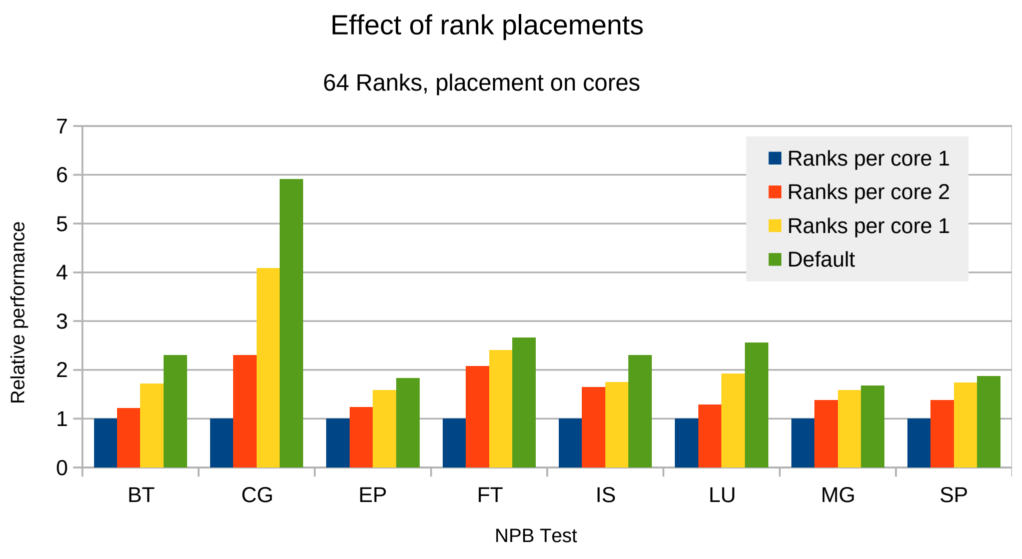
Argument	Information
all	All logical processors. Specify this subset to define the number of CPUs on a node.

Argument	Information
allcores	All cores (physical CPUs). Specify this subset to define the number of cores on a node. This is the default value.
allsocks	All packages/sockets. Specify this subset to define the number of sockets on a node.

There are a lot more options to fine control the rank placements, keeping the ranks close together or spread them in different orders, see this web page [https://software.intel.com/en-us/node/528898] for more detailed information.

The figure below show the effect of setting the rank placements hard to individual cores by setting the envirnment variable I_MPI_PIN_PROCESSOR_LIST to a list of cores (like I_MPI_PIN_PROCESSOR_LIST=0-63). In the case of count offset as one the 64 ranks are set to use the cores numbered from 0 to 63, which is one rank per physical core, as the physical cores are numbered from 0 though 63, use the utility *htop* [http://hisham.hm/htop/] to view this in real time. Comparing this to the layout done by the mpirun scheduling 64 ranks without any other hints that pinning with the environment variable I_MPI_PIN set to "on" make it clear that in many cases the mpirun does a decent job.

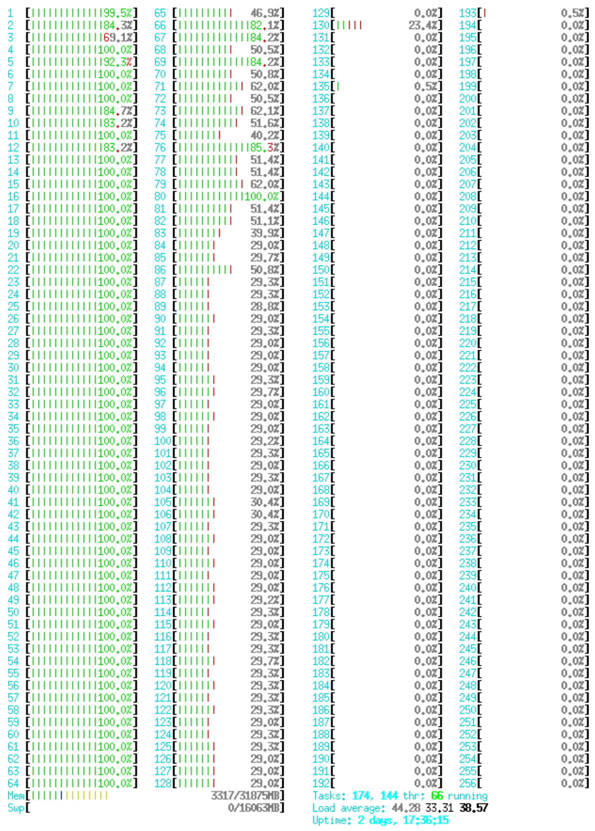
Figure 49. Effect of rank placements



It seems that pinning ranks to individual cores even when scheduling a single rank on a single physical core yield less performance than letting the OS and MPI run time system do the scheduling.

The core and threading layout is nicely illustrated by *htop* utility where each horisontal line is a physical core. The utility *htop* show below number the processor cores from 1 through 256, opposed to what the OS does, which number the cores from 0 through 255.

Figure 50. Screenshot htop



```
1   [|||||||||||||99.5%]  65 [||||||||||   46.9%]  129[              0.0%]  193[|             0.5%]
2   [|||||||||||||84.3%]  66 [|||||||||||||82.1%]  130[|||||        23.4%]  194[              0.0%]
3   [|||||||||||||69.1%]  67 [|||||||||||||84.2%]  131[              0.0%]  195[              0.0%]
4   [||||||||||||100.0%]  68 [||||||||||   50.5%]  132[              0.0%]  196[              0.0%]
5   [|||||||||||||92.3%]  69 [|||||||||||||84.2%]  133[              0.0%]  197[              0.0%]
6   [||||||||||||100.0%]  70 [||||||||||   50.8%]  134[              0.0%]  198[              0.0%]
7   [||||||||||||100.0%]  71 [|||||||||||| 62.0%]  135[|             0.5%]  199[              0.0%]
8   [||||||||||||100.0%]  72 [||||||||||   50.5%]  136[              0.0%]  200[              0.0%]
9   [|||||||||||||84.7%]  73 [|||||||||||| 62.1%]  137[              0.0%]  201[              0.0%]
10  [|||||||||||||83.2%]  74 [|||||||||||  51.6%]  138[              0.0%]  202[              0.0%]
11  [||||||||||||100.0%]  75 [||||||||     40.2%]  139[              0.0%]  203[              0.0%]
12  [|||||||||||||83.2%]  76 [|||||||||||||85.3%]  140[              0.0%]  204[              0.0%]
13  [||||||||||||100.0%]  77 [|||||||||||  51.4%]  141[              0.0%]  205[              0.0%]
14  [||||||||||||100.0%]  78 [|||||||||||  51.4%]  142[              0.0%]  206[              0.0%]
15  [||||||||||||100.0%]  79 [|||||||||||| 62.0%]  143[              0.0%]  207[              0.0%]
16  [||||||||||||100.0%]  80 [||||||||||||100.0%]  144[              0.0%]  208[              0.0%]
17  [||||||||||||100.0%]  81 [||||||||||   51.4%]  145[              0.0%]  209[              0.0%]
18  [||||||||||||100.0%]  82 [||||||||||   51.1%]  146[              0.0%]  210[              0.0%]
19  [||||||||||||100.0%]  83 [||||||||     39.9%]  147[              0.0%]  211[              0.0%]
20  [||||||||||||100.0%]  84 [|||||||      29.0%]  148[              0.0%]  212[              0.0%]
21  [||||||||||||100.0%]  85 [|||||||      29.7%]  149[              0.0%]  213[              0.0%]
22  [||||||||||||100.0%]  86 [||||||||||   50.8%]  150[              0.0%]  214[              0.0%]
23  [||||||||||||100.0%]  87 [||||||       29.3%]  151[              0.0%]  215[              0.0%]
24  [||||||||||||100.0%]  88 [||||||       29.3%]  152[              0.0%]  216[              0.0%]
25  [||||||||||||100.0%]  89 [||||||       28.8%]  153[              0.0%]  217[              0.0%]
26  [||||||||||||100.0%]  90 [|||||||      29.0%]  154[              0.0%]  218[              0.0%]
27  [||||||||||||100.0%]  91 [||||||       29.3%]  155[              0.0%]  219[              0.0%]
28  [||||||||||||100.0%]  92 [||||||       29.0%]  156[              0.0%]  220[              0.0%]
29  [||||||||||||100.0%]  93 [||||||       29.0%]  157[              0.0%]  221[              0.0%]
30  [||||||||||||100.0%]  94 [||||||       29.0%]  158[              0.0%]  222[              0.0%]
31  [||||||||||||100.0%]  95 [|||||||      29.3%]  159[              0.0%]  223[              0.0%]
32  [||||||||||||100.0%]  96 [|||||||      29.7%]  160[              0.0%]  224[              0.0%]
33  [||||||||||||100.0%]  97 [||||||       29.0%]  161[              0.0%]  225[              0.0%]
34  [||||||||||||100.0%]  98 [|||||||      29.0%]  162[              0.0%]  226[              0.0%]
35  [||||||||||||100.0%]  99 [||||||       29.0%]  163[              0.0%]  227[              0.0%]
36  [||||||||||||100.0%]  100[||||||       29.2%]  164[              0.0%]  228[              0.0%]
37  [||||||||||||100.0%]  101[||||||       29.3%]  165[              0.0%]  229[              0.0%]
38  [||||||||||||100.0%]  102[||||||       29.0%]  166[              0.0%]  230[              0.0%]
39  [||||||||||||100.0%]  103[||||||       29.3%]  167[              0.0%]  231[              0.0%]
40  [||||||||||||100.0%]  104[||||||       29.0%]  168[              0.0%]  232[              0.0%]
41  [||||||||||||100.0%]  105[|||||||      30.4%]  169[              0.0%]  233[              0.0%]
42  [||||||||||||100.0%]  106[|||||||      30.4%]  170[              0.0%]  234[              0.0%]
43  [||||||||||||100.0%]  107[||||||       29.3%]  171[              0.0%]  235[              0.0%]
44  [||||||||||||100.0%]  108[|||||||      29.0%]  172[              0.0%]  236[              0.0%]
45  [||||||||||||100.0%]  109[||||||       29.0%]  173[              0.0%]  237[              0.0%]
46  [||||||||||||100.0%]  110[|||||||      29.0%]  174[              0.0%]  238[              0.0%]
47  [||||||||||||100.0%]  111[||||||       29.0%]  175[              0.0%]  239[              0.0%]
48  [||||||||||||100.0%]  112[|||||||      29.0%]  176[              0.0%]  240[              0.0%]
49  [||||||||||||100.0%]  113[|||||||      29.2%]  177[              0.0%]  241[              0.0%]
50  [||||||||||||100.0%]  114[||||||       29.3%]  178[              0.0%]  242[              0.0%]
51  [||||||||||||100.0%]  115[|||||||      29.0%]  179[              0.0%]  243[              0.0%]
52  [||||||||||||100.0%]  116[||||||       29.3%]  180[              0.0%]  244[              0.0%]
53  [||||||||||||100.0%]  117[||||||       29.3%]  181[              0.0%]  245[              0.0%]
54  [||||||||||||100.0%]  118[|||||||      29.7%]  182[              0.0%]  246[              0.0%]
55  [||||||||||||100.0%]  119[||||||       29.3%]  183[              0.0%]  247[              0.0%]
56  [||||||||||||100.0%]  120[|||||||      29.3%]  184[              0.0%]  248[              0.0%]
57  [||||||||||||100.0%]  121[||||||       29.3%]  185[              0.0%]  249[              0.0%]
58  [||||||||||||100.0%]  122[|||||||      29.3%]  186[              0.0%]  250[              0.0%]
59  [||||||||||||100.0%]  123[||||||       29.0%]  187[              0.0%]  251[              0.0%]
60  [||||||||||||100.0%]  124[||||||       29.3%]  188[              0.0%]  252[              0.0%]
61  [||||||||||||100.0%]  125[|||||||      29.3%]  189[              0.0%]  253[              0.0%]
62  [||||||||||||100.0%]  126[|||||||      29.0%]  190[              0.0%]  254[              0.0%]
63  [||||||||||||100.0%]  127[|||||||      29.0%]  191[              0.0%]  255[              0.0%]
64  [||||||||||||100.0%]  128[|||||||      29.0%]  192[              0.0%]  256[              0.0%]
Mem[||||||||||||||                  3317/31875MB]  Tasks: 174, 144 thr: 66 running
Swp[                                   0/16063MB]  Load average: 44.28 33.31 38.57
                                                   Uptime: 2 days, 17:36:15
```

It's evident that providing more execution units yield higher performance. This is probably no supprise for MPI applications where the executable is essential independent of any other executable with no (little) benefit from shared caches.

Intel MPI core placement documentation [https://software.intel.com/en-us/node/528818].

7.7. Hybrid Programming

7.7.1. Optimal Tasks / Threads Strategy

In the case of the environment variable I_MPI_PIN_PROCESSOR_LIST is set to a list of processors, like I_MPI_PIN_PROCESSOR_LIST=0-63, only these 64 cores will be allocates by mpirun and the OS. Scheduling more than one thread per MPI rank will result that two threads will share the same logical core. Allocating more cores for MPI use does not yield free cores for new threads. Only the cores where there is ranks running will be used. There is no benefit from the 3 extra logical cores on the physical core (the usage of logical vs. physical might not be exatly true as the threads referred to as logical core are hardware threads). Hence be careful when using I_MPI_PIN_PROCESSOR_LIST in a Hybrid code environmet.

7.7.2. Thread placement

Pinning and limiting MPI ranks to cores as explained above is not a good option for the threads created within the rank processs (remember that the different MPI ranks are totally independent executables). The created threads need extra cores (physical or logical) to run on. One simple way of setting the environment is to pin the MPI ranks using I_MPI_PIN=on and place the threads close together using KMP_AFFINITY=granularity=core,compact (will help threads share the caches better as they are close) or spread them using KMP_AFFINITY=granularity=core,scatter (will provide higher memory bandwidth as they use more memory controllers). Whichever work best is not easy to tell, it should be tested as it's higly application dependent. The utility *htop* is as always helpful to see where ranks and threads are running, see Figure 50. The *htop* show physical cores line by line, and the hardware threads column by column making it easy to follow the placement in real time.

The thread control environment variables discussed in Section 7.5.3.3, “Excecution control” is also important.

7.7.3. Interoperability with MPI/OpenMP

There are envirment variables taylored for hybrid codes, the variable I_MPI_PIN_DOMAIN provide a domain containing the rank and threads specified by OMP_NUM_THREADS. The variable I_MPI_PIN_DOMAIN can be set to omp and in addition also specify the thread affinity using the following syntax:

- I_MPI_PIN_DOMAIN=omp:compact
- I_MPI_PIN_DOMAIN=omp:scatter

The meaning of compact and scatter are just as for KMP_AFFINITY.

One might also provide more specific binding such that the MPI rank has a domain containing a number of threads, e.g. new threads that are created by the rank will only be started on cores within the domain. To set such a domain the environment varaible KMP_HW_SUBSET. Setting it to the OMP_NUM_THREADS will allocate one core for each thread, syntax looks like:

```
KMP_HW_SUBSET=${OMP_NUM_THREADS}T
```

Intel also provide documentation for hybrid runs. [https://software.intel.com/en-us/node/522775] and interopability documentation [https://software.intel.com/en-us/node/528819].

7.7.4. Performance issues, suggested settings

Tests using the LS-Dalton application may serve as an example of what effect the different environment variables might have. The following settings produced the optimal peformance when running LD-Dalton in hybrid mode with 32 MPI ranks and 8 threads per rank, optimal settings for other applications might be different. Use these settings as a starting point.

- OMP_NUM_THREADS=8
- I_MPI_PIN=on
- I_MPI_PIN_DOMAIN=omp:scatter
- KMP_BLOCKTIME=20
- I_MPI_SHM_BYPASS=1

The KMP_BLOCKTIME did have a significant effect when running many threads per physical core, this need to be tuned to each hybrid application (see Table 24). The KMP_AFFINITY variable was not set, using I_MPI_PIN_DOMAIN=omp:scatter seem to have the same effect. Also the variable KMP_HW_SUBSET was not set, limiting the threads to the MPI domain did not have a significant effect.

However, limiting the available cores by setting domains etc might be conterproductive when less than all cores are used. In these cases I_MPI_PIN and KMP_AFFINITY are good combinations.

7.8. Memory Optimisation

7.8.1. Memory Affinity (MPI/OpenMP/Hybrid)

7.8.1.1. NUMA control

Numactl is a versatile tool for controlling memory allocation. When operating in flat or hybrid mode the high bandwidth memory (HBM) is visible as distinct NUMA memory banks and numactl can be used to control how memory allocations are done. A simple example look like:

```
numactl --preferred=1 ./prog.x
```

where the HBM is in NUMA bank number 1. This works even in the cases where the allocation exceeds the HBM, any allocation exceeding the HMB is done on the main memory.

7.8.2. Memory Allocation (malloc) Tuning

There are different ways of allocating memory, some allocate in different NUMA nodes while others use newer thread safe functions.

7.8.2.1. Fastmem allocation

Allocating memory in the high bandwidth memory make sense when the system is set up with a flat or hybrid memory profile. In the case when using the high bandwidth memory as a last level cache there are only one NUMA memory bank and no special memory is available.

Memory allocation can be controlled by means of directives to the compiler. The directive is used like this:

```
!dir$ attributes fastmem :: object
```

Alignment is also important and this can be added to the fast mem attribute:

```
!dir$ attributes fastmem, align:64 :: object
```

An example using the Stream benchmark might look like :

```
real(r8), allocatable :: a(:),b(:),c(:)
!dir$ attributes fastmem, align:64 :: a,b,c
allocate(a(n))
allocate(b(n))
allocate(c(n))
```

The object file need to be linked with a special library, libmemkind.

```
ifort -qopenmp -o stream.x stream.f -lmemkind
```

More information is found at Intel's web pages: Attributes Fastmem [https://software.intel.com/en-us/node/580357]. C/C++ are also supported, but this require changes to the source code, not only directives. Calls to malloc need to be renamed making it a bit less portable.

The performance is dramatically increased when the vectors are allocated in the high bandwidth memory. The table below show a Stream benchmark test using a total memory footprint at 4.5 GiB, well within the 16 GiB HMB.

Table 26. Stream results using Memkind library

Test	Main memory	High Bandwidth memory
Copy:	45.4 GiB/s	282.6 GiB/s
Scale:	45.8 GiB/s	282.7 GiB/s
Add:	52.4 GiB/s	297.7 GiB/s
Triad:	52.3 GiB/s	297.6 GiB/s

The alternative is to use *numactl --preferred=n*, but this provide far less control over which part of the allocation that is allocated in the high bandwidth memory, and for MPI programs *numactl* is not always simple to use. As always there is a choice betwen ease of use and degree of control.

7.8.2.2. Threaded Building Blocks memory allocating functions

TBB contains a thread scalable allocator which avoid the normal burden of locking a memory pool which all threads have to access to allocate memory. All threads maintain a thread local pool of memory which is then used to serve allocations in a thread local fashion. The normal malloc etc are not thread safe and hence place a lock around the allocation.

In some cases these thread safe memory allocating functions can provide better performance. A nice example is shown here [https://software.intel.com/videos/1-minute-intro-intel-tbb-malloc-proxy-library] and more information in this tutorial [https://www.threadingbuildingblocks.org/tutorial-intel-tbb-scalable-memory-allocator]

Usage is relatively simple: source the Intel compiler set first

```
LD_PRELOAD=$TBBROOT/lib/intel64/gcc*/libtbbmalloc_proxy.so.2:
$TBBROOT/lib/intel64/gcc*/libtbbmalloc.so.2
```

7.8.2.3. MKL with HBM

MKL can allocate from HBM. In order to control the amount of memory allocated by MKL the environment variable MKL_FAST_MEMORY_LIMIT can be used. Setting the

```
MKL_FAST_MEMORY_LIMIT=100
```

will limit the allocation to a maximum of 100 MiB. For more use of this functionallity see this web page [https://software.intel.com/en-us/node/589700].

7.8.3. Using Huge Pages

The address space in a moden virtual memory computer systems is partitioned into pages, usually a few kilobytes in size. Every time the CPU accesses virtual memory, a virtual address must be translated to the corresponding physical address. All of this hidden from the programmer by the operating system and memory management units.

7.8.3.1. Automatic Huge pages

Newer Red Hat (and CENTOS) distributions (staring with RHEL 6) introduce another mechanism for huge pages, transparent huge pages (THP). THP hides much of the complexity in using huge pages from system administrators and developers. THP can currently only map anonymous memory regions such as heap and stack space. See the web page Redhat huge page documentation [https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Performance_Tuning_Guide/s-memory-transhuge.html] for more information.

When allocating memory in a program the unit of allocation is a page. Small pages waste little memory while larger pages increases this. A large number of pages require a larger Translation Lookaside Buffer (TLB) of page addresses while pages of large size (megabytes) will limit the size and entries in the TLB. Finding an address is faster as fewer entries are needed in the TLB as the page-table walk is shorter.

The usage of huge pages can improve performance in some cases. the command

```
cat /proc/meminfo
```

will provide information abount the pages and page sizes (keywords to look for *HugePages* and *Hugepagesize*.

```
>cat /proc/meminfo|grep -i hugepages
AnonHugePages:   2201600 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
```

The AnonHugePages (anonymous huge pages) are the huge pages used by the THP, the huge page size is 2 MiB (kernel configuration). The huge page size is important when setting the number of huge pages as the total size of huge pages should be less than installed physical memory.

To set a given number of huge pages the following command can be used.

```
echo 10000 > /proc/sys/vm/nr_hugepages/
```

This will create 10000 huge pages with a total of 20 GiB memory available with huge pages.

To use huge pages in normal application a special library (libhugetlbfs and libhugetlbfs-devel if you want the documentation) need to be installed and some environment variables need to be set. An example is given here:

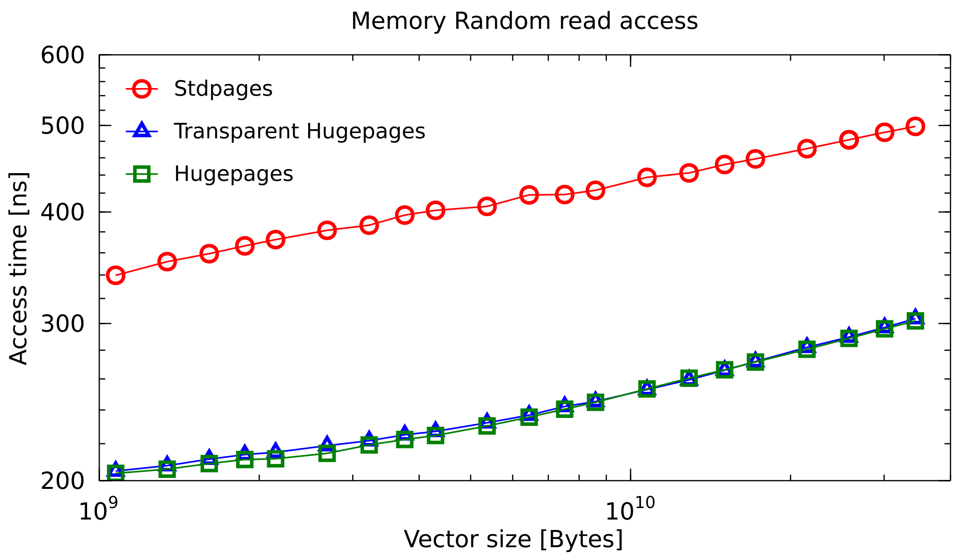
```
LD_PRELOAD=libhugetlbfs.so HUGETLB_MORECORE=yes HUGETLB_SHM=yes ./stream.x
```

More information can be found in the documentation found under /usr/share/doc/libhugetlbfs-2.16/ on systems that have the libhugetlbfs development package installed.

To examine if the application in question can benefit from huge pages examine the size of the *PageTables* in the */proc/meminfo* file. If this is large (several hundred MiB or more), much larger then the size of the current TBL it should show benefit from huge pages. Another parameter to look for is the minor pages faults reported by the time utility. For KNL the cache size is 64 entries in L1 and 256 (4k pages, only 128 with 2M pages) in the L2 cache. 256 entries times 4k is only 1M of memory, while with 2M pages it's 256 MiB.

The performance impact can be significant for some applications that do a lot of random memory access. The figure below show the memory random access time for three different scenarios without any huge pages, transparent huge pages and explicit huge pages. A small benchmark program that allocates and randomly access entries in a vector is used.

Figure 51. Memory access time using Hugepages



The ubiquitous tool "time" report minor page faults which is another term for TLB misses. It is used like this :

```
/usr/bin/time prog.x
```

Full path is needed as "time" is a shell keyword. The output contain a lot of useful information.

```
2175.42user 46.85system 37:03.76elapsed 99%CPU
(0avgtext+0avgdata 33555264maxresident)k
```

```
0inputs+0outputs (0major+139901minor)pagefaults 0swaps
```

Table 27. Minor page faults (TLB Misses)

Page setting	Minor page faults/TLB Misses (reported by /usr/bin/time)
Std 4k pages	54526142
Transparent Huge pages	139901
Explicit Huge pages	16648

As can be seen from the illustrations above the virtual memory system has performance implications. The translation from a virtual address space to the physical address space is an important aspect of a modern compute system. For applications that are sensitive to access time of random access to memory locations. For raw memory bandwidth like the "Stream" and matrix multiplications benchmark there is only minor differences. Data locallity is as always very important.

The advent of automatic usage of huge pages, transparent huge pages (THP) in newer distributions is good news. It makes usage of huge pages automatic without any manual setup or tuning. While not showing the same low number in TLB misses it does however show no significant difference is memory random access time.

7.8.3.2. Huge pages with TBB malloc functions

In addition the huge pages can be used with TBB:

```
export TBB_MALLOC_USE_HUGE_PAGES=1
```

also, the system need to be set up to allow for huge pages:

```
echo 2000 >  /proc/sys/vm/nr_hugepages
```

this will set up 2000 huge pages, equvialent to 20 GiB of memory.

7.9. Possible BIOS Parameter Tuning

There are a large range of different BIOS settings that impacts performance. The obvious ones like chip cluster layout and high bandwidth memory are already covered in Section 2.2. There are however a large range of other settings that might have an impact on performance.

Updating the BIOS settings can be done using a command line utility. Se Section 2.3.6 for details.

7.9.1. Snoop filter settings

The cache coherence protocol rely on a system to keep track of all memory references of cachelines used, the Tag Directory (TD) store the reference to the cache lines. From a processor view the smallest item in memory is a cache line. A core broadcast its memory requests and all other cores are snooping to check if the cache line is stored locally. Smart snoop filters can reduce the amount of system wide broadcasts. The snoop filter can have some impact on performance. There are several BIOS parameters affecting the snoop filter of the processor. The command :

```
syscfg /d biossettings "Snoop Latency Valid"
```

will list the snoop "Snoop Latency Valid" setting while

```
syscfg /bcs "" "Snoop Latency Valid" 01
```

will set it to enable.

8. Debugging

8.1. Available Debuggers

There are two GNU based debuggers installed, the ubiquitous gdb and the Intel version of gdb called gdb-ia. The way debuggers work in KNL are similar to how they work in a Xeon host, as there is no need to connect to an attached device, like in Knights Corner.

In addition the TotalView debugger from Rogue Wave is installed. The 2016.07.22 version is supporting KNL.

8.1.1. GNU based debuggers

- GNU, gdb

  There are two ways of debugging a program with GDB: running the program through GDB or attaching to a running program (process).

  In the example below, we show the output when attaching to a running process in KNL:

```
knl01:~> gdb -q -p 27245
Attaching to process 27245
Reading symbols from /bin/bash...(no debugging symbols found)...done.
Reading symbols from /lib64/libreadline.so.6...(no debugging symbols found)...done.
Reading symbols from /lib64/libtinfo.so.5...(no debugging symbols found)...done.
Reading symbols from /lib64/libdl.so.2...(no debugging symbols found)...done.
Reading symbols from /lib64/libc.so.6...(no debugging symbols found)...done.
Reading symbols from /lib64/ld-linux-x86-64.so.2...(no debugging symbols found)...done
0x00007f5a66a3195c in waitpid () from /lib64/libc.so.6
(gdb) bt
#0  0x00007f5a66a3195c in waitpid () from /lib64/libc.so.6
#1  0x0000000000429781 in ?? ()
#2  0x000000000042aa92 in wait_for ()
#3  0x000000000046222f in execute_command_internal ()
#4  0x0000000000462951 in execute_command ()
#5  0x000000000041b7f1 in reader_loop ()
#6  0x000000000041b4db in main ()
(gdb)
```

- Intel, gdb-ia

  It is an Intel-tuned version of GDB for IA-32/Intel® 64 Architecture systems. In order to set the environment to execute gdb-ia in a KNL node, you should follow these steps:

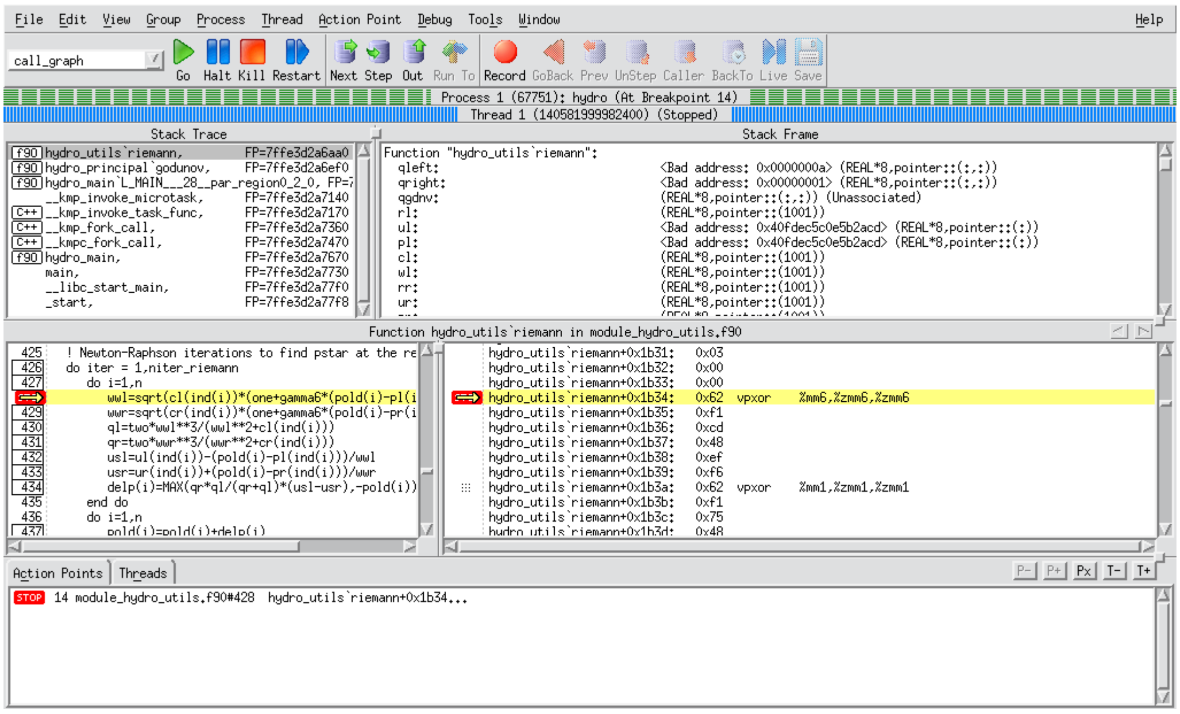
```
knl01:~> source /apps/INTEL/2017.0-035/bin/compilervars.sh intel64
knl01:~> gdb-ia
GNU gdb (GDB) 7.10-18.0.336
Copyright (C) 2015 Free Software Foundation, Inc; (C) 2016 Intel Corp.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-unknown-linux-gnu".
Type "show configuration" for configuration details.
For information about how to find Technical Support, Product Updates,
User Forums, FAQs, tips and tricks, and other support information, please visit:
<http://www.intel.com/software/products/support/>.For help, type "help".
Type "apropos word" to search for commands related to "word".
```

```
(gdb) show configuration
This GDB was configured as follows:
   configure --host=x86_64-unknown-linux-gnu --target=x86_64-unknown-linux-gnu
             --with-auto-load-dir=$debugdir:$datadir/auto-load
             --with-auto-load-safe-path=$debugdir:$datadir/auto-load
             --with-expat
             --with-gdb-datadir=/debugger_2017/gdb/intel64/share/gdb (relocatable)
             --with-jit-reader-dir=/debugger_2017/gdb/intel64/lib/gdb (relocatable)
             --without-libunwind-ia64
             --without-lzma
             --with-python=/nfs/iul/disks/gdb/local/gdb_deps/x86_64-Linux.pie/py27-sta
             --without-guile
             --with-separate-debug-dir=/debugger_2017/gdb/intel64/lib/debug (relocatab
             --without-babeltrace

("Relocatable" means the directory can be moved with the GDB installation
tree, and GDB will still find it.)
(gdb)
```

8.1.2. TotalView debugger

The TotalView debugger is currently (version 2016.07.22) supporting Knights Landing. Using it on a stand alone KNL system is just as any other x86-64 system. No special settings are needed. Documentation about installation and running are just as any other Intel based system.

Figure 52. TotalView example


File Edit View Group Process Thread Action Point Debug Tools Window Help
call_graph
Go Halt Kill Restart Next Step Out Run To Record GoBack Prev UnStep Caller BackTo Live Save
Stack Trace
Stack Frame
Action Points Threads
P- P+ Px T- T+
STOP

```
Process 1 (67751): hydro (At Breakpoint 14)
Thread 1 (140581999982400) (Stopped)
```

```
f90 hydro_utils`riemann,                 FP=7ffe3d2a6aa0
f90 hydro_principal`godunov,             FP=7ffe3d2a6ef0
f90 hydro_main`L_MAIN___28__par_region0_2_0, FP=7
    __kmp_invoke_microtask,              FP=7ffe3d2a7140
C++ __kmp_invoke_task_func,              FP=7ffe3d2a7170
C++ __kmp_fork_call,                     FP=7ffe3d2a7360
C++ __kmpc_fork_call,                    FP=7ffe3d2a7470
f90 hydro_main,                          FP=7ffe3d2a7670
    main,                                FP=7ffe3d2a7730
    __libc_start_main,                   FP=7ffe3d2a77f0
    _start,                              FP=7ffe3d2a77f8
```

```
Function "hydro_utils`riemann":
  qleft:               <Bad address: 0x0000000a> (REAL*8,pointer::(:,:))
  qright:              <Bad address: 0x00000001> (REAL*8,pointer::(:,:))
  qgdnv:               (REAL*8,pointer::(:,:)) (Unassociated)
  rl:                  (REAL*8,pointer::(1001))
  ul:                  <Bad address: 0x40fdec5c0e5b2acd> (REAL*8,pointer::(:))
  pl:                  <Bad address: 0x40fdec5c0e5b2acd> (REAL*8,pointer::(:))
  cl:                  (REAL*8,pointer::(1001))
  wl:                  (REAL*8,pointer::(1001))
  rr:                  (REAL*8,pointer::(1001))
  ur:                  (REAL*8,pointer::(1001))
```

Function hydro_utils`riemann in module_hydro_utils.f90

```
425   ! Newton-Raphson iterations to find pstar at the re
426   do iter = 1,niter_riemann
427      do i=1,n
            wwl=sqrt(cl(ind(i))*(one+gamma6*(pold(i)-pl(i
429         wwr=sqrt(cr(ind(i))*(one+gamma6*(pold(i)-pr(i
430         ql=two*wwl**3/(wwl**2+cl(ind(i)))
431         qr=two*wwr**3/(wwr**2+cr(ind(i)))
432         usl=ul(ind(i))-(pold(i)-pl(ind(i)))/wwl
433         usr=ur(ind(i))+(pold(i)-pr(ind(i)))/wwr
434         delp(i)=MAX(qr*ql/(qr+ql)*(usl-usr),-pold(i))
435      end do
436      do i=1,n
437         pold(i)=pold(i)+delp(i)
```

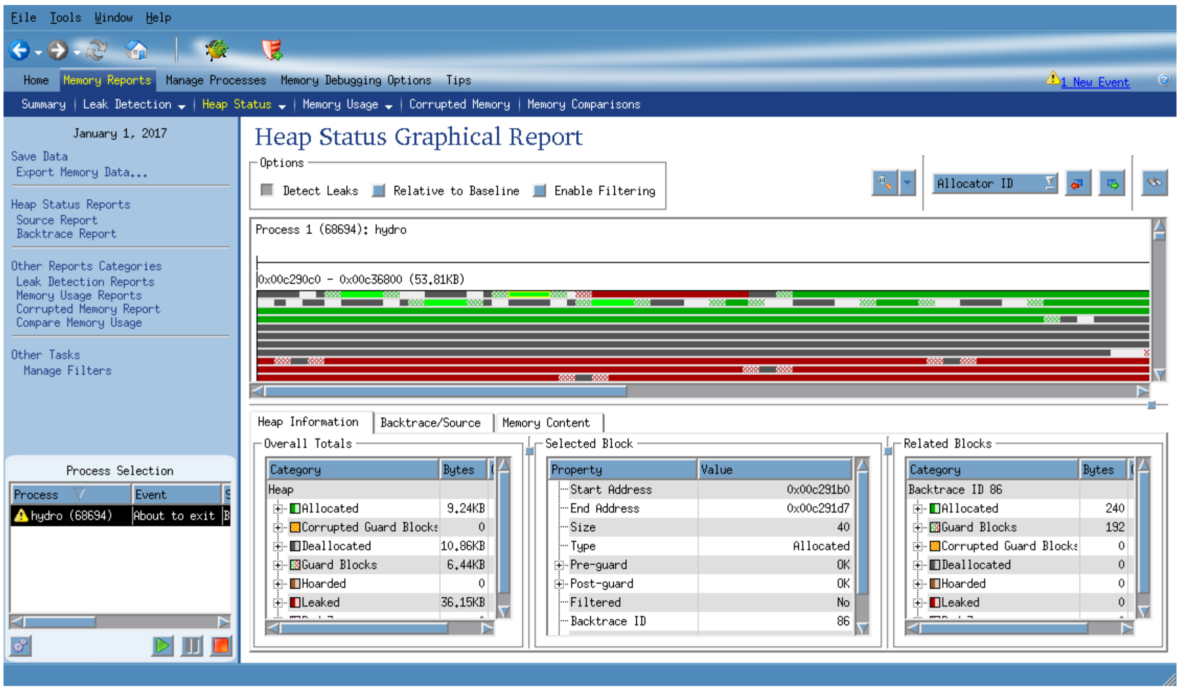
```
hydro_utils`riemann+0x1b31:   0x03
hydro_utils`riemann+0x1b32:   0x00
hydro_utils`riemann+0x1b33:   0x00
hydro_utils`riemann+0x1b34:   0x62  vpxor    %mm6,%zmm6,%zmm6
hydro_utils`riemann+0x1b35:   0xf1
hydro_utils`riemann+0x1b36:   0xcd
hydro_utils`riemann+0x1b37:   0x48
hydro_utils`riemann+0x1b38:   0xef
hydro_utils`riemann+0x1b39:   0xf6
hydro_utils`riemann+0x1b3a:   0x62  vpxor    %mm1,%zmm1,%zmm1
hydro_utils`riemann+0x1b3b:   0xf1
hydro_utils`riemann+0x1b3c:   0x75
hydro_utils`riemann+0x1b3d:   0x48
```

```
14 module_hydro_utils.f90#428  hydro_utils`riemann+0x1b34...
```

The figure illustrate the support for Knights Landing by showing examples of the 512bits vector instructions using the vector registers unique for KNL (the zmms), *vpxor %zmm6,%zmm6,%zmm6*. While it works in the current versions there are still some issues and communication with support suggest that future versions will have far better support for KNL. Not all AVX512 instructions are not yet supported.

The memory debugger and mapper Memscape is also running on the KNL. It seems to work fine, but in the time of writing there might still be parts that is not 100% ported to KNL.

Figure 53. Memscape example



Overall Totals

Category	Bytes
Heap	
Allocated	9.24KB
Corrupted Guard Blocks	0
Deallocated	10.86KB
Guard Blocks	6.44KB
Hoarded	0
Leaked	36.15KB

Selected Block

Property	Value
Start Address	0x00c291b0
End Address	0x00c291d7
Size	40
Type	Allocated
Pre-guard	OK
Post-guard	OK
Filtered	No
Backtrace ID	86

Related Blocks

Category	Bytes
Backtrace ID 86	
Allocated	240
Guard Blocks	192
Corrupted Guard Blocks	0
Deallocated	0
Hoarded	0
Leaked	0

The figure above illustrate the fact that Memscape runs fine on a stand alone KNL system and can analyze memory on a KNL system.

More information about TotalView is found at the Rogue Wave pages for TotalView [http://www.roguewave.com/products-services/totalview].

8.2. Compiler Flags

In this section, we show the basic flags in order to get debugging information. The table below shows common flags for most compilers.

Table 28. Compiler Flags

-g	Produce debugging information in the operating system's native format (stabs, COFF, XCOFF, or DWARF 2). GDB can work with this debugging information.
-p	Generate extra code to write profile information suitable for the analysis program prof. You must use this option when compiling the source files you want data about, and you must also use it when linking.
-pg	Generate extra code to write profile information suitable for the analysis program gprof. You must use this option when compiling the source files you want data about, and you must also use it when linking.

Further documentation

Books

[1] *Jim Jeffers, James Reinders, Avinash Sodani: Intel Xeon Phi Processor high performance Programming, Knights Landing Edition, Morgan Kaufman Publ. 2016.*

[2] *James Reinders, James Jeffers, Intel Xeon Phi Coprocessor High Performance Programming, Morgan Kaufman Publ Inc, 2013 http://lotsofcores.com [http://lotsofcores.com] .*

[3] *Rezaur Rahman: Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers, Apress 2013 .*

[4] *Parallel Programming and Optimization with Intel Xeon Phi Coprocessors, Colfax 2013 http://www.colfax-intl.com/nd/xeonphi/book.aspx [http://www.colfax-intl.com/nd/xeonphi/book.aspx] .*

[5] *Michael McCool, James Reinders, Arch Robison, Structured Parallel Programming: Patterns for Efficient Computation , Morgan Kaufman Publ Inc, 2013 http://parallelbook.com [http://parallelbook.com] .*

[6] *Barbara Chapman, Gabriele Jost and Ruud van der Pas, Using OpenMP, MIT Press Cambridge, 2007, http://mitpress.mit.edu/books/using-openmp [http://mitpress.mit.edu/books/using-openmp].*

Forums, Download Sites, Webinars

[7] *Intel Many Integrated Core Architecture User Forum, http://software.intel.com/en-us/forums/intel-many-integrated-core [http://software.intel.com/en-us/forums/intel-many-integrated-core].*

[8] *Intel Developer Zone: Intel Math Kernel Library, http://software.intel.com/en-us/forums/intel-math-kernel-library [http://software.intel.com/en-us/forums/intel-math-kernel-library].*

[9] *Intel Math Kernel Library Link Line Advisor, http://software.intel.com/sites/products/mkl/ [http://software.intel.com/sites/products/mkl/].*

[10] *OpenMP forum, http://openmp.org/wp/ [http://openmp.org/wp/].*

[11] *Intel Threading Building Blocks Documentation Site, http://threadingbuildingblocks.org/documentation [http://threadingbuildingblocks.org/documentation].*

Manuals, Papers

[12] *Using the Intel MPI Library on Intel Xeon Phi Coprocessor Systems, http://software.intel.com/en-us/articles/using-the-intel-mpi-library-on-intel-xeon-phi-coprocessor-systems [http://software.intel.com/en-us/articles/using-the-intel-mpi-library-on-intel-xeon-phi-coprocessor-systems].*

[13] *Intel Xeon Phi Coprocessor Instruction Set Reference Manual, http://software.intel.com/sites/default/files/forum/278102/327364001en.pdf [http://software.intel.com/sites/default/files/forum/278102/327364001en.pdf].*

[14] *An Overview of Programming for Intel Xeon processors and Intel Xeon Phi coprocessors, http://software.intel.com/sites/default/files/article/330164/an-overview-of-programming-for-intel-xeon-processors-and-intel-xeon-phi-coprocessors_1.pdf [http://software.intel.com/sites/default/files/article/330164/an-overview-of-programming-for-intel-xeon-processors-and-intel-xeon-phi-coprocessors_1.pdf].*

[15] *Programming and Compiling for Intel Many Integrated Core Architecture, http://software.intel.com/en-us/articles/programming-and-compiling-for-intel-many-integrated-core-architecture [http://software.intel.com/en-us/articles/programming-and-compiling-for-intel-many-integrated-core-architecture].*

[16] *Building a Native Application for Intel Xeon Phi Coprocessors, http://software.intel.com/en-us/articles/building-a-native-application-for-intel-xeon-phi-coprocessors [http://software.intel.com/en-us/articles/building-a-native-application-for-intel-xeon-phi-coprocessors].*

[17] *"Using Intel Trace Analyzer and Collector for Intel Many Integrated Core Architecture" in Intel Cluster Studio 2013 Tutorial, http://software.intel.com/sites/products/documentation/hpc/ics/ics2013/ics_tutorial/index.htm#Linux_itac.htm [http://software.intel.com/sites/products/documentation/hpc/ics/ics2013/ics_tutorial/index.htm#Linux_itac.htm].*

[18] *Advanced Optimizations for Intel MIC Architecture, http://software.intel.com/en-us/articles/advanced-optimizations-for-intel-mic-architecture [http://software.intel.com/en-us/articles/advanced-optimizations-for-intel-mic-architecture].*

[19] *Requirements for Vectorizable Loops, http://software.intel.com/en-us/articles/requirements-for-vectorizable-loops/ [http://software.intel.com/en-us/articles/requirements-for-vectorizable-loops/]*.

[20] *Data Alignment to Assist Vectorization, http://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization*.

[21] *VecAnalysis Python Script for Annotating Intel C++ and Fortran Compilers Vectorization Reports , http://software.intel.com/en-us/articles/vecanalysis-python-script-for-annotating-intelr-compiler-vectorization-report*.

[22] *Vectorization essentials, http://software.intel.com/en-us/articles/vectorization-essential*.

[23] *Open MP Thread Affinity Control, http://software.intel.com/en-us/articles/openmp-thread-affinity-control*.

[24] *Avoiding and Identifying False Sharing Among Threads, http://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads*.