Best Practice Guide
Intel Xeon Phi v2.0

Emanouil Atanassov, IICT-BAS, Bulgaria

Michaela Barth, KTH, Sweden

Mikko Byckling, CSC, Finland

Vali Codreanu, SURFsara, Netherlands

Nevena Ilieva, NCSA, Bulgaria

Tomas Karasek, IT4Innovations, Czech Republic

Jorge Rodriguez, BSC, Spain

Sami Saarinen, CSC, Finland

Ole Widar Saastad, University of Oslo, Norway

Michael Schliephake, KTH, Sweden

Martin Stachon, IT4Innovations, Czech Republic

Janko Strassburg, BSC, Spain

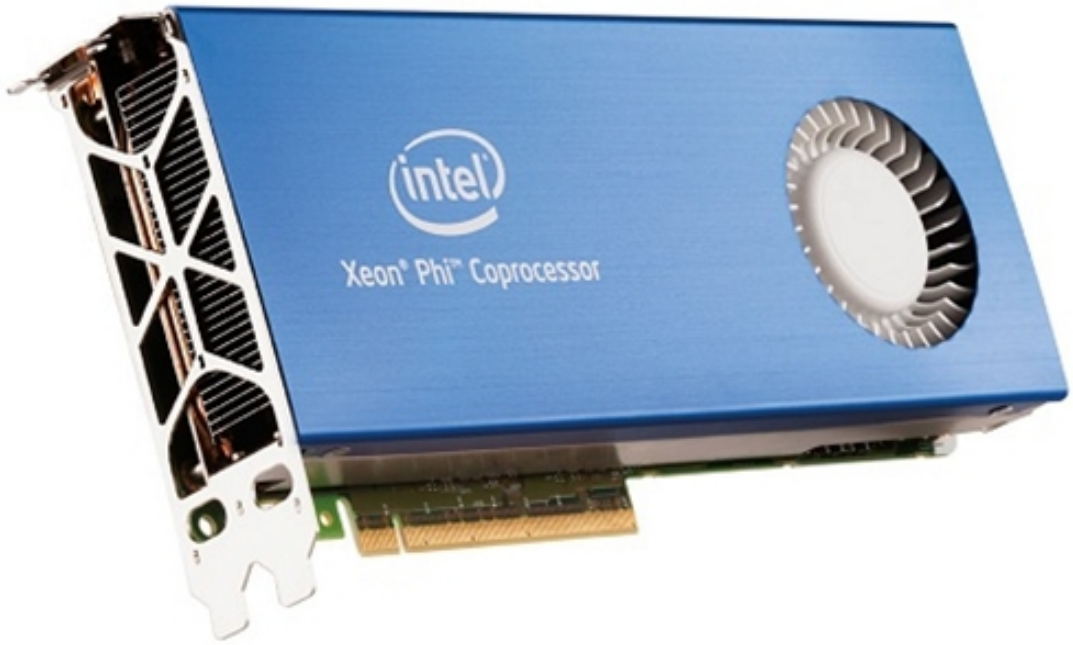
Volker Weinberg (Editor), LRZ, Germany

31-01-2017

Table of Contents

1. Introduction

Figure 1. Intel Xeon Phi coprocessor



This Best Practice Guide provides information about Intel's Many Integrated Core (MIC) architecture and programming models for the first generation Intel® Xeon Phi™ coprocessor named Knights Corner (KNC) in order to enable programmers to achieve good performance out of their applications.

The guide covers a wide range of topics from the description of the hardware of the Intel® Xeon Phi™ coprocessor through information about the basic programming models as well as information about porting programs up to tools and strategies how to analyse and improve the performance of applications. Through the highly parallel architecture and the use of high bandwidth memory, the MIC architecture allows higher performance than traditional CPUs for many types of scientific applications. The guide was created based on the PRACE-3IP Intel® Xeon Phi™ Best Practice Guide. New is the inclusion of information about applications, benchmarks and European Intel® Xeon Phi™ based systems. The following systems are now described:

- Avitohol @ IICT-BAS, Bulgary
- MareNostrum @ BSC, Spain
- Salomon @ IT4Innovations, Czech Republic
- SuperMIC @ LRZ, Germany

Several sections were added and updated. The Best Practice Guide also includes several informations from publicly available Intel documents and Intel webinars [14].

Information about the second generation Intel Xeon Phi coprocessor named Knights Landing (KNL) can be found in a separate guide named "Best Practice Guide - Knights Landing" available under http://www.prace-ri.eu/best-practice-guides/.

In 2013 the first books about programming the Intel Xeon Phi coprocessor [1] [3] [4] have been published. The Intel book about Xeon Phi programming [1] was updated in 2016 to cover the new Knights Landing architecture [2]. We also recommend the 2 volumes "High Performance Parallelism Pearls" by James Reinder and Jim Jeffers including contributions from many authors about porting experience [5] [6]. We further recommend a book about structured parallel programming [7]. Useful online documentation about the Intel Xeon Phi coprocessor can be found in Intel's developer zone for Xeon Phi Programming [9] and the Intel Many Integrated Core Architecture

User Forum [10]. To get things going quickly have a look on the Intel Xeon Phi Coprocessor Developer's Quick Start Guide [18] and also on the paper [27].

Various experiences with application enabling for Intel Xeon Phi gained within PRACE on the EURORA-cluster at CINECA (Italy) in late 2013 can be found in whitepapers available online at [19].

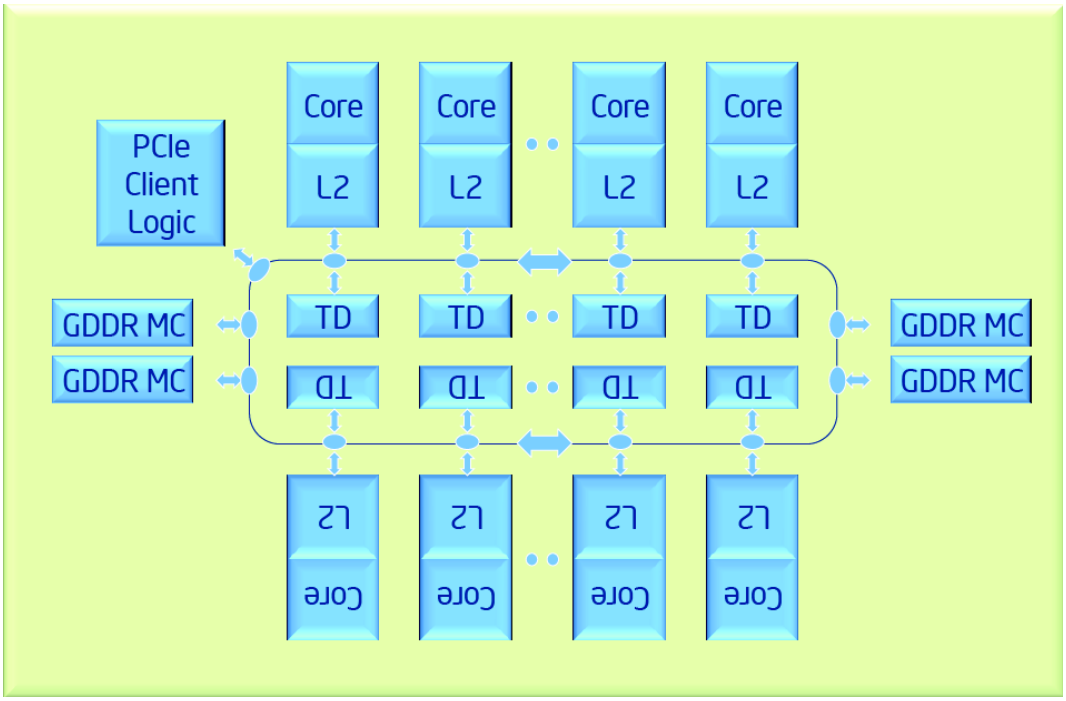
2. Intel MIC architecture & system overview

2.1. The Intel MIC architecture

2.1.1. Intel Xeon Phi coprocessor architecture overview

The Intel Xeon Phi coprocessor consists of up to 61 cores connected by a high performance on-die bidirectional interconnect. The coprocessor runs a Linux operating system and supports all important Intel development tools, like C/C++ and Fortran compiler, MPI and OpenMP, high performance libraries like MKL, debugger and tracing tools like Intel VTune Amplifier XE. Traditional UNIX tools on the coprocessor are supported via BusyBox, which combines tiny versions of many common UNIX utilities into a single small executable. The coprocessor is connected to an Intel Xeon processor - the "host" - via the PCI Express (PICe) bus. The implementation of a virtualized TCP/IP stack allows to access the coprocessor like a network node.

Figure 2. Intel Xeon Phi chip architecture layout. Notice the ring bus as interconnect.



Summarized information about the hardware architecture can be found in [20]. In the following we cite the most important properties of the MIC architecture from the System Software Developers Guide [21], which includes many details about the MIC architecture:

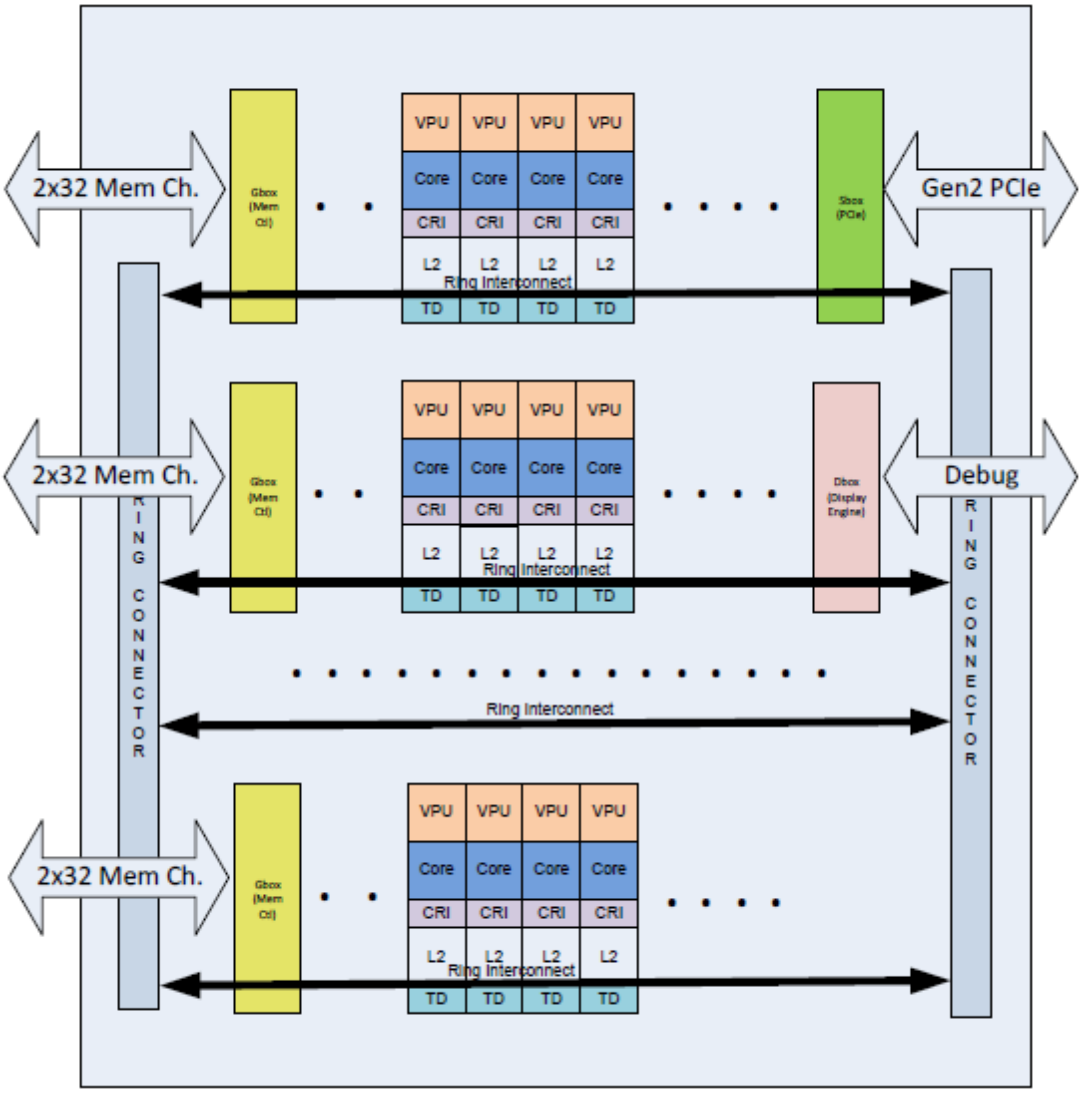
Core

- the processor core (scalar unit) is an in-order architecture (based on the Intel Pentium processor family)
- fetches and decodes instructions from *four hardware threads*
- supports a 64-bit execution environment, along with Intel Initial Many Core Instructions
- does not support any previous Intel SIMD extensions like MME, SSE, SSE2, SSE3, SSE4.1 SSE4.2 or AVX instructions

	• new vector instructions provided by the Intel Xeon Phi coprocessor instruction set utilize a dedicated 512-bit wide vector floating-point unit (VPU) that is provided for each core
	• high performance support for reciprocal, square root, power and exponent operations, scatter/gather and streaming store capabilities to achieve higher effective memory bandwidth
	• can execute 2 instructs per cycle, one on the U-pipe and one on the V-pipe (not all instruction types can be executed by the V-pipe, e.g. vector instructions can only be executed on the U-pipe)
	• contains the L1 Icache and Dcache
	• each core is connected to a ring interconnect via the Core Ring Interface (CRI) as shown in Figure 2.
Vector Processing Unit (VPU)	• the VPU includes the EMU (Extended Math Unit) and executes 16 single-precision floating point, 16 32bit integer operations or 8 double-precision floating point operations per cycle. Each operation can be a fused multiply-add, giving 32 single-precision or 16 double-precision floating-point operations per cycle.
	• contains the vector register file: 32 512-bit wide registers per thread context, each register can hold 16 singles or 8 doubles
	• most vector instructions have a 4-clock latency with a 1 clock throughput
Core Ring Interface (CRI)	• hosts the L2 cache and the tag directory (TD)
	• connects each core to an Intel Xeon Phi coprocessor Ring Stop (RS), which connects to the interprocessor core network
Ring	• includes component interfaces, ring stops, ring turns, addressing and flow control
	• a Xeon Phi coprocessor has 2 of theses rings, one travelling each direction
SBOX	• Gen2 PCI Express client logic
	• system interface to the host CPU or PCI Express switch
	• DMA engine
GBOX	• coprocessor memory controller
	• consists of the FBOX (interface to the ring interconnect), the MBOX (request scheduler) and the PBOX (physical layer that interfaces with the GDDR devices
	• There are 8 memory controllers supporting up to 16 GDDR5 channels. With a transfer speed of up to 5.5 GT/s a theoretical aggregated bandwidth of 352 GB/s is provided.
Performance Monitoring Unit (PMU)	• allows data to be collected from all units in the architecture
	• does not implement some advanced features found in mainline IA cores (e.g. precise event-based sampling, etc.)

The following picture (from [21]) illustrates the building blocks of the architecture.

Figure 3. Intel MIC architecture overview



Multithreading is a key element to hide latency and used extensively in the MIC architecture. There are also additional reasons for using several threads in flight. The vector unit can only issue a vector instruction from one stream every second clock cycle. This constraint requires that at least two threads per core are scheduled at the same time to fill the vector pipeline. The vector unit can issue execute one instruction per clock cycle if fed from two different threads. Issuing more threads helps both to hide latency and to fill the vector pipeline. Four hardware threads available help to accommodate this.

2.1.2. The cache hierarchy

Details about the L1 and L2 cache can be found in the System Software Developers Guide [21]. We only cite the most important features here.

The L1 cache has a 32 KB L1 instruction cache and 32 KB L1 data cache. Associativity is 8-way, with a cache line-size of 64 byte. It has a load-to-use latency of 1 cycle, which means that an integer value loaded from the L1 cache can be used in the next clock by an integer instruction. (Vector instructions have different latencies than integer instructions.)

The L2 cache is a unified cache which is inclusive of the L1 data and instruction caches. Each core contributes 512 KB of L2 to the total global shared L2 cache storage. If no cores share any data or code, then the effective total L2 size of the chip is up to 31 MB. On the other hand, if every core shares exactly the same code and data

in perfect synchronization, then the effective total L2 size of the chip is only 512 KB. The actual size of the workload-perceived L2 storage is a function of the degree of code and data sharing among cores and thread.

Like for the L1 cache, associativity is 8-way, with a cache line-size of 64 byte. The raw latency is 11 clock cycles. It has a streaming hardware prefetcher and supports ECC correction.

The main properties of the L1 and L2 caches are summarized in the following table (from [21]):

Parameter	**L1**	**L2**
Coherence	MESI	MESI
Size	32 KB + 32 KB	512 KB
Associativity	8-way	8-way
Line Size	64 bytes	64 bytes
Banks	8	8
Access Time	1 cycle	11 cycles
Policy	pseudo LRU	pseudo LRU
Duty Cycle	1 per clock	1 per clock
Ports	Read or Write	Read or Write

2.2. Network configuration & system access

Details about the system startup and the network configuration can be found in [22] and in the documentation coming with the Intel Manycore Platform Software Stack (Intel MPSS) [13].

To start the Intel MPSS stack and initialize the Xeon Phi coprocessor the following command has to be executed as root or during host system start-up:

```
weinberg@knf1:~> sudo service mpss start
```

During start-up details are logged to `/var/log/messages`.

If MPSS with OFED support is needed, further the following commands have to be executed as root:

```
weinberg@knf1:~> sudo service openibd start
weinberg@knf1:~> sudo service opensmd start
weinberg@knf1:~> sudo service ofed-mic start
```

In a production environment it is recommended to reboot the MIC nodes in the epilog script of the Batch system. This can be achieved by:

```
# At this point, we reboot both cards to have them cleanly prepared
# for new scheduled jobs.

# reset both cards
micctrl --reset --wait

# boot both cards
micctrl --boot --wait

# restart the ofed-mic service
service ofed-mic start
```

```
# cleanup scratch
rm -rf /scratch
```

The IP addresses of the attached coprocessors can be listed via the traditional `ifconfig` Linux program.

```
lu65fok@i01r13c01:~> ifconfig mic0
mic0      Link encap:Ethernet  HWaddr 4C:79:BA:44:04:E3
          inet6 addr: fe80::4e79:baff:fe44:4e3/64 Scope:Link
          UP BROADCAST RUNNING  MTU:1500  Metric:1
          RX packets:13137 errors:0 dropped:0 overruns:0 frame:0
          TX packets:717591 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1020768 (996.8 Kb)  TX bytes:143148709 (136.5 Mb)

lu65fok@i01r13c01:~> ifconfig mic1
mic1      Link encap:Ethernet  HWaddr 4C:79:BA:42:04:77
          inet6 addr: fe80::4e79:baff:fe42:477/64 Scope:Link
          UP BROADCAST RUNNING  MTU:1500  Metric:1
          RX packets:5785 errors:0 dropped:0 overruns:0 frame:0
          TX packets:704084 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:498952 (487.2 Kb)  TX bytes:124219803 (118.4 Mb)

lu65fok@i01r13c01:~>
```

Further information can be obtained by running the `micinfo` program on the host. To get also PCIe related details the command has to be run with root privileges. Here is an example output for a C0 stepping Intel Xeon Phi prototype:

```
lu65fok@i01r13c0:~> sudo micinfo
MicInfo Utility Log
Created Sat Jan 28 13:53:36 2017


 System Info
  HOST OS   : Linux
  OS Version  : 3.0.76-0.11-default
  Driver Version  : 3.4-1
  MPSS Version  : 3.4
  Host Physical Memory : 66078 MB

Device No: 0, Device Name: mic0

 Version
  Flash Version    : 2.1.02.0390
  SMC Firmware Version  : 1.16.5078
  SMC Boot Loader Version  : 1.8.4326
  uOS Version    : 2.6.38.8+mpss3.4
  Device Serial Number   : ADKC33400625

 Board
  Vendor ID    : 0x8086
  Device ID    : 0x2250
  Subsystem ID    : 0x2500
  Coprocessor Stepping ID  : 3
```

```
  PCIe Width    : x16
  PCIe Speed    : 5 GT/s
  PCIe Max payload size  : 256 bytes
  PCIe Max read req size  : 4096 bytes
  Coprocessor Model  : 0x01
  Coprocessor Model Ext  : 0x00
  Coprocessor Type  : 0x00
  Coprocessor Family  : 0x0b
  Coprocessor Family Ext  : 0x00
  Coprocessor Stepping   : B1
  Board SKU    : B1PRQ-5110P/5120D
  ECC Mode    : Enabled
  SMC HW Revision   : Product 225W Passive CS

 Cores
  Total No of Active Cores : 60
  Voltage    : 1026000 uV
  Frequency   : 1052631 kHz

 Thermal
  Fan Speed Control   : N/A
  Fan RPM    : N/A
  Fan PWM    : N/A
  Die Temp   : 58 C

 GDDR
  GDDR Vendor   : Elpida
  GDDR Version   : 0x1
  GDDR Density   : 2048 Mb
  GDDR Size   : 7936 MB
  GDDR Technology   : GDDR5
  GDDR Speed   : 5.000000 GT/s
  GDDR Frequency   : 2500000 kHz
  GDDR Voltage   : 1501000 uV

Device No: 1, Device Name: mic1

 Version
  Flash Version    : 2.1.02.0390
  SMC Firmware Version  : 1.16.5078
  SMC Boot Loader Version  : 1.8.4326
  uOS Version    : 2.6.38.8+mpss3.4
  Device Serial Number   : ADKC33300571

 Board
  Vendor ID    : 0x8086
  Device ID    : 0x2250
  Subsystem ID    : 0x2500
  Coprocessor Stepping ID  : 3
  PCIe Width    : x16
  PCIe Speed    : 5 GT/s
  PCIe Max payload size  : 256 bytes
  PCIe Max read req size  : 4096 bytes
  Coprocessor Model  : 0x01
  Coprocessor Model Ext  : 0x00
  Coprocessor Type  : 0x00
  Coprocessor Family  : 0x0b
  Coprocessor Family Ext  : 0x00
```

```
  Coprocessor Stepping   : B1
  Board SKU    : B1PRQ-5110P/5120D
  ECC Mode    : Enabled
  SMC HW Revision   : Product 225W Passive CS

 Cores
  Total No of Active Cores : 60
  Voltage    : 973000 uV
  Frequency   : 1052631 kHz

 Thermal
  Fan Speed Control   : N/A
  Fan RPM    : N/A
  Fan PWM    : N/A
  Die Temp   : 62 C

 GDDR
  GDDR Vendor   : Elpida
  GDDR Version   : 0x1
  GDDR Density   : 2048 Mb
  GDDR Size   : 7936 MB
  GDDR Technology   : GDDR5
  GDDR Speed   : 5.000000 GT/s
  GDDR Frequency   : 2500000 kHz
  GDDR Voltage   : 1501000 uV
```

Users can log in directly onto the Xeon Phi coprocessor via ssh.

```
lu65fok@i01r13c01:~> ssh i01r13c01-mic0
lu65fok@i01r13c01-mic0:~$ hostname
i01r13c01-mic0
lu65fok@i01r13c01-mic0:~$ cat /etc/issue
Intel MIC Platform Software Stack (Built by Poky 7.0) 3.4 \n \l

lu65fok@i01r13c01-mic0:~$
```

Per default the home directory on the coprocessor is `/home/username` .

Since the access to the coprocessor is ssh-key based users have to generate a private/public key pair via `ssh-keygen` before accessing the coprocessor for the first time.

After the keys have been generated, the following commands have to be executed as root to populate the filesystem image for the coprocessor on the host (`/opt/intel/mic/filesystem/mic0/home`) with the new keys. Since the coprocessor has to be restarted to copy the new image to the coprocessor, the following commands have to be used (preferrably only by the system administrator) with care.

```
  weinberg@knf1:~> sudo service mpss stop
  weinberg@knf1:~> sudo micctrl --resetconfig
  weinberg@knf1:~> sudo service mpss start
```

On production systems access to reserved cards might be realized by the job scheduler.

Informations how to set up and configure a cluster with hosts containing Intel Xeon Phi coprocessors, based on how Intel configured its own Endeavor cluster can be found in [23].

Since a Linux kernel is running on the coprocessor, further information about the cores, memory etc. can be obtained from the virtual Linux /proc or /sys filesystems:

```
lu65fok@i01r13c01-mic0:~$ tail -n 25 /proc/cpuinfo
processor       : 239
vendor_id       : GenuineIntel
cpu family      : 11
model           : 1
model name      : 0b/01
stepping        : 3
cpu MHz         : 1052.630
cache size      : 512 KB
physical id     : 0
siblings        : 240
core id         : 59
cpu cores       : 60
apicid          : 239
initial apicid  : 239
fpu             : yes
fpu_exception   : yes
cpuid level     : 4
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic mtrr mca pat fxsr ht syscall
bogomips        : 2113.10
clflush size    : 64
cache_alignment : 64
address sizes   : 40 bits physical, 48 bits virtual
power management:

lu65fok@i01r13c01-mic0:~$

lu65fok@i01r13c01-mic0:~$ head  /proc/meminfo
MemTotal:        7882352 kB
MemFree:         7499036 kB
Buffers:               0 kB
Cached:           129420 kB
SwapCached:            0 kB
Active:            41868 kB
Inactive:          90880 kB
Active(anon):      41796 kB
Inactive(anon):    82116 kB
Active(file):         72 kB
lu65fok@i01r13c01-mic0:~$
```

To run MKL, OpenMP or MPI based programs on the coprocessor, some libraries (exact path and filenames may differ depending on the version) need to be copied to the coprocessor or linked from the install directories of the Intel compiler suite. On production systems the libraries might be installed or mounted on the coprocessor already. Root privileges are necessary for the destination directories given in the following example. E.g. on the SuperMIC system @ LRZ the following files are linked. (Here `link.sh` is a simple script that generates softlinks for all files of a given directory.)

```
ssh ${HOSTNAME}-${CARD} 'link.sh /lrz/sys/intel/impi/5.1.3.181/mic/lib/  /lib64'
ssh ${HOSTNAME}-${CARD} 'link.sh /lrz/sys/intel/impi/5.1.3.181/mic/bin/  /bin'
ssh ${HOSTNAME}-${CARD} 'link.sh /lrz/sys/intel/studio2016_u4/
      compilers_and_libraries_2016.4.258/linux/compiler/lib/mic/ /lib64'
ssh ${HOSTNAME}-${CARD} 'link.sh /lrz/sys/intel/studio2016_u4/
```

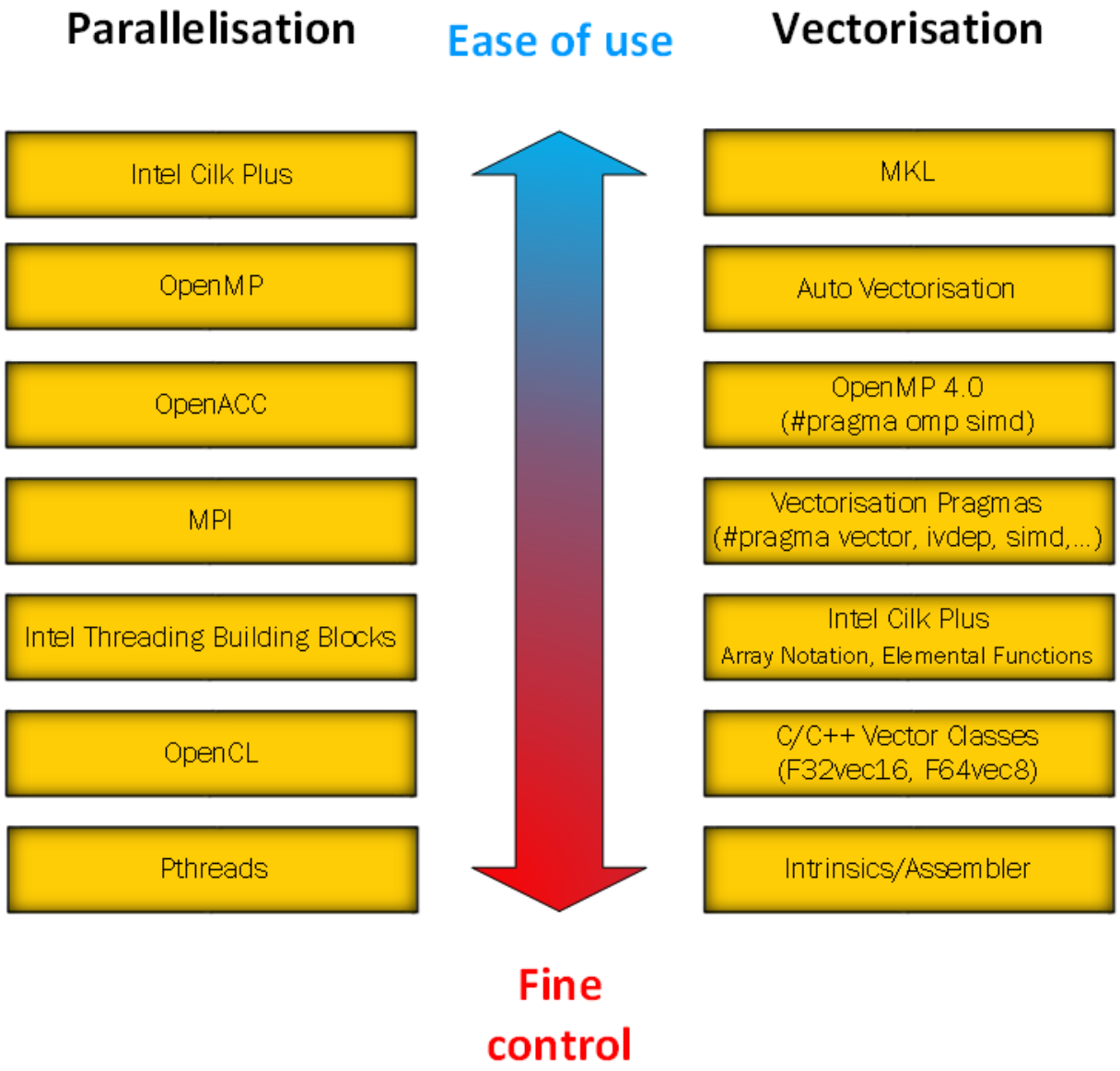
```
      compilers_and_libraries_2016.4.258/linux/mkl/lib/mic/ /lib64'
ssh ${HOSTNAME}-${CARD} 'link.sh /lrz/sys/intel/studio2016_u4/
      compilers_and_libraries_2016.4.258/linux/tbb/lib/mic/ /lib64'
ssh ${HOSTNAME}-${CARD} 'link.sh /lrz/sys/intel/studio2016_u4/
     compilers_and_libraries_2016.4.258/linux/ipp/lib/mic/ /lib64'
```

3. Programming Models

The main advantage of the MIC architecture is the possibility to program the coprocessor using plain C, C++ or Fortran and standard parallelisation models like OpenMP, MPI and hybrid OpenMP and MPI. The coprocessor can also be programmed using Intel Cilk Plus, Intel Threading Building Blocks, pthreads and OpenCL. Standard math-libraries like Intel MKL are supported and last but not least the whole Intel tool chain, e.g. Intel C/C++ and Fortran compiler, debugger and Intel VTune Amplifier. It is also possible to do hardware-specific tuning using Intrinsics or Assembler. However, we would not recommend doing this (except maybe for some critical kernel routines), since MIC vector Intrinsics and Assembler instructions are incompatible with SSE or AVX instructions.

An overview of the available programming models is shown in the following picture.

Figure 4. MIC Programming Models



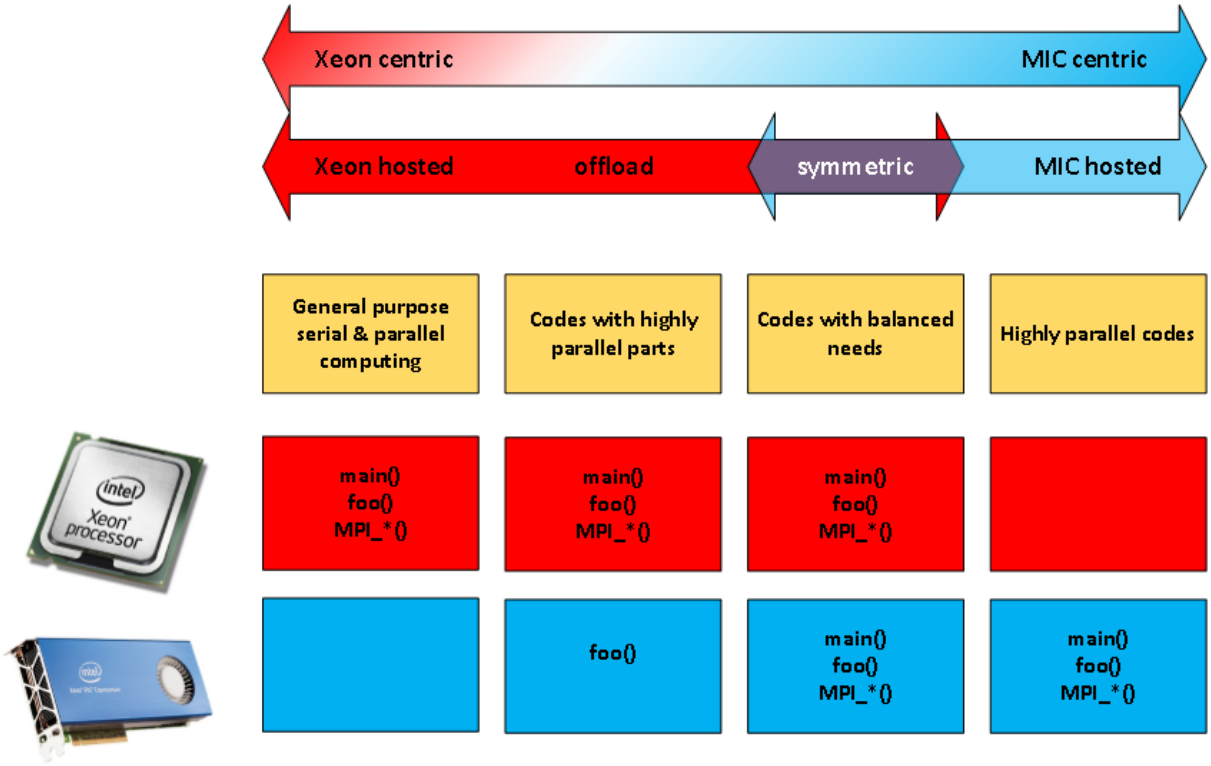
Generally speaking, two main execution modes can be distinguished: native mode and offload mode. In “native mode” the Intel compiler is instructed (through the use of the compiler-switch –mmic) to cross-compile for the MIC architecture. This is also possible for OpenMP and MPI codes. The generated executable has to be copied to the coprocessor and can be launched from within a shell running on the coprocessor.

In “offload mode” the code is instrumented with OpenMP-like pragmas in C/C++ or comments in Fortran to mark regions of code that should be offloaded to the coprocessor and be executed there at runtime. The code in the marked regions can be multithreaded by using e.g. OpenMP. The generated executable must be launched from the host. This approach is quite similar to the accelerator pragmas introduced by the PGI compiler, CAPS HMPP or OpenACC to offload code to GPGPUs.

For MPI programs, MPI ranks can reside on only the coprocessor(s), on only the host(s) (possibly doing offloading), or on both the host(s) and the coprocessor(s) allowing various combinations in clusters.

The following picture demonstrates the spectrum of executiuon modes supported, reaching from Xeon-centric over symmetric to MIC-centric modes.

Figure 5. Spectrum of MIC Programming Models



4. Native compilation

The simplest model of running applications on the Intel Xeon Phi coprocessor is native mode. Detailed information about building a native application for Intel Xeon Phi coprocessors can be found in [29].

In native mode an application is compiled on the host using the compiler switch `-mmic` to generate code for the MIC architecture. The binary can then be copied to the coprocessor and has to be started there.

```
weinberg@knf1:~/c> . /opt/intel/composerxe/bin/compilervars.sh intel64
weinberg@knf1:~/c> icc -O3 -mmic  program.c -o program
weinberg@knf1:~/c> scp program mic0:
program                                       100%   10KB  10.2KB/s   00:00
weinberg@knf1:~/c> ssh mic0 ~/program
hello, world
```

To achieve good performance one should mind the following items.

- Data should be **aligned to 64 Bytes (512 Bits)** for the MIC architecture, in contrast to 32 Bytes (256 Bits) for AVX and 16 Bytes (128 Bits) for SSE. See Section 13.1.1 for details.

- Due to the large SIMD width of 64 Bytes **vectorization is even more important for the MIC architecture than for Intel Xeon!** The MIC architecture offers new instructions like gather/scatter, fused multiply-add, masked vector instructions etc. which allow more loops to be parallelized on the coprocessor than on an Intel Xeon based host.

- Use pragmas like `#pragma ivdep`, `#pragma vector always`, `#pragma vector aligned`, `#pragma simd` etc. to achieve autovectorization. Autovectorization is enabled at default optimization level `-O2` . Requirements for vectorizable loops can be found in [38].

- Let the compiler generate vectorization reports using the compiler option `-qopt-report=n -qopt-report-phase=vec` to see if loops were vectorized for MIC (Message "*MIC* Loop was vectorized" etc) in the output files.

- Explicit vector programming is also possible via Intel Cilk Plus language extensions (C/C++ array notation, vector elemental functions, ...) or the new SIMD constructs from OpenMP 4.0 RC1.

- Vector elemental functions can be declared by using `__attributes__((vector)).` The compiler then generates a vectorized version of a scalar function which can be called from a vectorized loop.

- One can use intrinsics to have full control over the vector registers and the instruction set. Include `<immintrin.h>` for using intrinsics.

- Hardware prefetching from the L2 cache is enabled per default. In addition, software prefetching is on by default at compiler optimization level `-O2` and above. Since Intel Xeon Phi is an inorder architecture, care about prefetching is more important than on out-of-order architectures. The compiler prefetching can be influenced by setting the compiler switch `-opt-prefetch=n`. Manual prefetching can be done by using intrinsics (`_mm_prefetch()`) or pragmas (`#pragma prefetch var`).

5. Offloading

5.1. Introduction

Note: The offloading approach described in this section describes the Intel Language Extensions for Offload (LEO), which was the only method available for offloading using pragmas when the first Intel Xeon Phi coprocessors were shipped. Meanwhile OpenMP 4.x offloading is also supported by the Intel Compiler. This is covered in Section 6.3.

The offload programming model treats the Xeon Phi just like a co-processor. Part of the executable code is executed on the mic co-processor. The program is run on the host processor and is compiled for Sandy-Bridge/Haswell architecture. The only difference is that some part of the executable is execute on the co-processor. The part of the code run on the co-processor can be a library function like MKL, a user written function or routine or a region of the program. The cases involving user written code the compiler must be instructed to generate code to be cross compiled for the mic architecture. This is done with compiler directives much like OpenMP directives.

One can simply add OpenMP-like pragmas to C/C++ or Fortran code to mark regions of code that should be offloaded to the Intel Xeon Phi Coprocessor and be run there. This approach is quite similar to the accelerator pragmas introduced by the PGI compiler, CAPS HMPP or OpenACC to offload code to GPGPUs. When the Intel compiler encounters an offload pragma, it generates code for both the coprocessor and the host. Code to transfer the data to the coprocessor is automatically created by the compiler, however the programmer can influence the data transfer by adding data clauses to the offload pragma. Details can be found under "in the Intel compiler documentation.

5.2. Simple C example

In the following we show a simple example how to offload a matrix-matrix computation to the coprocessor.

```
main(){

double *a, *b, *c;
int i,j,k, ok, n=100;

// allocated memory on the heap aligned to 64 byte boundary
ok = posix_memalign((void**)&a, 64, n*n*sizeof(double));
ok = posix_memalign((void**)&b, 64, n*n*sizeof(double));
ok = posix_memalign((void**)&c, 64, n*n*sizeof(double));

// initialize matrices
...
//offload code
#pragma offload target(mic) in(a,b:length(n*n)) inout(c:length(n*n))
{
//parallelize via OpenMP on MIC
#pragma omp parallel for
for( i = 0; i < n; i++ ) {
for( k = 0; k < n; k++ ) {
#pragma vector aligned
#pragma ivdep
   for( j = 0; j < n; j++ ) {
   //c[i][j] = c[i][j] + a[i][k]*b[k][j];
   c[i*n+j] = c[i*n+j] + a[i*n+k]*b[k*n+j];
}}}}}
```

This simple example (without any blocking etc. and thus quite bad performance) shows how to offload the matrix computation to the coprocessor using the `#pragma offload target(mic)`. One could also specify the specific coprocessor `num` in a system with multiple coprocessors by using `#pragma offload target(mic:num)`.

Since the matrices have been dynamically allocated using `posix_memalign()`, their sizes must be specified via the `length()` clause. Using `in`, `out` and `inout` one can specify which data has to be copied in which direction. It is recommended that for Intel Xeon Phi data is 64-byte aligned. `#pragma vector aligned` tells the compiler that all array data accessed in the loop is properly aligned. `#pragma ivdep` discards any data dependencies assumed by the compiler.

Offloading is enabled per default for the Intel compiler. Use `-no-offload` to disable the generation of offload code.

5.3. Simple Fortran-90 example

In the following we show a simple example how to offload a Fortran 90 matrix-matrix computation to the co-processor.

5.3.1. User function offload.

The Intel compilers have support for user defined functions or regions inside a program to be offloaded onto the MIC co-processor. This is a bit more complex that just calling MKL routines. One can use regions in a program or write a complete function or subroutine to be compiled and run on the co-processor. In both cases the code marked for offload will be cross compiled for the mic architecture. The run time system will launch the code on the MIC processor and data is exchanged with yet another set of run time functions. Many possible combinations are possible, overlap of data transfer, load balancing between the host processor and the co-processor etc. However, all of this is left to the programmer. This makes usage of the co-processor a little bit harder to use in production where none og very few changes to the source code is wanted.

5.3.2. Offloading regions in program

This is the simplest solution where directives to instruct the compiler to generated offload code are just inserted into the program code. The following code show an example of how the nested do loop is effectively offloaded from the host processor to the mic co-processor.

```
!dir$ offload begin target(mic) in(a,b) out(c)
!$omp parallel do private(j,l,i)
  do j=1,n
     do l=1,n
        do i=1,n
           c(j,l)=c(j,l) + a(i,l)*b(i,l)
        enddo
     enddo
  enddo
  nt=omp_get_max_threads()

#ifdef __MIC__
  print*, "Hello MIC threads:",nt
#else
  print*, "Hello CPU threads:",nt
#endif
!dir$ end offload
```

The offloaded part of the code is executed on the MIC co-processor using an environment either set up by the system or bye the user via environment variables. Both the number of threads and thread placements on the mic processor can be controlled in this way. During the time the offload code is run on the co-processor the host processors are idle. To achieve load balancing the user must explicitly program the work partition.

5.3.3. Offloading functions or subroutines

This approach is making usage of the offload code simpler and is the more common way of programming. The user write a complete function or subroutine to be offloaded. Using this routine is straighforward it is called just as any other function with the only addition that data must be handled. Initiating data transfer between the two memories must be handled. This data transfer can be overlapping with other workload on the host processor hiding latency of the transfer. To follow the example above this piece of code can easily be put into a subroutine.

```
!dir$ attributes offload : mic :: mxm, omp_get_max_threads
subroutine mxm(a,b,c,n)
  use constants

  integer :: n
  real(r8),dimension(n,n) :: a,b,c
  integer :: i,j,l,nt

!$omp parallel do private(j,l,i)
  do j=1,n
     do l=1,n
        do i=1,n
           c(j,l)=c(j,l) + a(i,l)*b(i,l)
        enddo
     enddo
  enddo
  nt=omp_get_max_threads()

#ifdef __MIC__
```

```
  print*, "Hello MIC threads:",nt
#else
  print*, "Hello CPU threads:",nt
#endif

end subroutine mxm
```

This routine will now be compiled to an object file suitable for executing one the MIC co-processor. It can be called as any other routine, but data transfer must be accommodated for. The calling program need to arrange transfer.

```
time_start=mysecond()
!dir$ offload_transfer target(mic:0) in( a: alloc_if(.true.) free_if(.false.) )
!dir$ offload_transfer target(mic:0) in( b: alloc_if(.true.) free_if(.false.) )
!dir$ offload_transfer target(mic:0) in( c: alloc_if(.true.) free_if(.false.) )


!dir$ offload  target(mic:0)   in(a,b: alloc_if(.false.) free_if(.false.)) &
                               out(c: alloc_if(.false.) free_if(.false.))
  call mxm(a,b,c,n)

  time_end=mysecond()
```

The subroutine call will block using this construct. In order to utilize both host and co-processor resources concurrency and synchronization need to be introduced. However, the above setup works very for testing and timing purposes.

5.3.4. Load distribution and balancing, hosts cpus, co-processors (single and multiple)

It is relatively straightforward to set up concurrent runs, workload distribution and ultimately load balancing between the host cpus and the Xeon Phi mic processors. However, all of the administration is left to the programmer. Since there is no shared memory the work and memory partition must be explicitly handled. Only the buffers used on the co-processors need to be transferred as memory movement is limited by the PCIe bus' bandwidth. There are mechanisms for offline transfer and semaphores to synchronising both transfer and execution. All of this must be explicitly handled by the programmer. While each part is relatively simple it can become quite complex when trying to partition the problem while trying to load balancing. Some examples below will try to illustrate this.

Filling the matrices for the co-processors:

```
am0(:,:)=a(1:m,:)
am1(:,:)=a(m+1:2*m,:)
bm0(:,:)=b(1:m,:)
bm1(:,:)=b(m+1:2*m,:)
```

Initiate data transfer:

```
!dir$ offload_transfer target(mic:0) in( am0: alloc_if(.true.) free_if(.false.) )
!dir$ offload_transfer target(mic:0) in( bm0: alloc_if(.true.) free_if(.false.) )

!dir$ offload_transfer target(mic:1) in( am1: alloc_if(.true.) free_if(.false.) )
!dir$ offload_transfer target(mic:1) in( bm1: alloc_if(.true.) free_if(.false.) )
```

Variables for each co-processor have been declared and allocated. These are 1/3 og the size of the total matrix size held in the host memory. Each compute element (Host SB processors, mic0 and mic1) is doing 1/3 of the total calculation. No dynamic load balancing, fixed at 1/3 each. Calling the offloading subroutine:

```
time_start=mysecond()

!dir$ offload target (mic:0) in(am0,bm0) out(cm0) signal(s1)
  call mxm(am0,bm0,cm0,m,n)

  print *,"cm0",cm0(:,:)


!dir$ offload target (mic:1) in(am1,bm1) out(cm1) signal(s2)
  call mxm(am1,bm1,cm1,m,n)

  print *,"cm1",cm1(:,:)

  kc=2*m+1

!$omp parallel do private(j,l,i)
  do j=kc,n
     do l=1,n
        do i=1,n
           c(j,l)=c(j,l) + a(i,l)*b(i,l)
        enddo
     enddo
  enddo
  nt=omp_get_max_threads()
#ifdef __MIC__
  print*, "Hello MIC threads:",nt
#else
  print*, "Hello CPU threads:",nt
#endif
```

Here we wait for the co-processors if they have not yet finished. One semaphore for each co-processor.

```
dir$ offload_wait target(mic:0) wait(s1)
dir$ offload_wait target(mic:1) wait(s2)
```

Copy the data received from co-processors back in matrices on host memory:

```
! Put the parts computed on the mics into tha sub matrix of c.

  c(1:m,:)=cm0(:,:)
  c(m+1:2*m,:)=cm1(:,:)

!  c(2*m+1:3*m,:) is already in c, nothing to do.

  time_end=mysecond()
```

The amount of work is about evenly distributed with just a little time spend waiting for another compute element to finish it's work. In a production run the load balancing would be set up dynamic and hence a better load balancing obtained. However, this is left to the programmer and require detailed knowledge of the workload. Again we see that the actual programming is easy, but the administration of the workload can be complex. One advantage is that for we have more memory and can attack bigger problems without having to run offload parts in series with a chunk of data for each run. This must be done if the problem we want to offload exceed the 8 GiB of currently installed memory on the Xeon Phi cards.

Intel provide the tools needed to do the job, but the programmer need to to all the details himself. It must be noted that this is vastly simpler than doing similar programming using a GPU. Here the exact same Fortran 90 code is used for both processor classes, host processor and Xeon Phi/mic. Offloading is a very easy way to start

accelerating your code. It might not utilize the co-processors full potential but nevertheless any speedup with a small effort is worth harvesting.

5.4. Obtaining informations about the offloading

Using the compiler option `-vec-report2` one can see which loops have been vectorized on the host and the MIC coprocessor:

```
weinberg@knf1:~/c> icc  -vec-report2   -openmp offload.c
offload.c(57): (col. 2) remark: loop was not vectorized: vectorization
                                      possible but seems inefficient.
...
offload.c(57): (col. 2) remark: *MIC* LOOP WAS VECTORIZED.
offload.c(54): (col. 7) remark: *MIC* loop was not vectorized: not inner loop.
offload.c(53): (col. 5) remark: *MIC* loop was not vectorized: not inner loop.
```

The option `-vec-report2` has been deprecated in icc 15.0. Use `-qopt-report=n -qopt-report-phase=vec` and view the `*.optrpt files`.

By setting the environment variable `OFFLOAD_REPORT` one can obtain information about performance and data transfers at runtime:

```
weinberg@knf1:~/c> export OFFLOAD_REPORT=2
weinberg@knf1:~/c> ./a.out
[Offload] [MIC 0] [File]            offload2.c
[Offload] [MIC 0] [Line]            50
[Offload] [MIC 0] [CPU Time]        12.853562 (seconds)
[Offload] [MIC 0] [CPU->MIC Data]   9830416 (bytes)
[Offload] [MIC 0] [MIC Time]        12.208636 (seconds)
[Offload] [MIC 0] [MIC->CPU Data]   3276816 (bytes)
```

If a function is called within the offloaded code block, this function has to be declared with `__attribute__((target(mic)))` .

For example one could put the matrix-matrix multiplication of the previous example into a subroutine and call that routine within an offloaded block region:

```
__attribute__((target(mic))) void mxm( int n,  double * restrict a,
                           double * restrict b, double *restrict c ){
  int i,j,k;
  for( i = 0; i < n; i++ ) {
  ...
}

main(){

...

#pragma offload target(mic) in(a,b:length(n*n)) inout(c:length(n*n))
  {
    mxm(n,a,b,c);
  }

}
```

Mind the C99 restrict keyword that specifies that the vectors do not overlap. (Compile with -std=c99)

5.5. Syntax of pragmas

The following offload pragmas are available (from [14]):

Pragma	Syntax	Semantic
C/C++		
Offload pragma	`#pragma offload <clauses> <statement>`	Allow next statement to execute on coprocessor or host CPU
Variable/function offload properties	`_attribute__ ((target(mic)))`	Compile function for, or allocate variable on, both host CPU and coprocessor

Entire blocks of data/code defs	`#pragma offload_attribute(push, target(mic))`
	`...`
	`#pragma offload_attribute(pop)`
		Mark entire files or large blocks of code to compile for both host CPU and co-processor

Fortran		
Offload directive	`!dir$ omp offload <clauses> <state-ment>`	Execute OpenMP parallel block on coprocessor
Variable/function offload properties	`!dir$ attributes offload:<mic> :: <ret-name> OR <var1,var2,…>`	Compile function or variable for CPU and coprocessor

Entire code blocks	`!dir$ offload begin <clauses>`
	`...`
	`!dir$ end offload`
		Mark entire files or large blocks of code to compile for both host CPU and co-processor

The following clauses can be used to control data transfers:

Clause	Syntax	Semantic
Multiple coprocessors	`target(mic[:unit])`	Select specific coprocessors
Inputs	`in(var-list modifiers)`	Copy from host to co-processor
Outputs	`out(var-list modifiers)`	Copy from coprocessor to host
Inputs & outputs	`inout(var-list modifiers)`	Copy host to coprocessor and back when offload completes
Non-copied data	`nocopy(var-list modifiers)`	Data is local to target

The following (optional) modifiers are specified:

Modifier	Syntax	Semantic
Specify copy length	`length(N)`	Copy N elements of pointer's type
Coprocessor memory allocation	`alloc_if ( bool )`	Allocate coprocessor space on this offload (default: TRUE)

Modifier	Syntax	Semantic
Coprocessor memory release	`free_if ( bool )`	Free coprocessor space at the end of this offload (default: TRUE)
Control target data alignment	`align ( N bytes )`	Specify minimum memory alignment on coprocessor
Array partial allocation & variable relocation	`alloc ( array-slice ) into ( var-expr )`	Enables partial array allocation and data copy into other vars & ranges

5.6. Recommendations

5.6.1. Explicit worksharing

To explicitly share work between the coprocessor and the host one can use OpenMP sections to manually distribute the work. In the following example both the host and the coporcessor will run a matrix-matrix multiplication in parallel.

```
#pragma omp parallel
  {
#pragma omp sections
    {
#pragma omp section
      {
//section running on the coprocessor
#pragma offload target(mic) in(a,b:length(n*n)) inout(c:length(n*n))
 {
   mxm(n,a,b,c);
 }
      }
#pragma omp section
      {
//section running on the host
 mxm(n,d,e,f);
      }
    }

  }
```

5.6.2. Persistent data on the coprocessor

The main bottleneck of accelerator based programming are data transfers over the slow PCIe bus from the host to the accelerator and vice versa. To increase the performance one should minimize data transfers as much as possible and keep the data on the coprocessor between computations using the same data.

Defining the following macros

```
#define ALLOC   alloc_if(1)
#define FREE    free_if(1)
#define RETAIN  free_if(0)
#define REUSE   alloc_if(0)
```

one can simply use the following notation:

- to allocate data and keep it for the next offload

```
#pragma offload target(mic) in (p:length(l) ALLOC RETAIN)
```

- to reuse the data and still keep it on the coprocessor

```
#pragma offload target(mic) in (p:length(l) REUSE RETAIN)
```

- to reuse the data again and free the memory. (FREE is the default, and does not need to be explicitly specified)

```
#pragma offload target(mic) in (p:length(l) REUSE FREE)
```

More information can be found in the section "Managing Memory Allocation for Pointer Variables" under "Offload Using a Pragma" in the Intel compiler documentation.

5.6.3. Optimizing offloaded code

The implementation of the matrix-matrix multiplication given in Section 5.2 can be optimized by defining appropriate ROWCHUNK and COLCHUNK chunk sizes, rewriting the code with 6 nested loops (using OpenMP collapse for the 2 outermost loops) and some manual loop unrolling (thanks to A. Heinecke for input for this section).

```
#define ROWCHUNK 96
#define COLCHUNK 96

#pragma omp parallel for collapse(2) private(i,j,k)
    for(i = 0; i < n; i+=ROWCHUNK ) {
      for(j = 0; j < n; j+=ROWCHUNK ) {
 for(k = 0; k < n; k+=COLCHUNK ) {
   for (ii = i; ii < i+ROWCHUNK; ii+=6)  {
     for (kk = k; kk < k+COLCHUNK; kk++ ) {
#pragma ivdep
#pragma vector aligned
       for ( jj = j; jj < j+ROWCHUNK; jj++){
  c[(ii*n)+jj] += a[(ii*n)+kk]*b[kk*n+jj];
  c[((ii+1)*n)+jj] += a[((ii+1)*n)+kk]*b[kk*n+jj];
  c[((ii+2)*n)+jj] += a[((ii+2)*n)+kk]*b[kk*n+jj];
  c[((ii+3)*n)+jj] += a[((ii+3)*n)+kk]*b[kk*n+jj];
  c[((ii+4)*n)+jj] += a[((ii+4)*n)+kk]*b[kk*n+jj];
  c[((ii+5)*n)+jj] += a[((ii+5)*n)+kk]*b[kk*n+jj];
              }
            }
          }
        }
      }
    }
}
```

Using intrinsics with manual data prefetching and register blocking can still considerably increase the performance. Generally speaking, the programmer should try to get a suitable vectorization and write cache and register efficient code, i.e. values stored in registers should be reused as often as possible in order to avoid cache and memory access. The tuning techniques for native implementations discussed in Section 4 also apply for offloaded code, of course.

5.7. Intel Cilk Plus parallel extensions

More complex data structures can be handled by Virtual Shared Memory. In this case the same virtual address space is used on both the host and the coprocessor, enabling a seamless sharing of data. Virtual shared data is specified using the `_Cilk_shared` allocation specifier. This model is integrated in Intel Cilk Plus parallel

extensions and is only available in C/C++. There are also Cilk functions to specify offloading of functions and `_Cilk_for` loops. More information on Intel Cilk Plus can be found online under [15].

Intel Cilk Plus offers elemental functions, array syntax and "#pragma SIMD" to help with vectorization. To achieve the best performance, the use of array syntax should be done along with cache blocking. Unfortunately, when using constructs such as A[:] = B[:] + C[:]; for large arrays the performance may degrade. To make proper use of the array syntax ensures the vector length of single statements is short (some small multiple of the native vector length, perhaps only 1X). Finally, and perhaps most important to programmers today, Intel Cilk Plus offers mandatory vectorization pragmas for the compiler called "#pragma SIMD." The intent of "#pragma SIMD" is to do for vectorization what OpenMP has done for parallelization. Intel Cilk Plus requires compiler support. It is currently available from Intel for Windows, Linux and Apple OS X. It is also available in a branch of gcc.

Intel Cilk Plus provides three new keywords, support for reduction operations, and data parallel extensions. The keyword cilk_spawn can be applied to a function call, as in x = cilk_spawn fib(n-1), to indicate that the function fib can execute concurrently with the subsequent code. The keyword cilk_sync indicates that execution has to wait until all spawns from the function have returned. The use of the function as a unit of spawn makes the code readable, relies on the baseline C/C++ language to define scoping rules of variables, and allows Intel Cilk Plus programs to be composable. Below is an example calculating the Fibonacci series using Cilk Plus.

```
int fib (int n) {

  if (n < 2) return 1;
  else {
    int x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return x + y;
  }
}
```

Tools for Cilk Plus on the Xeon Phi

- Compiling: Intel C compiler - `icc cilkprogram.cpp -o cilkprogram`
- Check parallelism: `cilkview ./cilkprogram`
- Check race conditions: `cilkscreen ./cilkprogram`
- Profiling: `cilkprof ./cilkprogram`
- GCC 5.0 and LLVM also provide Cilk Plus support

6. OpenMP, hybrid and OpenMP 4.x Offloading

6.1. OpenMP

6.1.1. OpenMP basics

OpenMP is the de facto standard pragma-based parallel programming model for shared memory systems. The standard can be found online under http://www.openmp.org/specifications/ [46]. OpenMP parallelization on an Intel Xeon + Xeon Phi coprocessor machine can be applied both on the host and on the device with the `-openmp` compiler option. It can run on the Xeon host, natively on the Xeon Phi coprocessor and also in different offload schemes. It is introduced with regular pragma statements in the code for either case.

In applications that run both on the host and on the Xeon Phi coprocessor OpenMP threads do not interfere with each other and offload statements are executed on the device based on the available resources. If there are no free threads on the Xeon Phi coprocessor or less than requested, the parallel region will be executed on the host. Offload pragma statements and OpenMP pragma statements are independent from each other and must both be present in the code. Within the latter construct apply usual semantics of shared and private data.

There are 16 threads available on every Xeon host CPU and 4 times the number of cores threads on every Xeon Phi coprocessor. For offload schemes the maximal amount of threads that can be used on the device is 4 times the number of cores minus one, because one core is reserved for the OS and its services. Offload to the Xeon Phi coprocessor can be done at any time by multiple host CPUs until filling the resources available. If there are no free threads, the task meant to be offloaded may be done on the host.

6.1.2. Threading and affinity

The Xeon Phi coprocessor supports 4 threads per core. Unlike some CPU-intensive HPC applications that are run on Xeon architecture, which do not benefit from hyperthreading, applications run on Xeon Phi coprocessors do and using more than one thread per core is recommended.

The most important considerations for OpenMP threading and affinity are the total number of threads that should be utilized and the scheme for binding threads to processor cores. These are controlled with the environmental variables `OMP_NUM_THREADS` and `KMP_AFFINITY`.

The default settings are as follows:

	`OMP_NUM_THREADS`
OpenMP on host without HT	1 x ncore-host
OpenMP on host with HT	2 x ncore-host
OpenMP on Xeon Phi in native mode	4 x ncore-phi
OpenMP on Xeon Phi in offload mode	4 x (ncore-phi - 1)

For host-only and native-only execution the number of threads and all other environmental variables are set as usual:

```
% export OMP_NUM_THREADS=16
% export OMP_NUM_THREADS=240
% export KMP_AFFINITY=compact/scatter/balanced
```

Setting affinity to “compact” will place OpenMP threads by filling cores one by one, while “scatter” will place one thread in every core until reaching the total number of cores and then continue with the first core. When using “balanced”, it is similar to “scatter”, but threads with adjacent numbers will be placed on the same core. “Balanced” is only available for the Xeon Phi coprocessor. Another useful option is the verbosity modifier:

```
 % export KMP_AFFINITY=verbose,balanced
```

With compiler version 13.1.0 and newer one can use the `KMP_PLACE_THREADS` variable to point out the topology of the system to the OpenMP runtime, for example:

```
 % export OMP_NUM_THREADS=180
 % export KMP_PLACE_THREADS=60c,3t
```

meaning that 60 cores and 3 threads per core should be used. Still one should use the `KMP_AFFINITY` variable to bind the threads to the cores.

If OpenMP regions exist on the host and on the part of the code offloaded to the Phi, two separate OpenMP runtimes exist. Environment variables for controlling OpenMP behavior are to be set for both runtimes, for example the `KMP_AFFINITY` variable which can be used to assign a particular thread to a particular physical node. For Phi it can be done like this:

```
$ export MIC_ENV_PREFIX=MIC
# specify affinity for all cards
$ export MIC_KMP_AFFINITY=...
# specify number of threads for all cards
$ export MIC_OMP_NUM_THREADS=120
# specify the number of threads for card #2
$ export MIC_2_OMP_NUM_THREADS=200
# specify the number of threads and affinity for card #3
$ export MIC_3_ENV=”OMP_NUM_THREADS=60 | KMP_AFFINITY=balanced”
```

If `MIC_ENV_PREFIX` is not set and `KMP_AFFINITY` is set to “balanced” it will be ignored by the runtime.

One can also use special API calls to set the environment for the coprocessor only, e.g.

```
omp_set_num_threads_target()
omp_set_nested_target()
```

6.1.3. Loop scheduling

OpenMP accepts four different kinds of loop scheduling - static, dynamic, guided and auto. In this way the amount of iterations done by different threads can be controlled. The `schedule` clause can be used to set the loop scheduling at compile time. Another way to control this feature is to specify `schedule(runtime)` in your code and select the loop scheduling at runtime through setting the `OMP_SCHEDULE` environment variable.

6.1.4. Scalability improvement

If the amount of work that should be done by each thread is non-trivial and consists of nested for-loops, one might use the `collapse()` directive to specify how many for-loops are associated with the OpenMP loop construct. This often improves scalability of OpenMP applications (see Section 5.6.3).

Another way to improve scalability is to reduce barrier synchronization overheads by using the `nowait` directive. The effect of it is that the threads will not synchronize after they have completed their individual pieces of work. This approach is applicable combined with static loop scheduling because all threads will execute the same amount of iterations in each loop but is also a potential threat on the correctness of the code.

6.2. Hybrid OpenMP/MPI

6.2.1. Programming models

For hybrid OpenMP/MPI programming there are two major approaches: an MPI offload approach, where MPI ranks reside on the host CPU and work is offloaded to the Xeon Phi coprocessor and a symmetric approach in which MPI ranks reside both on the CPU and on the Xeon Phi. An MPI program can be structured using either model.

When assigning MPI ranks, one should take into account that there is a data transfer overhead over the PCIe, so minimizing the communication from and to the Xeon Phi is a good idea. Another consideration is that there is limited amount of memory on the coprocessor which favors the shared memory parallelism ideology so placing 120 MPI ranks on the coprocessor each of which starts 2 threads might be less effective than placing less ranks but allowing them to start more threads. Note that MPI directives cannot be called within a pragma offload statement.

6.2.2. Threading of the MPI ranks

For hybrid OpenMP/MPI applications use the thread safe version of the Intel MPI Library by using the `-mt_mpi` compiler driver option. A desired process pinning scheme can be set with the `I_MPI_PIN_DOMAIN` environment variable. It is recommended to use the following setting:

```
$ export I_MPI_PIN_DOMAIN=omp
```

By setting this to omp, one sets the process pinning domain size to be to `OMP_NUM_THREADS`. In this way, every MPI process is able to create `OMP_NUM_THREADS` number of threads that will run within the corresponding domain. If this variable is not set, each process will create a number of threads per MPI process equal to the number of cores, because it will be treated as a separate domain.

Further, to pin OpenMP threads within a particular domain, one could use the `KMP_AFFINITY` environment variable.

6.3. OpenMP 4.x Offloading

6.3.1. Execution Model

Starting with OpenMP 4.0 new device constructs have been added to the OpenMP standard to support heterogeneous systems and enable offloading of computations and data to devices.

The Intel compiler supports OpenMP 4.x for Intel Xeon Phi as an alternative to the Intel specific Offloading (LEO) described in the previous section.

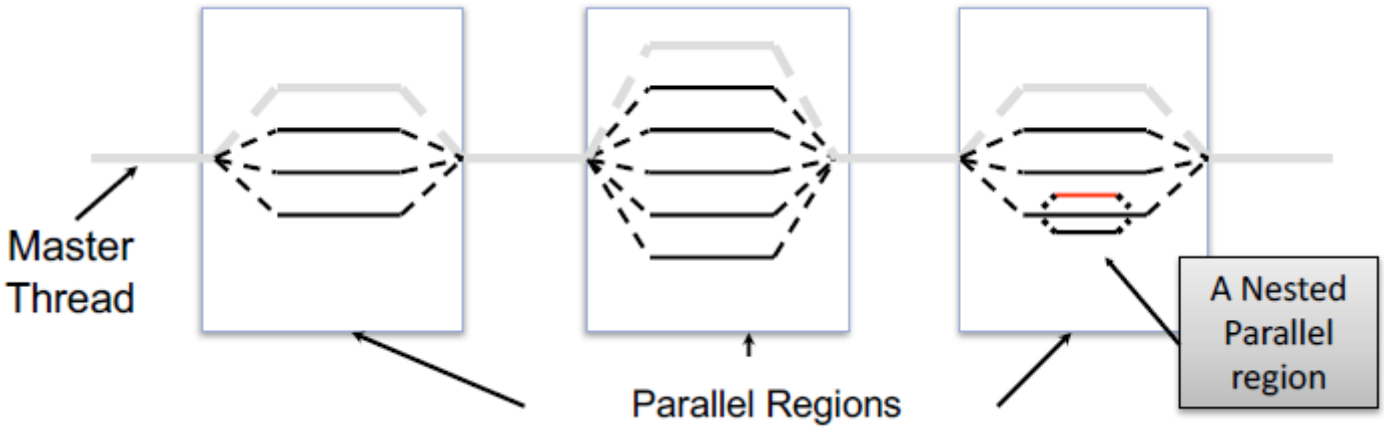
The OpenMP language terminology concerning devices is quite general. A device is defined as an implementation defined logical execution unit (which could have one or more processors). The execution model is host-centric: The **host device**, i.e. the device on which the OpenMP program begins execution, offloads code or data to a **target device**. If an implementation supports target devices, one or more devices are available to the host device for offloading. Threads running on one device are distinct from threads that execute on another device and cannot migrate to other devices.

The most important OpenMP device constructs are the **target** and the **teams** construct. When a target construct is encountered, a new **target task** is generated. When the target task is executed, the enclosed **target region** is executed by an initial thread which executes sequentially on a target device if present and supported. If the target device does not exist or the implementation does not support the target device, all target regions associated with that device execute on the host device. In this case, the implementation must ensure that the target region executes as if it were executed in the data environment of the target device.

The **teams** construct creates a **league of thread teams** where the master thread of each team executes the region sequentially.

This is the main difference in the execution model compared to versions < 4.x of the OpenMP standard. Before device directives have been introduced, a master thread could spawn a team of threads executing parallel regions of a program as needed (see Figure 6).

Figure 6. Execution model before OpenMP 4.x. From [48].



With OpenMP 4.x the master thread spawns a team of threads, and these threads can spawn leagues of thread teams as needed. Figure 7 shows an example of a teams region consisting of one team executing a parallel region, while Figure 8 shows an example using multiple teams.

Figure 7. Execution model after OpenMP 4.x executing one team. From [48].

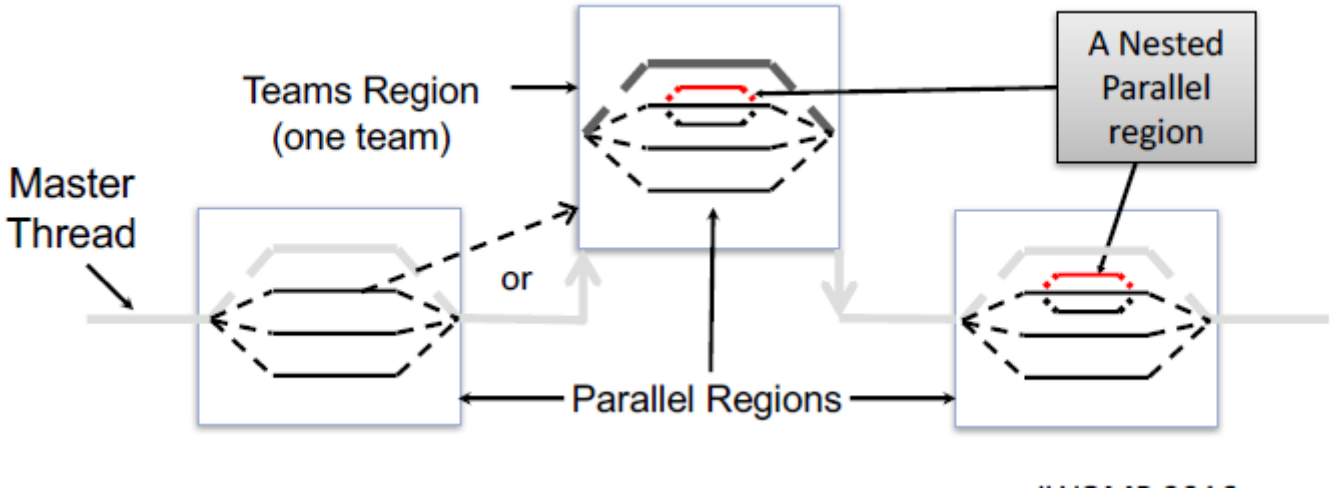
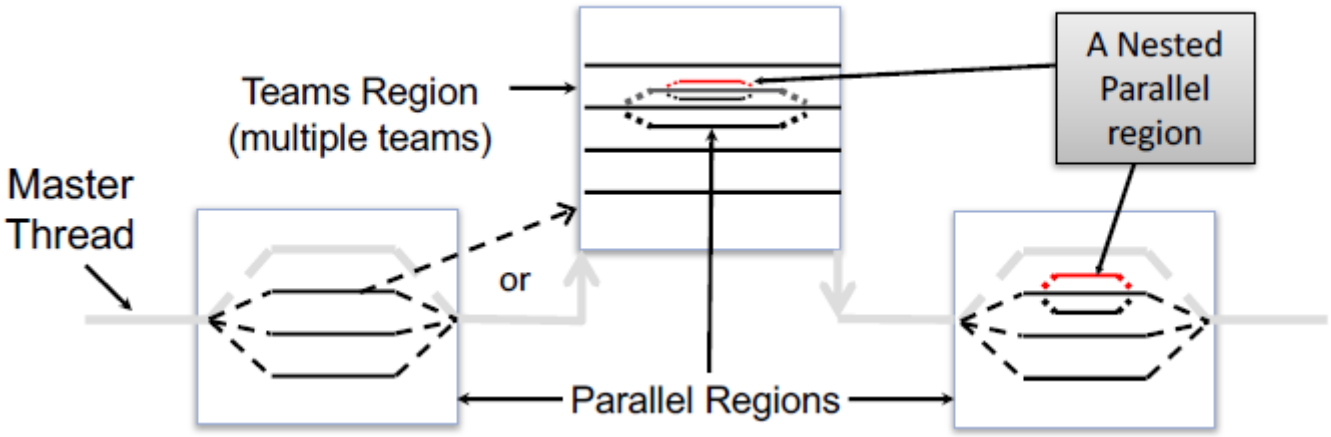


Figure 8. Execution model after OpenMP 4.x executing multiple teams. From [48].



6.3.2. Overview of the most important device constructs

The following table gives an overview of the OpenMP 4.x device constructs:

#pragma omp target data	Creates a data environment for the extent of the region.
#pragma omp target enter data	Specifies that variables are mapped to a device data environment.
#pragma omp target exit data	Specifies that list items are unmapped from a device data environment.
#pragma omp target	Map variables to a device data environment and execute the construct on the device.

#pragma omp target update	Makes the corresponding list items in the device data environment consistent with their original list items, according to the specified motion clauses (to/from).
#pragma omp declare target	A declarative directive that specifies that variables and functions are mapped to a device.
#pragma omp teams	Creates a league of thread teams where the master thread of each team executes the region.
#pragma omp distribute	Specifies loops which are executed by the thread teams
#pragma omp … simd	Specifies code that is executed concurrently using SIMD instructions.
#pragma omp distribute parallel for	Specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams.

The most important constructs are the **target**, **teams** and **distribute** constructs, which are explained in the following in more detail.

6.3.3. The target construct

The **target** construct maps variables to a device data environment and executes the construct on that device.

```
#pragma omp target [clause[ [,] clause] ... ] new-line
structured-block
```

where clause is one of the following:

- if([target :] scalar-expression)
- device(integer-expression)
- private(list)
- firstprivate(list)
- map([[map-type-modifier[,]] map-type:] list)
- is_device_ptr(list)
- defaultmap(tofrom:scalar)
- nowait
- depend(dependence-type: list)

6.3.4. The teams construct

The **teams** construct creates a league of thread teams and the master thread of each team executes the region.

```
#pragma omp teams [clause[ [,] clause] ... ] new-line
structured-block
```

where clause is one of the following:

- num_teams(integer-expression)
- thread_limit(integer-expression)

- default(shared | none)
- private(list)
- firstprivate(list)
- shared(list)
- reduction(reduction-identifier : list)

6.3.5. The distribute construct

The **distribute** construct specifies that the iterations of one or more loops will be executed by the thread teams in the context of their implicit tasks. The iterations are distributed across the master threads of all teams that execute the teams region to which the distribute region binds.

```
#pragma omp distribute [clause[ [,] clause] ... ] new-line
for-loops
```

Where clause is one of the following:

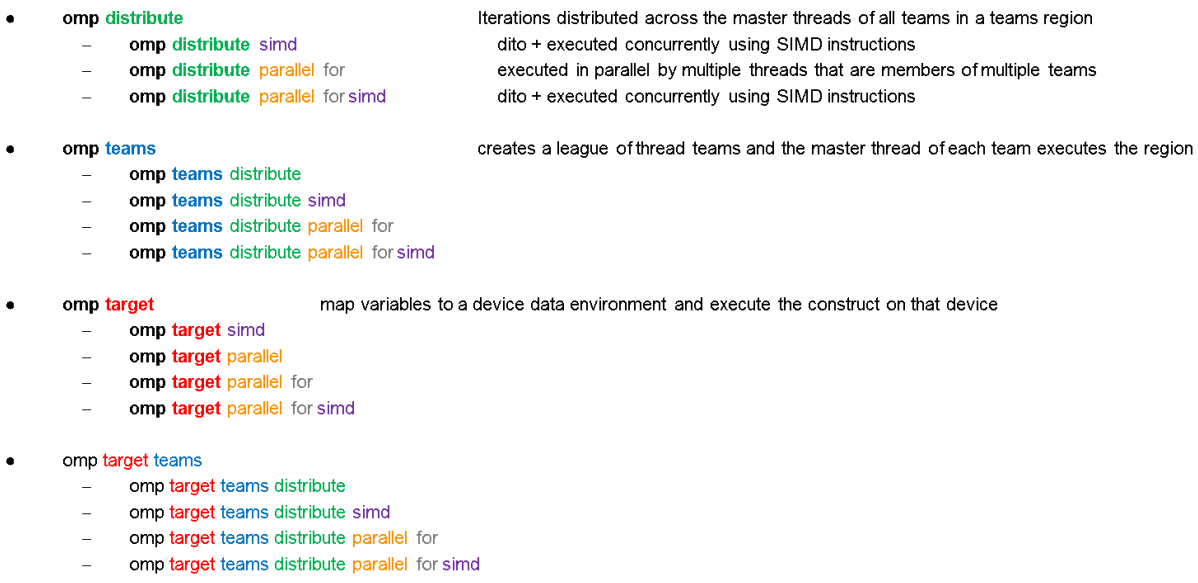
- private(list)
- firstprivate(list)
- lastprivate(list)
- collapse(n)
- dist_schedule(kind[, chunk_size])

A detailed description of the most important clauses can be found in the OpenMP standard.

6.3.6. Composite constructs and shortcuts in OpenMP 4.5

The OpenMP standard defines a couple of combined constructs. Combined constructs are shortcuts for specifying one construct immediately nested inside another construct. The semantics of the combined constructs are identical to that of explicitly specifying the first construct containing one instance of the second construct and no other statements. Figure 9 gives an overview of the supported composite constructs and shortcuts.

Figure 9. Overview of composite constructs and shortcuts in OpenMP 4.5.



The last construct (`omp target teams distribute parallel for simd`) contains the most levels of parallelism. The computations are offloaded to a target device and the iterations of a loop are executed in parallel by multiple threads that are member of multiple teams using SIMD instructions.

6.3.7. Examples

Many useful examples can be found in the examples book of the OpenMP standard [47].

We are showing some selected examples in the following.

The following example (from [47]) shows how to copy input data (`v1` and `v2`) from the host to the device, execute the computation of a parallel for loop on the device and copy the output data (`p`) back to the host.

```
#pragma omp target map(to: v1, v2) map(from: p)
#pragma omp parallel for
for (i=0; i<N; i++)
    p[i] = v1[i] * v2[i];
```

The next example (from [48]) shown in Figure 10 demonstrates how to create a device data environment, allocate memory in this environment, copy input data (`input`) before the execution of any code from the host data environment to the device data environment and copy output data (`res`) back from the device data environment to the host data environment. Both for loops are executed on the device, while the routine `update_input_array_on_the_host()` is executed on the host and updates the array input.

The `#pragma omp target update` is used to synchronise the value of a variable (`input`) in the host data environment with a corresponding variable in a device data environment.

Figure 10. Example for a device data environment. From [48].

```
#pragma omp target data map(alloc:tmp[:N]) map(to:input[:N]) map(from:res)
  {
#pragma omp target
#pragma omp parallel for
    for (i=0; i<N; i++)
      tmp[i] = some_computation(input[i], i);

    update_input_array_on_the_host(input);

#pragma omp target update to(input[:N])

#pragma omp target
#pragma omp parallel for reduction(+:res)
    for (i=0; i<N; i++)
      res += final_computation(input[i], tmp[i], i)
  }
```

host
target
host
target
host

The next example (from [47]) shows how the `target teams` and the `distribute parallel loop SIMD` constructs are used to execute a loop in a target teams region. The `target teams` construct creates a league of teams where the master thread of each team executes the teams region. The `distribute parallel loop SIMD` construct schedules the loop iterations across the master thread of each team and then across the threads of each team where each thread uses SIMD parallelism.

```
#pragma omp target teams map(to: v1[0:N], v2[:N]) map(from: p[0:N])
#pragma omp distribute parallel for simd
for (i=0; i<N; i++)
```

```
p[i] = v1[i] * v2[i];
```

More examples will be added in the next version of this guide.

6.3.8. Runtime routines and environment variables

The following runtime support routines deal with devices.

void omp_set_default_device(int dev_num)	Controls the default target device.
int omp_get_default_device(void)	Returns the number of the default target device.
int omp_get_num_devices(void);	Returns the number of target devices.
int omp_get_num_teams(void)	Returns the number of teams in the current teams region, or 1 if called from outside a teams region.
int omp_get_team_num(void);	Returns the team number of the calling thread (0 … omp_get_num_teams() - 1).
int omp_is_initial_device(void);	Returns true if the current task is executing on the host device; otherwise, it returns false.
int omp_get_initial_device(void);	Returns a device number representing the host device.

Seven different memory routines not mentioned in the table were added in OpenMP 4.5 to allow explicit allocation, deallocation, memory transfers and memory associations.

The default device can be set through the environment variable **OMP_DEFAULT_DEVICE** as well. The value of this variable must be a non-negative integer value.

6.3.9. Best Practices

Many of the following best practice recommendations aiming for performence portability are cited from [49] and [51]:

1. Keep data resistant on the device, since the copying of data between the host and the device is a bottle nack.
2. Overlap data transfers with computations on the host or the device using asynchronous data transfers.
3. Prefer to include the most extensive combined construct relevant for the loop nest, i.e. `#pragma omp target teams distribute parallel for simd` (however not available in GCC 6.1).
4. Always include `parallel for`, and `teams` and `distribute`, even if the compiler does not require them.
5. Include the `simd` directive above the loop you require to be vectorised.
6. Neither `collapse` nor `schedule` should harm functional portability, but might inhibit performance portability.
7. Avoid setting `num_teams` and `thread_limit` since each compiler uses different schemes for scheduling teams to a device.

7. MPI

Details about using the Intel MPI library on Xeon Phi coprocessor systems can be found in [24].

7.1. Setting up the MPI environment

The following commands have to be executed to set up the MPI environment:

```
# copy MPI libraries and binaries to the card (as root)
# only copying really necessary files saves memory

scp /opt/intel/impi/4.1.0.024/mic/lib/* mic0:/lib
scp /opt/intel/impi/4.1.0.024/mic/bin/* mic0:/bin

# setup Intel compiler variables
. /opt/intel/composerxe/bin/compilervars.sh intel64

# setup Intel MPI variables
. /opt/intel/impi/4.1.0.024/bin64/mpivars.sh
```

The following network fabrics are available for the Intel Xeon Phi coprocessor:

Fabric Name	Description
shm	Shared-memory
tcp	TCP/IP-capable network fabrics, such as Ethernet and InfiniBand (through IPoIB)
ofa	OFA-capable network fabric including InfiniBand (through OFED verbs)
dapl	DAPL–capable network fabrics, such as InfiniBand, iWarp, Dolphin, and XPMEM (through DAPL)

The Intel MPI library tries to automatically use the best available network fabric detected (usually shm for intra-node communication and InfiniBand (dapl, ofa) for inter-node communication).

The default can be changed by setting the `I_MPI_FABRICS` environment variable to `I_MPI_FABRICS=<fabric>` or `I_MPI_FABRICS=<intra-node fabric>:<inter-nodes fabric>`. The availability is checked in the following order: shm:dapl, shm:ofa, shm:tcp.

7.2. MPI programming models

Intel MPI for the Xeon Phi coprocessors offers various MPI programming models:

Symmetric model The MPI ranks reside on both the host and the coprocessor. Most general MPI case.

Coprocessor-only model All MPI ranks reside only on the coprocessors.

Host-only model All MPI ranks reside on the host. The coprocessors can be used by using offload pragmas. (Using MPI calls inside offloaded code is not supported.)

7.2.1. Coprocessor-only model

To build and run an application in coprocessor-only mode, the following commands have to be executed:

```
# compile the program for the coprocessor (-mmic)

mpiicc -mmic -o test.MIC test.c

#copy the executable to the coprocessor
scp test.MIC mic0:/tmp
```

```
#set the I_MPI_MIC variable
export I_MPI_MIC=1

#launch MPI jobs on the coprocessor mic0 from the host
#(alternatively one can login to the coprocessor and run mpirun there)
mpirun -host mic0  -n 2 /tmp/test.MIC
```

7.2.2. Symmetric model

To build and run an application in symmetric mode, the following commands have to be executed:

```
# compile the program for the coprocessor (-mmic)
mpiicc -mmic -o test.MIC test.c

# compile the program for the host
mpiicc -mmic -o test test.c


#copy the executable to the coprocessor
scp test.MIC mic0:/tmp/test.MIC

#set the I_MPI_MIC variable
export I_MPI_MIC=1


#launch MPI jobs on the host knf1 and on the coprocessor mic0
mpirun -host knf1 -n 1 ./test : -n 1  -host mic0 /tmp/test.MIC
```

7.2.3. Host-only model

To build and run an application in host-only mode, the following commands have to be executed:

```
# compile the program for the host,
# mind that offloading is enabled per default
mpiicc -o test test.c

# launch MPI jobs on the host knf1, the MPI process will offload code
# for acceleration
mpirun -host knf1 -n 1 ./test
```

7.3. Simplifying launching of MPI jobs

Instead of specifying the hosts and coprocessors via `-n hostname` one can also put the names into a hostfile and launch the jobs via

```
mpirun -f hostfile -n 4 ./test
```

Mind that the executable must have the same name on the hosts and the coprocessors in this case.

If one sets

```
export I_MPI_POSTFIX=.mic
```

the `.mix` postfix is automatically added to the executable name by mpirun, so in the case of the example above `test` is launched on the host and `test.mic` on the coprocessors. It is also possible to specify a prefix using

```
export I_MPI_PREFIX=./MIC/
```

In this case `./MIC/test` will be launched on the coprocessor. This is specially useful if the host and the coprocessors share the same NFS filesystem.

8. Intel MKL (Math Kernel Library)

8.1. Introduction

Intel provides support for automatic usage of the Intel Xeon Phi co-processor by means of automatic offloading of work using MKL routines. This is a very simple way to exploit the MIC co-processor. No recompilation is necessary. Presently, only a limited set of MKL functions and routines have been offload enabled.

Usage is very simple as will be shown is the following Fortran-90 example:

```
time_start=mysecond()
call dgemm('n', 'n', N, N, N, alpha, a, N, b, N, beta, c, N)
time_end=mysecond()
write(*,fmt=form) &
"dgemm end, timing :",time_end-time_start," secs, ",&
ops*1.0e-9/(time_end-time_start)," Gflops/s"
```

This f90 code is all it takes to do A*B => C . All the magic is done by MKL behind the scenes. Compiling is equally simple:

```
ifort -o dgemm-test.x -mcmodel=medium -O3 -openmp -mkl dgemm-test.f90
```

The Intel Xeon Phi coprocessor is supported since MKL 11.0. Details on using MKL with Intel Xeon Phi co-processors can be found in the Intel Parallel Studio documentation. Also the MKL developer zone [11] contains useful information. All functions can be used on the Xeon Phi, however the optimization level for wider 512-bit SIMD instructions differs.

As of Intel MKL 11.0 Update 2 the following functions are highly optimized for the Intel Xeon Phi coprocessor:

- BLAS Level 3, and much of Level 1 & 2
- Sparse BLAS: ?CSRMV, ?CSRMM
- Some important LAPACK routines (LU, QR, Cholesky)
- Fast Fourier Transformations
- Vector Math Library
- Random number generators in the Vector Statistical Library

Intel plans to optimize a wider range of functions in future MKL releases.

8.2. MKL usage modes

The following 3 usage models of MKL are available for the Xeon Phi:

1. Automatic Offload

2. Compiler Assisted Offload

3. Native Execution

8.2.1. Automatic Offload (AO)

In the case of automatic offload the user does not have to change the code at all. For automatic offload enabled functions the runtime may automatically download data to the Xeon Phi coprocessor and execute (all or part of) the computations there. The data transfer and the execution management is completely automatic and transparent for the user.

As of Intel MKL 11.0.2 only the following functions are enabled for automatic offload:

- Level-3 BLAS functions
  - ?GEMM (for m,n > 2048, k > 256)
  - ?TRSM (for M,N > 3072)
  - ?TRMM (for M,N > 3072)
  - ?SYMM (for M,N > 2048)
- LAPACK functions
  - LU (M,N > 8192)
  - QR
  - Cholesky

In the above list also the matrix sizes for which MKL decides to offload the computation are given in brackets. For most up to date information about enabled MKL functions, see the Intel Parallel Studio documentation.

To enable automatic offload either the function `mkl_mic_enable()` has to be called within the source code or the environment variable `MKL_MIC_ENABLE=1` has to be set. If no Xeon Phi coprocessor is detected the application runs on the host without penalty.

To build a program for automatic offload, the same way of building code as on the Xeon host is used:

```
icc -O3 -mkl file.c -o file
```

By default, the MKL library decides when to offload and also tries to determine the optimal work division between the host and the targets (MKL can take advantage of multiple coprocessors). In case of the BLAS routines the user can specify the work division between the host and the coprocessor by calling the routine

```
mkl_mic_set_Workdivision(MKL_TARGET_MIC,0,0.5)
```

or by setting the environment variable

```
MKL_MIC_0_WORKDIVISION=0.5
```

Both examples specify to offload 50% of computation only to the 1st card (card #0).

8.2.2. Compiler Assisted Offload (CAO)

In this mode of MKL the offloading is explicitly controlled by compiler pragmas or directives. In contrast to the automatic offload mode, all MKL function can be offloaded in CAO-mode.

A big advantage of this mode is that it allows for data persistence on the device.

For Intel compilers it is possible to use AO and CAO in the same program, however the work division must be explicitly set for AO in this case. Otherwise, all MKL AO calls are executed on the host.

MKL functions are offloaded in the same way as any other offloaded function (see section Section 5). An example for offloading MKL's sgemm routine looks as follows:

```
#pragma offload target(mic) \
  in(transa, transb, N, alpha, beta) \
  in(A:length(N*N)) \
  in(B:length(N*N)) \
  in(C:length(N*N)) \
  out(C:length(N*N) alloc_if(0)) {

     sgemm(&transa, &transb, &N, &N, &N, &alpha, A, &N, B, &N, &beta, C, &N);

  }
```

To build a program for compiler assisted offload, the following command is recommended by Intel:

```
icc –O3 -openmp -mkl \
–offload-option,mic,ld, “-L$MKLROOT/lib/mic -Wl,\
--start-group -lmkl_intel_lp64 -lmkl_intel_thread \
-lmkl_core -Wl,--end-group” file.c –o file
```

To avoid using the OS core, it is recommended to use the following environment setting (in case of a 61-core coprocessor):

```
MIC_KMP_AFFINITY=explicit,granularity=fine,proclist=[1-240:1]
```

Setting larger pages by the environment setting `MIC_USE_2MB_BUFFERS=16K` usually increases performance. It is also recommended to exploit data persistence with CAO.

Table 1. Relevant environment flags

Environment variable	Function
MKL_MIC_ENABLE	Enables Automatic Offload
OFFLOAD_DEVICES	List offload devices
MKL_MIC_WORKDIVISION	Specifies the fraction of work to do on all the Intel Xeon Phi co-processors on the system, including autoa
OFFLOAD_REPORT	Specifies the profiling report level for any offload
MIC_LD_LIBRARY_PATH	Search path for coprocessor-side dynamic libraries

8.2.3. Native Execution

In this mode of MKL the Intel Xeon Phi coprocessor is used as an independent compute node.

To build a program for native mode, the following compiler settings should be used:

```
icc -O3 -mkl -mmic file.c -o file
```

The binary must then be manually copied to the coprocessor via ssh and directly started on the coprocessor.

8.3. Example code

Example code can be found under `$MKLROOT/examples/mic_ao` and `$MKLROOT/examples/mic_offload` .

8.4. Intel Math Kernel Library Link Line Advisor

To determine the appropriate link line for MKL the Intel Math Kernel Library Link Line Advisor available under [12] has been extended to include support for the Intel Xeon Phi specific options.

9. TBB: Intel Threading Building Blocks

The Intel TBB library is a template based runtime library for C++ code using threads that allows us to fully utilize the scaling capabilities within our code by increasing the number of threads and supporting task oriented load balancing.

Intel TBB is open source and available on many different platforms with most operating systems and processors. It is already popular in the C++ community. You should be able to use it with any compiler supporting ISO C++. So this is one of the advantages of Intel TBB when you intend to keep your code as easily portable as possible.

Typically as a rule of thumb an application must scale well past one hundred threads on Intel Xeon processors to profit from the possible higher parallel performance offered with e.g. the Intel Xeon Phi coprocessor. To check if the scaling would profit from utilising the highly parallel capabilities of the MIC architecture, you should start to create a simple performance graph with a varying number of threads (from one up to the number of cores).

From a programming standpoint we treat the coprocessor as a 64-bit x86 SMP-on-a-chip with an high-speed bi-directional ring interconnect, (up to) four hardware threads per core and 512-bit SIMD instructions. With the available number of cores we have easily 200 hardware threads at hand on a single coprocessor. The multi-threading on each core is primarily used to hide latencies that come implicitly with an in-order microarchitecture. Unlike hyper-threading these hardware threads cannot be switched off and should never be ignored. Generally it should be impossible for a single thread per core to approach the memory or floating point capability limit. Highly tuned codesnippets may reach saturation already at two threads, but in general a minimum of three or four active threads per cores will be needed. This is one of the reasons why the number of threads per core should be parameterized as well as the number of cores. The other reason is of course to be future compatible.

TBB offers programming methods that support creating this many threads in a program. In the easiest way the one main production loop is transformed by adding a single directive or pragma enabling the code for many threads. The chunk size used is chosen automatically.

The new Intel Cilk Plus which offers support for a simpler set of tasking capabilities fully interoperates with Intel TBB. Apart from that Intel Cilk Plus also supports vectorization. So shared memory programmers have Intel TBB and Intel Cilk Plus to assist them with built-in tasking models. Intel Cilk Plus extends Intel TBB to offer C programmers a solution as well as help with vectorization in C and C++ programs.

Intel TBB itself does not offer any explicit vectorization support. However it does not interfere with any vectorization solution either.

In relevance to the Intel Xeon Phi coprocessor TBB is just one available runtime-based parallel programming model alongside OpenMP, Intel Cilk Plus and pthreads that are also already available on the host system. Any code running natively on the coprocessor can put them to use just like it would on the host with the only difference being the larger number of threads.

9.1. Advantages of TBB

There exists a variety of approaches to parallel programming, but there are several advantages to using Intel TBB when writing scalable applications:

- TBB relies on generic programming: Writing the best possible algorithms with the fewest possible constraints enables to deliver high performance algorithms which can be applied in a broader context. Other more traditional libraries specify interfaces in terms of particular types or base classes. Intel TBB specifies the requirements on the types instead and in this way keeps the algorithms themselves generic and easily adaptable to different data representations.

- It is easy to start: You don't have to be a threading expert to leverage multi-core performance with the help of TBB. Normally you can successfully thread some programs just by adding a single directive or pragma to the main production loop.

- It obeys to logical parallelism: Since with TBB you specify tasks instead of threads, you automatically produce more portable code which emphasizes scalable, data parallel programming. You are not bound to platform-dependent threading primitives; most threading packages require you to directly code on low-level constructs close to the hardware. Direct programming on raw thread level is error-prone, tedious and typically hard work since it forces you to efficiently map logical tasks into threads and it is not always leading to the desired results. With the higher level of data-parallel programming on the other hand, where you have multiple threads working on different parts of a collection, performance continues to increase as you add more cores since for a larger number of processors the collections are just divided into smaller chunks. This is a great feature when it comes to portability.

- TBB is compatible with other programming models: Since the library is not designed to address all kinds of threading problems, it can coexist seamlessly with other threading packages.

- The template-based approach allows Intel TBB to make no excuses for performance. Other general-purpose threading packages tend to be low-level tools that are still far from the actual solution, while at the same time supporting many different kinds of threading. In TBB every template solves a computationally intensive problem in a generic, simple way instead.

All of these advantages make TBB popular and easily portable while at the same time facilitating data parallel programming.

Further advanced concepts in TBB that are not MIC specific can be found in the Intel TBB User Guide or in the Reference Manual, both available under [17].

9.2. Using TBB natively

Code that runs natively on the Intel Xeon Phi coprocessor can apply the TBB parallel programming model just as they would on the host, with no unusual complications beyond the larger number of threads.

In order to initialize your compiler environment variables needed to set up TBB correctly, typically the `/opt/intel/composerxe/tbb/bin/tbbvars.csh` or `tbbvars.sh` script with `intel64` as the argument is called by the `/opt/intel/composerxe/bin/compilervars.csh` or `compilervars.sh` script with `intel64` as argument. (e.g. `source /opt/intel/composerxe/bin/compilervars.sh intel64`)

Normally there is no need to call the `tbbvars` script directly and it is not advisable either since the `compilervars` script also calls other subscripts taking i.e. care of the debugger or Intel MKL and running the subscripts out of order might result in unpredictable behavior.

A minimal C++ TBB example looks as follows:

```
#include “tbb/task_scheduler_init.h”
#include “tbb/parallel_for.h”
#include "tbb/blocked_range.h"

using namespace tbb;

int main() {

  task_scheduler_init init;
```

```
  return 0;
}
```

The `using` directive imports the namespace `tbb` where all of the library's classes and functions are found. The namespace is explicit in the first mention of a component, but implicit afterwards. So with the `using namespace` statement present you can use the library component identifiers without having to write out the namespace prefix `tbb` before each of them.

The task scheduler is initialized by instantiating a `task_scheduler_init` object in the main function. The definition for the `task_scheduler_init` class is included from the corresponding header file. Actually any thread using one of the provided TBB template algorithms must have such an initialized `task_scheduler_init` object. The default constructor for the `task_scheduler_init` object informs the task scheduler that the thread is participating in task execution, and the destructor informs the scheduler that the thread no longer needs the scheduler. With the newer versions of Intel TBB as used in a MIC environment the task scheduler is automatically initialized, so there is no need to explicitely initialize it if you don't need to have control over when the task scheduler is constructed or destroyed. When initializing it you also have the further possibility to tell the task scheduler explicitly how many worker threads there are to be used and what their stack size would be.

In the simplest form scalable parallelism can be achieved by parallelizing a loop of iterations that can each run independently from each other.

The `parallel_for` template function replaces a serial loop where it is safe to process each element concurrently.

A typical example would be to apply a function `Foo` on all elements of an array over the iterations space of type `size_t` going from 0 to n-1:

```
void SerialApplyFoo( float a[], size_t n ) {
  for( size_t i=0; i!=n; ++i )
    Foo(a[i]);
}
```

becomes

```
void ParallelApplyFoo( float a[], size_t n) {
  parallel_for(size_t(0), n, [=](size_t i) {Foo(a[i]);});
}
```

This is the TBB short form of a `parallel_for` over a loop based on a one-dimensional iteration space consisting of a consecutive range of integers (which is one of the most common cases). The expression `parallel_for(first,last,step,f)` is synonymous to `for(auto i=first; i!=last; i+=step) f(i)` except that each `f(i)` can be evaluated in parallel if resources permit. The omitted step parameter is optional. The short form implicitly uses automatic chunking.

The long form would be:

```
void ParallelApplyFoo( float* a, size_t n ) {
  parallel_for( blocked_range<size_t>(0,n),
  [=](const blocked_range<size_t>& r) {
    for(size_t i=r.begin(); i!=r.end(); ++i)
      Foo(a[i]);
  });
}
```

Here the key feature of the TBB library is more clearly revealed. The template function `tbb::parallel_for` breaks the iteration space into chunks, and runs each chunk on a separate thread. The first parameter of tem-

plate function call `parallel_for` is a `blocked_range` object that describes the entire iteration space from 0 to n-1. The `parallel_for` divides the iteration space into subspaces for each of the over 200 hardware threads. `blocked_range<T>`is a template class provided by the TBB library describing a one-dimensional iteration space over type `T`. The `parallel_for` class works just as well with other kinds of iteration spaces. The library provides `blocked_range2d` for two-dimensional spaces. There exists also the possibility to define own spaces. The general constructor of the `blocked_range` template class is `blocked_range<T>(begin,end,grainsize)`. The `T` specifies the value type. `begin` represents the lower bound of the half-open range interval `[begin,end)` representing the iteration space. `end` represents the excluded upper bound of this range. The `grainsize` is the approximate number of elements per sub-range. The default `grainsize` is 1.

A parallel loop construct introduces overhead cost for every chunk of work that it schedules. The MIC adapted Intel TBB library chooses chunk sizes automatically, depending upon load balancing needs. The heuristic normally works well with the default `grainsize`. It attempts to limit overhead cost while still providing ample opportunities for load balancing. For most use cases automatic chunking is the recommended choice. There might be situations though where controlling the chunk size more precisely might yield better performance.

When compiling programs that employ TBB constructs, be sure to link in the Intel TBB shared library with `–ltbb`. If you don't undefined references will occur.

```
icc -mmic –ltbb foo.cpp
```

Afterwards you can use `scp` to upload the binary and any shared libraries required by your application to the coprocessor. On the coprocessor you can then export the library path and run the application.

9.3. Offloading TBB

The Intel TBB header files are not available on the Intel MIC target environment by default (the same is also true for Intel Cilk Plus). To make them available on the coprocessor the header files have to be wrapped with `#pragma offload` directives as demonstrated in the example below:

```
#pragma offload_attribute (push,target(mic))
#include “tbb/task_scheduler_init.h”
#include “tbb/parallel_for.h”
#include "tbb/blocked_range.h"
#pragma offload_attribute (pop)
```

Functions called from within the offloaded construct and global data required on the Intel Xeon Phi coprocessor should be appended by the special function attribute `__attribute__((target(mic)))`.

Codes using Intel TBB with an offload should be compiled with `-tbb` flag instead of `-ltbb`.

9.4. Examples

The TBB library provides support for parallel algorithms, concurrent containers, tasks, synchronization, memory allocation and thread control. Amongst it, the templates cover well-known parallel-programming patterns like parallel loops or reduction operations, task-based constructs or pipelines. An easy and flexible formulation of parallel computations is possible. The programmer's task is mainly to expose a high degree of parallelism to the processor, to allow or support the vectorization of calculations, and to maximize the cache-locality of algorithms. Principles of vectorization and striving for implementations with high cache-locality are general topics of performance optimization.

9.4.1. Exposing parallelism to the processor

The concrete parallelization of a certain algorithm is of course dependent on the specific floating-point operations used, its memory access pattern, the possible decoupling into parallel operations resp. vice-versa the dependencies

between the calculations. However, some estimation can be made what degree the needed parallelism should have in order to achieve efficient execution on the coprocessor.

Xeon Phi processors can execute 4 hyperthreads on each core. That results in 240 threads active at the same time on the 60 cores of the coprocessor. A rule of thumb is that one should have about 10 computational tasks per thread in order to compensate delays from load imbalances. That results in 2400 parallel tasks. Efficient calculations have to apply the vector units that can process 16 single precision floating-point numbers in one instruction. Therefore, we would need several ten thousand operations that can be executed independently from each other in order to keep the processor busy.

On the other hand, good load balancing or the placement of several MPI tasks on the Xeon Phi can lower the needed degree of parallelism. Considerations also have to be made if it would really increase the performance to use all 4 hyperthreads of a core. Those hyperthreads share certain hardware resources, and depending on the algorithm, competition for the resources can occur causing an increase of the overall execution speed.

9.4.1.1. Parallelisation with the task construct

This example shows how one can implement a multi-threaded, task-oriented algorithm. Nevertheless, there is no need to deal with the subtleties of thread creation and their control for the implementer of the algorithm.

The class `tbb:task` represents a low-level task with low overhead. It will be used by higher-level constructs like `parallel_for` or `task_group`.

Our example shows the use of task groups in a walkthrough through a binary tree. The sum of values stored in each node will be calculated during this walkthrough.

The tree nodes are defined as follows:

```
struct tree {
    struct tree *l, *r;  // left and right subtree
    long depth;          // depth of the tree starting from here
    float v;             // an important value
};
```

The serial implementation could be written as:

```
float tree_sum( struct tree *t )
{
    float mysum = t->v;

    if ( t->l != NULL ) mysum += tree_sum( t->l );
    if ( t->r != NULL ) mysum += tree_sum( t->r );

    return mysum;
}
```

A TTB task that performs the same computation is given with the following class `ParallelSumTask`. It must contain a method `exec()`. This method does either a serial computation of the value sum if its subtree has a depth lower than 10 levels or creates two subtasks that calculate the value sums of the left resp. right child tree. These tasks will be added to the task list of the active scheduler. After their termination will the current task add the value sums of both child trees and their own value and terminate itself.

```
class ParallelSumTask : public tbb::task {
    float       *mysum;
    struct tree *myr;

public:
```

```
    ParallelSumTask( TreeNode *r, float *sum ) : myr(r), mysum(sum) {}

    task *execute() {
        if( root->depth < 10 ) {
            *mysum = tree_sum(myr);
        }
        else {
            float sum_l, sum_r;
            tbb::task_list tlist;

            // Create subtasks for processing of left and right subtree.
            int count = 1;
            if( myr->l ) {
                ++count;
                tlist.push_back( *new(allocate_child())
                                  ParallelSumTask(myr->l, &sum_l));
            }
            if( myr->r ) {
                ++count;
                list.push_back( *new(allocate_child())
                                    ParallelSumTask(myr->r, &sum_r));
            }
            // Start task processing
            set_ref_count(count);
            spawn_and_wait_for_all(list);

            // Add own value and sums of left and right subtree.
            *mysum = myr->v;
            if( myr->l ) *mysum += sum_l;
            if( myr->r ) *mysum += sum_r;
        }
        return NULL;
    }
};
```

The calculation of the value sum could be started in the following way:

```
float value_sum;
struct tree *tree_root;

// ... Initialize tree ...

// Initialize a task scheduler with the maximum number of threads.
tbb::task_scheduler_init init( 240 );
// Perform the calculation.
ParallelSumTask *a = new(tbb::task::allocate_root())
                           ParallelSumTask(tree_root, &value_sum);
tbb::task::spawn_root_and_wait*(a);
// The sum of all values stored in the tree nodes
// has been calculated in variable "value_sum"
```

9.4.1.2. Loop parallelisation

TBB provides several algorithms that can be used for the parallelisation of loops, again without forcing the implementer do deal with the details of low-level threads. The following example provides the details how a loop can be parallelized with the `parallel_for` template. The implemented algorithm shall calculate the sum of the vector elements.

The serial implementation could be written as follows:

```
float serial_vecsum( float *vec, int len) {
   float sum = 0;

   for (int i = 0; i < len; ++i)
      sum += vec[i];
   return sum;
}
```

The parallel algorithm shall compute the partial sums of partial vectors. The overall sum of all partial vectors will be computed as sum of these partial sums.

The TBB template `parallel_reduce` provides the basic implementation of subdividing a range to iterate over into several subranges. The subranges will be assigned to different threads that perform the iterations on them. The program has to provide a "loop body" that that will be applied to each array element as well as a function that processes the results of the iterations over the subranges of the vector.

These functionality for the vector summation will be provided in the class `adder`. The transformation function working on the vector elements has to be provided as `operator()` method. The function that reduces the results of the parallel work on different subvectors has to be implemented as method `join()`, which takes a reference to another instance of the adder class as argument. Finally, we have to equip the adder class with a copy constructor because each thread will get one copy of the originally provided adder instance.

```
class adder {
    float *myvec // pointer to array with values to sum.

public:
    float sum;

    adder ( float *a ): myvec(a), sum(0) { }

    adder ( adder &a, split ): myvec(a.myvec, su (0) { }

    void operator()( const tbb::blocked_range<float> &r ) {
        for( float i = r.begin(); i != r.end( ); ++i )
            sum += myvec[i];
    }

    void join( const adder &other ) {
        sum += other.sum;
    }
};
```

The listing shows how the partial sum will be calculated by iterating over the elements of the subvector in `operator()`. The reduction of two partial sums to one value is implemented in `join()`.

The summation of all array elements can be performed with the following code piece:

```
float *vec; // Vector data, initialise them.
int veclen; // ...

adder vec_adder(vec); // Create root object for decomposition.
parallel_reduce(tbb::blocked_range<int>(0, size, 5), vec_adder);
   // Define range and grainsize for iteration.
// vec_adder.sum contains the sum of all vector elements.
```

9.4.2. Vectorization and Cache-Locality

Vectorization and cache-locality should be used in order to increase the program efficiency. Many details and examples are given in the programming and compilation guide of the MIC architecture [28].

Vectorization can be achieved by auto vectorization and checked with the vectorisation report. There is the possibility to apply user mandated vectorization for constructs that are not covered by automatic vectorization. Users of Cilk Plus also have an option to use extensions for array notation.

9.4.3. Work-stealing versus Work-sharing

Using work-stealing for task scheduling means that idle threads will take over tasks from other busy threads. This approach has been mentioned already above when it was said that there should be a certain amount of tasks per thread available in order to compensate load imbalances. The scheduler could keep idle processors busy if sufficient many tasks are available.

Work-sharing as method of the task scheduling is worthwhile for well-balanced workloads. Work-sharing is typically implemented as task pool and achieves near optimal occupancy for balanced workload.

10. IPP: The Intel Integrated Performance Primitives

Remark. This section gives a general overview about IPP based on version 8.0 (which did not support IPP, yet). Meanwhile, Intel® IPP libraries along with IA-32 and Intel® 64 versions contain native Intel® MIC architecture libraries, which allow to build the applications running totally (natively) or particularly (offload) on Intel® MIC Architecture. Examples are available in the file linux/ipp/examples/ipp_examples_mic.tgz (components_and_examples_mic.tgz) delivered with the Intel Parallel Studio 2016 (2017) compiler suite. This tgz File also contains useful documentation for using IPP on Xeon Phi (documentation/ipp-examples.htm).

10.1. Overview of IPP

Intel Integrated Performance Primitives (IPP) is a software library, which provides a large collection of functions for signal processing and multimedia processing. It is useable on MS-Windows, Linux, and Mac OS X platforms.

The covered topics are

- Signal processing, including
  - FFT, FIR, Transformations, Convolutions etc
- Image & frame processing, including
  - Transformations, filters, color manipulation
- General functionality, including
  - Statistical functions
  - Vector, matrix and linear algebra for small matrices.

The functions in this library are optimised to use advanced features of the processors like SSE and AVX instruction sets. Many functions are internally parallelised using OpenMP.

Practical aspects. IPP requires support of Streaming SIMD extensions (SSE) or Advanced Vector Extensions (AVX). The validated operating systems include newer versions of MS Windows (Windows 2003, Windows 7, Windows 2008), Redhat Enterpreise Linux (RHEL) 5 and 6, and Mac OS X 10.6 or higher. More detailed information can be found in the product documentation.

The library is compatible to Intel, Microsoft, GNU and cctools compilers. It contains freely distributable runtime libraries in order to allow the execution of programs for users without having the need to install the development tools.

10.2. Using IPP

10.2.1. Getting Started

The following example has been taken from the IPP User's Guide.

```
#include "ipp.h"
#include <stdio.h>

int main(int argc, char* argv[])
{

    const IppLibraryVersion *lib;
    Ipp64u fm;

    ippInit();                  //Automatic best static dispatch
    //ippInitCpu(ippCpuSSSE3); //Select a specific static library level
                               //Can be useful for debugging/profiling
    // Get version info
    lib = ippiGetLibVersion();
    printf(%s %s\n, lib->Name, lib->Version);
    //Get CPU features enabled with selected library level
    fm = ippGetEnabledCpuFeatures();
    printf(SSE2  %c\n,(fm>>2)&1? 'Y' : 'N');
    printf(SSE3  %c\n,(fm>>3)&1? 'Y' : 'N');
    printf(SSSE3 %c\n,(fm>>4)&1? 'Y' : 'N');
    printf(SSE41 %c\n,(fm>>6)&1? 'Y' : 'N');
    printf(SSE42 %c\n,(fm>>7)&1? 'Y' : 'N');
    printf(AVX   %c OS Enabled %c\n,
           (fm>>8)&1 ? 'Y' : 'N', (fm>>9)&1 ? 'Y' : 'N');
    printf(AES   %c CLMUL      %c\n,
           (fm>>10)&1 ? 'Y' : 'N', (fm>>11)&1 ? 'Y': 'N');

    return 0;
}
```

This program contains three steps. The initialisation with `ippInit()` makes sure that the best possible implementation of IPP functions will be used during the program execution. The next step provides nformation about the used library version, and finally will be the enabled hardware features listed.

Building of the program. The first step to build the program is to provide the correct environment settings for the compiler. Intel's default solution is a shellscript compilervars.sh in the bin directory of the installation that should be executed. Some other software like the modules environment is available on some HPC computer systems too. Please refer to the locally available documentation.

The program, saved in the file ipptest.cpp can be compiled and linked with the following command line:

```
icc ipptest.cpp -o ipptest \
     -I$IPPROOT/include -L$IPPROOT/lib/intel4 \
     -lippi -lipps -lippcore
```

The executable can be started with the following command:

```
./ipptest
```

10.2.2. Linking of Applications

IPP contains different implementations of each function that provide the best performance on different processor architectures. They will be selected by means of dispatching while the programmer uses always the same API.

Dynamic Linking. Dynamic linking of IPP can be achieved by using Intel's compiler switch `-ipp` or by linking with the default libraries that have names that do not end in `_l` resp. `_t`. The dynamic libraries use internally OpenMP in order to achive multi-threaded execution. The multithreading within the IPP routines can be turned off by calling the function `ippSetNumThreads(1)`. Other options are static single-threaded linking or to build a single-threaded shared library from the static single-threaded libraries.

Static Linking. The libraries with names ending in `_l` must be used for static single- threaded linking, while the libraries with names ending in `_t` provide will provide multi-threaded implementations based again on OpenMP.

More information. The exact choice of the linking model depends on several factors like intended processor architectures for the execution, useable memory size during the execution, installation aspects and more. A white paper available online focuses on such aspects and provides an in-depth discussion of the topic.

10.3. Multithreading

The use of treads within the IPP should also be taken into consideration by the application developer.

- IPP uses OpenMP in order to achieve internally a multi-threaded execution as mentioned in the section about linking. IPP uses processors up to the minimum of `$OMP_NUM_THREADS` and the number of available processors. Another possibility to get and set the number of used processors are the functions `ippSetNumThreads(int n)` and `ippGetNumThreads()`.

- Some functions (for example FFT) are designed to use two threads that should be mapped onto the same die in order to use a shared L2 cache if available. The user should set the following environment variable when using processors with more than two cores per die in order to ensure best performance: `KMP_AFFINITY=compact`.

- There is a risk of thread oversubscription and performance degradation in the case that an application uses OpenMP additionally. The use of threads within IPP can be turned off by calling `ippSetNumThreads(1)`. However, some OpenMP-related functionality could be active regardless of that. Therefore, single-threaded execution can be achieved best by using the single-threaded libraries.

11. Further programming models

The programming models OpenCL and OpenACC have become popular to program GPGPUs and have also been enabled for the Intel MIC architecture.

11.1. OpenCL

OpenCL (Open Computing Language) [32] is the first open, royalty-free standard for cross-platform, parallel programming of modern processors found in personal computers, servers and handheld/embedded devices. OpenCL is maintained by the non-profit technology consortium Khronos Group. It has been adopted by Apple, Intel, Qualcomm, Advanced Micro Devices (AMD), Nvidia, Altera, Samsung, Vivante and ARM Holdings. OpenCL 2.0 is the latest significant evolution of the OpenCL standard, designed to further simplify cross-platform programming, while enabling a rich range of algorithms and programming patterns to be easily accelerated.

OpenCL views a computing system as consisting of a number of compute devices. The compute devices can be either CPUs or accelerators, such as graphics processing units (GPUs), or the many-core Xeon Phi, attached to a CPU. OpenCL defines a C-like language for writing programs, and the functions that are executed on an OpenCL device are called "kernels". Besides the C-like programming language, OpenCL defines an application

programming interface (API) that allows host programs to launch compute kernels on the OpenCL devices and manage device memory. Programs in the OpenCL language are intended to be compiled at run-time (although it's not strictly required), so that OpenCL-using applications are portable between implementations for various host devices.

The main competitor of OpenCL is NVIDIA's CUDA. However, CUDA can run only on NVIDIA devices, so OpenCL's main power is it's versatility. Another major advantage of OpenCL over CUDA is that it can harness all resources in a computing system for a given task, hence enabling heterogeneous computing. In the case of a Xeon+Xeon Phi system, OpenCL can make use of both the Xeon host as well as the Xeon Phi accelerator. The Intel Xeon Phi co-processor supports OpenCL 1.2.

While OpenCL is a portable programming model, the performance portability is not guaranteed. Traditional GPUs and the Intel Xeon Phi coprocessor have different HW designs. Their differences are such that they benefit from different application optimizations. For example, traditional GPUs rely on the existence of fast shared local memory, which the programmer needs to program explicitly. Intel Xeon Phi coprocessor includes fully coherent cache hierarchy, similar to regular CPU caches, which automatically speed up memory accesses. Another example: while some traditional GPUs are based on HW scheduling of many tiny threads, Intel Xeon Phi coprocessors rely on the device OS to schedule medium size threads. These and other differences suggest that applications usually benefit from tuning to the HW they’re intended to run on.

In the following example we show a naive implementation of matrix-matrix multiplication on the Xeon Phi using OpenCL. The following kernel uses only global memory, and hence is not performance-optimized.

```
__kernel void matMatMultKern(__global double *output,__global double *d_MatA,
__global double *d_MatB, __global int* d_rows, __global int* d_cols)
{
    int globalIdx = get_global_id(0);
    int globalIdy = get_global_id(1);
    double sum =0.0;
    int i;
    double tmpA,tmpB;
    for ( i=0; i < (*d_rows); i++)
    {
      tmpA = d_MatA[globalIdy * (*d_rows) + i];
      tmpB = d_MatB[i * (*d_cols) + globalIdx];
      sum += tmpA * tmpB;
    }
    output[globalIdy * (*d_rows) + globalIdx] = sum;
}
```

Tools for OpenCL on the Xeon Phi

To run OpenCL applications on the Phi, there are two additional toolsets required. The first necessary set to actually execute OpenCL applications is the Intel SDK for OpenCL. To compile OpenCL applications for Intel, the Intel Parallel Studio XE is also required. It is possible to compile OpenCL applications simply with G++, but OpenCL headers are only available as part of the Intel Parallel Studio XE.

To optimize OpenCL kernels, the following tools are recommended:

- Intel VTune Amplifier XE 2013, which enables you to fine-tune for optimal performance, ensuring the CPU or coprocessor device facilities are fully utilized.
- OpenCL Code Builder - Offline Compiler command-line tool, which offers full offline OpenCL language compilation, including an OpenCL syntax checker, cross hardware compilation support, Low-Level Virtual Machine (LLVM) viewer, Assembly language viewer, and intermediate program binaries generator.
- OpenCL Code Builder - Kernel Builder, which enables you to build and analyze your OpenCL kernels. The tool provides full offline OpenCL language compilation.

- OpenCL Code Builder – Debugger, which enables you to debug OpenCL kernels with the GNU Project Debugger (GDB).
- OpenCL Code Builder - Offline Compiler and Debugger plug-ins for Microsoft Visual Studio IDE.

A coding guide for developing OpenCL applications for the Intel Xeon Phi coprocessor can be found in [33].

11.2. OpenACC

The OpenACC Application Program Interface [34] describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs and accelerators. OpenACC is designed for portability across a wide range of accelerators and coprocessors, including APUs, GPUs, many-core and multi-core implementations. The standard was originally developed by Cray, CAPS, Nvidia and PGI. Currently OpenACC has many member and supporter organizations. See http://www.openacc.org/About_OpenACC [http://www.openacc.org/About_OpenACC].

Currently there is no OpenACC support for Intel Xeon Phi corpocessors. Intel Xeon Phi support was provided by the French company CAPS, which does not exist anymore. Based on the directive-based OpenACC and OpenHMPP standards, CAPS compilers enabled developers to incrementally build portable applications for various many-core systems such as NVIDIA and AMD GPUs, and Intel Xeon Phi. Currently, there is no OpenACC implementation available. However, there are plans that the PGI compiler suite will support Knights Landing in future.

12. Debugging

Information about debugging on Intel Xeon Phi coprocessors can be found in [25].

The GNU debugger (gdb) has been enabled by Intel to support the Intel Xeon Phi coprocessor. The debugger is now part of the recent MPSS release and does not have to be downloaded separately any more.

There are 2 different modes of debugging supported: native debugging on the coprocessor or remote cross-debugging on the host.

12.1. Native debugging with gdb

- Run gdb on the coprocessor

```
ssh -t mic0 /usr/bin/gdb
```

- One can then attach to a running application with process ID pid via

```
(gdb) attach pid
```

- or alternatively start an application from within gdb via

```
(gdb) file /path/to/application
(gdb) start
```

12.2. Remote debugging with gdb

- Run the special gdb version with Xeon Phi support on the host

```
/usr/linux-k1om-4.7/bin/x86_64-k1om-linux-gdb
```

- Start the gdbserver on the coprocessor by typing on the host gdb

```
(gdb) target extended-remote| ssh -T mic0 gdbserver -multi -
```

- Attach to a remotely running application with the remote process ID pid

```
(gdb) file /local/path/to/application
(gdb) attach pid
```

- It is also possible to run an application directly from the host gdb

```
(gdb) file /local/path/to/application
(gdb) set remote exec-file /remote/path/to/application
```

12.3. Debugging with Totalview

It is possible to debug applications running natively on the Intel Xeon Phi coprocessor. As with the offload directives mode, TotalView launches its debug server on the coprocessor that will start the remote graphic interface.

12.3.1. Using a Standard Single Server Launch string

This method is useful when the filesystem where TotalView is installed is shared between host and Xeon Phi. If the filesystem is not shared with the coprocessor, you can copy the "tvdsvrmain_mic" binary in /tmp on both the host and coprocessor/s:

```
cp /path/to/totalview/tvdsvrmain_mic /tmp
scp /path/to/totalview/tvdsvrmain_mic $HOSTNAME-mic0:/tmp
```

and then modify the launch string (File, Preferences, Launch Strings, Command) from the default to:

```
%C %R -n "%B/tvdsvr%K -working_directory %D -callback %L -set_pw %P %F"
```

Afterwards, you can open Totalview as:

```
totalview -r $HOSTNAME-mic0 /path/to/binary.mic
```

And proceed with the debugging of "binary.mic" from Totalview, the same way as running in local (Go, Halt, Step, Breakpoint, etc.)

12.3.2. Using the Xeon Phi Native Launch string

This method is useful in two primary use cases:

- To perform debugging on both the host and coprocessor while maintaining separate server launch string.
- To run TotalView in an environment where TotalView servers are not installed on an accessible, shared file system.

Open TotalView and your remote debugging session using this launch string:

```
totalview -mmic -r $HOSTNAME-mic0 binary.mic
```

And proceed with the debugging of "binary.mic" from Totalview, the same way as running in local (Go, Halt, Step, Breakpoint, etc.)

13. Tuning

Information and material on performance tuning from Intel can be found in [35] and [36].

A single Xeon Phi core is slower than a Xeon core due to lower clock frequency, smaller caches and lack of sophisticated features such as out-of-order execution and branch prediction. To fully exploit the processing power of a Xeon Phi, parallelism on both intruction level (SIMD) and thread level (OpenMP) is needed.

The following sections describe a few basic methodologies for improving the performance of a code on a Xeon Phi. As a first step, we consider methods for improving the performance on a single core. We then continue by thread-level parallelization in shared memory.

13.1. Single core optimization

13.1.1. Memory alignment

Xeon Phi can only perform memory reads/writes on 64-byte aligned data. Any unaligned data will fetched and stored by performing a masked unpack or pack operation on the first and last unaligned bytes. This may cause performance degredation, especially if the data to be operated on is small in size and mostly unaligned. In the following, we list a few of the most common ways to let the compiler to align the data or to assume that the data has been aligned.

Compiler flags for alignment

C/C++ `n/a`

Fortran Align arrays: `-align array64byte`

Align fields of derived types: `-align rec64byte`

Compiler directives for alignment

C/C++ Align variable `var`:

```
float var[100] __attribute__((aligned(64)));
```

Inform the compiler of the alignment of variable `var`:

```
__assume_aligned(var, 64)
```

Declare a loop to be aligned:

```
#pragma vector aligned
```

Fortran Align variable `var`:

```
real var(100)
```

```
!dir$ attributes align:64::var
```

Inform the compiler of the alignment of variable `var`:

```
!dir$ assume_aligned var:64
```

Declare a loop to operate on aligned data:

```
!dir$ vector aligned
```

Allocation of aligned dynamic memory

C/C++	Use `_mm_malloc()` and `_mm_free()` to allocate and free memory. These take the desired byte-alignment as a second input argument. When using a aligned variable `var`, use `__assume_aligned(var, 64)` to inform the compiler about the alignment
Fortran	Use `-align array64byte` compiler flag to enforce aligned heap memory allocation.

For a more detailed description of memory alignment on Xeon Phi, see [40].

13.1.2. SIMD optimization

Each Xeon Phi core has a 512-bit VPU unit which is capable of performing 16SP flops or 8 DP flops per clock cycle. VPU units are also capable of Fused Multiply-Add (FMA) or Fused Multiply-Subtract (FMS) operations which effectively double the theoretical floating point performance.

Intel compilers have several directives to aid vectorization of loops. These are listed in the following in short. For details, refer to the compiler manuals.

Let the compiler know there are no loop carried dependencies, but only affect compiler heuristics.

C/C++	`#pragma IVDEP`
Fortran	`!DIR$ IVDEP`

Let the compiler know the loop should be vectorized, but only affect compiler heuristics.

C/C++	`#pragma VECTOR`
Fortran	`!DIR$ VECTOR`

Force the compiler vectorize a loop, independent of heuristics.

C/C++	`#pragma SIMD` or `#pragma vector always`
Fortran	`!DIR$ SIMD` or `!DIR$ VECTOR ALWAYS`

The modern way of doing this in a portable way is to use OpenMP 4.0 SIMD directives. OpenMP SIMD directives are used to instruct the compiler to use the SIMD vector unit as the programmer wants. 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 parallelism. It 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.

```
!$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 contains links to more information from Intel.

13.2. OpenMP optimization

Threading parallelism with Intel Xeon Phi can be readily exploited with OpenMP. All threading constructs are equivalent for both offload and native models. We expect the basic concepts and syntax of OpenMP to be known. For OpenMP, see for instance "Using OpenMP" By Chapman et al [8], the OpenMP forum [16], and references therein.

The high level of parallelism available on a Xeon Phi available is very likely to reveal any performance problems related to threading previously been unnoticed in the code. In the following, we introduce a few of the most common OpenMP performance problems and suggest some ways to correct them. We begin by considering thread to core affinity and thread placement among the cores. For further details, we refer to [42] .

13.2.1. OpenMP thread affinity

Each Xeon Phi card contains a shared-memory environment with approximately 60 physical cores which, in turn, are divided into 4 logical cores each. We refer to this as node topology.

Each memory bank resides closer to some of the cores in the topology and therefore access to data laying in a memory bank attached to another socket is generally more expensive. Such a non-uniform memory access (NUMA) can create performance issues if threads are allowed to migrate from one logical core to another during their execution.

In order to extract maximum performance, consider binding OpenMP threads to logical and physical cores across different sockets on a single Xeon Phi card. The layout of this binding in respect to the node topology has performance implications depending on the computational task and is referred as thread affinity.

We now briefly show how to set thread affinity using Intel compilers and OpenMP-library. For a complete description, see "Thread affinity interface" in the Intel compiler manual.

The thread affinity interface of the Intel runtime library can be controlled by using the `KMP_AFFINITY` environment variable or by using a proprietary Intel API. We now focus on the former. A standardized method for setting affinity is available with OpenMP 4.0.

`KMP_AFFINITY`=[modifier,...]<type>[,<permute>][,<offset>]]

modifier	**default**=noverbose, respect, granularity=core
	granularity=<{fine, thread, core}>, norespect, noverbose, nowarnings, proclist={<proc-list>}, respect, verbose, warnings.
type	**default**=none
	balanced, compact, disabled, explicit, none, scatter
permute	**default**=0.
	$\geq$ 0, integer. Not valid with type=explicit,none,disabled.
offset	**default**=0.
	$\geq$ 0, integer. Not valid with type=explicit,none,disabled

In most cases it is sufficient only to specify the affinity and granularity. The most important affinity types supported by Intel Xeon Phi are

balanced	Thread affinity balanced is a mixture of scatter and compact affinities. Threads from <1> to <n_p> will be spread across the topology as evenly as evenly as possible in the granularity context, where <n_p> denotes the number of physical cores. For thread <k> from threads <n_p+1> to <n> will be assigned as close as possible to thread <k+1>.
compact	Thread <k+1> will be assigned as close as possible to thread <k> in the granularity context according to which the threads are placed.
none	Threads are not bound to any contexts. Use of affinity none is not recommended in general.
scatter	Threads from <1> to <n> will be spread across the topology as evenly as possibly in the granularity context according to which the threads are placed.

The most important granularity types supported by Intel Xeon Phi are

core	Threads are bound to a single core, but allowed to float within the context of a physical core.
fine/thread	Threads are bound to a single context, i.e., a logical core.

13.2.2. Example: Thread affinity

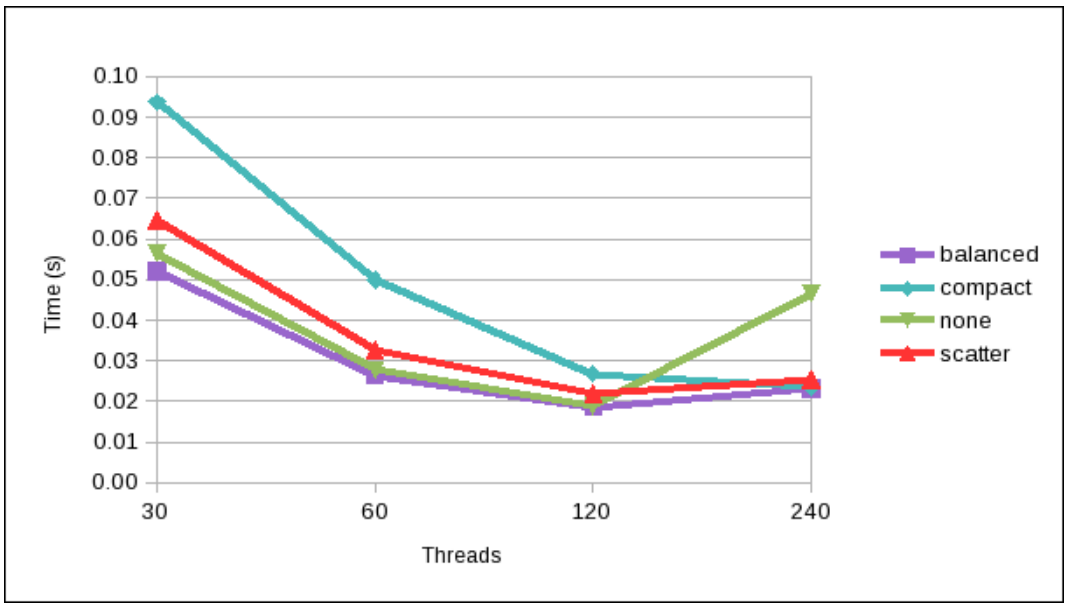
We now consider the effect of thread affinity to matrix-matrix multiply. Let A, B and C=AB be real matrices of size 4000-by-4000. We implement the data initialization and multiplication operation in Fortran90 (without blocking) by using jki loop-ordering and OpenMP as follows

```
! Initialize data per thread
!$OMP PARALLEL DO DEFAULT(NONE) &
!$OMP SHARED(A,B,C) &
!$OMP PRIVATE(j)
DO j=1,n
   ! Initialize A
   CALL RANDOM_NUMBER(A(1:n,j))
   ! Initialise B
   CALL RANDOM_NUMBER(B(1:n,j))
   ! Initialise C
   C(1:n,j)=0D0
END DO
!$OMP END PARALLEL DO

! C=A*B
!$OMP PARALLEL DO DEFAULT(NONE) &
!$OMP SHARED(A,B,C) &
!$OMP PRIVATE(i,j,k)
DO j=1,n
  DO k=1,n
    DO i=1,n
      C(i,j)=C(i,j)+A(i,k)*B(k,j)
    END DO
  END DO
END DO
!$OMP END PARALLEL DO
```

Running and timing the matrix-matrix multiplication part on a single Intel Xeon Phi 7110 card, we have the following results.

Threads	balanced, t(s)	compact, t(s)	none, t(s)	scatter, t(s)
1	1.43E+00	1.45E+00	1.42E+00	1.58E+00
30	5.21E-02	9.37E-02	5.65E-02	6.45E-02
60	2.64E-02	5.00E-02	2.77E-02	3.26E-02
120	1.86E-02	2.68E-02	1.89E-02	2.18E-02
240	2.31E-02	2.35E-02	4.66E-02	2.54E-02

Figure 11. Results of Example: Thread affinity



As seen from the results, changing the thread affinity has implications to the performance of even a very simple test case. Using affinity `none` is generally not recommended and is the slowest when all available threads are being used. Using affinity `balanced` is generally a good compromise, especially if one wishes to use less than the number of threads available at maximum.

13.2.3. OpenMP thread placement

Xeon Phi runtime for OpenMP includes an extension for placing the threads over the cores on a single card. For a given number of cores and threads, the user has control where (relative to the first physical core, i.e., core 0) the OpenMP threads are placed. In conjunction with offloading from several MPI processes to a single Xeon Phi card, such control can be very useful to avoid oversubscription of cores.

The placement of the threads can be controlled with the environment variable `KMP_PLACE_THREADS`. It specifies the number of cores to allocate with an optional offset value and number of threads per core to use. Effectively `KMP_PLACE_THREADS` defines the node topology for `KMP_AFFINITY`.

`KMP_PLACE_THREADS`= (*int* ["C" | "T"] [*delim*] | *delim*) [*int* ["T"] [*delim*]] [*int* ["O"]],

where *int* is a simple integer constant and *delim* is either "," or "x".

C **default** if "C" or "T" not specified

Indicates number of cores

T Indicates number of threads

O **default**=0O.

Indicates number of cores to offset, starting from core 0. Offset ignores granularity.

As an example, we consider the case with `OMP_NUM_THREADS`=20.

`KMP_PLACE_THREADS` value	**Thread placement**
5C,4T,0O or 5,4,0 or 5C or 5	Threads are placed on cores from 0 to 4 with 4 threads per core.
5C,4T,10O or 5,4,10 or 5C,10O	Threads are placed on cores from 10 to 14 with 4 threads per core.
20C,1T,20O or 20,1,20	Threads are placed on cores from 20 to 40 with 1 thread per core.

13.2.4. Multiple parallel regions and barriers

Whenever an OpenMP parallel region is encountered, a team of threads is formed and launched to execute the computations. Whenever the parallel region is ended, threads are joined and the computation proceeds with a single thread. Between different parallel regions it is up to the OpenMP implementation to decide whether the threads are shut down or left in an idle state.

Intel OpenMP -library leaves the threads in a running state for a predefined amount of time before setting them to sleep. The time is defined by `KMP_BLOCKTIME` and `KMP_LIBRARY` environment variables. The default is 200ms. For more details, see Sections "Intel Environment Variables Extensions" and "Execution modes" in the Intel compiler manual.

Repeatedly forming and disbanding thread-teams and setting idle threads to sleep has some overhead associated with it. Another common source of threading overhead in OpenMP computations are implicit or explicit barriers. Recall that many OpenMP constructs have an implicit barrier attached to the end of the construct. Then, especially if the amount of work done inside an OpenMP construct is relatively small, thread synchronization with several threads may be a source of significant overhead. If the computations are independent, the implicit barrier at the end of OpenMP constructs can be removed with the optional NOWAIT parameter.

13.2.5. Example: Multiple parallel regions and barriers

We now consider the effect of multiple parallel regions and barriers to performance. Let `v` be a vector with real entries with size `n`=1000000. Let `f(x)` denote a function, defined as `f(x)=x+1`.

We implement an OpenMP loop to apply `f(x)` successively `repeats`=10000 times to a given vector `v`. We consider three different implementations. In the first one, OpenMP parallel region is re-initialized for each successive application of `f(x)`. The second one initializes the parallel region once, but contains two implicit barriers from OpenMP constructs. In the third implementation the parallel region is initialized once and one barrier is used to synchronize the repetitions.

Implementation 1: parallel region re-initialized repeatedly.

```
DO rep=1,repeats
  !$OMP PARALLEL DO DEFAULT(NONE) NUM_THREADS(threads) &
  !$OMP SHARED(vec1, rep, n) &
  !$OMP PRIVATE(i)
  DO i=1,n
    vec1(i)=vec1(i)+1D0
  END DO
  !$OMP END PARALLEL DO

  ops = ops + 1
END DO
```

Implementation 2: parallel region initialized once, two implicit barriers from OpenMP constructs.

```
!$OMP PARALLEL DEFAULT(NONE) NUM_THREADS(threads) &
!$OMP SHARED(vec1, repeats, ops, n) &
!$OMP PRIVATE(i, rep)
DO rep=1,repeats
  !$OMP DO
  DO i=1,n
    vec1(i)=vec1(i)+1D0
  END DO
  !$OMP END DO

  !$OMP SINGLE
  ops = ops + 1
  !$OMP END SINGLE
END DO
!$OMP END PARALLEL
```

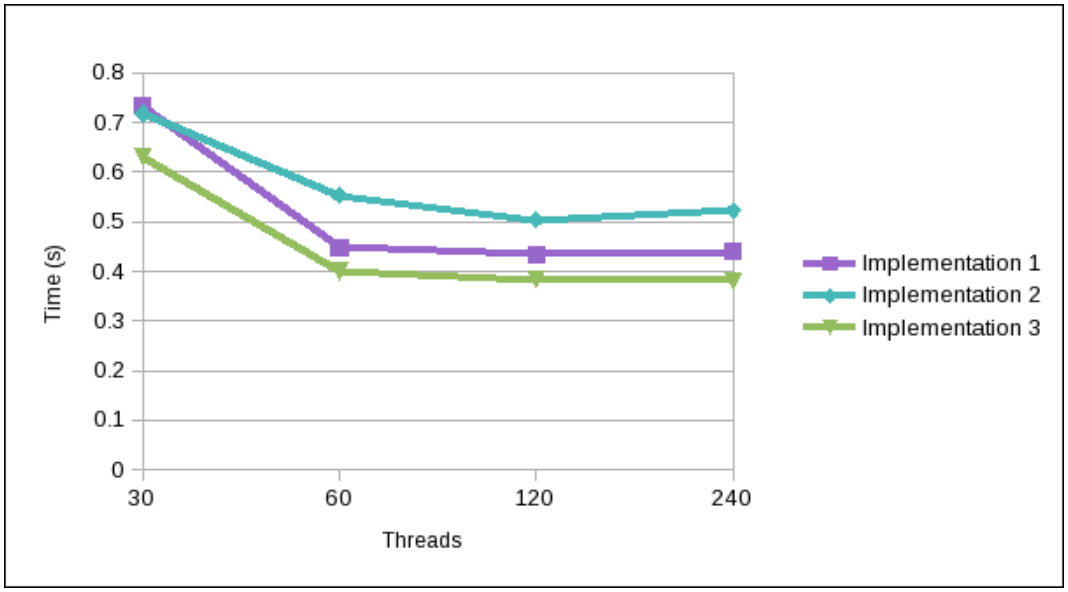
Implementation 3: parallel region initialized once, one explicit barrier from OpenMP construct.

```
!$OMP PARALLEL DEFAULT(NONE) NUM_THREADS(threads) &
!$OMP SHARED(vec1, repeats, ops, n) &
!$OMP PRIVATE(i, rep)
DO rep=1,repeats
  !$OMP DO
  DO i=1,n
    vec1(i)=vec1(i)+1D0
  END DO
  !$OMP END DO NOWAIT

  !$OMP SINGLE
  ops = ops + 1
  !$OMP END SINGLE NOWAIT

  ! Synchronize threads once per round
  !$OMP BARRIER
  END DO
!$OMP END PARALLEL
```

The results on a single Intel Xeon Phi 7110 card with `KMP_AFFINITY`=granularity=fine,balanced and `KMP_BLOCKTIME`=200 are presented in the following table.

Threads	**Implementation 1, t(s)**	**Implementation 2, t(s)**	**Implementation 3, t(s)**
1	2.40E+01	2.37E+01	2.37E+01
30	7.33E-01	7.18E-01	6.31E-01
60	4.48E-01	5.53E-01	4.00E-01
120	4.34E-01	5.04E-01	3.83E-01
240	4.40E-01	5.23E-01	3.81E-01

Figure 12. Results of Example: Multiple parallel regions and barriers



As the number of threads used increases, parallel threading overhead becomes more apparent. The implementation with only one barrier is the fastest by a fair margin. Implementation 1 comes out as the second fastest. This is due to Implementation 1 having only one barrier (at the end of the parallel region) versus two in Implementation 2 (at the end of both of the OpenMP constructs). With Implementation 1, the parallel region is re-initialized immediately after it has ended and thus waiting time for threads is less than `KMP_BLOCKTIME`, i.e., the threads are not being put to sleep before the next parallel iteration begins.

13.2.6. False sharing

On a multiprocessor shared-memory system, each core has some local cache, which must be kept coherent the among the cores in the system. Processor cache is organized into several cache lines, each of which map to some part of the main memory. On an Intel Xeon Phi, cache line size is 64 bytes. For reference and details, see [44] .

If more than one core accesses data from the same cache line, a cache line is shared. Whenever a shared cache line is updated, to maintain coherency an update is forced to the caches of all the cores accessing the cache line.

False sharing occurs when several cores access and update different variables which reside on a single shared cache line. The resulting updates to maintain cache coherency may cause a significant performance degradation. The processors may not be actually sharing any data, it is sufficient that the data resides on a same cache line, hence the name false sharing. Due to the ring-bus architecture of the Xeon Phi, false sharing among the cores can cause severe performance degredation.

False sharing can be avoided by carefully considering write access to shared variables. If a variable is updated often, it may be worthwhile to use a private variable in stead of a shared one and use reduction at the end of the work sharing loop.

Given a code with performance problems, false sharing may be hard to localize. Intel VTune Performance Analyzer can be used to locate false sharing. For details, we refer to [43].

13.2.7. Example: False sharing

We now consider a simple example where false sharing occurs. Let `v` be a vector with real entries and size `n`=1E+08. Let `f(x)` denote a function which counts the number of entries in `v` which are smaller than zero.

We implement `f(x)` with OpenMP in two different ways. In the first implementation, each thread counts the number of negative entries it has found in `v` to a globally shared array. To avoid race conditions, each thread uses its own entry in the shared array, uniquely determined by thread id. When a thread has finished its portion

of vector, a global counter is atomically incremented. The second implementation is practically equivalent to the first one, except that each thread has its own private array for counting the data.

Implementation 1: False sharing with an array counter.

```
!$OMP PARALLEL DEFAULT(NONE) NUM_THREADS(threads) &
!$OMP SHARED(vec, count, counter, n) &
!$OMP PRIVATE(i,TID)

TID=1
!$ TID=omp_get_thread_num()+1

!$OMP DO
DO i=1,n
  IF (vec(i)<0) counter(TID)=counter(TID)+1
END DO
!$OMP END DO

!OMP ATOMIC
count = counter(TID)+count

!$OMP END PARALLEL
```

Implementation 2: Private array used to avoid false sharing.

```
!$OMP PARALLEL DEFAULT(NONE) NUM_THREADS(threads) &
!$OMP SHARED(vec, count, n) &
!$OMP PRIVATE(i, counter, TID)

TID=1
!$ TID=omp_get_thread_num()+1

!$OMP DO
DO i=1,n
  IF (vec(i)<0) counter(TID)=counter(TID)+1
END DO
!$OMP END DO

!OMP ATOMIC
count = counter(TID)+count

!$OMP END PARALLEL
```

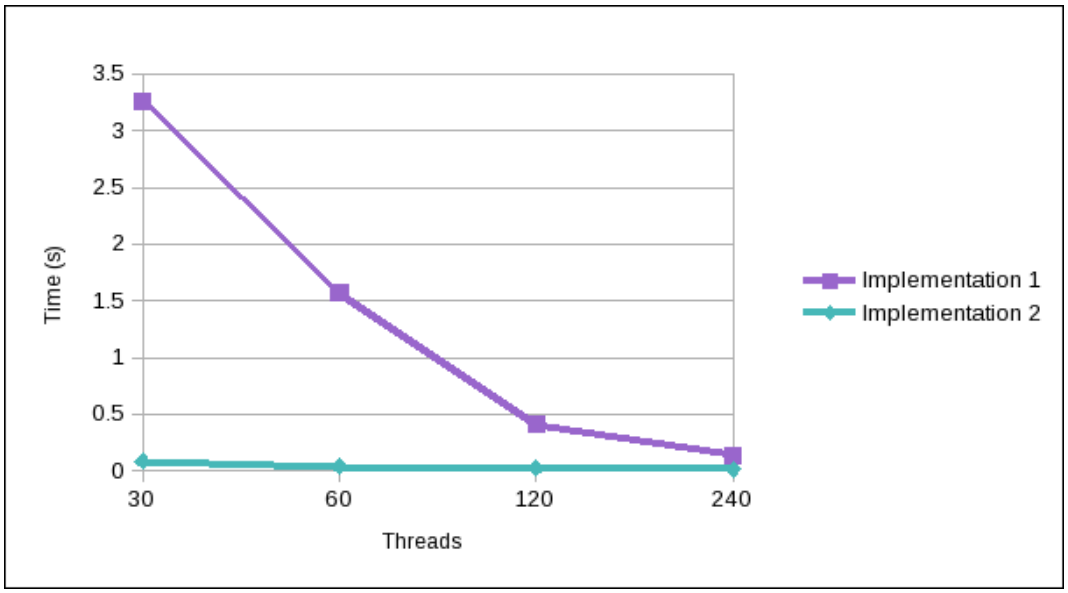
We note that a better implementation for this particular problem will be given in the next section.

The results on a single Intel Xeon Phi 7110 card with `KMP_AFFINITY`=granularity=fine,balanced are presented in the following table. We note that to obtain the results, we compiled the test code with `-O1`. On optimization level `-O2` and higher, at least in this case, the possibility of false sharing with multiple threads was recognized and corrected by the Intel Fortran compiler.

Threads	**Implementation 1, t(s)**	**Implementation 2, t(s)**
1	1.94E+00	2.05E+00
30	3.26E+00	8.30E-02
60	1.57E+00	4.18E-02

Threads	Implementation 1, t(s)	Implementation 2, t(s)
120	4.13E-01	2.53E-02
240	1.40E-01	1.34E-02

Figure 13. Results of Example: False sharing



As expected, Implementation 2 is faster than Implementation 1, with a difference of an order of magnitude. Although in this case we had to lower the optimization level to prevent the compiler from correcting the situation, we cannot completely rely on the compiler to detect false sharing, especially if the code to be compiled is relatively complex.

13.2.8. Memory limitations

Available memory per core on Xeon Phi is very limited. When an application is run using all the available threads, approximately 30Mb of memory is available per thread assuming none of the data is shared. Excessive memory allocation per thread is therefore highly discouraged. Care should be also taken when assigning private variables in order to avoid unnecessary data duplication among threads.

13.2.9. Example: Memory limitations

We now return to the example given in the previous section. In the example, we prevented threads from doing false sharing by modifying the definition of the vector containing the counters. What is important to note is that in doing so, each thread now implicitly allocates a vector of length nthreads, i.e., the memory consumption is quadratic in terms of the number of threads. A better alternative is to let each thread to store the local result in a temporary variable and use a reduction to count the number of elements smaller than zero.

Implementation 3: Temporary variable with reduction used to store local results.

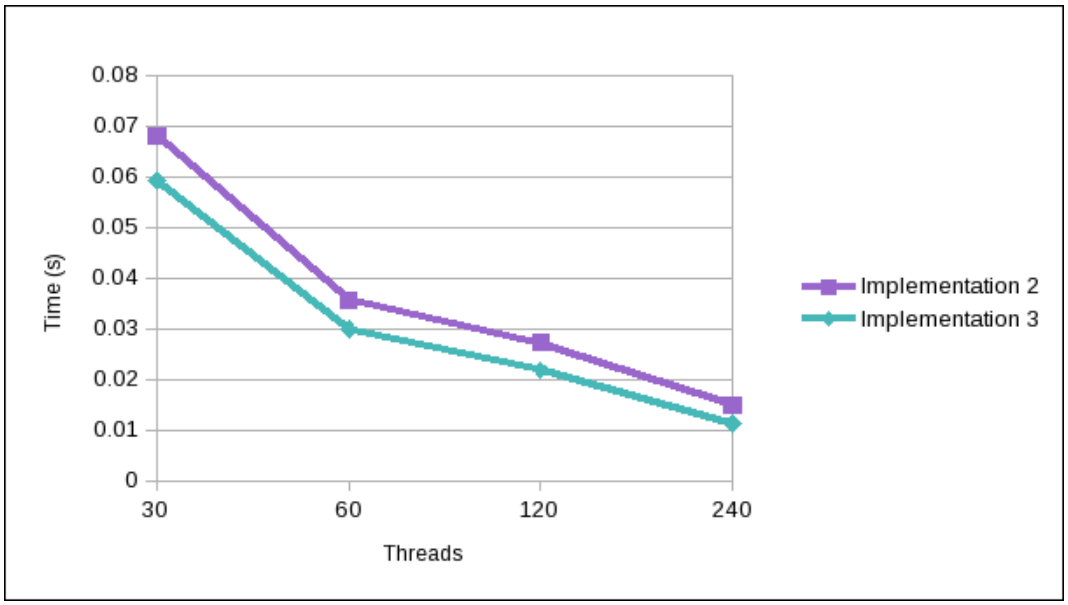
```
count = 0
!$OMP PARALLEL DEFAULT(NONE) NUM_THREADS(threads) &
!$OMP SHARED(vec, n) &
!$OMP PRIVATE(i, Lcount, TID) &
!$OMP REDUCTION(+:count)

TID=1
!$ TID=omp_get_thread_num()+1
```

```
Lcount = 0
!$OMP DO
DO i=1,n
  IF (vec(i)<0) Lcount=Lcount+1
END DO
!$OMP END DO

count = Lcount+count
!$OMP END PARALLEL
```

We have the following results, where Implementation 2 refers to second implementation given in the previous section. As previously, results have been computed on a single Intel Xeon Phi 7110 card by using `KMP_AFFINITY`=granularity=fine,balanced and optimization level `-O1`.

Threads	Implementation 2, t(s)	Implementation 3, t(s)
1	2.04E+00	1.78E+00
30	6.81E-02	5.93E-02
60	3.57E-02	2.99E-02
120	2.73E-02	2.19E-02
240	1.50E-02	1.13E-02

Figure 14. Results of Example: Memory limitations



The main difference between Implementation 2 and 3 is memory use. This is because Implementation 2 uses a private array for storing the result (memory usage grows quadratically with the number of threads), whereas in Implementation 3 the result is stored a single scalar per thread. In total 57600 elements have to be stored in Implementation 2 with 240 threads, whereas just 240 elements suffice in Implementation 3 for the same amount of threads. We note that if an array of results is to be computed, one should prefer using an implementation where the size of the work arrays does grow with the number of threads used.

13.2.10. Nested parallelism

Due to the limited amount of memory available, sometimes using all available threads on an Intel Xeon Phi to parallelize the outer loop of some computation is not possible. In some cases this may not be due to inefficient structure of the code, but because the data needed for computations per thread is too large. In this case, to take advantage of all the processing power of the coprocessor, an option is to use nested OpenMP parallelism.

When nested parallelism is enabled, any inner OpenMP parallel region which is enclosed within an outer parallel region will be executed with multiple threads. The performance impact of using nested parallelism is similar to performance impact of using multiple parallel regions and barriers.

Enabling OpenMP nested parallelism is done by setting environment variable `OMP_NESTED=TRUE` or with an API call to `omp_set_nested` -function. The number of nested threads within each OpenMP parallel region is done by setting the environment variable `OMP_NUM_THREADS=n_1, n_2,n_3,...`, where `n_j` refers to the number of threads on the `j`th level. The number or threads within each nesting level can be also set with an API call to `omp_set_num_threads` -function.

13.2.11. Example: Nested parallelism

Consider a case where several independent matrix-matrix multiplications have to be computed. Let `A` be an `n`-by-`n` matrix and let matrix B_k be defined as $B_k = A^T A$.

We let `m`=1000, `n`=1000 and `k`=240 and study the effect of parallelizing the computation of B_k's in three different ways. The first case is to use parallelize the loop over `k` with all available threads. A second case is to parallelize the computation over different `k`'s to physical Intel Xeon Phi cores and use nested parallelism with varying levels of hardware threads in the computation of the matrix-matrix multiplications. The third case uses parallelization only over physical cores. For this, we have the following implementation.

Implementation: possibly nested parallel loop for computing $A^T A$.

```
!$OMP PARALLEL DO DEFAULT(NONE) NUM_THREADS(nthreads) &
!$OMP SCHEDULE(STATIC) &
!$OMP SHARED(A, B, m, n, k)
DO i=1,k
  CALL DGEMM('T','N', n, n, m, 1D0, A, m, A, m, 0D0, B(1,1,i), n)
END DO
!$OMP END PARALLEL DO
```

The results on a single Intel Xeon Phi 7110 card with `KMP_AFFINITY`=granularity=fine,balanced are presented in the following table.

Threads	**240/1, t(s)**	**60/4, t(s)**	**120/1, t(s)**	**60/2, t(s)**	**60/1, t(s)**
time	1.06E+01	4.98E+00	5.64E+00	4.80E+00	4.81E+00

Intuitively one would expect using 240 threads to be the most efficient in this case. The results show otherwise, however. Since the threads are competing for the same cache, the performance is lowered. Nested parallelism does not offer significant improvements from just using 60 threads in a flat fashion. One reason for this might be that with the affinity policies used, it is difficult to control the thread placement on the second level of thread parallelism.

13.2.12. OpenMP load balancing

With any parallel processing, some processes or threads may require more resources and be more time consuming than others. In a regular distributed memory program, load balancing requires programming effort to redistribute parts of the computation among the processors. In a shared memory program with a runtime, such as OpenMP, load balancing can be in some cases automatically handled by the runtime itself with little overhead.

OpenMP loop constructs support an additional SCHEDULE-clause. The syntax for this is

SCHEDULE(<kind>[,chunk_size]),

kind **default**=STATIC.

<STATIC,DYNAMIC,GUIDED,RUNTIME>

chunk_size	**default**=value depends on the schedule kind.
	>0, integer.

Different schedule kinds supported by OpenMP runtime on a Xeon Phi are

STATIC	With the static scheduling policy, iteration indices are divided into chucks of chunk_size and distributed to threads in a round-robin fashion. If chuck_size is not defined, iteration indices are divided into chucks that are roughly equal in size.
DYNAMIC	With the dynamic scheduling policy, iteration indices are assigned to threads in chunks of chunk_size. Threads request and process new chucks with assigned iteration indices until the whole index range has been processed.
GUIDED	With the guided scheduling policy, iteration indices are assigned to threads in chunks of size chunk_size at minimum. In the beginning of the iteration, the chunk_size actually assigned to be processed is proportional to the number of unassigned iteration indices versus the number of available threads. The assigned chunk_size and can be larger than the minimum.
RUNTIME	The scheduling policy and chunk size will be decided at runtime based on the `OMP_SCHEDULE` environment variable.

14. Performance analysis tools

14.1. Intel performance analysis tools

The following Intel performance analysis tools have been enabled for the Intel Xeon Phi coprocessor:

- Intel trace analyzer and collector (ITAC)
- Intel VTune Amplifier XE

More information can be found in the Intel Parallel Studio documentation.

14.2. Scalasca

Scalasca [39] is a scalable automatic performance analysis toolset designed to profile large scale parallel Fortran, C and C++ applications that use MPI, OpenMP and hybrid MPI+OpenMP parallelization models. It is portable on Intel Xeon Phi architecture in native, symmetric and offload models. Version 2.x uses the Score-P instrumenter and measurment libraries. The following examples are taken from [45].

14.2.1. Compilation of Scalasca

On the host Scalasca can be normally compiled, while on the device one must perform a cross-compilation and add -mmic compiler option.

14.2.2. Usage

Instrumentation of the code to be profiled is done with the command skin.

```
$ skin mpiifort -O -openmp *.f
```

This produces the instrumented executable that will be executed on the host, while

```
$ skin mpiifort -O -openmp -mmic *.f
```

produces an executable for the coprocessor. Further, measurement is than performed with the scan command:

```
$ scan mpiexec -n 2 a.out.cpu       // on the host
% scan mpiexec -n 61 a.out.mic    // on the device
```

It can also be launched on more than one node on the host:

```
$ scan mpiexec.hydra -host node0 -n 1 a.out.cpu : -host node1 -n 1 a.out.cpu
```

and on more than one coprocessor, if available:

```
$ scan mpiexec.hydra -host mic0 -n 30 a.out.mic : -host mic1 -n 31 a.out.mic
```

For symmetric execution one can use:

```
$ scan mpiexec.hydra -host node0 -n 2 a.out.cpu : -host mic0 -n 61 a.out.mic
```

Finally, the collected data can be analyzed with the square command. The scan output would look something like `epik_a_2x16_sum` for the runs performed on the host and `epik_a_mic61x4_sum` for those performed on the coprocessor. For data collected on the host and on the device we have:

```
$ square epik_a_2x16_sum
$ square epik_a_61x4_sum
```

respectively. To analyze the data collected in a run from a symmetric execution type:

```
$ square epik_a_2x16+mic61x4_sum.
```

15. Benchmarks

15.1. Performance evaluation of native execution

In the following we show the performance of some known benchmarks using the native execution model.

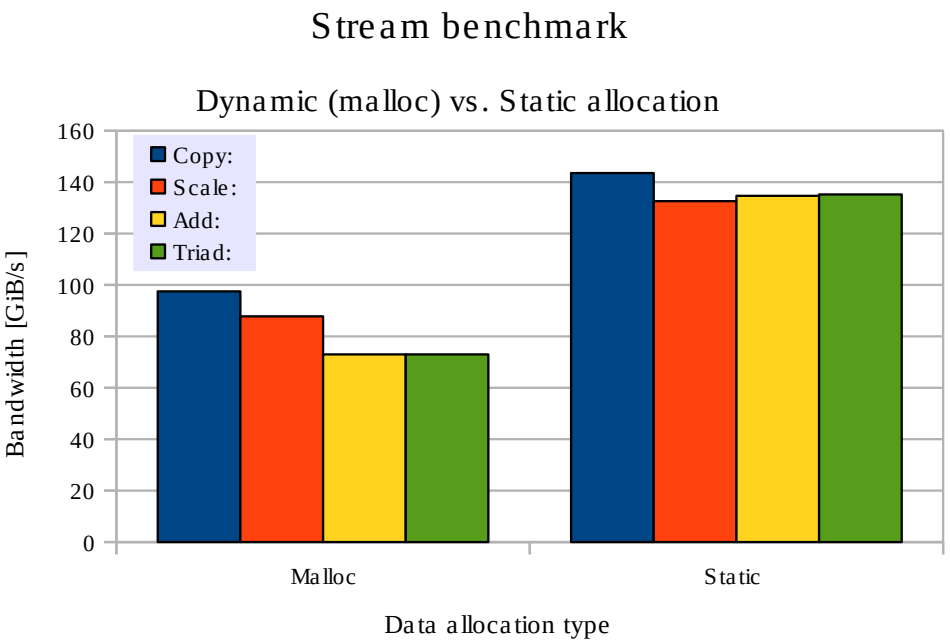
15.1.1. Stream memory bandwidth benchmark

Stream is a well known benchmark for measuring memory bandwidth written by John D. Calpin of TACC. TACC also happens to host the large supercomputer system called “Stampede” which is an accelerated system using a large array of Intel Xeon Phis. Stream can be built in several ways, it turned out that static allocation of the three vectors provided the best results. The follwing source code illustrates 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
```

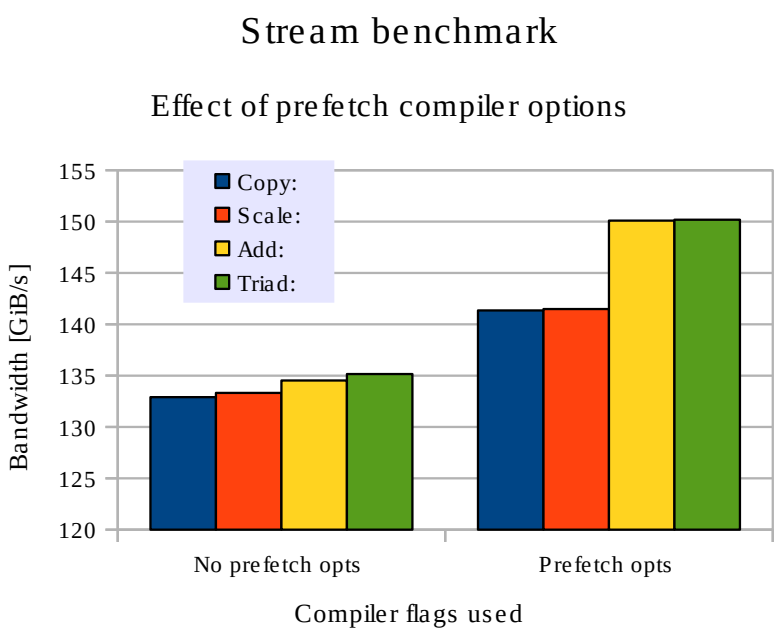
The figure below shows the difference between using malloc and static allocation. For this reason all subsequent runs using stream were done using static allocation.

Figure 15. Effect of the data allocation scheme, C malloc versus static allocation. Both runs werde done with compact placements.



To achieve optimum performance on the Xeon Phi architecture one needs to take extra care when compiling programs. There are many compiler switches that can help with this. However, the switches may have different effect on the MIC architecture than on the Xeon architecture. One possible optimalisation is prefetching. The MIC relies much more on software prefetching as MIC cores lack the same efficient hardware prefetcher. More is left to the programmer and the compiler.

Figure 16. Effect of the compiler prefetch options (compiler options used: -opt-prefetch=4 -opt-prefetch-distance=64,32).



The figure above shows the beneficial effect of providing prefetch compiler options to the C-compiler enabling it to issue prefetch instructions in loops and other places where memory latencies impacts performance. The distance numbers of iterations given to the prefetcher options are found by trial and error. There is in most cases not a magic number that will be optimal in all cases. Additionally one might use streaming store instructions to prevent store instructions to write to cache. There is no reason to pollute the cache with data if the data is not reused. For the tests done here it had a small effect. Bandwidth for Triad went up from 134.9 to 135.2 GiB/s, hardly significant.

Placement of threads per core is also a major performance issue. Hardware threads are grouped together onto cores and share resources. This sharing of resources can benefit or harm particular algorithms, depending on their behaviors. Understanding the behavior of your algorithms will guide you in selecting the optimal affinity. Affinity can be specified in the environment (KMP_AFFINITY) or via a function call (kmp_set_affinity). There are basically three models, compact, scatter and balanced. In addition there is granularity. Granularity is set to fine so each OpenMP thread is constrained to a single HW thread. Alternatively, setting core granularity groups the OpenMP threads assigned to a core into little quartets with free reign over their respective cores.

The affinity types compact and scatter either clump OpenMP threads together on as few cores as possible or spread them apart so that adjacent thread numbers are on different cores. Sometimes though, it is advantageous to enable a limited number of threads distributed across the cores in a manner that leaves adjacent thread numbers on the same cores. For this, there is a new affinity type available, BALANCED, which does exactly that. Using the verbose setting shown above, you can determine how the OpenMP thread numbers are dispersed. Some illustartions of the various placement models are shown in the following figures.

Figure 17. Illustration of the compact placement model. All 60 threads are scheduled to be as close together as possible. Four threads are sharing a single core even if there are idle cores. Beneficial when the threads can share the L2 cache.

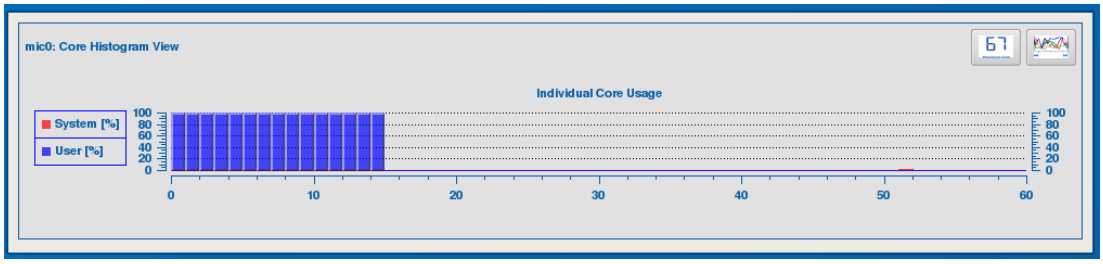
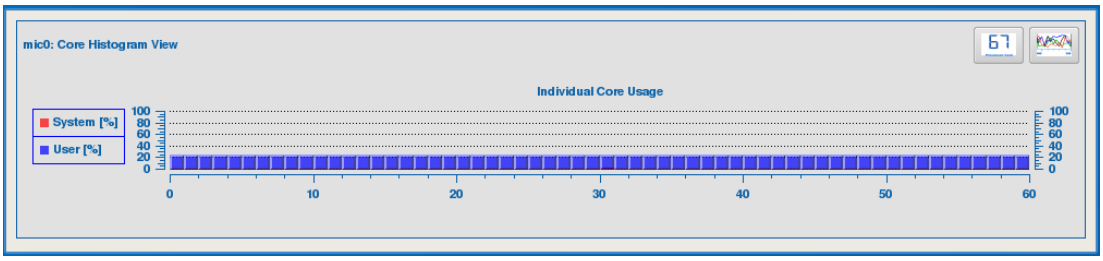
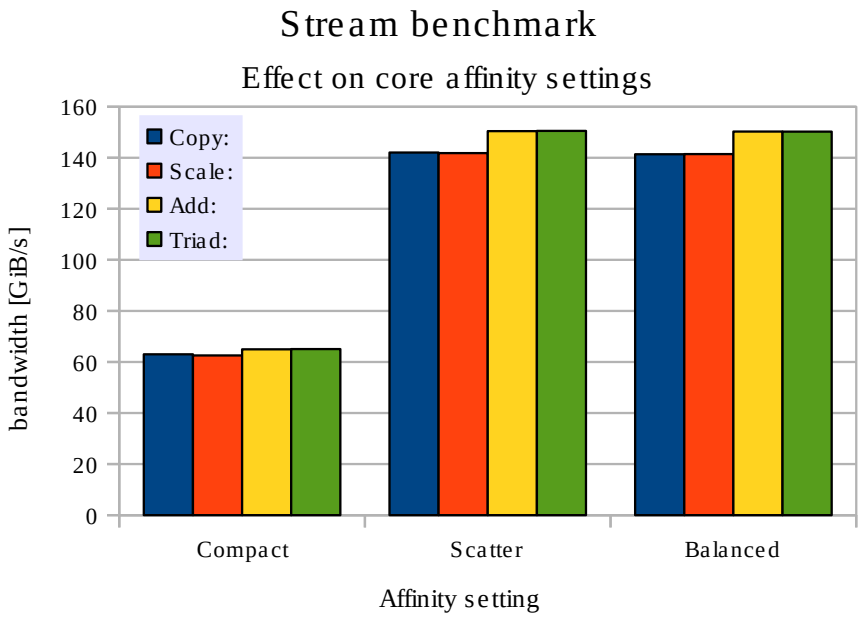


Figure 18. Illustration of scatter placement model. All 60 threads are scheduled as far apart as possible. In this case one thread per core. Beneficial when memory bandwidth is required.



The following figure shows the effect on performance when using one, two, three or four threads per core. When effectively run the 60 cores are capable of saturating the memory bandwidth. The stream benchmark is just copying data and does not perform any significant calculation. Effects using more cores do not show up when all the memory bandwidth is already utilized.

Figure 19. Effect on core affinity setting. 60 threads are used in this test. For 240 threads the effect is small.



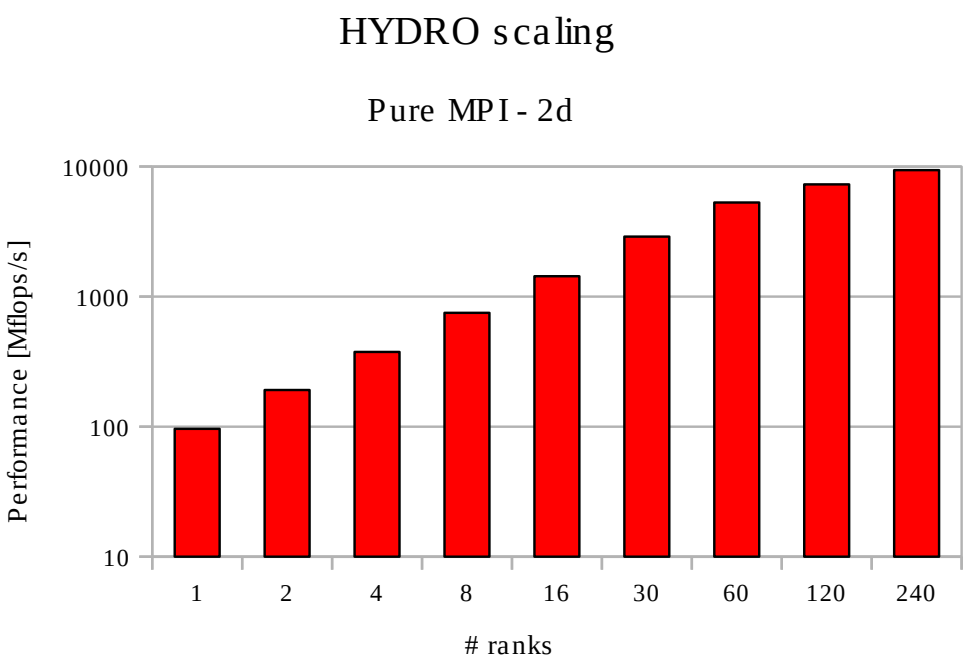
15.1.2. HYDRO benchmark

HYDRO is a heavily 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 exists 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 version and both OpenMP and MPI versions have been used for evaluation.

How well this architecture scales using MPI or a hybrid mode using both MPI and OpenMP is of interest in the initial testing of HYDRO. Unfortunately, there the hybrid model does not provide performance numbers in Mflops/s, so run time is used to measure scaling.

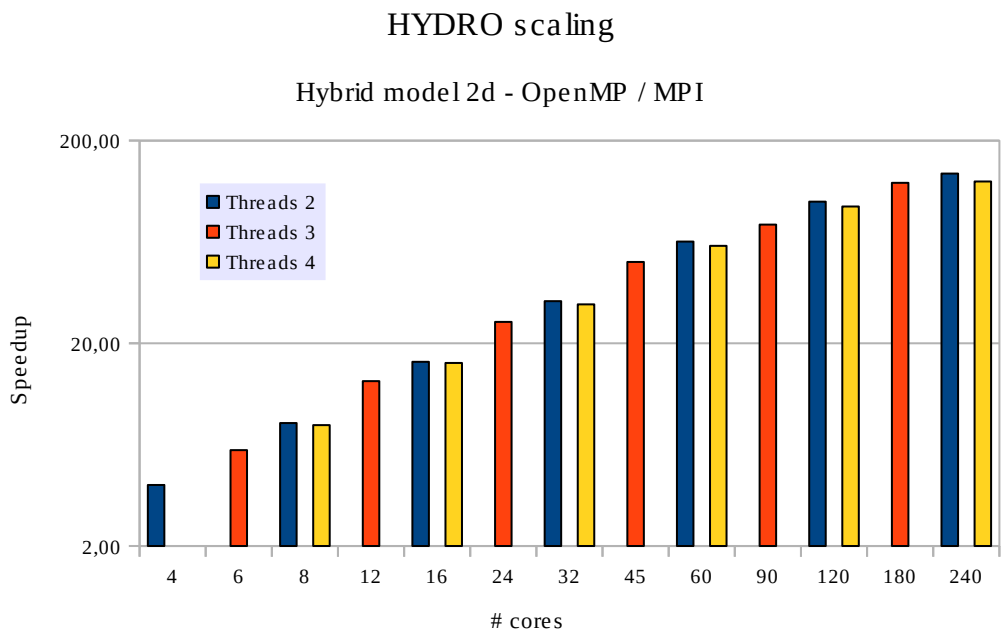
Figure 20. HYDRO scaling using a pure MPI model solving the 2d problem.



The figure above shows the scaling using a pure MPI implementation of HYDRO. The scaling seems to be quite good up to a point where 2 or more threads are scheduled per core. The placements are at this test the defaults. Better tuning of the placements might provide slightly better results.

For hybrid models the implementation of HYDRO does not yield performance numbers, so run times are taken as indicators and a scaling is calculated. As there are no runs with only one core, perfect scaling is assumed from 1 to 4 cores, e.g. a scaling factor of 4 for the 4 core run. The next figure shows the obtained scaling using 2, 3 and 4 OpenMP threads per MPI rank. Scaling is quite good up till 3 threads per rank, e.q. 180 cores, but the highest performance was measured using all 240 cores.>

Figure 21. Scaling using a hybrid MPI/OpenMP model solving the 2d problem. Runs are performed using 2,3 or 4 OpenMP thraeds per MPI rank. The core count is the total number of cores employed. The speedup using 4 cores is assumed to be perfect from a single core.



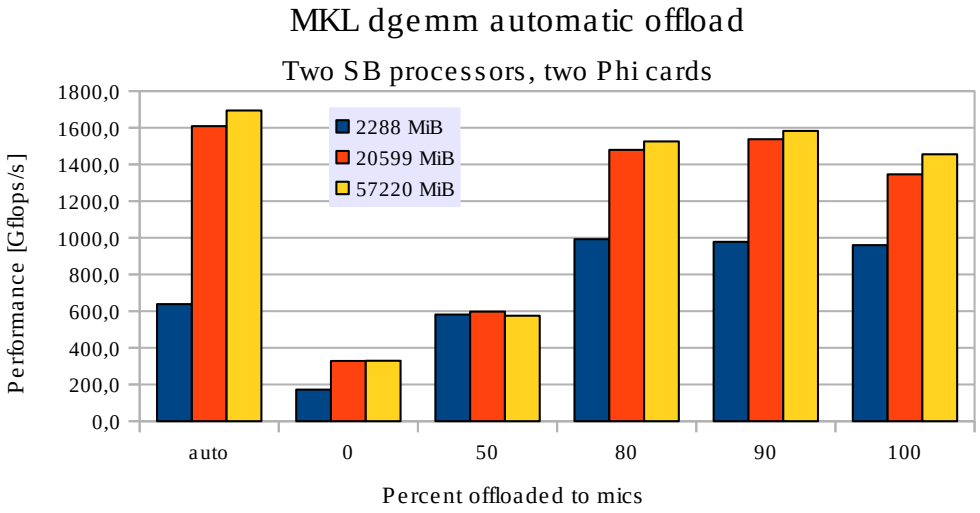
15.2. Offloading Performance

Gaining performance using the offload model is not a simple task. The load balancing between the host cores and the co-processor cores needs special attention.

15.2.1. Offloading with MKL

Intel provides support for automatic usage of the Xeon Phi as a co-processor by means of automatic offloading of work using MKL routines. This is a very simple way to exploit the MIC co-processor. and run. An example is shown in the following picture:

Figure 22. MKL automatic offload using two Xeon Phi cards and both SB host processors.



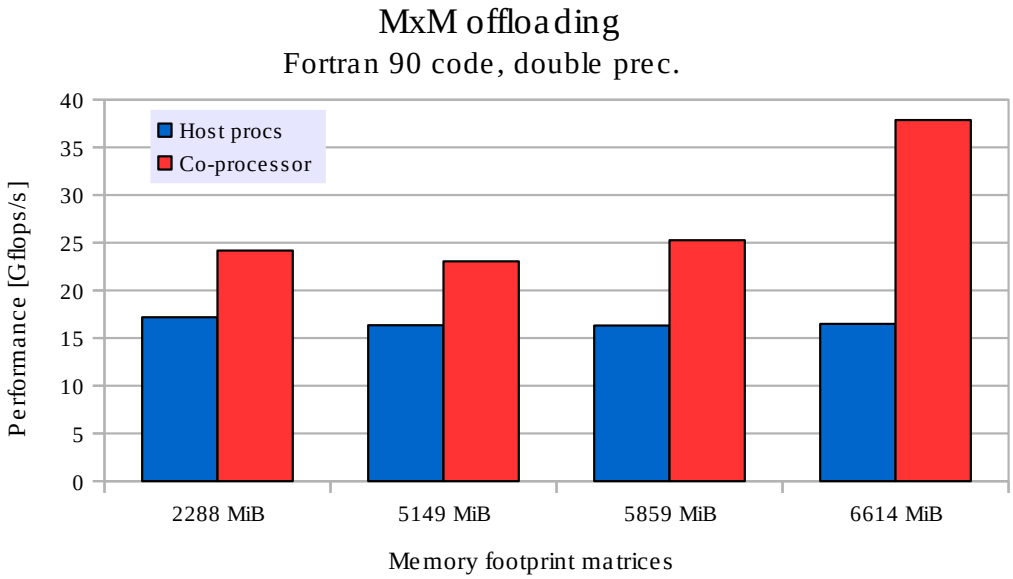
This figure shows performance measures when performing dense matrix matrix multiplication using MKL and automatic offloading. Only some environment variables are set to achieve this performance. It is quite remarkable how well the automatic load balancing in MKL's run time system works. It's actually quite hard to beat the automatic load partitioning between the MIC processor and the host processor. Looking at the actual numbers there is reason be be really impressed. A normal compute node used can perform at about 320 Gflops/s when doing dgemm. With two installed Xeon Phi cards with MIC coprocessors this performance clocks in at 1694 Gflops/s, or 1.7 Tflops/s per compute node.

15.2.2. Offloading user functions

In this apprach the user writes a complete function or subroutine to be offloaded. Using this routine is straighforward: it is called just as any other function with the only addition that data must be handled. Initiating data transfers between the two memories must be handled. This data transfer can be overlapping with other workload on the host processor hiding latency of the transfer.

The figure below shows the performance measured when comparing host CPUs and co-processors running Fortran 90 code with OpenMP threading of three nested loops doing matrix multiplication in double precision. Far inferior of the MKL library, but serves as an illustration of what can be expected using user Fortran 90 code. No special form of optimization have been performed, only compiler flags like -O3, -mavx in addition to -openmp have been used.

Figure 23. Comparing Host processors, SandyBridge, using 16 threads, with Xeon Phi MIC co-processor. Fortran 90 code using OpenMP threading. One single MIC processor is using 240 threads. Scatter placement is used.



It is relatively straightforward to set up concurrent runs, workload distribution and ultimately load balancing between the host CPUs and the Xeon Phi MIC coprocessors. However, all of the administration is left to the programmer. Since there is no shared memory the work and memory partition must be explicitly handled. Only the buffers used on the co-processors need to be transferred as memory movement is limited by the PCIe bus bandwidth. There are mechanisms for offline transfer and semaphores for synchronising both transfer and execution. All of this must be explicitly handled by the programmer. While each part is relatively simple it can become quite complex when trying to partition the problem while trying to do load balancing.

15.2.3. Accelerator Benchmark Suite - Synthetic Benchmarks

As part of PRACE-4IP, we have put together a benchmark suite suitable for evaluating accelerators. It is comprised mostly by application benchmarks, but also includes a synthetic benchmark part, based on the Scalable HeterOgeneous Computing (SHOC) benchmark suite (available under [https://github.com/vetter/shoc]). More details about the Accelerator Benchmark Suite will be presented in a separate PRACE deliverable, D7.5 Applications Performance on Accelerators. We will present below the results of running a subset of SHOC under the following settings: an OpenCL version of SHOC for the Knights Corner architecture, an offload version of the benchmarks also for the Knights Corner, and an OpenCL version for a 24-core dual-socket Intel Haswell-based system. The benchamrks from SHOC are organized in three different categories. The Level 0 benchmarks are low-level benchmarks measuring the system throughput. Level 1 benchmarks are basic algorithms, while Level 2 benchmarks are application-level benchmarks. The table below gives a short description of each benchmark with their corresponding level.

Name	Description
PCIe Download (L0)	Measures bandwidth of transferring data across the PCIe bus to a device.
PCIe Readback (L0)	Measures bandwidth of reading data back from a device.

Name	Description
GMEM_readBW (L0)	Measures bandwidth of read memory accesses to the global device memory.
GMEM_writeBW (L0)	Measures bandwidth of write memory accesses to the global device memory.
FFT_sp (L1)	Forward and reverse 1D FFT with single-precision floating-point accuracy.
FFT_dp (L1)	Forward and reverse 1D FFT with double-precision floating-point accuracy.
SGEMM (L1)	Single-precision matrix-matrix multiplication.
DGEMM (L1)	Double-precision matrix-matrix multiplication.
MD (L1)	Computation of the Lennard-Jones potential from molecular dynamics.
MD5Hash (L1)	Computation of many small MD5 digests, heavily dependent on bitwise operations.
Reduction (L1)	Reduction operation on an array of floating-point values.
Scan (L1)	Parallel prefix sum operation on an array with floating-point values.
Sort (L1)	Sorts an array of key-value pairs using a radix sort algorithm.
Stencil2D_sp (L1)	A 9-point stencil operation applied to a 2D data set in single-precision.
Stencil2D_dp (L1)	A 9-point stencil operation applied to a 2D data set in double-precision.
Triad (L1)	A version of the STREAM Triad benchmark that includes PCIe transfer time.
S3D (L2)	A computationally intensive kernel from the S3D turbulent combustion simulation program.

The following table shows the results of running the syntethic benchmarks from the Accelerator Benchmark Suite on both an Intel Xeon Phi 7120P-based system and on a Haswell-based system.

Name	OpenCL KNC	Offload KNC	OpenCL Haswell
PCIe Download(GB/s)	6.8	6.6	12.4
PCIe Readback(GB/s)	6.8	6.7	12.5
GMEM_readBW(GB/s)	49.7	170	20.2
GMEM_writeBW(GB/s)	41	72	13.6
FFT_sp (GFLOPS)	71	135	80
FFT_dp (GFLOPS)	31	69.5	55
SGEMM (GFLOPS)	217	645	554
DGEMM (GFLOPS)	100	190	196
MD (GFLOPS)	33	28	114
MD5Hash (GH/s)	1.7	N/A	1.29
Reduction (GB/s)	10	99	91
Scan (GB/s)	4.5	11	15
Sort (GB/s)	0.11	N/A	0.35

Name	OpenCL KNC	Offload KNC	OpenCL Haswell
Stencil2D_sp (GFLOPS)	8.95	89	34
Stencil2D_dp (GFLOPS)	7.92	16	30
Triad (GB/s)	5.57	5.76	8
S3D (GFLOPS)	18	109	27

16. Intel Xeon Phi Application Enabling

16.1. Acceleration of Blender Cycles Render engine by Intel Xeon Phi

This section was contributed by IT4Innovations, Czech Republic, to represent their experience using Intel Xeon Phi.

Blender is the name of a project, open to everybody who wants to join and has experience with programming or 3D graphics. It's founded by Tom Roosendaal and part of this project is development of open source software. This software could be used to create pictures, movies, animations, and even computer games. In general, we could say that this software could be used in any discipline dealing with 3D graphics, from design through manufacturing of parts to the medical applications such as visualization of 3D models of human organs.

However, the main domain is creation of movies, which is another focus area of this project.

Blender is entirely created in Python which could be beneficial for developing new functions, features, or scripts or to modify the current ones. Because Python is not very suitable for computationally extensive tasks such as rendering, it could be extended using C++ language.

The cycles rendering engine is based on methods simulating real behaviour of light. This is described by a so called rendering equation presented by Kajiya in 1986. Solving the rendering equation is computationally very extensive so effective utilization of accelerators such as Intel Xeon Phi will help to reduce computational time significantly.

The Intel Xeon Phi has different possibilities how the coprocessor could be used. There are four possible modes:

1. Multicore only - this mode uses only the CPU.

2. Multicore Hosted with Many-Core Offload - there is one main process on the CPU host and it calls the function on the Xeon Phi coprocessor using Offload technology (discussed later).

3. Symmetric mode – this mode uses MPI. The CPU and Xeon Phi is on the same level. We don't care about reading data and calling function from/on Xeon Phi.

4. Native – we don't support it. The compiled code can run only on Xeon Phi. For example you can connect via ssh directly to the coprocessor and runs the application in the same way like on a standard computer with Linux operating system.

In our implementation we extended the Blender Cycles engine with support for: OpenMP, MPI and Intel® Xeon Phi™ Offload as well as the Symmetric mode. We call this extension CyclesPhi.

Symmetric mode

In symmetric mode computations on the processor and co-processor are on the same level which means that we treat the co-processor in the same way as another host. Symmetric mode is very simple and there is no need for special code for Intel Xeon Phi. To create applications for symmetric mode several flags have to be used during compilation of the source code. This will open communication between Blender and the client.

Offload mode

Offload mode is more complicated and it could be activated in Blender from its menu. If we wish to execute part of the code on MIC (which could be an OpenMP parallel construct) we need to create a block with the pragma offload syntax. There are two options how to tell the compiler which part of the code whould run on the host and which part on the coprocessor or on both. One can use for example the `pragma offload_attribute(push/pop, target(mic))` command or one can add the flag `-qoffload-attribute-target=mic` to the compiler.

The next set of important offload mode clauses is shown in Figure 24 and Figure 25 (more at https://software.intel.com/en-us/articles/ixptc-2013-presentations [https://software.intel.com/en-us/articles/ixptc-2013-presentations]). To use the offload mode we use the following directives:

Figure 24. Important Offload clauses.

	C/C++ Syntax	Semantics
Offload pragma	#pragma offload <*clauses*> <*statement*>	Execute next statement on MIC (which could be an OpenMP parallel construct)
Function and variable	__declspec (target (mic)) <*func/var*> __attribute__ ((target (mic)) <*func/var*> #pragma offload_attribute (target (mic)) <*func/var*>	Compile function and variable for CPU and MIC

Figure 25. Important offload clauses.

Clauses	Syntax	Semantics
Target specification	target (mic [: <expr>])	Where to run construct
If specifier	if (condition)	Offload statement if condition is TRUE
Inputs	in (var-list modifiers)	Copy CPU to target
Outputs	out (var-list modifiers)	Copy target to CPU
Inputs & outputs	inout (var-list modifiers)	Copy both ways
Non-copied data	nocopy (var-list modifiers)	Data is local to target
Modifiers		
Specify pointer length	length (element-count-expr)	Copy that many pointer elements
Control pointer memory allocation	alloc_if (condition) free_if (condition)	Allocate/free new block of memory for pointer if condition is TRUE

First we have to specify the identification number of the resource we would like to use. For example if we first want to use the first co-processor we use the clause `pragma omp target mic:0.`

To send or receive data to/from co-processor we have to specify a list of variables we would like to enable on this co-processor. Moreover we have to specify for each variable what should be done with it. For example, if we want to copy data from the host to the target before, and again back to the host after the calculation is done, we use the clause `inout`.

Another important parameter is `alloc_if` and `free_if`. Those two commands will allocate or de-allocate memory on the co-processor.

The principle of MIC computing is as follows: the rendered scene is stored in `KernelData` and `KernelTextures` structures. At first the basic information about the scene is sent to the selected compute devices, in this case the OMPDevice, using `const_copy_to method`. After that the objects and textures are saved to the memory of the MIC using the tex_alloc method. In addition, we have to allocate buffers for rendered pixels and initialize the buffer for the random number generator state, the `rng_state`. An example of the implementation of the memory allocation and the data transfers to the co-processor is depicted in Figure 26.

Figure 26. Blender example code.

```
//blender/intern/cycles/kernel/kernels/mic/kernel_mic.cpp
#define ALLOC alloc_if(1) free_if(0)
#define FREE alloc_if(0) free_if(1)
#define REUSE alloc_if(0) free_if(0)
#define ONE_USE
device_ptr mic_alloc_kg(int numDevice) {
   device_ptr kg_bin;
   #pragma offload target(mic:numDevice) out(kg_bin)
    {
        KernelGlobals *kg = new KernelGlobals();
        kg_bin = (device_ptr) kg;
    }
   return (device_ptr) kg_bin;
}
void mic_free_kg(int numDevice, device_ptr kg_bin) {
     #pragma offload target(mic:numDevice) in(kg_bin)
      {
        KernelGlobals *kg = (KernelGlobals *) kg_bin;
        delete kg;
} }
void mic_const_copy(int numDevice, /*…*/)  {
  #pragma offload target(mic:numDevice) \\
  in(host_bin:length(size) ONE_USE) in(kg_bin) in(size)
  {
     KernelGlobals *kg = (KernelGlobals *)kg_bin;
     memcpy(&kg->__data, host_bin, size);
     kg->__data_size = size;
} }
```

```
void mic_mem_alloc(int numDevice, char *mem, size_t memSize) {
  #pragma offload target(mic:numDevice) in(mem:length(memSize) ALLOC)
}
void mic_mem_copy_to(int numDevice, char *mem, size_t memSize, char*
signal_value) {
      if (signal_value == NULL)  {
  #pragma offload target(mic:numDevice) in(mem:length(memSize) REUSE)
      } else  {
  #pragma offload_transfer target(mic:numDevice) in(mem:length(memSize) REUSE)
signal(signal_value)
      }
}
void mic_mem_copy_from(int numDevice, char *mem, size_t offset, size_t memSize,
char* signal_value) {
      if (signal_value == NULL)
      {
  #pragma offload target(mic:numDevice) out(mem[offset:memSize]: REUSE)
      }
      else
      {
  #pragma offload_transfer target(mic:numDevice) out(mem[offset:memSize]: REUSE)
signal(signal_value)
      }
}
void mic_mem_free(int numDevice, char *mem, size_t memSize) {
  #pragma offload target(mic:numDevice) in(mem:length(0) FREE)
}
```

This example shows functions for sending send data from the CPU host to the MIC. For memory allocation on the MIC the clause with `offload target` and `set alloc_if` set to one is used. Because we use complex structures such as `KernelData`, we transfer all data before sending them to the co-processor to the array of bytes. The next step is to decompose the rendered image, or task, into tiles (rule of the thumb on MIC is that the bigger tile is faster). There is one POSIX thread that calls the `path_tracing method` on MIC. The results are stored in the buffer vector and sent back from the MIC to the CPU host.

Example code showing how the tasks are created is depicted in Figure 27.

Figure 27. Blender example code.

```
//blender/intern/cycles/kernel/kernels/mic/kernel_mic.cpp
void mic_path_trace(int numDevice, /*...*/)
{
#pragma offload target(mic:numDevice) \
        in(buffer_bin : length(0) REUSE) \
        in(rng_state_bin : length(0) REUSE) \
        in(sample_finished_mic : length(0) REUSE) \
        in(reqFinished_mic : length(0) REUSE) \
        in(rgba_byte_bin : length(0) REUSE) \
        in(kg_bin) in(start_sample) in(end_sample) \
        in(tile_x) in(tile_y) in(offset) in(stride) \
        in(tile_h) in(tile_w) in(nprocs_cpu) \
        signal(signal_value)
        {
#pragma omp parallel for num_threads(nprocs_cpu) schedule(dynamic, 1)
  for (int i = 0; i < size; i++)
  {
   int y = i / tile_w;
   int x = i - y * tile_w;

   for (int sample = start_sample; sample < end_sample; sample++)
   {
     kernel_path_trace((KernelGlobals *)kg_bin, /*…*/);
} } }
```

```
//blender/intern/cycles/device/device_omp.cpp
omp_set_nested(1);
#pragma omp parallel num_threads(2)          {
#pragma omp single nowait              {
#pragma omp task {
              while (reqFinished == 0) {
                 #pragma omp flush
                 if (omp_path_trace_req != 0)  {
                    cpu_path_trace((KernelGlobals *)kg_bin, /*...*/);
                    omp_path_trace_req = 0;
                 }
                 usleep(100);
              } }
#pragma omp task {
              while (true) {
                 for (int dev = 0; dev < num_devices_cpu_mics; dev++) {
                    if (dev > 0)
                       mic_mem_copy_from(dev - 1, (char*) buffer, /*...*/);
                    if (sample_finished_devices[dev] == end_sample) {
                       if (dev == 0) omp_path_trace_req = 1;
                       else  mic_path_trace(dev - 1, /*...*/);
                 }  }
                 task.update_progress(&tile);
                 //...
              }  }  }
#pragma omp taskwait   }
```

Building Blender 2.77a with Intel Compiler 2016

- https://wiki.blender.org/index.php/Dev:Doc/Building_Blender
- Intel® Parallel Studio XE Cluster Edition (free for students)
- Intel® Manycore Platform Software Stack (Intel® MPSS)
- Microsoft Visual Studio 2013 (Windows) or Netbeans 8.1, CMake 3.3, GCC 5.3 (Linux)
- Libraries (GCC/ICC): boost (./bjam install toolset=intel), ilmbase, openexr, tiff, openimageio, zlib, Python, …

Building CyclesPhi with Intel Compiler 2016

git clone git@code.it4i.cz:blender/cyclesphi.git

new build flags:

blender:

- WITH_IT4I_MIC_OFFLOAD=ON/OFF
- WITH_IT4I_MPI=ON/OFF
- WITH_OPENMP=ON/OFF (old flag)

client:

- WITH_IT4I_MIC_NATIVE=ON/OFF
- WITH_IT4I_MIC_OFFLOAD=ON/OFF

new folders:

it4i/client

- api/client_api.h – main header with predefined communication tags and structures
- cycles_mic – the shared libraries for rendering on Intel Xeon Phi
- cycles_mpi – the shared libraries for communication with Blender (root)
- cycles_omp - the shared libraries for rendering on CPU using OpenMP
- main – blender_client application

it4i/scripts (created for Salomon supercomputer with PBS job management system)

- build_lib.sh – build some basic libraries (boost, …)
- build_blender.sh – build Blender CyclesPhi, which supports Intel Xeon Phi
- run_blender.sh – run Blender CyclesPhi without MPI
- run_mpi.sh – run Blender CyclesPhi with MPI support

Run CyclesPhi

New scripts in it4i/scripts (created for Salomon supercomputer):

- run_blender.sh – run Blender CyclesPhi without MPI: ./${ROOT_DIR}/install/blender/Blender
- run_mpi.sh – run Blender CyclesPhi with MPI (only CPU / MIC‘s offload mode): mpirun -n 1 ${ROOT_DIR}/install/blender/blender : -n 1 ${ROOT_DIR}/install/blender_client/bin/blender_client
- run_mpi_mic.sh – run Blender CyclesPhi with MPI support with Symmetric mode: mpirun -genv LD_LIBRARY_PATH $MIC_LD_LIBRARY_PATH -machine $NODEFILECN -n 1 ${ROOT_DIR}/install/blender/blender : -n $NUMOFCN ${ROOT_DIR}/install/blender_client/bin/blender_client

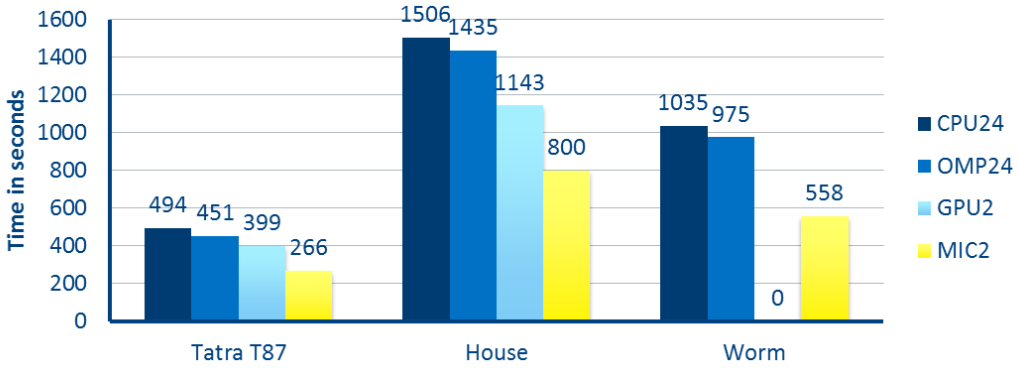
Benchmarking

Three benchmarks were performed. The benchmarks were run on one compute node of the Salomon supercomputer equipped with two Intel Xeon E5-2680v3 CPUs (24 cores) and two Intel Xeon Phi 7120P. GPU tests were run on

two NVIDIA GeForce GTX 970. For testing purposes three different scenes were chosen. The first one is Tatra T87 car with HDRI light. The second one is a scene with a house and a water pool. The realistic water surface increases the rendering time. The last scene is taken from Laundromat movie that we created using Blender software. The Worm scene has blur background which increases the rendering time as well. To show performance of our MIC implementation we perform tests on different devices.

Specifically, the original version of the renderer (CPU24) was run on two Intel Xeon CPUs (24 cores) of a Salomon compute node. The MIC2 version run on two Intel Xeon Phi accelerators and two Intel Xeon CPUs (full utilization of a compute node). The OMP24 version run on two Intel Xeon CPUs (24 cores). The GPU2 version of the code run on two NVIDIA GeForce accelerators. Figure 28 shows rendering times on specified devices for three different scenes. As can be seen the fastest rendering is achieved by MIC2. This is true in all tested scenes. Speedup is almost 50% compared to original renderer's implementation running on 24 CPU cores. There is also about 30% speedup in case of MIC2 compared to the GPU2 version. The Worm scene could not be computed on the GPU device. This is the reason why there is the zero value in the graph. Comparison is depicted on Figure 28 .

Figure 28. Performance comparison.



16.2. The BM3D Filter on Intel Xeon Phi

This section was contributed by IT4Innovations, Czech Republic, to represent their experience using Intel Xeon Phi.

16.2.1. BM3D filtering method

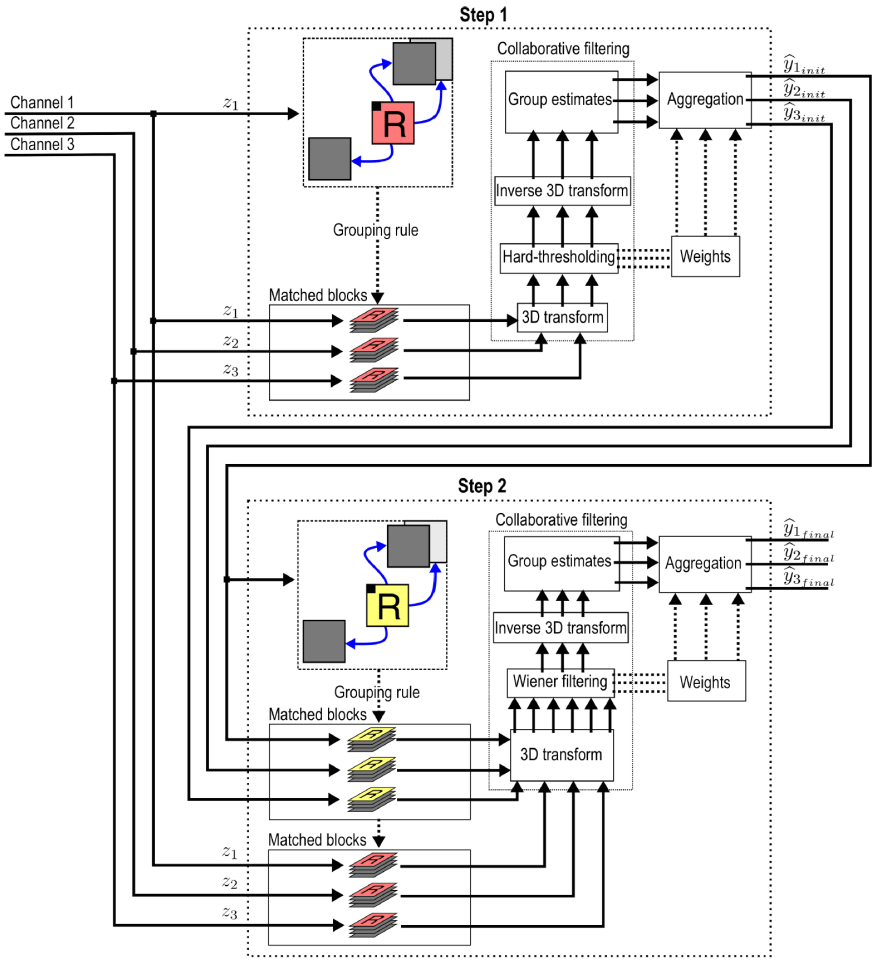
The block-matching and the 3D collaborative filtering method operate over the image trying to minimize the amount of noise based on sparsity of similar image blocks. Generally, the noise is assumed as Additive White Gaussian Noise (AWGN) that has been added to the original noiseless image.

The detailed description of the BM3D method is covered in [52]. The color version of the method is presented in [53].

BM3D is a two-step method. The first-step image is divided into several overlapping areas where similar smaller parts of the image called patches are searched for. Searching for the similar patches is done in a sparse domain provided by wavelet transform of each patch. The found similar patches are stacked in the 3D array and the whole stack is transformed from image to sparse representation by 3D wavelet transform. Here the filtering operation in the form of the hard thresholding is performed, and then the stack is back transformed to the image domain. Each patch from the filtered stack is redistributed back to its position within the image and all overlapping pixels are aggregated and averaged out by weighted average. Summation goes through all the patches within one stack and through all the stacks within one image.

The filtering steps of the method are graphically summarized in Figure 29.

Figure 29. Workflow of the collaborative filtering method



16.2.2. BM3D - computationally extensive parts

The BM3D method operates in sparse domain, where all the filtering is performed either by hard-thresholding in Step 1 or by Wiener filtering in Step 2. Conversion to sparse domain is done by matrix multiplication of each selected image patch with transformation matrices. The number of patches that are transformed and then further processed is high. Based on the recommended setting of the method, as elaborated in [54] and [55], it is around 1300 patches for every single reference patch. The number of the reference patches depends on the image resolution, but generally it is about 1/16th of the number of image pixels. In the case of rendering tasks, image resolution is typically HD (1920x1080 pixels) or higher, which gives approximately 130 000 of reference patches at least. Summing it all up, it gives around 340 000 000 of image patches that are being processed just in Step 1. Similar counting holds also for the Step 2. Luckily computations can be parallelized around each reference patch. Limitations are set if one tries for concurrency between Step 1 and Step 2. It is not possible since the results of Step 1 are used right at the start of Step 2.

16.2.3. Parallel implementation of BM3D filter

To achieve the best possible implementation in terms of the algorithm speed, we have implemented it in C++. We have used OpenMP standard with its #pragma directives for parallel programming and SIMD directives for vectorization.

We use our own vectorized code for operations with matrices (summation, subtraction, multiplication, norm) because it performs better than Eigen or MKL libraries on small matrices as we have tested. After that, we have tried to use implicit vectorization on matrix operations. As a further step, we have used SIMD vectorization, which brings another processing speed-up. By profiling the code there was still quite large bottleneck in matrix

multiplications. This was solved by direct assembler implementation, which helps especially on MIC architecture, as can be seen in results. The assembler code was generated by library for small matrix multiplication (XSMM). The examples of source code are shown in figures bellow.

Figure 30. The source code of multiplication generated by XSMM for KNC

```
#ifdef __MIC__
void mul_8_8_8_knc(const float* A, const float* B, float* C) {
//#ifdef __MIC__
  __asm__ __volatile__("movq %0, %%rsi\n\t"
                       "movq %1, %%rdi\n\t"
                       "movq %2, %%rdx\n\t"
                       "movq $0, %%r12\n\t"
                       "movq $0, %%r13\n\t"
                       "movq $0, %%r14\n\t"
                       "movq $255, %%r11\n\t"
                       "kmov %%r11d, %%k1\n\t"
                       "vpxord %%zmm24, %%zmm24, %%zmm24\n\t"
                       "vprefetch1 0(%%rdx)\n\t"
                       "vpxord %%zmm25, %%zmm25, %%zmm25\n\t"
                       "vprefetch1 32(%%rdx)\n\t"
                       "vpxord %%zmm26, %%zmm26, %%zmm26\n\t"
                       "vprefetch1 64(%%rdx)\n\t"
                       "vpxord %%zmm27, %%zmm27, %%zmm27\n\t"
                       "vprefetch1 96(%%rdx)\n\t"
                       "vpxord %%zmm28, %%zmm28, %%zmm28\n\t"
                       "vprefetch1 128(%%rdx)\n\t"
                       "vpxord %%zmm29, %%zmm29, %%zmm29\n\t"
                       "vprefetch1 160(%%rdx)\n\t"
                       "vpxord %%zmm30, %%zmm30, %%zmm30\n\t"
                       "vprefetch1 192(%%rdx)\n\t"
                       "vpxord %%zmm31, %%zmm31, %%zmm31\n\t"
                       "vprefetch1 224(%%rdx)\n\t"
                       "vloadunpacklps 0(%%rdi), %%zmm0%{%%k1%}\n\t"
                       "vloadunpackhps 64(%%rdi), %%zmm0%{%%k1%}\n\t"
                       "vprefetch0 32(%%rdi)\n\t"
                       "vfmadd231ps 0(%%rsi)%{1to16%}, %%zmm0, %%zmm24\n\t"
                       "vprefetch1 64(%%rdi)\n\t"
                       "vfmadd231ps 32(%%rsi)%{1to16%}, %%zmm0, %%zmm25\n\t"
                       "vprefetch0 64(%%rsi)\n\t"
                       "vfmadd231ps 64(%%rsi)%{1to16%}, %%zmm0, %%zmm26\n\t"
```

Figure 31. The macro is used bellow. The value of alignment is for KNC (128 is faster than 64) or for AVX2.

```
#if defined(__MIC__)
#   define ALIGNMENT 128
#else
#   define ALIGNMENT 32
#endif

#define DECLSPEC_ALIGN __declspec(align(ALIGNMENT))
#define STATIC_INLINE static inline
```

Figure 32. The matrix multiplication with SIMD

```
STATIC_INLINE void mulSIMD(
    DATA_TYPE *res, int res_rows, int res_cols,
    DATA_TYPE *a, int a_rows, int a_cols,
    DATA_TYPE *b, int b_rows, int b_cols,
    int res_size)
{
    setValued(res, 0, res_size);

#pragma omp simd
    for (int i = 0; i < a_rows; i++)
    {
        for (int k = 0; k < b_cols; k++)
        {
            for (int j = 0; j < a_cols; j++)
            {
                res[i + k * res_rows] = res[i + k * res_rows] + a[i + j * a_rows] * b[j + k * b_rows];
            }
        }
    }
}
```

Figure 33. The matrix multiplication with MKL libraries

```
STATIC_INLINE void mulMKL(
    DATA_TYPE *res, int res_rows, int res_cols,
    DATA_TYPE *a, int a_rows, int a_cols,
    DATA_TYPE *b, int b_rows, int b_cols,
    int res_size)
{
    DATA_TYPE *A = a, *B = b, *C = res;
    int m = a_rows, n = b_cols, k = a_cols;
    DATA_TYPE alpha = 1.0, beta = 0.0;

    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, n, k, alpha, A, k, B, n, beta, C, n);
}
```

Figure 34. The matrix multiplication with Eigen libraries

```
void mulEigen()
{
    Matrix2d A, B, C;
    A << 1, 2, 3, 4;
    B << 1, 2, 3, 4;

    C = A * B;
}
```

Figure 35. The adding of two matrices with SIMD vectorization

```
STATIC_INLINE void add(DATA_TYPE *res, DATA_TYPE *a, DATA_TYPE *b, int res_size)
{
#pragma omp simd aligned(res:ALIGNMENT) aligned(a:ALIGNMENT) aligned(b:ALIGNMENT)
    for (int i = 0; i < res_size; i++)
    {
        res[i] = a[i] + b[i];
    }
}
```

Figure 36. The norm of matrix with SIMD vectorization

```
STATIC_INLINE DATA_TYPE norm(DATA_TYPE *a, int a_size)
{
    DATA_TYPE sum = 0;
#pragma omp simd reduction(+:sum)
    for (int i = 0; i < a_size; i++)
    {
        sum = sum + a[i] * a[i];
    }
    return sqrt(sum);
}
```

16.2.4. Results

Tatra scene was used as examples for the tests of filter processing time. We have tested the filter for the set of sampling. Specific sampling does not influence the filter runtime directly, but what does is the value of sigma. There are also shown measurements on different architectures (CPU, MIC) with specific optimizations (AVX2, AVX2+SIMD, AVX2+xsmm+SIMD, knc-offload, knc-offload+SIMD, knc-offload+xsmm+SIMD).

Our speed improvements by different optimization of the filtering algorithm are in graphical form represented in Figure 40.

All the tests were performed on one computing node of Salomon supercomputer. Specifically on 2x Intel Xeon E5-2680v3, 2.5GHz for CPU tests and 1x Intel Xeon Phi 7120P, 61cores for MIC tests.

Figure 37. Rendered scene of Tatra car without filtering; from left to right- 64 samples/pixel, 1 sample/pixel

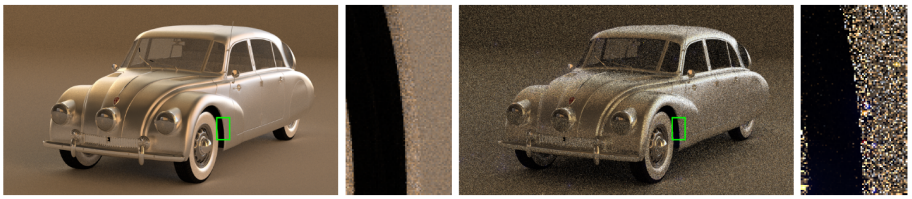


Figure 38. Rendered scene of Tatra car with filtering; from left to right- 64 samples/pixel, 1 sample/pixel

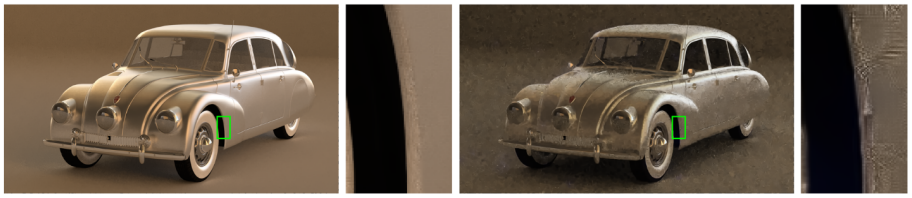
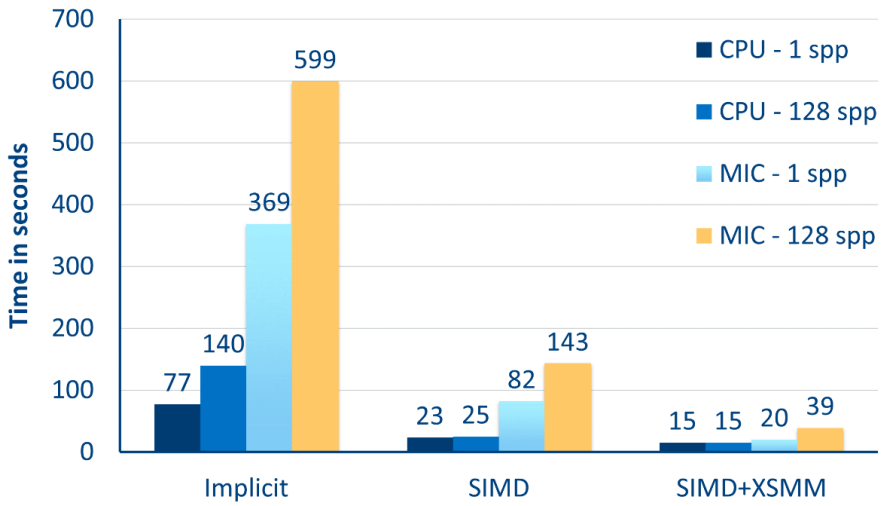


Figure 39. Tatra - BM3D runtime for different optimizations and architectures. Comparison of our implementation with original version of filter by Dabov et all for two different values of noise (sigma)

Filter runtime	Ours [s]		Dabov et al. [s]	
Arch., Inst. set	$\sigma > 40$	$\sigma \leq 40$	$\sigma > 40$	$\sigma \leq 40$
CPU, AVX2	76.860	139.539	**107.436**	**66.389**
CPU, AVX2+SIMD	22.973	24.709		
CPU, AVX2+xsmm+SIMD	**12.739**	**14.974**		
MIC, knc-offload	369.607	598.845		
MIC, knc-offload+SIMD	81.569	142.315		
MIC, knc-offload+xsmm+SIMD	**19.844**	**38.683**		

Figure 40. Tatra - BM3D runtime for different optimizations and architectures. Comparison of our implementation with original version of filter by Dabov et all for two different values of noise (sigma)



16.3. GEANT4 and GEANTV on Intel Xeon Phi Systems

This section was contributed by NCSA, Bulgaria, to represent their experience using Intel Xeon Phi.

16.3.1. Introduction

GEANT4 [http://geant4.cern.ch/] and GEANTV [http://geant.cern.ch/] (GEometry ANd Tracking) are Monte Carlo packages for simulation of the passage of particles through matter. This is an important task for several scientific areas, ranging from fundamental science (high-energy physics experiments, astrophysics and astroparticle physics, nuclear investigations) to applied science (radiation protection, space science), biology and medicine (DNA investigations, hadron therapy, medical physics and medical imaging), and education.

GEANT4 and GEANTV are complex software libraries meant to be linked to a user-specific code. These libraries require a number of additional libraries and packages to be configured and installed in a specific way in advance. Here we discuss the installation of GEANT4 and GEANTV libraries on hybrid architectures with Intel Xeon Phi co-processors [http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html]. In addition, we give some guidelines for the configuration, compilation and running the user-specific codes to be linked to these libraries. For file transfer, one can use Gsatellite [https://github.com/fr4nk5ch31n3r/gsatellite/wiki/Gsatellite-ex-

plained] (see “USE CASE – GSATELLITE” for details and discussion of possible advantages/disadvantages of this set of tools).

As a test system was used Avitohol [http://www.hpc.acad.bg/system-1/], with LINPACK [https://software.intel.com/en-us/articles/intel-linpack-benchmark-download-license-agreement] performance of 264.2 TFlop/s and operating system Red Hat Enterprise Linux release 6.7 [https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/6.7_Release_Notes/]. The coprocessors run MPSS version 3.6.

16.3.2. GEANT4 pre-requisites

1. GEANT4 version 10.0 patch-04
2. Intel C/C++ version 16.X. or higher. We have used ICC and ICPP version 16.0.2 (compatible with gcc version 4.4.7)
3. Intel MPSS (Manycore Platform Support Stack) version 3.4
4. CMake [https://cmake.org/] version 3.3 or higher
5. C++ Compiler and Standard Library supporting the C++11 Standard
6. Linux: GNU Compiler Collection [https://gcc.gnu.org/] 4.8.2 or higher

16.3.3. GEANT4 configuration and installation

Setting-up directories and downloading the code:

```
export  GEANT_DIR=/opt/soft/geant4/mic/non-mic
cd $GEANT_DIR
wget https://geant4.web.cern.ch/geant4/support/source/geant4.10.00.p04.tar.gz
tar -xzvf geant4.10.00.p04.tar.gz
rm geant4.10.00.p04.tar.gz
mkdir geant4.10.00.p04-build geant4.10.00.p04-install geant4.10.00.p04-data
cd geant4.10.00.p04-build/
```

It is worth mentioning that the only version of GEANT4 which can be compiled with Intel version 16.X compiler is 4.10.00. Compilation of older or newer versions results in various compilation errors.

GEANT4 needs several databases with physical/interaction constants. The databases can be downloaded from http://geant4.cern.ch/support/source/ in the geant4.10.00.p04-data folder.

Setting-up cmake, compilers and linkers:

The compiler and linker have to be set with appropriate environment variables. In what follows, we use bash [https://www.gnu.org/software/bash/manual/] shell, but the compilation procedure can be carried out under any other shell (tcsh for instance).

The shell commands listed below set the compilers (in our case ICC and ICPP version 16.0.2, which are compatible with gcc version 4.4.7) and MPSS (in our case version 3.6-1) as well as the linker (GNU LD) and archiver (GNU AR - needed to create libraries). Both the linker and the archiver are part of GNU Binutils [https://www.gnu.org/software/binutils/]version 2.22.52.20120302.

```
/opt/intel/compilers_and_libraries_2016.2.181/linux/bin/iccvars.sh intel64 \
      -arch intel64 -platform linux
export CC=/opt/intel/compilers_and_libraries_2016.2.181/linux/bin/intel64/icc
export CXX=/opt/intel/compilers_and_libraries_2016.2.181/linux/bin/intel64/icpc
export LD=/usr/linux-k1om-4.7/bin/x86_64-k1om-linux-ld
```

```
export AR=/usr/linux-k1om-4.7/bin/x86_64-k1om-linux-ar
export PATH=/opt/soft/geant4/mic/cmake-3.3.2/bin/:${PATH}
export
LD_LIBRARY_PATH=/opt/intel/compilers_and_libraries_2016.2.181/linux/
       compiler/lib/intel64_lin
```

These environment variables are needed also for compiling the applications. For convenience, we have gathered them in geant4_compilers_setup_nonmic.sh, to be sourced before user code compilation.

The file mic-toolchain-file.cmake has to be created, containing the following configuration:

```
# this one is important
SET(CMAKE_SYSTEM_NAME Linux)
#this one not so much
SET(CMAKE_SYSTEM_VERSION 1)
# specify the cross compiler
SET(CMAKE_C_COMPILER /opt/intel/compilers_and_libraries_2016.2.181/linux/bin/
       intel64/icc)
SET(CMAKE_CXX_COMPILER /opt/intel/compilers_and_libraries_2016.2.181/linux/bin/
       intel64/icpc)
# where is the target environment
SET(CMAKE_FIND_ROOT_PATH  /opt/intel/compilers_and_libraries_2016.2.181/linux)
# search for programs in the build host directories
SET(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
# for libraries and headers in the target directories
SET(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
SET(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
```

The information in the .cmake file seems to repeat the information set in the environment variables, but this redundancy is necessary to instruct the cmake utility for the compiler and binutils, to prevent cmake using the standard compiler installed on the machine, neglecting the environment variables.

```
cp mic-toolchain-file.cmake $GEANT_DIR
cd $GEANT_DIR/geant4.10.00.p04-build/

cmake  -DGEANT4_BUILD_MULTITHREADED=ON -DGEANT4_USE_SYSTEM_EXPAT=OFF \
-DGEANT4_USE_FREETYPE=ON -DGEANT4_INSTALL_DATA=OFF \
-DGEANT4_INSTALL_DATADIR=../geant4.10.00.p04-data/  -DCMAKE_C_COMPILER=${CC} \
-DCMAKE_CXX_COMPILER=${CXX} -DCMAKE_LINKER=${LD} -DCMAKE_AR=${AR} \
-DCMAKE_TOOLCHAIN_FILE=../mic-toolchain-file.cmake -DBUILD_SHARED_LIBS=OFF \
-DBUILD_STATIC_LIBS=ON -DCMAKE_INSTALL_PREFIX=../geant4.10.00.p04-install \
../geant4.10.00.p04
```

GEANT4 compilation takes some time, so it is convenient using for this all the cores. The location of the database should also be set.

```
basepath=$GEANT_DIR/geant4.10.00.p04-data
export G4LEVELGAMMADATA=${basepath}/PhotonEvaporation3.0
export G4NEUTRONXSDATA=${basepath}/G4NEUTRONXS1.4
export G4LEDATA=${basepath}/G4EMLOW6.35
export G4NEUTRONHPDATA=${basepath}/G4NDL4.4
export G4RADIOACTIVEDATA=${basepath}/RadioactiveDecay4.0
export G4ABLADATA=${basepath}/G4ABLA3.0
export G4PIIDATA=${basepath}/G4PII1.3
export G4SAIDXSDATA=${basepath}/G4SAIDDATA1.1
```

```
export G4REALSURFACEDATA=${basepath}/RealSurface1.0
```

The user code has to be compiled and linked with GEANT:

```
#setting up compiler and binutils
source  geant4_compilers_setup_nonmic.sh
cd $WORK_DIR
cmake -DCMAKE_LINKER=${LD} -DCMAKE_AR=${AR} \
-DCMAKE_TOOLCHAIN_FILE=/opt/soft/geant4/mic/non-mic/geant4.10.00.p04/
    mic-toolchain-file.cmake  \
    -DGeant4_DIR=$GEANT_DIR/geant4.10.00.p04-install $APPLICATION_SOURCE
make -j $NCORES
```

with WORK_DIR and APPLICATION_SOURCE variables pointing to the folder where the executable will be created and to the user source code respectively.

The compiled code can be used on the CPU or on the Xeon Phi in offload mode. It is not suitable for execution in native mode. The native code cross compilation is guided in the next chapter.

16.3.4. GEANT4 cross compilation as a native Xeon Phi application

The native mode cross compilation fof Xeon Phi does not differ too much from the usual compilation, only the differences from the CPU compilation will be highlighted.

To instruct the compiler to generate Xeon Phi native code, some additional environment variable should be set:

```
export LDFLAGS=-mmic
export CXXFLAGS=-mmic
export CFLAGS=-mmic
```

The difference is that the variables LDFLAGS, CXXFLAGS and CFLAGS are set to option -mic, which instructs the compilers and the linker to cross-compile to native Xeon Phi code. Again, we recommend this setup to be saved in a file (geant4_compilers_setup.sh) to be sourced prior to code compilation.

16.3.5. GEANTV installation

GEANTV needs many specific packages to be pre-installed, most of them to be compiled and installed from scratch. The installation order is determined by the package dependences and the compilation options have to be explicitly specified. The default system compiler on Red Hat Enterprise Linux release 6.7 is not sufficient to compile GEANTV and even the GNU compiler should be compiled from scratch, together with some libraries needed by the compiler. Any version above 4.8.0 should be sufficient. We have tested that version 4.8.5 works.

Before GEANTV compilation, the following packages should be properly configured and installed:

- ROOT6 [https://root.cern.ch/]
- Vc [https://github.com/VcDevel/Vc]
- VecGeom [https://gitlab.cern.ch/VecGeom/VecGeom]
- Pythia8 [http://home.thep.lu.se/~torbjorn/Pythia.html]
- HepMC3 [https://hepmc.web.cern.ch/hepmc/]
- Xerces-c [http://xerces.apache.org/xerces-c/]

- GEANT4 [https://geant4.web.cern.ch/geant4/]

16.3.6. GEANT4 user code compilation and execution on CPU

User code compilation: Supposing the user code is in the directory pointed by the environment variable $USR_CODE and the directory to build and store executable is pointed by $USR_BUILD, the compilation instructions are:

```
#setting up cmake, compiler and compiler flags
/opt/intel/compilers_and_libraries_2016.2.181/linux/bin/iccvars.sh intel64 \
      -arch intel64 -platform linux
export CC=/opt/intel/compilers_and_libraries_2016.2.181/linux/bin/intel64/icc
export CXX=/opt/intel/compilers_and_libraries_2016.2.181/linux/bin/intel64/icpc
export LD=/usr/linux-k1om-4.7/bin/x86_64-k1om-linux-ld
export AR=/usr/linux-k1om-4.7/bin/x86_64-k1om-linux-ar
export PATH=/opt/soft/geant4/mic/cmake-3.3.2/bin/:${PATH}
#export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/intel/
       compilers_and_libraries_2016.2.181/linux/compiler/lib/intel64_lin
export LD_LIBRARY_PATH=/opt/intel/compilers_and_libraries_2016.2.181/
       linux/compiler/lib/intel64_lin

cd $USR_BUILD

cmake -DCMAKE_LINKER=${LD} -DCMAKE_AR=${AR} \
    -DCMAKE_TOOLCHAIN_FILE=/opt/soft/geant4/
          /mic/non-mic/geant4.10.00.p04/mic-toolchain-file.cmake \
   -DGeant4_DIR=/opt/soft/geant4/mic/non-mic/geant4.10.00.p04/
           /geant4.10.00.p04-build $USR_CODE
```

Running GEANT4 user code on CPU: To run the code one needs to set the necessary variables first and then to execute the program. Here is a generic example:

```
#set GEANT_DIR environment variable
export  GEANT_DIR=/opt/soft/geant4/geant4.10.00.p04

#set variables to point to the databases
basepath=$GEANT_DIR/geant4.10.00.p04-data
export G4LEVELGAMMADATA=${basepath}/PhotonEvaporation3.0
export G4NEUTRONXSDATA=${basepath}/G4NEUTRONXS1.4
export G4LEDATA=${basepath}/G4EMLOW6.35
export G4NEUTRONHPDATA=${basepath}/G4NDL4.4
export G4RADIOACTIVEDATA=${basepath}/RadioactiveDecay4.0
export G4ABLADATA=${basepath}/G4ABLA3.0
export G4PIIDATA=${basepath}/G4PII1.3
export G4SAIDXSDATA=${basepath}/G4SAIDDATA1.1
export G4REALSURFACEDATA=${basepath}/RealSurface1.0
export G4ENSDFSTATE=${basepath}/G4ENSDFSTATE1.0

# entering the directory with the user code executable
cd ~/user_executable/B2a

#set the number of threads
export G4FORCENUMBEROFTHREADS=32

# launch the user application (exampleB2a) with the configuration file
# (configuration.in)
./exampleB2a configuration.in
```

16.3.7. GEANT4 user code compilation and execution on XeonPhi in native mode

User code compilation: The user code is in the directory pointed by the environment variable $USR_CODE and the directory to build and store executable is pointed by $USR_BUILD:

```
#setting up cmake, compiler and compiler flags
/opt/intel/compilers_and_libraries_2016.2.181/linux/bin/iccvars.sh intel64 \
      -arch intel64 -platform linux
export CC=/opt/intel/compilers_and_libraries_2016.2.181/linux/bin/intel64/icc
export CXX=/opt/intel/compilers_and_libraries_2016.2.181/linux/bin/intel64/icpc
export LD=/usr/linux-k1om-4.7/bin/x86_64-k1om-linux-ld
export AR=/usr/linux-k1om-4.7/bin/x86_64-k1om-linux-ar
export LDFLAGS=-mmic
export CXXFLAGS=-mmic
export CFLAGS=-mmic
export PATH=/opt/soft/geant4/mic/cmake-3.3.2/bin/:${PATH}

cd $USR_BUILD
cmake -DCMAKE_LINKER=${LD} -DCMAKE_AR=${AR} -DCMAKE_TOOLCHAIN_FILE= \
   /opt/soft/geant4/mic/geant4.10.00.p04/mic-toolchain-file.cmake  \
  -DGeant4_DIR=/opt/soft/geant4/mic/geant4.10.00.p04/geant4.10.00.p04-build \
  $USR_CODE
```

The difference form CPU case is that the flags LDFLAGS,CXXFLAGS,CFLAGS are set to -mmic option.

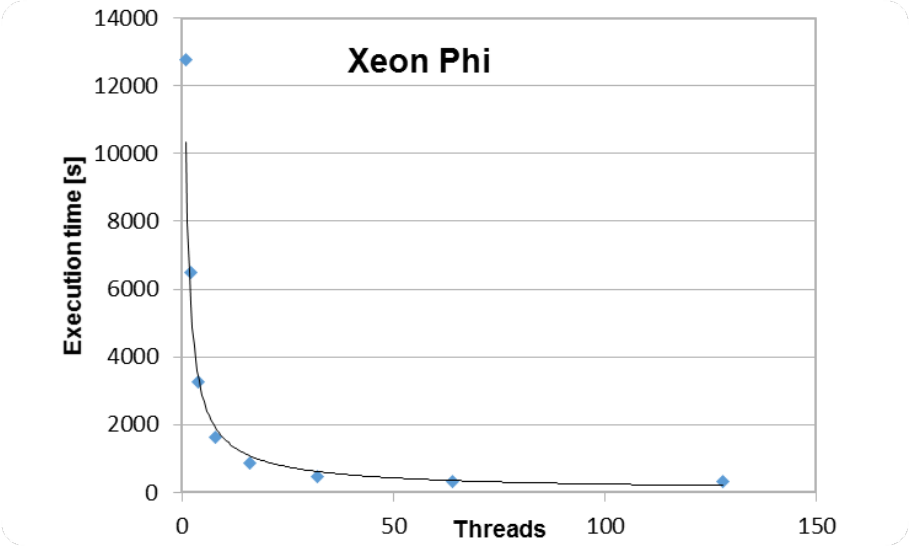
Running in native mode on Xeon Phi: One way to run a native application is to connect to the co-processor by ssh, to set the necessary environment and to execute the code directly. Here is an example:

```
#ssh to mic0 on sl051 host
ssh sl051-mic0
#set GEANT_DIR environment variable
export  GEANT_DIR=/opt/soft/geant4/mic/geant4.10.00.p04

#set variables to point to the databases
basepath=$GEANT_DIR/geant4.10.00.p04-data
export G4LEVELGAMMADATA=${basepath}/PhotonEvaporation3.0
export G4NEUTRONXSDATA=${basepath}/G4NEUTRONXS1.4
export G4LEDATA=${basepath}/G4EMLOW6.35
export G4NEUTRONHPDATA=${basepath}/G4NDL4.4
export G4RADIOACTIVEDATA=${basepath}/RadioactiveDecay4.0
export G4ABLADATA=${basepath}/G4ABLA3.0
export G4PIIDATA=${basepath}/G4PII1.3
export G4SAIDXSDATA=${basepath}/G4SAIDDATA1.1
export G4REALSURFACEDATA=${basepath}/RealSurface1.0
export G4ENSDFSTATE=${basepath}/G4ENSDFSTATE1.0

export LD_LIBRARY_PATH=/opt/intel/lib/mic:$LD_LIBRARY_PATH

# entering the directory with the user code executable
cd ~/mic_user_executable/B2a

#set the number of threads
export G4FORCENUMBEROFTHREADS=32

# launch the user application (exampleB2a) with the configuration file
```

```
# (configuration.in)
./exampleB2a configuration.in
```

The Figure below shows the acceleration with the number of threads in the execution of a particular GEANT4 example on Xeon Phi in native mode.

Figure 41. Program execution time vs the number of threads (128k events).



17. European Intel Xeon Phi based systems

17.1. Avitohol @ BAS (NCSA)

17.1.1. Introduction

In June 2015, the new supercomputer Avitohol was deployed at IICT-BAS. The cluster consists of 150 HP Cluster Platform SL250S GEN8 servers.

Each compute node has 2 Intel Xeon CPU E5-2650 v2 @ 2.6 GHz (8 cores each), 64GB RAM and 2 Intel Xeon Phi 7120P coprocessors (with 61 cores each). Thus, in total there are 20700 cores.

Additionally, there are 4 I/O nodes to serve the 96 TB of disk storage.

The Avitohol cluster currently has the following capabilities: theoretical peak performance of 412.32 TFlop/s, total memory 9600 GB and total disk storage of 96 TB.

It was ranked at 389th place on the November 2015 Top 500 list (http://www.top500.org) with LINPACK performance of 264.2 TFlop/s.

Figure 42. The Avitohol HPC system.

17.1.2. System Architecture / Configuration

The current system configuration includes:

- 150 dual-socket nodes HP ProLiant SL250S Gen8.
- 1 of them reserved for login/job submission and application development.
- 4 I/O nodes HP ProLiant DL380p Gen8 with 2 Intel Xeon CPU E5-2650 v2, 64GB RAM, Fibre Channel cards.
- 2 management nodes HP ProLiant DL380p Gen8 with 2 Intel Xeon CPU E5-2650 v2, 64GB RAM.
- 96 TB of online disk space, connected with Fibre Channel with the I/O nodes.
- Fully non-blocking 56Gbps FDR InfiniBand network interconnecting all the nodes above.

The configuration of the computing nodes is as follows:

- Processors: dual Intel Xeon 8-core CPU E5-2650 v2 @ 2.6 GHz,

- Coprocessors: two Intel Xeon Phi 7120P coprocessors, 16 GB RAM and 61 cores each one
- Main memory of the compute nodes: 64 GB (9.6 TB in total)
- Memory of the accelerators: 16 GB (4.8 TB in total)

The operating system on Avitohol is Red Hat Enterprise Linux release 6.7.

The coprocessors run MPSS version 3.6-1.

17.1.2.1. Filesystems

Currently the only filesystem that is read-write for all users is the `/home` filesystem, which is of type Lustre. Groups of users may request to have a shared directory to share data between them.

Some software that is made available to all users is available under `/opt` and is typically deployed using environment modules. This filesystem is shared via NFS and is read-only for users.

Currently the Lustre filesystem is provided from only two OSTs. This effectively means that one big file may be served from two storage servers, if this file is created in appropriate way.

To convert a directory to “stripe” files in this way one can do the following:

```
lfs setstripe <filename|dirname> --count 2
```

to enforce a file or directory to use striping with 2 OSTs in parallel.

Setting the count to two may be advisable if high I/O rates are expected for files in a given directory.

The Lustre filesystem is optimized for processing of large files. This means that creating large amount of small files may be relatively slow process. However, one can use the /dev/shm filesystem on each node, which is in the RAM, if fast processing of high number of small files is necessary. This filesystem is not shared between nodes and does not survive system restarts! For most users its use is not needed.

Further reading

For more information about the Lustre filesystem, consult the Lustre documentation. See, e.g., https://wiki.hpdd.intel.com/display/PUB/HPDD+Wiki+Front+Page

Further details

Applications with high I/O requirements can request dedicated storage.

17.1.3. System Access

17.1.3.1. Remote access

The gateway node for accessing Avitohol is the node **gw.avitohol.acad.bg**. The key fingerprints are currently:

1024 cd:97:16:41:2c:cd:3d:59:c4:f0:d2:67:ce:1c:5e:62 /etc/ssh/ssh_host_dsa_key.pub (DSA)

2048 b9:e3:6f:f5:a2:ea:8f:95:0a:96:5b:55:03:0c:bf:e9 /etc/ssh/ssh_host_rsa_key.pub (RSA)

For security reasons it is advisable for users to use keys with at least 2048 bits.

Access can be done with ssh protocol.

The gateway node has identical hardware and software configuration (as much as possible) as the execution nodes.

Only this node can be used for launching jobs.

Once inside the cluster, the user can login with ssh to any of the execution nodes where he or she has a running job. It is also possible to login to the Xeon Phi accelerators that are associated with such nodes.

17.1.3.2. Using compute resources on Avitohol

The login node **gw.avitohol.acad.bg** is mainly intended for editing, compiling and submitting compute-intensive programs. Some lightweight testing of parallel programs is allowed. In case of doubt, users are encouraged to submit a job requesting one whole compute node and use it for debugging and development.

The execution nodes are named **sl001-sl150**.

17.1.3.3. Interactive debug runs

If you need to debug your program code you may login to the node **gw.avitohol.acad.bg** and run your code interactively. This node has two CPUs and two Xeon Phi coprocessors similar to the execution nodes.

Please monitor resource utilization and in case the system becomes overloaded (as can be seen from running top) cancel the debug runs and perform them via job submission on an execution node, where you will get dedicated access. Users who cause system crashes of the node or system crashes or hangs on the Xeon Phi coprocessors should try to resolve the issues by codes using execution nodes allocated by the batch system. If the problems persist they are encouraged to contact system administrators.

The support email is:

avitohol-support@parallel.bas.bg.

Further details

There are other ways of accessing storage from the system, which do not concern most users. If your storage requirements exceed one terabyte, please describe them fully in the access form. Access to different interfaces can be provided based on evaluation of these needs.

The Xeon Phi coprocessors are visible as **sl001-mic0** to **sl150-mic0** and **sl001-mic1** to **sl150-mic1**. Users are expected to use them only after obtaining exclusive access to the corresponding execution node.

17.1.4. Production Environment

17.1.4.1. Batch System

The login node **gw.avitohol.acad.bg** of the Avitohol HPC system is intended mainly for editing and compiling of parallel programs. Interactive usage of **mpirun/mpiexec** is allowed on the login node and its coprocessors, although it should not be necessary. It is advisable to switch to using a dedicated node obtained through the batch system if such parallel runs take more than a few minutes or use lots of memory.

To run test or production jobs, submit them to the torque batch system, which will find and allocate the resources required for your job (e.g. the compute nodes to run your job on).

Short test jobs (shorter than 15 min) with 2, 4 or 8 cores will run with short turn-around times. However, the use of partial nodes (i.e., less than 16 cores from one node) is discouraged. The use of multiple partial nodes, e.g., by requesting 10 nodes with 4 cores, is strongly discouraged.

By default, the job run limit is set to 24 hours on Avitohol. If your batch jobs can't run independently from each other, please use job steps or contact the system administrators helpdesk using the email address

avitohol-support@parallel.bas.bg.

The Intel processors support the "Hyperthreading / Simultaneous Multithreading (SMT)" mode which might increase the performance of your application by up to 20%. However, we leave it to the users to decide whether to use hyperthreading or not. Currently the execution nodes are declared to have 16 CPU cores in torque, although the Linux OS sees 32 (logical) cores.

It is encouraged for users to request multiples of full nodes, e.g., by adding

```
#PBS -l nodes=100:ppn=16
```

in the PBS script you can ask for 100 dedicated nodes.

With hyperthreading, you have to increase the number of actual MPI processes or OpenMP threads to 32 instead of 16 for each node.

However, it is best to first measure the performance on a realistic benchmark problem to see if hyperthreading is actually useful. In both cases, with and without hyperthreading, the request for nodes to PBS will be the same, but the mpirun command line would change.

For example, one can compare the performance of the same executable **testprogram** with and without hyperthreading on 100 nodes measuring the execution times of:

```
mpirun –f hostfile -np 1600 -ppn 16 ./testprogram
```

vs.

```
mpirun –f hostfile -np 3200 -ppn 32 ./testprogram
```

The file with name **hostfile** should contain the list of all nodes in the job. This can be achieved by doing

```
cat $PBS_NODEFILE| sort |uniq > hostfile
```

Please be aware that with 32 tasks-per-node each process gets only half of the memory by default. In the Avitohol cluster, there are 150 compute nodes available with 64 GB of real memory (some is taken by the OS). So, on these 150 nodes, you can expect to have 2 GB per core for MPI jobs with hyperthreading or 4 GB per core for MPI jobs without hyperthreading.

The default Parallel Environment is Intel MPI. You can use executables that were built with other MPI libraries. For example, openmpi is also available system-wide.

For each execution node there are two associated Xeon Phi coprocessors. For example, on node sl039 the coprocessors are available via ssh or for mpiexec with the names sl039-mic0 and sl039-mic1.

Each coprocessor is running its own operating system, which means that it appears as separate machine.

For each execution node there are alternative names that correspond to alternative network interfaces, associated with the InfiniBand cards. For example, ibsl039 corresponds to the IP-over-InfiniBand network interface ib0 on host sl039.

The same type of IP-over-InfiniBand interfaces are available on the coprocessors, for example ibsl039-mic0 and ibsl039-mic1.

Although these interfaces offer higher bandwidth than the standard Ethernet interfaces, they are not necessary for normal use, because MPI jobs are expected by default to use native InfiniBand and not IP-over-InfiniBand.

The Lustre filesystem already uses native InfiniBand transport and since the files are shared between nodes and it is even available on the coprocessors, necessity for data movement between nodes via the IP-over-InfiniBand interface may be indication for sub-optimal use of the system.

InfiniBand is also the default transport for MPI when using the coprocessors via MPI.

Once a given user has obtained exclusive access to an execution node he or she can also use the corresponding Xeon Phi coprocessors. Inside a job script one may read the contents of the file pointed by the environment variable PBS_NODEFILE and obtain the names of the corresponding Xeon Phi nodes. The access to the Xeon Phi coprocessors is not managed separately.

However, users are not allowed to login with ssh to nodes where they do not have running jobs. To determine which nodes have been allocated to a given job, the user can run

```
qstat –na <jobnumber>
```

Users are not expected to use Xeon Phi coprocessors without first requesting their host node from the batch system.

Further reading

For more information about using **torque** batch system and **moab** scheduler please see the documentation from the website of the software provider: www.adaptivecomputing.com.

17.1.5. Programming Environment

17.1.5.1. Compilers

The Intel Fortan (ifort) and C/C++ (icc, icpc) compilers are the default compilers on the HPC cluster Avitohol and are provided automatically at login (see the output of module list for details on the version).

To compile and link MPI codes using Intel compilers, use the commands mpiifort, mpiicc or mpiicpc, respectively.

The GNU compiler collection (gcc, g++, gfortran) is available as well. A default version comes with the operating system. Version 4.4.7 is installed on Avitohol system. More recent versions can be accessed via environment modules. To compile and link MPI codes using GNU compilers, use the commands mpicc, mpic++/mpicxx, mpif77 or mpif90, respectively.

The compilation for the Xeon Phi is usually done at the host, i.e., using cross-compilation.

To load the development environment for the Xeon Phi one can issue

```
source /opt/mpss/3.6/environment-setup-k1om-mpss-linux
```

Then the version of the compiler and other development tools can be accessed with something like:

```
k1om-mpss-linux-gcc
k1om-mpss-linux-g++
k1om-mpss-linux-nm
```

Invoke the command module avail to get an overview on all compilers and versions available on Avitohol.

Modules environment

For more information on general Modules usage please refer to the PRACE Generic x86 Best Practice Guide http://www.prace-ri.eu/IMG/pdf/Best-Practice-Guide-Generic-x86.pdf.

For more information about using MPSS consult the documentation from Intel

https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-developers-quick-start-guide

17.1.5.2. Parallel Programming

On the Avitohol HPC cluster two basic parallelization methods are available.

MPI: The Message Passing Interface provides a maximum of portability for distributed memory parallelization over a large numer of nodes.

OpenMP: OpenMP is a standardized set of compiler directives for shared memory parallelism. On Avitohol a pure OpenMP code is restricted to run on a single node (whether the host or the coprocessor system).

It is possible to mix MPI and OpenMP parallelism within the same code in order to achieve large scale parallelism.

MPI and OpenMP are also available on the coprocessors, and all the modes of using coprocessors - native, symmetric and offload, are available.

Each Xeon Phi coprocessor can run OpenMP programs in native mode. The number of physical cores on a coprocessor is 61. It is advisable to reserve one core for the operating system use and to use at most 60 cores. However, the number of logical cores on the coprocessor is 4 times the number of physical cores, thus 244. Experience has shown that using more threads than the number of physical cores can be beneficial. Users can try to find which setting of the number of threads for OpenMP via the variable OMP_NUM_THREADS (in native

mode) or MIC_OMP_NUM_THREADS (in offload mode) is acceptable. In many cases 120 seems to be a good compromise.

For the programs that run on the host the number of threads may be set equal to the number of CPU cores (16) or number of logical cores (32) if only OpenMP is used. If a hybrid MPI/OpenMP program is run, one must take into account how many MPI processes are running on one node. For example, if two MPI processes are run on each node, then OMP_NUM_THREADS should be set to 8 to use all physical cores or 16 to use all logical cores.

17.1.5.3. Parallel MPI applications

By default, the Intel compilers for Fortran/C/C++ are available on Avitohol.

The MPI wrapper executables are mpiifort, mpiicc and mpiicpc. These wrappers pass include and link information for MPI together with compiler-specific command line flags to the Intel compiler.

The parallel program can be started with mpiifort. We remind that environmental variables can be passed via options. For example, to set the variable MKL_MIC_ENABLE to 1, thus enabling automatic offload for all MPI processes, one can add

```
-genv MKL_MIC_ENABLE 1
```

to the mpiexec command line.

Batch jobs

For production runs it is necessary to run the MPI program as a batch job.

Environment variables

Some environment variables may give information about software that does not come directly from the operating system distribution. Examples are some build tools like ant/maven, which are provided.

17.1.5.4. Multithreaded (OpenMP or hybrid MPI/OpenMP) applications

To compile and link OpenMP applications pass the flag –qopenmp (which replaces the former option –openmp, which is now deprecated) to the Intel compiler, or –fopenmp to the gcc compiler.

In some cases it is necessary to increase the private stack size of the threads at runtime, e.g. when threaded applications exit with segmentation faults. On Avitohol the thread private stack size is set via the environment variable KMP_STACKSIZE. The default value is 4 megabytes (4MB). For example, to request a stack size of 128 megabytes on Avitohol, set KMP_STACKSIZE=128m.

Beware that on the coprocessor the practical limit for the number of threads is 244 (61 hardware threads times 4 due to hyperthreading). Thus using high numbers for KMP_STACKSIZE may crush the application.

Note

Applications occasionally may use swap memory. In general such application behavior is undesirable and strongly discouraged. It is presumed to be a result of application mis-configuration or error in the input. Users are expected to cancel jobs that they observe to be swapping and try and correct the errors. If they believe that this is normal behavior, they should discuss this with system administrators.

For information on compiling applications which use pthreads please consult the Intel compiler documentation.

17.1.5.5. Using MKL mathematical library on Avitohol

The Intel Math Kernel Library (MKL) is provided on Avitohol. MKL provides highly optimized implementations of (among others)

- LAPACK/BLAS routines,
- direct and iterative solvers,

- FFT routines,
- ScaLAPACK.

Parts of the library support thread or distributed memory parallelism.

By setting the environment variable MKL_MIC_ENABLE to 1, e.g., by

```
MKL_MIC_ENABLE=1
```

in a bash shell, one can enable the automatic offload of some library routines to the Xeon Phi coprocessors that are available. For using this option with MPI, see above.

Note:

Extensive information on the features and the usage of MKL is provided by the official Intel MKL documentation

About the automatic offload one can consult:

https://software.intel.com/sites/default/files/11MIC42_How_to_Use_MKL_Automatic_Offload_0.pdf

Linking programs with MKL

By default, an MKL environment module is already loaded. The module sets the environment variables MKL_HOME and MKLROOT to the installation directory of MKL. These variables can then be used in makefiles and scripts.

The Intel MKL Link Line Advisor is often useful to obtain information on how to link programs with MKL. For example, to link statically with the threaded version of MKL on Avitohol (Linux, Intel64) using standard 32 bit integers, something like the following must be added to the Makefile when invoking the Intel compiler:

```
    -Wl,--start-group
$(MKLROOT)/lib/intel64/libmkl_intel_lp64.a
$(MKLROOT)/lib/intel64/libmkl_intel_thread.a
$(MKLROOT)/lib/intel64/libmkl_core.a
-Wl,--end-group -lpthread -lm –qopenmp
```

(in one line). If compiling directly in bash, one must use ${MKLROOT} instead of $(MKLROOT).

17.1.5.6. Numerical Libraries

FFTW

The so-called "Fastest Fourier transforms in the West/World" software is installed. Currently the version of FFTW 3 that comes with the OS is 3.2.3 and is available in the standard locations for development software.

PETSc

A suite of data structures and routines for the scalable (MPI parallel) solution of scientific applications modeled by partial differential equations. Available as a module.

SLEPc

Library for the solution of large scale sparse eigenvalue problems on parallel computers. Available as a module.

GSL

The GNU Scientific Library (GSL) is a numerical library for C and C++ programmers (the FGSL FORTRAN addon interface is installed). GSL provides a wide range of mathematical routines such as random number generators, special functions and least-squares fitting. Available as part of the OS distribution.

Python modules

The Python version from the OS distribution is of the 2.6 series. The latest current 2.7 series version is available as a module. The tensorflow module is available as part of the 2.7 installation.

17.1.5.7. I/O Libraries

NetCDF

NetCDF is a set of software libraries and self-describing, machine-independent data formats that support the creation, access, and sharing of array-oriented scientific data. Available as a module (version 4).

17.1.5.8. Miscellaneous Libraries

- **Boost**: The Boost C++ libraries provide many classes and routines for various applications. The version of boost provided by the OS is rather old. Newer versions are available under /opt/soft/ and are available through the environmental modules system.
- **TBB: Intel Threading Building Blocks**. TBB enables the C++ programmer to integrate (shared memory) parallel capability into the code. Available as part of the Intel Compiler Suite.
- **IPP: Intel Integrated Performance Primitives (IPP)**. The IPP API contains highly optimized primitive operations used for digital filtering, audio and image processing.
- **PAPI: The Performance Application Programming Interface** is a library for reading performance event counters in a portable way.

17.1.6. Programming Tools

To access the software described below please use the module command (module avail, module load).

17.1.6.1. Debugging

- Compiler options: Compilers usually have some debugging features which allow e.g. to check violations of array boundaries. Please consult the compiler's manual pages and documentation for details.
- gdb, the GNU debugger.
- Intel Inspector enables the debugging of threaded applications.
- If debugging features will not be used, the executable can be stripped of debugging information by running strip.

17.1.6.2. Profiling and Performance Analysis

1. Intel VTune/Amplifier is a powerful tool for analyzing the single core performance of a code.
2. gprof, the GNU profiler.
3. Intel Trace Analyzer and Collector is a tool for profiling MPI communication.
4. Scalasca enables the analysis of MPI/OpenMP/hybrid codes.
5. Optimised executables can be obtained by first compiling for profile generation (option -prof-gen for Intel, -fprofile-generate for gcc) and then running the executable and using the generated profile with options like

Further reading

For more information related to “Programming Tools” please refer to the Avitohol website:

http://www.hpc.acad.bg/system-1/

17.1.7. Final remarks

For updates please check http://www.hpc.acad.bg [http://www.hpc.acad.bg]

Users that require additional tools or software to be installed or upgrades to existing software should contact the support team at **avitohol-support@parallel.bas.bg**.

17.2. MareNostrum @ BSC

17.2.1. System Architecture / Configuration

The cluster consists of one iDataPlex rack with 42 nodes dx360 M4. Each node contains 2 Sandy-Bridge-EP (2 x 8 cores) host processors E5-2670 @ 2.6 GHz and 2 Intel Xeon Phi (MIC) coprocessors 5110P with 60 cores @ 1.1 GHz. The memory per node is 64 GB (host) and 2 x 8 GB (Intel Xeon Phi). The compute nodes are connected via Mellanox Infiniband FDR10 using Mellanox OFED 2.2. Through a vritual bridge interface all MIC coprocessors can be directly accessed from the host compute nodes. The Intel Xeon compute nodes are stored in compute rack c37 and named s18r3b01, …, s18r3b42. The 2 MIC coprocessors attached to a compute node using PCIe are named (e.g. for the host sr18r3b01) s18r3b01-mic0 and s18r3b01-mic1. Mellanox ConnextX-3 Dual-port FDR10 QSFP IB Mezz Card attached to CPU socket 0. Linux is running as operation system on both the compute hosts (SLES 11 SP3) and the Intel Xeon Phi coprocessors (Intel MIC Platform Software Stack (Built by Poky 7.0) 3.4.3 based on Busybox). Every compute node has a 455 GB local disk attached to it, which is also mounted on both Xeon Phi.

17.2.2. System Access

The Intel Xeon Phi partition in MareNostrum can be accessed only for users who have already a MareNostrum user account. After you are registered, use ssh to connect to MareNostrum's login nodes via: ssh username@mn1.bsc.es. Login nodes should be only used for editing, compiling and submitting programs. Jobs must be submitted via LSF. Mind that compilation on the compute nodes and coprocessors is not possible (no include files installed etc.). Login to compute nodes via an interactive job: Login to the corresponding MIC coprocessors (passwordless): `ssh $HOST-mic0 ssh $HOST-mic1`

17.2.3. Production Environment

17.2.3.1. Budget and quota

There is a budget system enabled for PRACE users in the Intel Xeon Phi partition.

17.2.3.2. Filesystems

The GPFS user home directories under `/home/group/user` is only mounted on the host compute nodes. The Xeon Phi partition has its own home directory, which is local to the MIC, mounted in RAM memory. The local disk of the host compute node is mounted on both associated MICs: `/scratch` (455 GB) for user scratch files (read + write). Content is cleaned after the job finishes.

Mind that the I/O performance of the mounted /home filesystem is quite low. In the case of MPI, the MPI executable and necessary input files should be better copied explicitly to the scratch partitions of the local disks.

17.2.3.3. Batch Mode

Batch jobs must use the LSF class "mic". This is currently the only available class with a runtime limit of 48h. In the batch file the number of compute nodes, (i.e. the number of hosts, not the number of Intel Xeon Phi coprocessors) must be specified. An sample Job file allocating 1 compute node and the 2 Intel Xeon Phi cprocessors attached to that host is shown below:

```
#BSUB -n 16
#BSUB -R"span[ptile=16]"
#BSUB -o %J.out
#BSUB -e %J.err
#BSUB -J Test
#BSUB -W 02:00
#BSUB -q mic
```

```
    program-doing-offloading
```

Submitting batch jobs:

```
bsub < jobscript
    Job <1874136> is submitted to queue <mic>
```

Showing the queue:

```
bjobs
    JOBID   USER    STAT  QUEUE      FROM_HOST   EXEC_HOST   JOB_NAME SUBMIT_TIME
    1874136 username RUN  mic        login1                  Test       May 31 15:43
```

Cancelling a job:

```
bkill 1874136
    Job <1874136> is being terminated
```

17.2.3.4. Interactive Jobs

To interactively work with the system submit a simple batch command to the mic-interactive queue:

```
bsub -Is -q mic-interactive -W 01:00 /bin/bash
```

It will open a bash session in a Xeon Phi host for an hour.

17.2.3.5. Running Applications

Application can be run in two modes.

17.2.3.5.1. Native Mode

Run an interactive job and copy the binary compiled for MIC to the coprocessor using scp or copy it to the local filesystem `/scratch/`. Login to the Intel Xeon Phi coprocessor to get a shell and execute the binary.

17.2.3.5.2. Mixed Mode

The wrapper script “bsc_micwrapper” has been developed to ease the utilization of MPI across different hosts and their MICs. Usage:

```
Parameters
    $1   : Number of MPI processes to execute in each host
    $2   : Number of MPI processes to execute in each mic of each host
    $3   : binary file *
```

*since host and mic binary are different, you must have <binary> and <binary>.mic in the same folder somewhere under /gpfs/scratch filesystem Example:

```
#!/bin/bash
      #BSUB -n 64
      #BSUB -o %J.out
      #BSUB -e %J.err
      #BSUB -J Test
      #BSUB -W 02:00
      #BSUB -q mic
```

```
# The job will be allocated in 4 Xeon Phi nodes
# this will submit 16 MPI tasks in each node (host) and
# 240 MPI tasks in each mic (mic0 and mic1) of the 4 nodes allocated
# note that both
#    /gpfs/scratch/xxx/$USER/MIC/test_MPI_impi
#    /gpfs/scratch/xxx/$USER/MIC/test_MPI_impi.mic
# must exist

bsc_micwrapper 16 240 /gpfs/scratch/xxx/$USER/MIC/test_MPI_impi
```

17.2.4. Programming Environment

The following software stack is currently installed on MareNostrum:

software stack

- MPSS 3.4.3
- MLNX_OFED.2.2.125
- Intel Parallel Studio XE 2017 beta compilers
- Intel MPI
- Intel MKL
- Intel VTune Amplifier

The following modules are loaded per default:

```
module list
  Currently Loaded Modulefiles:
  1) intel/13.0.1    2) openmpi/1.8.1   3) transfer/1.0    4) bsc/current
```

17.2.5. Tuning

Depending on the mode MPI programs are using the MICs, the I_MPI_DAPL_PROVIDER_LIST, I_MPI_ADJUST_BARRIER, I_MPI_MIC, I_MPI_HYDRA_BOOTSTRAP, I_MPI_DEVICE and I_MPI_DYNAMIC_CONNECTION must be tuned.

17.2.6. Debugging

The GNU Debugger gdb is installed also on the coporcessors under `/bin/gdb`

17.3. Salomon @ IT4Innovations

17.3.1. System Architecture / Configuration

The Salomon cluster consists of 1008 compute nodes, totaling 24192 compute cores with 129TB RAM and giving over 2 Pflop/s theoretical peak performance. Each node is a powerful x86-64 computer, equipped with 24 cores, at least 128GB RAM. Nodes are interconnected by 7D Enhanced hypercube Infiniband network and equipped with Intel Xeon E5-2680v3 processors. The Salomon cluster consists of 576 nodes without accelerators and 432 nodes equipped with Intel Xeon Phi MIC accelerators.

The cluster runs CentOS Linux operating system, which is compatible with the RedHat Linux family.

Figure 43. iDataPlex rack of SuperMIC

Table 2. System Summary

In general	
Primary purpose	High Performance Computing
Architecture of compute nodes	x86-64
Operating system	CentOS 6.7 Linux
Compute nodes	
Totally	1008
Processor	2x Intel Xeon E5-2680v3, 2.5GHz, 12cores
RAM	128GB, 5.3GB per core, DDR4@2133 MHz
Local disk drive	no
Compute network / Topology	InfiniBand FDR56 / 7D Enhanced hypercube
w/o accelerator	576
MIC accelerated	432
In total	
Total theoretical peak performance	(Rpeak) 2011 Tflop/s
Total amount of RAM	129.024 TB

Table 3. Compute nodes

Node	Count	Processor	Cores	Memory	Accelerator
w/o accelerator	576	2x Intel Xeon E5-2680v3, 2.5GHz	24	128GB	-
MIC accelerated	432	2x Intel Xeon E5-2680v3, 2.5GHz	24	128GB	2x Intel Xeon Phi 7120P, 61cores, 16GB RAM

For remote visualization two nodes with NICE DCV software are available each configured:

Table 4. Remote visualization nodes

Node	Count	Processor	Memory	GPU Accelerator
visualization	2	2x Intel Xeon E5-2695v3, 2.3GHz	28	512GB NVIDIA QUADRO K5000, 4GB RAM

For large memory computations a special SMP/NUMA SGI UV 2000 server is available:

Table 5. SGI UV 2000

Node	Count	Processor	Cores	Memory	Extra HW
UV 2000	1	14x Intel Xeon E5-4627v2, 3.3GHz, 8cores	112	3328GB DDR3@1866MHz	2x 400GB local SSD 1x NVIDIA GM200 (GeForce GTX TITAN X), 12GB RAM

17.3.2. System Access

17.3.2.1. Applying for Resources

Computational resources may be allocated by any of the following Computing resources allocation mechanisms. Academic researchers can apply for computational resources via Open Access Competitions. Anyone is welcomed to apply via the Directors Discretion. Foreign (mostly European) users can obtain computational resources via the PRACE (DECI) program. In all cases, IT4Innovations' access mechanisms are aimed at distributing computational resources while taking into account the development and application of supercomputing methods and their benefits and usefulness for society. The applicants are expected to submit a proposal. In the proposal, the applicants apply for a particular amount of core-hours of computational resources. The requested core-hours should be substantiated by scientific excellence of the proposal, its computational maturity and expected impacts. Proposals do undergo a scientific, technical and economic evaluation. The allocation decisions are based on this evaluation.

17.3.2.2. Shell access and data transfer

The Salomon cluster is accessed by SSH protocol via login nodes login1, login2, login3 and login4 at address salomon.it4i.cz. The login nodes may be addressed specifically, by prepending the login node name to the address, eg. login1.salomon.it4i.cz. The address salomon.it4i.cz is a round-robin DNS alias for the four login nodes.

17.3.3. Production Environment

17.3.3.1. Application Modules

In order to configure your shell for running particular application on Salomon we use Module package interface.

Application modules on Salomon cluster are built using EasyBuild. The modules are divided into the following structure:

```
base: Default module class
bio: Bioinformatics, biology and biomedical
cae: Computer Aided Engineering (incl. CFD)
chem: Chemistry, Computational Chemistry and Quantum Chemistry
compiler: Compilers
data: Data management & processing tools
debugger: Debuggers
devel: Development tools
geo: Earth Sciences
ide: Integrated Development Environments (e.g. editors)
lang: Languages and programming aids
lib: General purpose libraries
math: High-level mathematical software
mpi: MPI stacks
numlib: Numerical Libraries
perf: Performance tools
```

```
phys: Physics and physical systems simulations
system: System utilities (e.g. highly depending on system OS and hardware)
toolchain: EasyBuild toolchains
tools: General purpose tools
vis: Visualization, plotting, documentation and typesetting
```

The modules set up the application paths, library paths and environment variables for running particular application. The modules may be loaded, unloaded and switched, according to momentary needs.

To check available modules use:

```
$ module avail
```

To load a module, for example the OpenMPI module use:

```
$ module load OpenMPI
```

loading the OpenMPI module will set up paths and environment variables of your active shell such that you are ready to run the OpenMPI software

To check loaded modules use:

```
$ module list
```

To unload a module, for example the OpenMPI module use:

```
$ module unload OpenMPI
```

17.3.4. Programming Environment

17.3.4.1. EasyBuild Toolchains

As we wrote earlier, we are using EasyBuild for automatised software installation and module creation. EasyBuild employs so-called compiler toolchains or, simply toolchains for short, which are a major concept in handling the build and installation processes. A typical toolchain consists of one or more compilers, usually put together with some libraries for specific functionality, e.g., for using an MPI stack for distributed computing, or which provide optimized routines for commonly used math operations, e.g., the well-known BLAS/LAPACK APIs for linear algebra routines. For each software package being built, the toolchain to be used must be specified in some way. The EasyBuild framework prepares the build environment for the different toolchain components, by loading their respective modules and defining environment variables to specify compiler commands (e.g., via $F90), compiler and linker options (e.g., via $CFLAGS and $LDFLAGS), the list of library names to supply to the linker (via $LIBS), etc. This enables making easyblocks largely toolchain-agnostic since they can simply rely on these environment variables; that is, unless they need to be aware of, for example, the particular compiler being used to determine the build configuration options. Recent releases of EasyBuild include out-of-the-box toolchain support for: various compilers, including GCC, Intel, Clang, CUDA common MPI libraries, such as Intel MPI, MPICH, MVAPICH2, OpenMPI various numerical libraries, including ATLAS, Intel MKL, OpenBLAS, ScalaPACK, FFTW

On Salomon, we have currently following toolchains installed:

Toolchain Module(s)

- GCC GCC
- ictce icc, ifort, imkl, impi
- intel GCC, icc, ifort, imkl, impi
- gompi GCC, OpenMPI
- goolf BLACS, FFTW, GCC, OpenBLAS, OpenMPI, ScaLAPACK
- iompi OpenMPI, icc, ifort
- iccifort icc, ifort

17.4. SuperMIC @ LRZ

17.4.1. System Architecture / Configuration

The cluster consists of one iDataPlex rack with 32 nodes dx360 M4. Each node contains 2 Ivy-Bridge (2 x 8 cores) host processors E5-2650 @ 2.6 GHz and 2 Intel Xeon Phi (MIC) coprocessors 5110P with 60 cores @ 1.1 GHz.

Memory per node is 64 GB (host) and 2 x 8 GB (Intel Xeon Phi).

The following picture shows the iDataPlex rack of SuperMIC.

Figure 44. iDataPlex rack of SuperMIC

Intel's Xeon Phi coprocessor is based on x86 technology and consists of 60 cores interconnected by a bidirectional ring interconnect. Each core can run up to 4 hardware threads, so max. 240 hardware threads are supported per coprocessor.

The connection from the host to the attached coprocessors is via PCIe 2.0, which limits the bandwidth to 6.2 GB/s.

The compute nodes are connected via Mellanox Infiniband FDR14 using Mellanox OFED 2.2. Through a vritual bridge interface all MIC coprocessors can be directly accessed from the SuperMIC login node and the compute nodes.

The SuperMIC login node supermic.smuc.lrz.de (=login12) can be accessed from the supermuc login nodes and user-registered IP addresses.

The Intel Xeon compute nodes in rack column A are named i01r13a01, i01r13a02, ..., i01r13a16, those in rack column B are named i01r13c01, i01r13c02, ..., i01r13c16.

The 2 MIC coprocessors attached to a compute node using PCIe are named (e.g. for the host i01r13a01) i01r13a01-mic0 and i01r13a01-mic1.

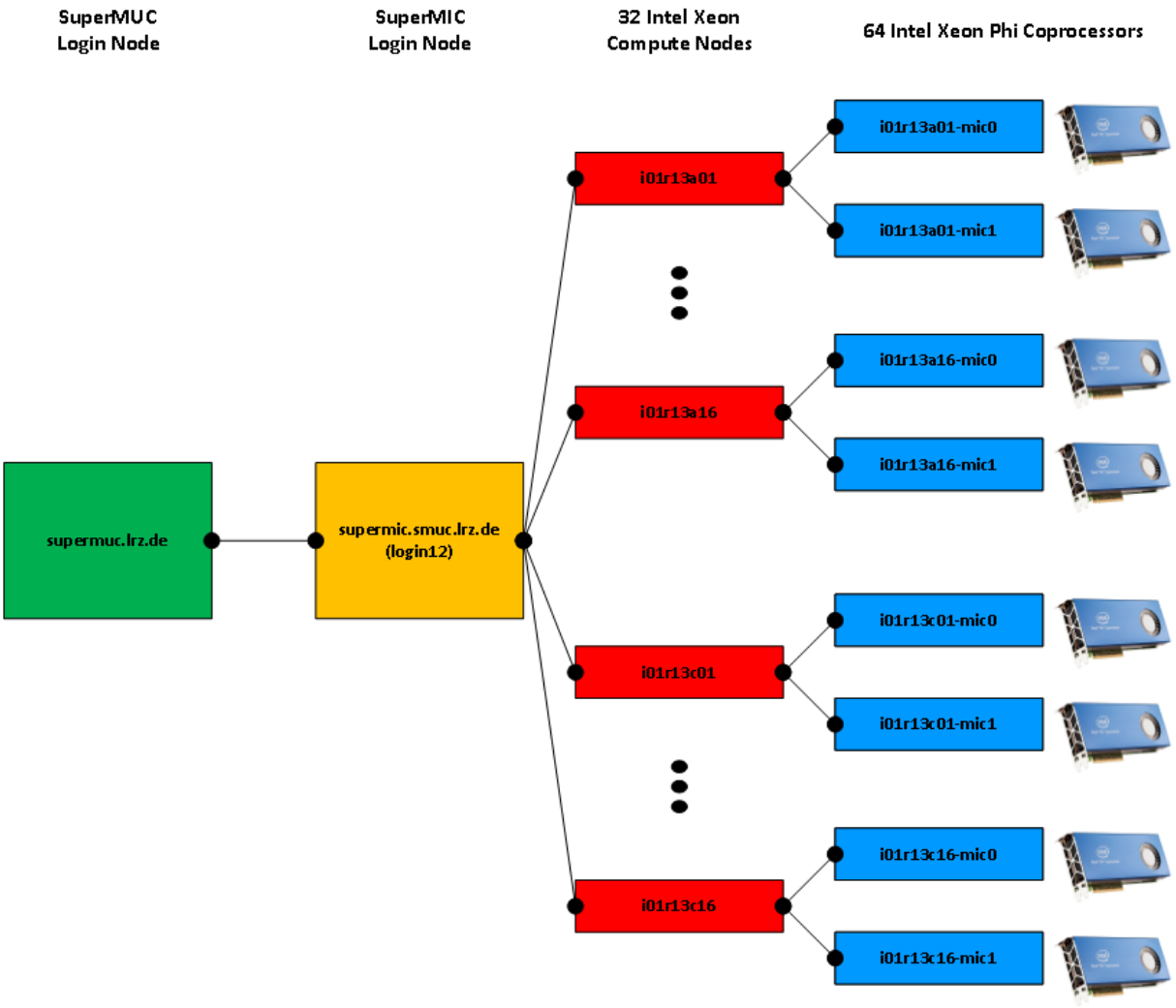
2 MLNX CX3 FDR PCIe cards are attached to each CPU socket (Reiser cards).

Linux is running as operation system on both the compute hosts and the Intel Xeon Phi coprocessors.

Every compute node has a local disk attached to it.

An overview of the vaious nodes and coprocessors is shown in the following picture.

Figure 45. SuperMIC configuration



17.4.2. System Access

SuperMIC can be accessed only for users have already a SuperMUC user account with a registered IP-addresse. Please apply for access by contacting the LRZ HPC support providing your name, user-ID and your static IP-addresse. After you are registered then use ssh to connect to the SuperMIC login node via:

```
ssh supermic.smuc.lrz.de
```

The SuperMIC Login node (supermic = login12) should be only used for editing, compiling and submitting programs. Compilation is also possible on the SuperMUC login nodes.

Jobs must be submitted via LoadLeveler. Once you get the reservation from the batch system, also interactive login to the reserved compute nodes and the attached MIC coprocessors is possible.

Mind that compilation on the compute nodes and coprocessors is not possible (no include files installed etc.).

Login to compute nodes:

```
ssh i01r13???
```

Login to the corresponding MIC coprocessors:

```
ssh i01r13???-mic0
ssh i01r13???-mic1
```

To be able to login to the coprocessors you have to create ssh-keys using the command "ssh-keygen". These keys must be named id_rsa and id_rsa.pub and must reside within your $HOME/.ssh directory _before_ you submit a SuperMIC job.

The content of id_rsa.pub has to be appended to the file $HOME/.ssh/authorized_keys to access also the compute nodes.

It is then possible to login to the MIC coprocessors from the reserved compute nodes or directly from the SuperMIC login node, too.

17.4.3. Production Environment

17.4.3.1. Budget and Quota

There are no quotas for computing time on the SuperMIC cluster yet.

17.4.3.2. Filesystems

The user home directories under /home/hpc/group/user and the LRZ software repositories under /lrz/sys/ (same as on SuperMUC) and /lrz/mic/ (MIC specific software and modules) are mounted on the SuperMIC login-node, compute nodes and the MIC coprocessors.

To offer a clean environment on the MICs, the default home-directory on the MICs is a local directory /home/user.

The local disks have 2 partitions:

/scratch (410 GB) for user scratch files (read + write). Content is cleaned after the job finishes.

/lrzmic (50 GB) for MIC specific software (read-only), maintained by LRZ. Depricated. MIC-specific software is now mounted globally under /lrz/mic.

Both partitions are mounted on the compute host and the 2 attached Intel Xeon Phi coprocessors.

Mind that the I/O performance of the mounted /home/hpc filesystem is quite low. In the case of MPI, the MPI executable and necessary input files should be better copied explicitly to the scratch partitions of the local disks attached to the allocated compute nodes.

17.4.3.3. Interactive and Batch Jobs

17.4.3.3.1. Batch Mode

Batch jobs must use the LoadLeveler class "phi". This is currently the only available class with a runtime limit of 48h. In the batch file the number of compute nodes, (i.e. the number of hosts, not the number of Intel Xeon Phi coprocessors) must be specified.

To offer a clean system environment the Xeon Phi coprocessors are reset and rebooted after every job.

Example script: Sample Job file allocating 1 compute node and the 2 Intel Xeon Phi cprocessors attached to that host:

```
#!/bin/bash
#@ wall_clock_limit = 01:00:00
#@ job_name =test
#@ job_type = parallel
#@ class = phi
#@ node = 1
#@ node_usage = not_shared
#@ initialdir = $(home)/jobs/
#@ output = test-$(jobid).out
#@ error = test-$(jobid).out
#@ notification=always
#@ notify_user=user@lrz.de
#@ queue

. /etc/profile
. /etc/profile.d/modules.sh

program-doing-offloading #for noninteractive execution
sleep 10000 #for interactive access while the job is running
```

Submitting batch jobs:

```
lu65fok@login12:~/jobs> llsubmit  job1.ll
llsubmit: Processed command file through Submit Filter: "/lrz/loadl/SuperMIC/filter/subm
llsubmit: The job "i01xcat3-ib.287" has been submitted.
```

Showing the queue:

```
lu65fok@login12:~/jobs> llq
Id                       Owner      Submitted   ST PRI Class        Running On
------------------------ ---------- ----------- -- --- ------------ -----------
i01xcat3-ib.287.0        lu65fok     6/4  16:51 ST 50  phi          i01r13c11-ib

1 job step(s) in queue, 0 waiting, 1 pending, 0 running, 0 held, 0 preempted
```

Cancelling a job:

```
lu65fok@login12:~/jobs> llcancel i01xcat3-ib.288.0
llcancel: Cancel command has been sent to the central manager.
```

Due to the rebooting of the cards cancellation of a job takes some time.

17.4.3.3.2. Interactive Jobs

To interactively work with the system submit a simple batch file and insert a "sleep 10000" command at the very end of the script.

During the sleeping period the allocated compute-nodes and Intel Xeon Phi coprocessors can be accessed interactively.

17.4.3.4. Running Applications

17.4.3.4.1. Native Mode

Run an interactive job (using sleep within a job script) and copy the binary compiled for MIC to the coprocessor using scp or copy it to the local filesystem /scratch/. Login to the Intel Xeon Phi coprocessor to get a shell and execute the binary.

For simple programs not needing any input files etc. native programs can be also launched from the host using

```
micnativeloadex micbinary
```

17.4.3.4.2. MPI Programs

MPI tasks can be run on the host and/or on the Xeon Phi coprocessors. The MPI code can contain Offload Pragmas for offloading to the coprocessors and/or OpenMP to implement multithreading.

Only Intel MPI is supported. Since Intel MPI tasks are launched via ssh on remote nodes / coprocessors (default on SuperMIC: I_MPI_HYDRA_BOOTSTRAP=ssh), users have to create passwordless ssh-keys named id_rsa and id_rsa.pub using "ssh-keygen". The content of id_rsa.pub has to be appended to the file $HOME/.ssh/authorized_keys. It is not allowed to copy the keys to other systems outside of SuperMUC/SuperMIC.

An overview of the various MPI/OpenMP modes is presented in the following picture.

mpimodes

17.4.3.4.3. MPI Programs: Offload Mode

Sample MPI program doing offloading:

```
Test
```

Sample Jobscript allocating 2 nodes and running 2 MPI processes on every compute node. Every MPI process offloads to another MIC coprocessor.

```
#!/bin/bash
#@ wall_clock_limit = 01:00:00
#@ job_name =test
#@ job_type = parallel
#@ class = phi
#@ node = 2
#@ tasks_per_node= 2
#@ node_usage = not_shared
#@ initialdir = $(home)/jobs/
#@ output = test-$(jobid).out
#@ error = test-$(jobid).err
#@ notification=always
#@ notify_user=user@lrz.de
#@ queue
```

```
. /etc/profile
. /etc/profile.d/modules.sh


export I_MPI_DAPL_PROVIDER_LIST=ofa-v2-mlx4_0-1u
mpiexec ./testmpioffload
```

Output:

```
Hello world from process 2 of 4: host: i01r13a06
Hello world from process 1 of 4: host: i01r13a07
Hello world from process 3 of 4: host: i01r13a06
Hello world from process 0 of 4: host: i01r13a07
MIC: I am i01r13a07-mic1 and I have 240 logical cores. I was called by  process 1 of 4:
MIC: I am i01r13a06-mic1 and I have 240 logical cores. I was called by  process 3 of 4:
MIC: I am i01r13a07-mic0 and I have 240 logical cores. I was called by  process 0 of 4:
MIC: I am i01r13a06-mic0 and I have 240 logical cores. I was called by  process 2 of 4:
```

17.4.3.4.4. MPI Programs: Native Mode

To run MPI tasks natively on the Intel Xeon Phi coprocessors I_MPI_MIC must be enabled (default on SuperMIC: I_MPI_MIC=enable). The preferred fabrics is using Infiniband via dapl (default on SuperMIC: I_MPI_FABRICS=shm:dapl). The preferred value of I_MPI_DAPL_PROVIDER_LIST depends on the MPI mode used (tasks on host and MIC vs. host and host etc). Currently all MPI modes work with setting I_MPI_DAPL_PROVIDER_LIST=ofa-v2-mlx4_0-1u, but may not deliver max. bandwidth.

In some cases I_MPI_DAPL_PROVIDER_LIST=ofa-v2-mlx4_0-1u,ofa-v2-scif0 delivers better performance.

Below we show a sample jobscript allocating two nodes and running 1 MPI process per host and per MIC co-processor.

The binary $bin_mic must have been compiled natively for MIC using "-mmic". It is copied to the local /scratch/ directory on every allocated compute-node by the script.

The binary $bin_host must have been compiled for Intel Xeon.

$taskspermic and $tasksperhost should be properly set (in the example both are set to 1).

```
#!/bin/bash
#@ wall_clock_limit = 01:00:00
#@ job_name =TEST2
#@ job_type = parallel
#@ class = phi
#@ node = 2
#@ node_usage = not_shared
#@ initialdir = $(home)/jobs/
#@ output = test-$(jobid).out
#@ error = test-$(jobid).err
#@ notification=always
#@ notify_user=user@lrz.de
#@ queue

. /etc/profile
. /etc/profile.d/modules.sh


export I_MPI_DAPL_PROVIDER_LIST=ofa-v2-mlx4_0-1u
```

```
bin_mic=testmpi-mic
dir_mic=/scratch
bin_host=./testmpi-host
arg=""

taskspermic=1
tasksperhost=1


command="mpiexec"

for i in `cat $LOADL_HOSTFILE`;do
    numhosts=$((numhosts+1))
    host=`echo $i| cut -d- -f1`;
    hosts="$hosts $host";
    mic0=$host-mic0
    mic1=$host-mic1
    mics="$mics $mic0 $mic1"
    scp $bin_mic $host:/$dir_mic

    command="$command  -host $host-ib -n $tasksperhost $bin_host $arg : "
    command="$command  -host $mic0 -n $taskspermic $dir_mic/$bin_mic $arg : -host $mic1

    if test $numhosts  -lt $LOADL_TOTAL_TASKS; then
    command="$command : "
    fi
done

echo Hosts = $hosts
echo MICS = $mics
echo $command
$command
```

Output:

```
Hosts = 01r13c16 i01r13a02
MICS = i01r13c16-mic0 i01r13c16-mic1 i01r13a02-mic0 i01r13a02-mic1

mpiexec -host i01r13c16-ib -n 1 ./testmpi-host :
        -host i01r13c16-mic0 -n 1 /scratch/testmpi-mic :
        -host i01r13c16-mic1 -n 1 /scratch/testmpi-mic :
        -host i01r13a02-ib -n 1 ./testmpi-host :
        -host i01r13a02-mic0 -n 1 /scratch/testmpi-mic :
        -host i01r13a02-mic1 -n 1 /scratch/testmpi-mic

Hello world from process 3 of 6: host: i01r13a02
Hello world from process 0 of 6: host: i01r13c16
Hello world from process 5 of 6: host: i01r13a02-mic1
Hello world from process 4 of 6: host: i01r13a02-mic0
Hello world from process 2 of 6: host: i01r13c16-mic1
Hello world from process 1 of 6: host: i01r13c16-mic0
```

17.4.4. Programming Environment

The following software stack is currently installed on SuperMIC:

- MPSS 3.4

- MLNX_OFED_LINUX-2.2-1.0.1.1
- Intel Parallel Studio XE 2016 compilers
- Intel MPI
- Intel MKL, IPP, TBB (module load ipp tbb)
- Intel VTune Amplifier (module load amplifier_xe/2016)

The following modules are loaded per default:

```
lu65fok@login12:~> module list
Currently Loaded Modulefiles:
 1) admin/1.0      3) intel/16.0     5) mpi.intel/5.1
 2) tempdir/1.0    4) mkl/11.3       6) lrz/default
lu65fok@login12:~>
```

There is a small module system installed also on the coprocessors.

Currently the following modules are available on the Xeon Phi corpocessors:

```
u65fok@i01r13c07-mic0:~$ module av
------------ /lrz/mic/share/modules/files/tools ------------
likwid/3.1
------------ /lrz/mic/share/modules/files/compilers ------------
ccomp/intel/14.0            fortran/intel/15.0(default)
ccomp/intel/15.0(default)   fortran/intel/16.0
ccomp/intel/16.0            intel/16.0
fortran/intel/14.0
------------ /lrz/mic/share/modules/files/libraries ------------
mkl/11.1          mkl/11.2          mkl/11.3(default) tbb/4.4
------------ /lrz/mic/share/modules/files/parallel ------------
mpi.intel/4.1          mpi.intel/5.0          mpi.intel/5.1(default)
lu65fok@i01r13c07-mic0:~$
```

17.4.5. Performance Analysis

17.4.5.1. Likwid

liwkid (Lightweight performance tools) can be loaded interactively on the coprocessors using

```
module load likwid
```

For more details see the likwid home page [https://code.google.com/archive/p/likwid/].

17.4.5.2. Intel Amplifier XE

The new Sampling Enabling Product (SEP) kernel modules are now loaded on the compute nodes and MIC coprocessors per default. Only use module amplifier_xe/2016 and no older versions, since with incompatible versions the sep kernel module will often hang.

Since a lot of output is produced, always use /scratch as the output directory. Write access to /home is very slow.

To collect data, use the command line version "amplxe-cl" on the COMPUTE NODES. To visualise the collected data, use the GUI version "amplxe" on the LOGIN NODE.

To detect hotspots on the MIC run something similar to the following:

```
# module load amplifier_xe/2016
# rm -rf /scratch/result
# amplxe-cl --collect advanced-hotspots -target-system=mic-native:`hostname`-mic0  -r
```

To use SEP directly, use a command like (use /scratch as output directory!)

```
  # module load amplifier_xe/2016
  # sep -start -mic -verbose -out /scratch/test  -app /usr/bin/ssh -args
  "lu65fok@i01r13c01-mic0 /home/lu65fok/micapp"

to sample on the MIC. To convert tb6 Files to amplxe and view within Amplifier XE use (

  # amplxe-cl -import test.tb6 -r test
  # amplxe-gui  test
```

17.4.6. Tuning

When running MPI tasks on several hosts AND Xeon Phi coprocessors, several collective MPI functions like MPI Barriers do not return properly (cause deadlocks).

In this case set i.e.

export I_MPI_DAPL_PROVIDER_LIST=ofa-v2-mlx4_0-1u

export I_MPI_ADJUST_BARRIER=1

export I_MPI_ADJUST_BCAST=1

More details can be found under https://software.intel.com/en-us/articles/intel-mpi-library-collective-optimization-on-intel-xeon-phi

To improve the performance of MPI_Put operations use:

export I_MPI_SCALABLE_OPTIMIZATION=off

Make sure the following variables are not set for SuperMIC (these variables are currently set by intel.mpi for all systems except for SuperMIC):

unset I_MPI_HYDRA_BOOTSTRAP_EXEC (makes MPI jobs block at start-up)

unset I_MPI_HYDRA_BRANCH_COUNT (causes strange ssh authorization errors)

More details can be found under https://software.intel.com/en-us/node/528782

17.4.7. Debugging

The GNU Debugger gdb is installed also on the coporcessors under /bin/gdb.

Further documentation

Books

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

[2] *James Reinders, James Jeffers, Avinash Sodani, Intel Xeon Phi Processor High Performance Programming, Knights Landing Edition, Morgan Kaufman Publ Inc, 2016 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] *James Reinders, Jim Jeffers: High Performance Parallelism Pearls, Morgan Kaufman Publ Inc, 2015 http://www.lotsofcores.com/ [http://www.lotsofcores.com/] .*

[6] *James Reinders, Jim Jeffers: High Performance Parallelism Pearls Vol. 2, Morgan Kaufman Publ Inc, 2015 http://www.lotsofcores.com/ [http://www.lotsofcores.com/] .*

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

[8] *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

[9] *Intel Developer Zone: Intel Xeon Phi Coprocessor, http://software.intel.com/en-us/mic-developer [http://software.intel.com/en-us/mic-developer].*

[10] *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].*

[11] *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].*

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

[13] *Intel Manycore Platform Software Stack (MPSS), http://software.intel.com/en-us/articles/intel-many-core-platform-software-stack-mpss [http://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss].*

[14] *Intel Xeon Processors & Intel Xeon Phi Coprocessors – Introduction to High Performance Applications Development for Multicore and Manycore – Live Webinar, 26.-27.2.2013, recorded http://software.intel.com/en-us/articles/intel-xeon-phi-training-m-core [http://software.intel.com/en-us/articles/intel-xeon-phi-training-m-core].*

[15] *Intel Cilk Plus Home Page, http://cilkplus.org/ [http://cilkplus.org/].*

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

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

Manuals, Papers

[18] *Intel Xeon Phi Coprocessor Developer's Quick Start Guide, http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-developers-quick-start-guide [http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-developers-quick-start-guide]*.

[19] *PRACE-1IP Whitepapers, Evaluations on Intel MIC, http://www.prace-ri.eu/Evaluation-Intel-MIC*.

[20] *Intel Xeon Phi Coprocessor (codename Knights Corner), http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner [http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner]*.

[21] *Intel Xeon Phi Coprocessor System Software Developers Guide, https://secure-software.intel.com/sites/default/files/article/334766/intel-xeon-phi-systemsoftwaredevelopersguide.pdf [https://secure-software.intel.com/sites/default/files/article/334766/intel-xeon-phi-systemsoftwaredevelopersguide.pdf]*.

[22] *System Administration for the Intel Xeon Phi Coprocessor, http://software.intel.com/sites/default/files/article/373934/system-administration-for-the-intel-xeon-phi-coprocessor.pdf [http://software.intel.com/sites/default/files/article/373934/system-administration-for-the-intel-xeon-phi-coprocessor.pdf]*.

[23] *Configuring Intel Xeon Phi coprocessors inside a cluster, http://software.intel.com/en-us/articles/configuring-intel-xeon-phi-coprocessors-inside-a-cluster*.

[24] *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]*.

[25] *Debugging on Intel Xeon Phi Coprocessor Use Case Overview, http://software.intel.com/en-us/articles/debugging-on-intel-xeon-phi-coprocessor-use-case-overview [http://software.intel.com/en-us/articles/debugging-on-intel-xeon-phi-coprocessor-use-case-overview]*.

[26] *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]*.

[27] *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]*.

[28] *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]*.

[29] *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]*.

[30] *Intel Math Kernel Library on the Intel Xeon Phi Coprocessor, http://software.intel.com/en-us/articles/intel-mkl-on-the-intel-xeon-phi-coprocessors [http://software.intel.com/en-us/articles/intel-mkl-on-the-intel-xeon-phi-coprocessors]*.

[31] *Intel Software Documentation Library, http://software.intel.com/en-us/intel-software-technical-documentation*.

[32] *OpenCL Webpage, https://www.khronos.org/opencl/*.

[33] *OpenCL Design and Programming Guide for the Intel® Xeon Phi Coprocessor, http://software.intel.com/en-us/articles/opencl-design-and-programming-guide-for-the-intel-xeon-phi-coprocessor.*

[34] *OpenACC Webpage, http://www.openacc-standard.org/.*

[35] *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].*

[36] *Optimization and Performance Tuning for Intel Xeon Phi Coprocessors - Part 1: Optimization Essentials, http://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-1-optimization [http://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-1-optimization].*

[37] *Optimization and Performance Tuning for Intel Xeon Phi Coprocessors, Part 2: Understanding and Using Hardware Events, http://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-2-understanding [http://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-2-understanding].*

[38] *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/].*

[39] *Scalasca , http://www.scalasca.org/ [http://www.scalasca.org/].*

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

[41] *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.*

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

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

[44] *Intel Xeon Phi Core Micro-architecture, http://software.intel.com/en-us/articles/intel-xeon-phi-core-micro-architecture.*

[45] *Brian Wylie, Wolfgang Frings: Scalasca support for Intel Xeon Phi, XSEDE13 22-25 July 2013, San Diego, https://www.xsede.org/documents/384387/561679/XSEDE13-Wylie.pdf [https://www.xsede.org/documents/384387/561679/XSEDE13-Wylie.pdf].*

[46] *OpenMP Application Programming Interface Version 4.5 – November 2015, http://openmpcon.org [http://openmpcon.org]>.*

[47] *Application Programming Interface Examples Version 4.5.0 – November 2016, http://openmpcon.org [http://openmpcon.org] .*

[48] *James Beyer, Bronis R. de Supinski, OpenMP Accelerator Model, IWOMP 2016 Tutorial .*

[49] *Simon MacIntosh-Smith, Evaluating OpenMP's Effectiveness in the Many-Core Era, OpenMPCon'16 talk .*

[50] *OpenMP: Memory, Devices, and Tasks12th International Workshop on OpenMP, IWOMP 2016, Nara, Japan, October 5-7, 2016, Proceedings, Springer .*

[51] *Matt Martineau, James Price, Simon McIntosh-Smith, and Wayne Gaudin (Univ. of Bristol, UK Atomic Weapon Est.), Pragmatic Performance Portability with OpenMP 4.x, published in [50] .*

[52] *Dabov,K. and Foi,A. and Katkovnik,V. and Egiazarian,K., 2006, Image denoising with block-matching and 3D filtering, Proceedings of SPIE - The International Society for Optical Engineering .*

[53] *Dabov,K. and Foi,A. and Katkovnik,V. and Egiazarian,K., 2006, Color image denoising via sparse 3D collaborative filtering with grouping constraint in luminance-chrominance space, Proceedings - International Conference on Image Processing, ICIP, p. I313-I316* .

[54] *Dabov,K. and Foi,A. and Katkovnik,V. and Egiazarian,K., 2007, Image denoising by sparse 3-D transform-domain collaborative filtering, IEEE Transactions on Image Processing, p. 2080-2095* .

[55] *Lebrun, Marc, 2012, An Analysis and Implementation of the BM3D Image Denoising Method, Image Processing On Line, p. 175—213* .