Best Practice Guide Haswell/Broadwell

Vali Codreanu, SURFsara

Joerg Hertzer, HLRS

Cristian Morales, BSC

Jorge Rodriguez, BSC

Ole Widar Saastad, University of Oslo

Martin Stachon, IT4Innovations

Volker Weinberg (Editor), LRZ

31-01-2017

Table of Contents

1. Introduction

This Best Practice Guide provides information about Intel's Haswell/Broadwell architecture in order to enable programmers to achieve good performance of their applications. The guide covers a wide range of topics from the description and comparison of the hardware of the Haswell/Broadwell processor, through information about the compiler usage as well as information about porting programs up to tools and strategies how to analyse and improve the performance of applications.

With the introduction of extra vector instructions with these processors (fused multiply add etc.) stronger focus is placed on vectorisation. In the tuning context several of the tuning tools provided by Intel are demonstrated and examples are given on how to use command line tools to collect data in batch mode.

Furthermore, the guide provides information about the following European Intel Haswell/Broadwell based European systems:

- Hazel Hen @ HLRS, Germany
- MinoTauro @ BSC, Spain
- Salomon @ IT4Innovations, Czech Republic
- SuperMUC Phase 2 @ LRZ, Germany

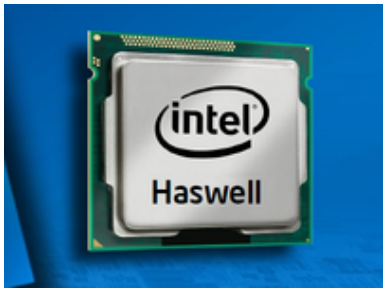
General information on Intel's x86 architecture can be found in the "Best Practice Guide – Generic x86" (May 2013) available under http://www.prace-ri.eu/best-practice-guides/. [http://www.prace-ri.eu/best-practice-guides/]

2. System Architecture

2.1. Overview System Architecture

Haswell is the codename for a processor microarchitecture developed by Intel as the "fourth-generation core" successor to the Ivy Bridge microarchitecture. The Haswell microarchitecture is based on the 22 nm process for mobile, desktops, and servers. Haswell was introduced in 2013 and is named after the town Haswell located in Kiowa County, Colorado, United States. In 2014 Intel introduced Haswell's successor, Broadwell. Broadwell is Intel's codename for the 14 nanometer die shrink of the Haswell microarchitecture. In Intel's Tick-Tock model Haswell represents a "Tock", while Broadwell represents a "Tick". In this model every "Tick" represents a shrinking of the process technology of the previous microarchitecture (sometimes introducing new instructions as with Broadwell) and every "Tock" represents a new microarchitecture. This guide concentrates on features common to both Haswell and Broadwell.

Figure 1. The Intel Haswell Processor.



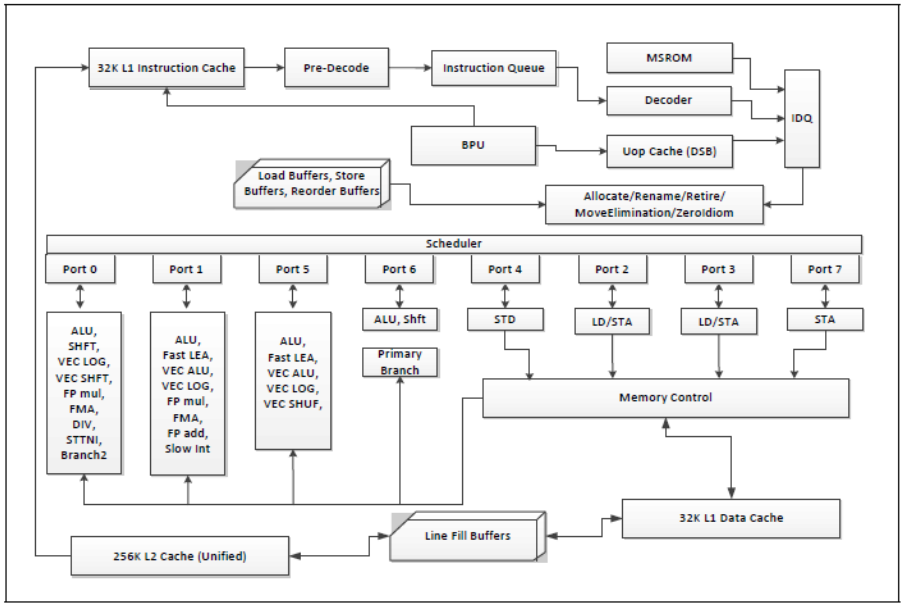
2.2. Processor Architecture

The Haswell microarchitecture builds on the successes of the Sandy Bridge and Ivy Bridge microarchitectures. The basic pipeline functionality of the Haswell microarchitecture is depicted in Figure 2. In general, most of the

features described below also apply to the Broadwell microarchitecture. Some of the innovative feature of the Haswell architecture are:

- Support for Intel Advanced Vector Extensions 2 (Intel AVX2), FMA.
- Support for general-purpose, new instructions to accelerate integer numeric encryption.
- Support for Intel® Transactional Synchronization Extensions (Intel® TSX).
- Each core can dispatch up to 8 micro-ops per cycle.
- 256-bit data path for memory operation, FMA, AVX floating-point and AVX2 integer execution units.
- Improved L1D and L2 cache bandwidth.
- Two FMA execution pipelines.
- Four arithmetic logical units (ALUs).
- Three store address ports.
- Two branch execution units.
- Advanced power management features for IA processor core and uncore sub-systems.
- Support for optional fourth level cache.

Figure 2. Intel Haswell Architecture



To compare the Haswell ("Tock") / Broadwell ("Tick") architecture with previous Intel architectures like Sandy Bridge ("Tock") or Ivy Bridge ("Tick") we show some tables based on material from Intel's code modernization workshop series. The following table shows the main differences between the Xeon E5-2600 v2 (Ivy Bridge) and the E5-2600 v3 (Haswell).

	Xeon E5-2600 v2 (Ivy Bridge)	**Xeon E5-2600 v3 (Haswell)**
Core Count	Up to 12 Cores	Up to 18 Cores
Frequency	TDP and Turbo Frequencies	TDP and Turbo Frequencies, AVX and AVX Turbo Frequencies

	Xeon E5-2600 v2 (Ivy Bridge)	**Xeon E5-2600 v3 (Haswell)**
AVX Support	AVX 1, 8 DP Flops/Clock/Core	AVX 2, 16 DP Flops/Clock/Core
Memory Type	4xDDR3 channels, RDIMM, UDIMM, LRDIMM	4xDDR4 channels, RDIMM, LRDIMM
Memory Frequency (MHz)	1866 (1DPC), 1600, 1333, 1066	RDIMM: 2133 (1DPC), 1866 (2DPC), 1600; LRDIMM: 2133 (1 and 2 DPC), 1600
QPI Speed	Up to 8.0 GT/s	Up to 9.6 GT/s
TDP	Up to 130W Server, 150W Workstation	Up to 145W Server, 160W Workstation (Increase due to Integrated VR)
Power Management	Same P-states for all cores, same core and uncore frequency	Per-core P-states, independent uncore frequency scaling, energy efficient turbo

The following table shows the main differences between the Xeon E5-2600 v3 (Haswell) and the E5-2600 v4 (Broadwell) processor.

	Xeon E5-2600 v3 (Haswell-EP)	**Xeon E5-2600 v3 (Broadwell-EP)**
Cores per socket	Up to 18	Up to 22
Threads per socket	Up to 36 threads	Up to 44 threads
Last-level Cache	Up to 45 MB	Up to 55 MB
QPI Speed	2 x QPI 1.1 channels 6.4, 8.0, 9.6 GT/s	
Memory Population	4 channels of up to 3 RDIMMs or 3 LRDIMMs	+3DS LRDIMM
Max. Memory Speed	Up to 2133	Up to 2400
Max. memory bandwidth	68 GB/s	77 GB/s

One of the main improvements of the Haswell architecture can be seen in the cache bandwidths. The following table compares the Cache sizes, latencies and bandwidths for Nehalem, Sandy-Bridge and Haswell:

Metric	**Nehalem**	**Sandy Bridge**	**Haswell**
L1 Instruction Cache	32k, 4-way	32k, 8way	32k, 8-way
L1 Data cache	32k, 8way	32k, 8way	32k, 8way
L1 Fastest Load-to-use	4 cycles	4 cycles	4 cycles
L1 Load bandwidth	16 Bytes/cycle	32 Bytes/cycle (banked)	*64 Bytes/cycle*
L1 Store bandwidth	16 Bytes/cycle	16 Bytes/cycle	*32 Bytes/cycle*
L2 Unified Cache	256K, 8-way	256K, 8-way	256K, 8-way
L2 Fastest load-to-use	10 cycles	11 cycles	11 cycles
L2 Bandwidth to L1	32 Bytes/cycle	32 Bytes/cycle	*64 Bytes/cycle*
L1 Instruction TLB	4K: 128, 4-way; 2M/4M: 7/thread	4K: 128, 4-way; 2M/4M: 8/thread	4K: 128, 4-way; 2M/4M: 8/thread
L1 Data TLB	4K: 64, 4-way; 2M/4M: 32, 4-way;1G: fractured	4K: 64, 4-way; 2M/4M: 32, 4-way;1G: 4-way	4K: 64, 4-way; 2M/4M: 32, 4-way;1G: 4-way
L2 Unified TLB	4K: 512, 4-way	4K: 512, 4-way	*4K+2M shared: 1024, 8-way*

With each new processor line, Intel introduces new architecture optimizations. The design of the Haswell architecture acknowledges that highly-parallel/vectorized applications place the highest load on the processor cores (requiring more power and thus generating more heat). While a CPU core is executing intensive vector tasks (AVX

instructions), the clock speed may be reduced to keep the processor within its power limits (TDP). In effect, this may result in the processor running at a lower frequency than the “base” clock speed advertised for each model. For that reason, each Haswell processor model is assigned two “base” frequencies:

- AVX mode: due to the higher power requirements of AVX instructions, clock speeds may be somewhat lower while executing AVX instructions.
- Non-AVX mode: while not executing AVX instructions, the processor will operate at what would traditionally be considered the “stock” frequency.

Just as in previous architectures, Haswell CPUs include the Turbo Boost feature which causes each processor core to operate well above the “base” clock speed during most operations. The precise clock speed increase depends upon the number and intensity of tasks running on each CPU. With the Haswell architecture, Turbo Boost speed increases also depend upon the types of instructions (AVX vs. Non-AVX). For more details on this see e.g. https://www.microway.com/knowledge-center-articles/detailed-specifications-intel-xeon-e5-2600v3-haswell-ep-processors/ [https://www.microway.com/knowledge-center-articles/detailed-specifications-intel-xeon-e5-2600v3-haswell-ep-processors/].

2.2.1. Instruction set

Both Intel Haswell and Broadwell processors support an extension to the AVX instruction set, called AVX2. AVX2 uses the same set of 256-bit registers present since AVX, but has added new instructions on top of the AVX instruction set. Perhaps the most important set of added instructions (for HPC) between the pre-Haswell and Haswell architectures is the FMA (Fused Multiply–Add). This extension effectively doubles the peak performance of the chip, allowing a fused multiplication and addition as a single instruction. Thus, with Haswell, each vector unit is able to process 16 floating-point instructions per cycle, compared to a maximum of 8 in prior x86 architectures.

The FMA instruction set is an extension to the 128 and 256-bit Streaming SIMD Extensions instructions to perform fused multiply–add (FMA) operations. The variant supported by Haswell architecture is FMA3 and is described in the table below.

Table 1. FMA Instructions

FMA	Each [z] is the string 132 or 213 or 231, giving the order the operands A,B,C are used in: 132 is A=AC+B 213 is A=AB+C 231 is A=BC+A
VFMADD[z][P/S][D/S]	Fused multiply add A = r1 * r2 + r3 for packed/scalar of double/single
VFMADDSUB[z]P[D/S]	Fused multiply alternating add/subtract of packed double/single A = r1 * r2 + r3 for odd index, A = r1 * r2-r3 for even
VFMSUBADD[z]P[D/S]	Fused multiply alternating subtract/add of packed double/single A = r1 * r2-r3 for odd index, A = r1 * r2+r3 for even
VFMSUB[z][P/S][D/S]	Fused multiply subtract A = r1 * r2-r3 of packed/scalar double/single
VFNMADD[z][P/S][D/S]	Fused negative multiply add of packed/scalar double/single A = -r1 * r2+r3
VFNMSUB[z][P/S][D/S]	Fused negative multiply subtract of packed/scalar double/single A = -r1 * r2-r3

Restricted Transactional Memory (RTM) provides a flexible software interface for transactional execution. RTM provides three new instructions—XBEGIN, XEND, and XABORT—for programmers to start, commit, and abort a transactional execution.

Table 2. RTM Instructions

Function	Description
`void _xabort (const unsigned int imm8)`	Force an RTM abort. The EAX register is updated to reflect an XABORT instruction caused the abort, and the imm8 parameter will be provided in bits [31:24] of EAX. Following an RTM abort, the logical processor resumes execution at the fallback address computed through the outermost XBEGIN instruction.

Function	Description
`unsigned int _xbegin (void)`	Specify the start of an RTM code region. If the logical processor was not already in transactional execution, then this call causes the logical processor to transition into transactional execution. On an RTM abort, the logical processor discards all architectural register and memory updates performed during the RTM execution, restores architectural state, and starts execution beginning at the fallback address computed from the outermost XBEGIN instruction.
`void _xend (void)`	Specify the end of an RTM code region. If this corresponds to the outermost scope, the logical processor will attempt to commit the logical processor state atomically. If the commit fails, the logical processor will perform an RTM abort.
`unsigned char _xtest (void)`	Query the transactional execution status, return 0 if inside a transactionally executing RTM or HLE region, and return 1 otherwise.

The purpose of the Bit Manipulation Instructions Sets (BMI sets) is to improve the speed of bit manipulation. All the instructions in these sets are non-SIMD and operate only on general-purpose registers. There are two sets published by Intel: BMI (here referred to as BMI1) and BMI2; they were both introduced with the Haswell microarchitecture.

Table 3. BMI1 Instructions

Instruction	Description	Equivalent C expression	Intrinsics
ANDN	Logical and not	~x & y	
BEXTR	Bit field extract (with register)	(src >> start) & ((1 << len)-1)	_bextr_u[32/64]
BLSI	Extract lowest set isolated bit	x & -x	_blsi_u[32/64]
BLSMSK	Get mask up to lowest set bit	x ^ (x - 1)	_blsmsk_u[32/64]
BLSR	Reset lowest set bit	x & (x - 1)	_blsr_u[32/64]
TZCNT	Count the number of trailing zero bits	N/A	_mm_tzcnt_[32/64] - _tzcnt_u[32/64]

Table 4. BMI2 Instructions

Instruction	Description	Intrinsics
BZHI	Copy all bits from unsigned integer a to dst, and reset (set to 0) the high bits in dst starting at index.	dst = _bzhi_u[32/64] (a, index)
PDEP	Deposit contiguous low bits from unsigned integer a to dst at the corresponding bit locations specified by mask; all other bits in dst are set to zero.	dst = _pdep_u[32/64] (a, mask)
PEXT	Extract bits from unsigned integer a at the corresponding bit locations specified by mask to contiguous low bits in dst; the remaining upper bits in dst are set to zero.	dst = _pext_u[32/64] (a, mask)

2.2.1.1. New Instructions

Intel AVX2 extends Intel AVX by promoting most of the 128-bit SIMD integer instructions with 256-bit numeric processing capabilities. AVX2 instructions follow the same programming model as AVX instructions. In addition, AVX2 provides enhanced functionalities for broadcast/permute operations on data elements, vector shift in-

structions with variable-shift count per data element, and instructions to fetch non-contiguous data elements from memory. Thus, AVX2 adds support for:

- Expansion of most vector integer SSE and AVX instructions to 256 bits. AVX2's integer support is particularly useful for processing visual data commonly encountered in consumer imaging and video processing workloads.
- Three-operand general-purpose bit manipulation instructions. These are useful for compressed database, hashing , large number arithmetic, and a variety of general purpose codes.
- Gather support, enabling vector elements to be loaded from non-contiguous memory locations. This is useful for vectorizing codes with nonadjacent data elements.
- DWORD- and QWORD-granularity any-to-any permutes allowing shuffling accross an entire 256-bit register.
- Vector shifts. These are very useful in vectorizing loops with variable shifts.

The new AVX2 instructions are listed in the table below.

Table 5. New Instructions

Instruction	Description
VBROADCASTSS, VBROADCASTSD	Copy a 32-bit or 64-bit register operand to all elements of a XMM or YMM vector register. These are register versions of the same instructions in AVX1. There is no 128-bit version however, but the same effect can be simply achieved using VINSERTF128.
VPBROADCASTB, VPBROADCASTW, VPBROADCASTD, VPBROADCASTQ	Copy an 8, 16, 32 or 64-bit integer register or memory operand to all elements of a XMM or YMM vector register.
VBROADCASTI128	Copy a 128-bit memory operand to all elements of a YMM vector register.
VINSERTI128	Replaces either the lower half or the upper half of a 256-bit YMM register with the value of a 128-bit source operand. The other half of the destination is unchanged.
VEXTRACTI128	Extracts either the lower half or the upper half of a 256-bit YMM register and copies the value to a 128-bit destination operand.
VGATHERDPD, VGATHERQPD, VGATHERDPS, VGATHERQPS	Gathers single or double precision floating point values using either 32 or 64-bit indices and scale.
VPGATHERDD, VPGATHERDQ, VPGATHERQD, VPGATHERQQ	Gathers 32 or 64-bit integer values using either 32 or 64-bit indices and scale.
VPMASKMOVD, VPMASKMOVQ	Conditionally reads any number of elements from a SIMD vector memory operand into a destination register, leaving the remaining vector elements unread and setting the corresponding elements in the destination register to zero. Alternatively, conditionally writes any number of elements from a SIMD vector register operand to a vector memory operand, leaving the remaining elements of the memory operand unchanged.
VPERMPS, VPERMD	Shuffle the eight 32-bit vector elements of one 256-bit source operand into a 256-bit destination operand, with a register or memory operand as selector.
VPERMPD, VPERMQ	Shuffle the four 64-bit vector elements of one 256-bit source operand into a 256-bit destination operand, with a register or memory operand as selector.
VPERM2I128	Shuffle the four 128-bit vector elements of two 256-bit source operands into a 256-bit destination operand, with an immediate constant as selector.

Instruction	Description
VVPBLENDD	Doubleword immediate version of the PBLEND instructions from SSE4.
VPSLLVD, VPSLLVQ	Shift left logical. Allows variable shifts where each element is shifted according to the packed input.
VPSRLVD, VPSRLVQ	Shift right logical. Allows variable shifts where each element is shifted according to the packed input.
VPSRAVD	Shift right arithmetically. Allows variable shifts where each element is shifted according to the packed input.

2.2.1.2. Applications

Most applications will benefit from the new AVX2 instruction set present on Haswell and Broadwell CPUs. The only change required for legacy applications to benefit from AVX2, is recompilation with the required flags (`-xCORE-AVX2` on Intel Compilers). In terms of performance, particularly the applications that were compute bound before, will out of the box achieve nice speed-ups by fusing the multiply with the addition instructions. Theoretically, performance can be 2x higher for this type of applications. In practice, however, users should expect more modest performance increases, up to 30-40%, mostly for programs with high arithmetic intensity, such as the LINPACK benchmark. However, if the application is memory-bound, simply switching from AVX to AVX2 will not give any performance benefits. In order for AVX2 to matter, the memory bandwidth bottleneck has to be removed first. In general, any loop accessing memory beyond L1/L2 and/or performing double precision div/sqrt or permutations will not fully profit of AVX/AVX2.

2.2.1.3. Compiler Support

AVX2 is supported on all major compiling infrastructures such as: GNU/Intel/PGI/LLVM. All compilers support auto-vectorization, with various level of performance. Some more details and examples on auto-vectorization are presented in Section 5.3.2 and Section 5.3.3. To enable AVX2 when compiling with the various available compilers use the following compiler switches:

Table 6. AVX2 flags for the various compilers

Instruction	Description
Intel Compilers	icc/ifort -xCORE-AVX2 ... to compile for AVX2 hardware exclusively, or icc/ifort -axCORE-AVX2 ... to compile for backward compatibility.
PGI compilers	pgcc -tp=haswell ...
LLVM/Clang compiler	clang -mavx2 ...
GNU compilers	gcc -march=haswell ...

More details on the various compiler switches are given in the following chapter, Programming Environments.

2.2.1.4. Example using Intrinsics

AVX instructions improve an application's performance by processing large chunks of values at the same time instead of processing the values individually. These chunks of values are called vectors, and AVX vectors can contain up to 256 bits of data. Common AVX vectors contain four doubles (4 x 64 bits = 256), eight floats (8 x 32 bits = 256), or eight ints (8 x 32 bits = 256). The simple example below will demonstrate the power of AVX/AVX2 processing. Suppose a function needs to multiply eight floats of one array by eight floats of a second array and add the result to a third array. Without vectors, the function might look like this:

```
multiply_and_add(const float* a, const float* b, const float* c, float* d) {

  for(int i=0; i<8; i++) {
```

```
    d[i] = a[i] * b[i];
    d[i] = d[i] + c[i];
  }
}
```

Here's what the function looks like with AVX2:

```
__m256 multiply_and_add(__m256 a, __m256 b, __m256 c) {

  return _mm256_fmadd_ps(a, b, c);
}
```

As can be seen, AVX/AVX2 offers benefit not only by packing the 16 floating-point operations (8 multiplies and 8 additions) into a single instruction, but also by reducing the jump and comparison instructions that were previously required by the for loop. A few intrinsics accept traditional data types like ints or floats, but most operate on data types that are specific to AVX and AVX2. There are six main vector types, listed in the table below.

Table 7. AVX/AVX2 Data Types

Data Type	Description
__m128	128-bit vector containing 4 floats.
__m128i	128-bit vector containing integers.
__m128d	128-bit vector containing 2 doubles.
__m256	256-bit vector containing 8 floats.
__m256d	256-bit vector containing 4 doubles.
__m256i	256-bit vector containing integers.

Each type starts with two underscores, an m, and the width of the vector in bits. AVX512 supports 512-bit vector types that start with _m512, while AVX/AVX2 support 256-bit vector types.

If a vector type ends in d, it contains doubles, and if it doesn't have a suffix, it contains floats. It might look like _m128i and _m256i vectors must contain ints, but this isn't always the case. An integer vector type can contain any type of integer, from chars to shorts to unsigned longs. That is, an _m256i may contain 32 chars, 16 shorts, 8 ints, or 4 longs. These integers can be signed or unsigned.

Although the names of AVX/AVX2 intrinsics can be confusing at the beginning, the convention is actually quite straightforward. A generic AVX/AVX2 intrinsic function is given below: _mm<bit_width>_<name>_<data_type>

The parts of this format are:

- <bit_width> identifies the size of the vector returned by the function. For 128-bit vectors, this is empty. For 256-bit vectors, this is set to 256.
- <name> describes the operation performed by the intrinsic.
- <data_type> identifies the data type of the function's primary arguments.

The last part, <data_type>, identifies the content of the input values, and can be set to any of the following values:

- ps - vectors contain floats (ps stands for packed single-precision)
- pd - vectors contain doubles (pd stands for packed double-precision)
- epi8/epi16/epi32/epi64 - vectors contain 8-bit/16-bit/32-bit/64-bit signed integers

- epu8/epu16/epu32/epu64 - vectors contain 8-bit/16-bit/32-bit/64-bit unsigned integers
- si128/si256 - unspecified 128-bit vector or 256-bit vector
- m128/m128i/m128d/m256/m256i/m256d - identifies input vector types when they're different to the type of the returned vector

As an example, consider _mm256_srlv_epi64. Even if we don't know what the srlv instruction refers to, the _mm256 prefix tells us that the function returns a 256-bit vector and the _epi64 tells us that the arguments contain 64-bit signed integers.

2.3. Memory Architecture

The cache hierarchy of the Haswell/Broadwell systems is similar to prior generations, including an instruction cache, a first-level data cache and a second-level unified cache in each core. On top of these, Haswell systems feature a 3rd-level unified cache with size dependent on specific product configuration. The 3rd-level cache is organized as multiple cache slices, the size of each slice may depend on product configurations, connected by a ring interconnect. It resides in the “uncore” sub-system that is shared by all the processor cores from within a socket. The capabilities of the uncore and integrated I/O sub-system vary across the processor family implementing the Haswell-E microarchitecture. You can find more details by checking the data sheets of respective Intel Xeon E5 v3 processors.

Non-Uniform Memory Access (NUMA) means that memory from different locations may have different access times. For example, in the case of a 2-socket Haswell system, each CPU is linked to separate memories. This effectively creates separate non-uniform memory architecure (NUMA) domains, and extra care should be taken in order to maintain the same performance level when moving from a single-socket to a multi-socket system. Extra care should be taken in the case of a multi-threaded application, as each thread should try to access as much as possible memory that is local to the socket on which the thread is executing. Since each CPU will be connected to a separate set of memories, if CPU0 wants to access data hosted in the memory attached to CPU1, performance hits will be incurred, as both the memory request and the serving of the memory request will pass through the QPI bus connecting the two sockets, prior to addressing the appropriate memory controller. In order to list the existing NUMA domains, we can use the numactl utility. Numactl offers the possibility to control both the scheduling policy (which cores are used), and the memory placement policy (where to allocate data): The command below prints the NUMA configuration of the system.

```
 numactl -H
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11
node 0 size: 32533 MB
node 0 free: 23488 MB
node 1 cpus: 12 13 14 15 16 17 18 19 20 21 22 23
node 1 size: 32768 MB
node 1 free: 25334 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
```

Depending on the way the processors are interconnected (glue-less or glued), remote memory access latency can be two times or even three times as slow as local memory access latency. Also, inter-processor links such as the QPI bus are not used just for memory traffic. They are also used for I/O and cache coherency traffic. For an application that is memory bandwidth constrained, it may run up to 2-3 times slower when most of the memory accesses are remote instead of local. For a non NUMA-aware application, the higher the number of processor sockets, the higher the chance of remote memory access, leading to poorer performance. NUMA effects can be seen in many OpenMP applications utilizing multi-socket systems. This is most common in the case where the master thread initializes all the data structures used by the worker threads. Afterwards, when accessing the data,

a significant number of workers will access remote data, making program execution slower. The solution is to initialize the data structures from the worker threads that will actually work on these structures. This way, all data will be local to the socket that is processing it. For a concrete example of the performance benefits when using OpenMP in a NUMA-aware fashion, we advise the reader to check out the Matrix-matrix multiplication example from Section 5.3.3n.

3. Programming Environment / Basic Porting

3.1. Available Compilers

All of the compilers described in the following support C, C++ and FORTRAN (90 and 2003), all with OpenMP threading support. For information about OpenMP support please refer to Section 3.4

Available Compilers:

- Intel compiler suite, *icc, ifortran, icpc*
- GNU compiler suite, *gcc, gfortran, g++*
- Portland group compiler suite, *pgcc, pgfortran, pgCC*

3.1.1. Intel Compilers

The Intel Compiler suite includes compilers for C/C++ and Fortran, and offers support for:

- Features from Fortran 2003 and Fortran 2008
- ANSI/ISO C99 and C++ standards
- OpenMP 4.0

Table 8. Intel Compiler Flags

Compiler Option	Purpose
-g	Enables debugging; disables optimization
-O0	Disable optimization
-O1	Light optimization
-O2	Heavy optimization (default)
-O3	Aggressive optimization; may change numerical results
-ipo	Inline function expansion for calls to procedures defined in separate files
-funroll-loops	Loop unrolling
-parallel	Automatic parallelization
-openmp	Enables translation of OpenMP directives
-xHost [-xCORE-AVX2]	Sets the AVX2 architecture in Intel compilers

Table 9. Suggested compiler flags for Intel compilers

Compiler	Suggested flags
Intel C compiler	-O3 --xCORE-AVX2 -fma -align -finline-functions
Intel C++ compiler	-std=c11 -O3 -xCORE-AVX2 -fma -align -finline-functions
Intel Fortran compiler	-O3 -xCORE-AVX2 -fma -align array64byte -finline-functions

3.1.2. PGI Compilers

The Portland Group (PGI) compiler suite is a commercial software product containing compilers for Fortran and C/C++. Today, the PGI compilers and tools are made available through NVIDIA under "The Portland Group Compilers and Tools" brand name. The PGI compilers offer full support for:

- Fortran 2003
- ANSI C99 with K&R extensions
- ISO/ANSI and GNU standards for C++
- OpenMP 3.1

Table 10. PGI Compiler Flags

Compiler Option	Purpose
-g	Enables debugging; disables optimization
-O0	Disable optimization; default if -g is specified
-O1	Light optimization; default if -g is not specified
-O or -O2	Heavy optimization
-O3	Aggressive optimization; may change numerical results
-Mipa	Inline function expansion for calls to procedures defined in separate files; implies -O2
-Munroll	Loop unrolling; implies -O2
-Mconcur	Automatic parallelization; implies -O2
-Mprefetch	Control generation of prefetch instructions to improve memory performance in compute-intensive loops.
-Msafeptr	Ignore potential data dependencies between C/C++ pointers
-Mfprelaxed	Relax floating point precision; trade accuracy for speed.
-Mpfi/-Mpfo	Profile Feedback Optimization; requires two compilation passes and an interim execution to generate a profile
-Msmartalloc	Use optimized memory allocation
-fast	A generally optimal set of options including global optimization, SIMD vectorization, loop unrolling and cache optimizations
-mp	Enables translation of OpenMP directives
-tp=haswell	Specific optimization for Haswell

Table 11. Suggested compiler flags for PGI compilers

Compiler	Flags
PGFORTRAN	-tp=haswell -fast -Mipa=fast,inline
PGCC	-tp=haswell -fast -Mipa=fast,inline -Msmartalloc
PGC++	-tp=haswell -fast -Mipa=fast,inline -Msmartalloc

3.1.3. GCC

The GNU Compiler Collection (GCC) includes compilers for C, C++ and Fortran, and libraries for these languages on a variety of platforms including x86. GCC offers:

- Features from Fortran 2003 and Fortran 2008
- Partial support for ANSI/ISO C99
- Support for the ISO/ANSI C++ standard and partial C++11 compatibility
- OpenMP 4.5 support

Table 12. GNU Compiler Flags

Compiler Option	Purpose
-O0	Disable optimization
-O1 or -O	Ligh optimization
-O2	Heavy optimization
-O3	Most expensive optimization (Recommended)
-fopenmp	Enables compiler recognition of OpenMP directives
-march=native	Sets the native architecture of the platform where GCC is running

Table 13. Suggested compiler flags for GNU compilers

Compiler	Suggested Flags
gcc compiler	-march=haswell -O3 -mfma -malign-data=cacheline -finline-functions
g++ compiler	-std=c11 -march=haswell -O3 -mfma -malign-data=cacheline -finline-functions
gfortran compiler	-march=haswell -O3 -mfma -malign-data=cacheline -finline-functions

3.1.4. Compiler Flags

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

Below we only list the most important default flags for the Intel and GNU FORTRAN compilers. For C the default compiler flags are quite similar, however some flags may differ. For a discussion of the common flags for optimization please refer to Section 5.1.

Table 14. Default Intel FORTRAN compiler flags

Default flag	Description
-O2	Optimize for maximum speed
-f[no-]protect-parens	enable/disable (DEFAULT) a reassociation optimization for REAL and COMPLEX expression evaluations by not honoring parenthesis.
-f[no-]omit-frame-pointer	enable (DEFAULT)/disable use of EBP as general purpose register. -fno-omit-frame-pointer replaces -fp
-f[no-]exceptions	enable (DEFAULT)/disable exception handling
-[no-]ip	enable (DEFAULT)/disable single-file IP optimization within files
-[no-]scalar-rep	enable (DEFAULT)/disable scalar replacement (requires -O3)
-[no]pad	enable/disable (DEFAULT) changing variable and array memory layout
-[no-]ansi-alias	enable (DEFAULT)/disable use of ANSI aliasing rules optimizations; user asserts that the program adheres to these rules
-[no-]complex-limited-range	enable/disable (DEFAULT) the use of the basic algebraic expansions of some complex arithmetic operations. This can allow for some performance improvement in programs which use a lot of complex arithmetic at the loss of some exponent range.
-[no-]ansi-alias	enable (DEFAULT)/disable use of ANSI aliasing rules optimizations; user asserts that the program adheres to these rules

Default flag	Description
-[no-]complex-limited-range	enable/disable (DEFAULT) the use of the basic algebraic expansions of some complex arithmetic operations. This can allow for some performance improvement in programs which use a lot of complex arithmetic at the loss of some exponent range.
-no-heap-arrays	temporary arrays are allocated on the stack (DEFAULT)
-[no-]vec	enables (DEFAULT)/disables vectorization
-coarray	enable/disable (DEFAULT) coarray syntax for data parallel programming
-q[no-]opt-matmul	replace matrix multiplication with calls to intrinsics and threading libraries for improved performance (DEFAULT at -O3 -parallel)
-[no-]simd	enables (DEFAULT)/disables vectorization using SIMD directive
-qno-opt-prefetch	disable (DEFAULT) prefetch insertion. Equivalent to -qopt-prefetch=0
-qopt-dynamic-align	enable (DEFAULT) dynamic data alignment optimizations. Specify -qno-opt-dynamic-align to disable
-[no-]prof-data-order	enable/disable (DEFAULT) static data ordering with profiling
-pc80	set internal FPU precision to 64 bit significand (DEFAULT)
-auto-scalar	make scalar local variables AUTOMATIC (DEFAULT)
-[no]zero	enable/disable (DEFAULT) implicit initialization to zero of local scalar variables of intrinsic type INTEGER, REAL, COMPLEX, or LOGICAL that are saved and not initialized
-init=<keyword>	enable/disable (DEFAULT) implicit initialization of local variables of intrinsic type INTEGER, REAL, COMPLEX, or LOGICAL that are asaved and not initialized The <keyword> specifies the initial value keywords: zero (same as -zero), snan (valid only for floating point variables), arrays
-Zp[n]	specify alignment constraint for structures (n=1,2,4,8,16 -Zp16 DEFAULT)
-fstack-security-check	enable overflow security checks. -fno-stack-security-check disables (DEFAULT)
-fstack-protector	enable stack overflow security checks. -fno-stack-protector disables (DEFAULT)
-fstack-protector-strong	enable stack overflow security checks for routines with any buffer. -fno-stack-protector-strong disables (DEFAULT)
-fstack-protector-all	enable stack overflow security checks including functions. -fno-stack-protector-all disables (DEFAULT)
-fpic, -fPIC	generate position independent code (-fno-pic/-fno-PIC is DEFAULT)
-fpie, -fPIE	generate position independent code that will be linked into an executable (-fno-pie/-fno-PIE is DEFAULT)
[no-]global-hoist	enable (DEFAULT)/disable external globals are load safe
-f[no-]keep-static-consts	enable/disable (DEFAULT) emission of static const variables even when not referenced
-mcmodel=<size>	use a specific memory model to generate code and store data. • small - Restricts code and data to the first 2GB of address space (DEFAULT) • medium - Restricts code to the first 2GB; it places no memory restriction on data • large - Places no memory restriction on code or data
-falign-functions=[2|16]	align the start of functions on a 2 (DEFAULT) or 16 byte boundary

Table 15. Default GNU Fortran compiler flags

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

3.2. Available Numerical Libraries

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

3.2.1. Math Kernel Library, MKL

Intel MKL is an integrated part of the Intel compiler suite. The simplest way of enabling MKL is to just issue the flag *-mkl*, this will link the default version of MKL. For a multicore system this links the threaded version. There are both a single threaded sequential version and a threaded multicore version available. The sequential version is very often used with non-hybrid MPI programs. The parallel version honors the environment variable OMP_NUM_THREADS.

Table 16. Invoking different versions of MKL

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

MKL contains a huge range of functions. Several of the common widely used libraries and functions have been incorporated into MKL.

Libraries contained in MKL:

- BLAS and BLAS95
- FFT and FFTW (wrapper)
- LAPACK
- BLACS
- ScaLAPACK
- Vector Math

Most of the common functions that are needed are part of the MKL core routines, Basic linear algebra (BLAS 1,2,3), FFT and the wrappers for FFTW. Software often requires the FFTW package. With the wrappers there is no need to install FFTW, which is outperformed by MKL in most cases.

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

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

Another example using FFTW syntax:

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

The calling syntax for the matrix matrix multiplication is just as for dgemm from the reference Netlib BLAS implementation. The same applies to the FFTW wrappers and other functions. Calling semantics and parameter lists are kept as close to the reference implementation as practically possible.

The usage and linking sequence of some of the libraries can be somewhat tricky. Please consult the Intel compiler and MKL documentation for details. There is a Intel Math Kernel Library Link Line Advisor [https://software.intel.com/en-us/articles/intel-mkl-link-line-advisor] available from Intel. An example for building the application VASP is given below (list of object files contracted to *.o):

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

or even simpler for a fftw example:

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

3.2.2. Intel Performance Primitives, IPP

The Intel Performance Primitives (IPP) library contains functions that do simple operations on scalars, vectors and smaller matrices. It is best suited for image and signal processing, data compression and cryptography.

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

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

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

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

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

```
}


#define N 207374182

main() {

  int16_t x[N];
  __declspec(align(32))  float a[N],b[N],c[N];
  float sum;
  int i,j;
  double t0,t;
  extern double mysecond();

  libinfo();

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

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

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

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

3.2.3. Math library, libimf

Intel Math Libraries (libimf) is an addition to libm.{a,so}, the standard math library provided with gcc and Linux.

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

This is why libm is always listed when one checks the library dependencies using ldd, even when linking with libimf.

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

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

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

3.2.4. Short vector math library, libsvml

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

The svml is linked by default and the functions are not easily available directly from source code. They are, however, accessible through intrinsics. Intel provides a nice overview of the intrinsic functions available [https://software.intel.com/en-us/node/583200]. Usage of intrinics can yield a quite good performance gain. In addition, intrinsics are compatible with newer versions of processors. Usage of inline assembly might not be forward compatible. Intel strongly suggests using intrinsics instead of inline assembly.

A simple example of intrinsic usage is show below:

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

The performance using svml functions can be quite good compared to the serial code below, mostly due to the difference in calls to the *svml_d_pow8* vector function and the libimf serial *pow* function. However, at high optimization the compiler will recognize the simple expression below and vectorize it and performance gain will be smaller. The usage of these intrinsics is best when the compiler fails to vectorize the code.

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

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

3.3. Available MPI Implementations

There are mainly two different MPI implementations available:

- Intel MPI
- OpenMPI

The two implementations use quite similar syntax for compilation and simple runs. However, the mpirun command has many options, which are quite different for the two implementations. Please consult the help files and documentation for details about these options. On most systems the batch system sets up the execution in an adequate way.

Table 17. Implementations of MPI

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

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

3.4. OpenMP

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

Table 18. Versions of OpenMP supported

Compiler suite	Compiler version	Version supported
Intel	2016.1	OpenMP 4.0
GNU	5.2.0	OpenMP 4.0
Portland	15.7	OpenMP 3.1

3.4.1. Compiler Flags

Table 19. OpenMP enabling flags

Compiler	Flag to enable OpenMP
Intel	-qopenmp
GNU	-fopenmp
Portland	-mp

4. Performance Analysis

4.1. Performance Counters

Intel Haswell processors support the Architectural Performance Monitoring Version 3. The complexity of computing systems has tremendously increased over the last decades. Hierarchical cache subsystems, non-uniform memory, simultaneous multithreading and out-of-order execution have a huge impact on the performance and compute capacity of modern processors. CPU utilization numbers obtained from the operating system (OS) is a metric that has been used for many purposes like product sizing, compute capacity planning, job scheduling, and so on. The current implementation of this metric (the number that the UNIX "top" utility and the Windows task manager report) shows the portion of time slots that the CPU scheduler in the OS could assign to execution of running programs or the OS itself; the rest of the time is idle. For compute-bound workloads, the CPU utilization metric calculated this way predicted the remaining CPU capacity very well for architectures of 80ies that had much more uniform and predictable performance compared to modern systems. The advances in computer architecture made this algorithm an unreliable metric because of introduction of multi core and multi CPU systems, multi-level caches, non-uniform memory, simultaneous multithreading (SMT), pipelining, out-of-order execution, etc.

4.1.1. Counters

- 3 Fixed Function Performance Counters: FIXC0, FIXC1, FIXC2.
- 4 General Purpose Performance Counters: PMC0, PMC1, PMC2, PMC3.
- 4 RAPL energy Counters: PWR0, PWR1, PWR2, PWR3.

4.1.2. Events

This architecture has 276 events.

Most events can be measured per SMT thread.

For a recent list of supported events execute `likwid-perfctr` (contained in the likwid toolsuite presented in Section 4.2.3) with the -e switch.

4.1.3. Use-case: Measuring memory bandwidth using uncore counters

There are two kinds of access modes for uncore. One is to use model specific registers (MSRs) which are chip-specific special registers that can be read by privileged rdmsr instructions. This is also how traditional hardware performance counters are implemented. The other access method is via the PCI config space. This is different than MSR access, and requires a different set of tools to read. If your operating system has uncore support it will abstract away the differences between these interfaces.

4.2. Available Performance Analysis Tools

4.2.1. Intel Performance Counter Monitor

Intel’s hardware counters are a real gem to anyone trying to delve into the performance characteristics of the system. There exist hardware counters in almost every corner of the processor. One can measure instructions, cycles, cache misses, memory bandwidth, and even inter-socket details such as Intel QPI traffic. The full list of H/W counters can be found in Intel’s Software Developer manuals. And they are ever-expanding with every new architecture. There are several ways to measure H/W counters. For most stuff, there is *perf*, free and included in most Linux distribution repositories, which supports many H/W counters out-of-the-box and you can also access more H/W counters by simply specifying their details (which you can find on the Intel manuals for your architecture). For more advanced measurements, you can use Intel’s VTune Amplifier, which offers quick templates to

measure classes of H/W counters with tons of configurations options. The UI is extremely helpful as well. Another great free and open-source solution is likwid. And for the experts who need to quickly gain access to custom H/W counters from within their code, there's the free open-source Intel Performance Counter Monitor (PCM) tool. The PCM tool accesses H/W counters through the /dev/cpu/*/msr virtual files (accessible if the msr kernel module is loaded). The problem is that on recent Linux distributions, access to the msr is allowed only with root privileges. Unless you can run your application always with sudo rights, this is a problem. A potential solution is forking off to a separate process with elevated privileges, that can access the MSR counters, and you communicate with it — see the likwid MSRDaemon. This, however, also comes with some disadvantages: (a) you need a separate process, and (b) there's a small time overhead accessing the H/W counters. Ideally, you would like to access the H/W counters from within your application.

4.2.2. Add instrumentation instructions using Extrae

Extrae is a dynamic instrumentation package to trace programs compiled and run with the shared memory model (like OpenMP and pthreads), the message passing (MPI) programming model or both programming models (different MPI processes using OpenMP or pthreads within each MPI process). Extrae generates trace files that can be later visualized with Paraver.

In order to trace an execution, the users typically have to load the module extrae and write a script (called trace.sh in the example below) that sets the variables to configure the tracing tool. It must be executable (`chmod +x ./trace.sh`). The job must run this script before executing the application.

Example for MPI jobs:

```
   #!/bin/bash
# @ output = tracing.out
# @ error = tracing.err
# @ total_tasks = 4
# @ cpus_per_task = 1
# @ tasks_per_node = 12
# @ wall_clock_limit = 00:10

module load extrae

srun ./trace.sh ./app.exe
```

Example of trace.sh script:

```
#!/bin/bash

export EXTRAE_CONFIG_FILE=./extrae.xml
export LD_PRELOAD=${EXTRAE_HOME}/lib/(tracing-library)
$*
```

EXTRAE_CONFIG_FILE points to the Extrae configuration file. Editing this file users can control the type of information that is recorded during the execution and where the resulting trace file is written, among other parameters. (tracing-library) depends on the programming model the application uses:

Table 20. Tracing library Extrae

Job type	Meaning
MPI	libmpitrace.so (C codes) libmpitracef.so (Fortran codes)
OpenMP	libomptrace.so
Pthreads	libpttrace.so

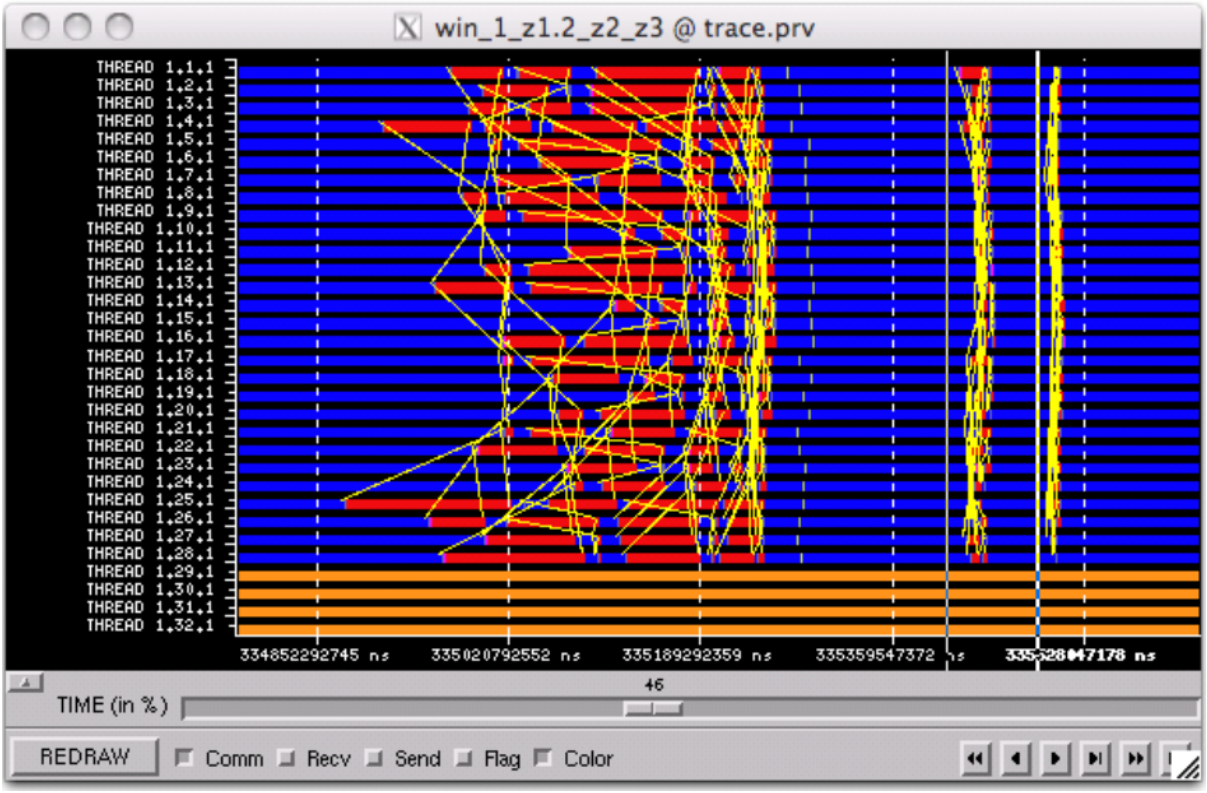
CUDA	libcudatrace.so
MPI+CUDA	libcudampitrace.so (C codes) libcudampitracef.so (Fortran codes)
OmpSs	-
Sequential job (manual instrumentation) (*)	libseqtrace.so
Automatic instrumentation of user functions and parallel runtime calls (**)	-

(*) Jobs that make explicit calls to the Extrae API do not load the tracing library via LD_PRELOAD, but link with the libraries instead.

(**) Jobs using automatic instrumentation via Dyninst neither load the tracing library via LD_PRELOAD nor link with it.

The next figure shows an example of a Paraver trace. Specifically, it is a zoom in on a WRF iteration, executed with 32 processes (MPITrace). The blue area represents computation, orange and red areas represent communication and the yellow lines represent the source and destination of MPI communications.

Figure 3. Extrae + Paraver



More details can be found under https://tools.bsc.es/paraver and https://tools.bsc.es/extrae [https://tools.bsc.es/extrae].

4.2.3. Likwid (Lightweight performance tools)

Likwid stands for "Like I knew what I am doing". This project contributes easy to use command line tools for Linux to support programmers in developing high performance multi threaded programs.

It contains the following tools:

- likwid-topology: Show the thread and cache topology

- likwid-perfctr: Measure hardware performance counters on Intel and AMD processors
- likwid-features: Show and Toggle hardware prefetch control bits on Intel Core 2 processors
- likwid-pin: Pin your threaded application without touching your code (supports pthreads, Intel OpenMP and gcc OpenMP)
- likwid-bench: Benchmarking framework allowing rapid prototyping of threaded assembly kernels
- likwid-mpirun: Script enabling simple and flexible pinning of MPI and MPI/threaded hybrid applications
- likwid-perfscope: Frontend for likwid-perfctr timeline mode. Allows live plotting of performance metrics.
- likwid-powermeter: Tool for accessing RAPL counters and query Turbo mode steps on Intel processor.
- likwid-memsweeper: Tool to cleanup ccNUMA memory domains and force eviction of dirty cachelines from caches.
- likwid-setFrequencies: Tool to set specific processor frequencies.

For more details see the likwid home page [https://code.google.com/archive/p/likwid/].

4.2.4. Intel Tools: Amplifier, Advisor, Inspector, ...

See Section 5.2.

4.3. Intel Software Development Emulator

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

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

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

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

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

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

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

```
Total Mflops double  7798
Total Mflops including FMA instructions
Total Mflops single  0
Total Mflops double  8782
Total Mlops :  8782
```

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

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

5. Tuning

5.1. Guidelines for tuning

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

5.1.1. Top down

If possible try establishing an estimate of the number of floating point operations that is done in total. If it's not possible to calculate, try to make an educated guess. Then compare the run time of the application with the theoretical performance. Intel Haswell/Broadwell offers a theoretical performance of 32 single-precision floating point operations per core per cycle (2 AVX2 FMA units). Thus, by multiplying this number with the clock frequency and with the core count, we get the theoretical peak performance of a specific Haswell/Broadwell part. This is a number arrived to when using all the cores with full vector registers performing only FMA. This number has to be regarded as a marketing number, but the top 500 benchmark HPL might come quite close. Comparing the application in question is a top down approach and a good place to start to establish a performance baseline.

If the application is memory bound the performance measures might be very low compared to the above number. A quick way of checking this is to look at the source and try to establish how many floating point operations you need per byte of data. If this number is very low the application is memory bound and a different set of tuning is needed as opposed to a compute bound application.

5.1.2. Bottom up

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

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

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

5.2. Intel tuning tools

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

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

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

- Intel compiler, analysis of source code.

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

5.2.1. Intel compiler

This section provides some hints and tricks with the Intel compiler tools with some additional reference to the Intel tuning tools. It is not a tutorial for using the compiler and the tuning tools. It will, however, give an overview of what these tools do and provide a guide where to start.

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

5.2.1.1. Compiler optimization

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

5.2.1.1.1. Optimization flags

The following table gives an overview of optimization flags which are useful for a simple start of the tuning process.

Table 21. Common optimization flags

Compiler flag	Information
-O1	Optimize for maximum speed, but disable some optimizations which increase code size for a small speed benefit
-O2	Optimize for maximum speed (default)
-O3	Optimize for maximum speed and enable more aggressive optimizations that may not improve performance on some programs. *For some memory bound programs this has proven to be true.*
-Ofast	Enable -O3 -no-prec-div -fp-model fast=2 optimizations. *This might not be safe for all programs.*
-fast	enable -xHOST -O3 -ipo -no-prec-div -static -fp-model fast=2 *This might not be safe for all programs.*
-xCORE-AVX2	AVX and CORE-AVX2 generate Intel Advanced Vector Extensions code. This is supported by the Haswell/Broadwell and upcoming Intel CPUs. Using -x generates code that runs exclusively on the processor indicated and on future Intel CPUs.
-funroll-loops	Unroll loops based on default heuristics.
-ipo	Enable multi-file IP optimization between files. This option perform whole program optimization. This combined with optimization reports can yield significant insight and performance gain.

5.2.1.1.2. Optimization reporting flags

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

Table 22. Common optimization flags

Compiler flag	Information
-vec-report[=n]	Control amount of vectorizer diagnostic information. Any number from 0 to 7. 3 is most common as it provide information about both vectorized loops and non-vectorized loops, as well as data depenencies.
-qopt-report[=n]	Generate an optimization report. Any level from 0 to 5 is valid. Higher number yield more information.
-qopenmp-report[n]	Control the OpenMP parallelizer diagnostic level, valid numbers are 0 to 2.

5.2.1.1.3. Floating point Optimization

Some flags affect the way the compiler optimizes floating point operations.

Table 23. Floating point flags

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

5.2.1.2. Code generation

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

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

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

This provides output like shown below, where parts of a loop is show:

```
       DO 60 j = 1,n
 10eb:       48 83 c1 04             add    $0x4,%rcx
 10ef:       48 81 f9 00 c2 eb 0b    cmp    $0xbebc200,%rcx
 10f6:       72 e0                   jb     10d8 "MAIN__+0x10d8"
 10f8:       0f ae f0                mfence
         a(j) = b(j) + scalar*c(j)
```

```
60     CONTINUE
       t = mysecond() - t
 10fb:       33 c0                   xor    %eax,%eax
 10fd:       c5 f8 77                vzeroupper
 1100:       e8 00 00 00 00          callq  1105 "MAIN__+0x1105"
       a(n) = a(n) + t
       times(4,k) = t
 1105:       48 b9 00 00 00 00 00    mov    $0x0,%rcx
 110c:       00 00 00
```

5.2.2. Intel MPI library

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

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

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

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

Below is an example of this output:

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

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

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

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

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

The log files contain a detailed log of the MPI communication, one section for each rank. The log provides information about data transfers (which rank communicated with which), MPI calls and message sizes, collectives

with corresponding message sizes, and collectives in context. This is a very good starting point, but the sheer amount of data call for the trace analyzer application. Below is given a small sample of the log file, only the rank 0 communication in summary:

```
   Data Transfers
Src Dst Amount(MB) Transfers
---------------------------------------
000 --> 000 0.000000e+00 0
000 --> 001 2.598721e+02 8515
000 --> 002 5.228577e-01 1427
000 --> 003 4.708290e-01 135
000 --> 004 3.904312e+02 19407
000 --> 005 1.042298e+01 825
000 --> 006 4.864540e-01 143
000 --> 007 4.561615e-01 135
000 --> 008 3.485235e+02 19407
000 --> 009 1.042298e+01 825
000 --> 010 4.864540e-01 143
000 --> 011 4.561615e-01 135
000 --> 012 6.762256e+01 8239
000 --> 013 3.765289e+00 1903
000 --> 014 4.712982e-01 143
000 --> 015 4.419518e-01 135
=======================================
Totals  1.094853e+03 61517
```

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

5.2.3. Intel XE-Advisor

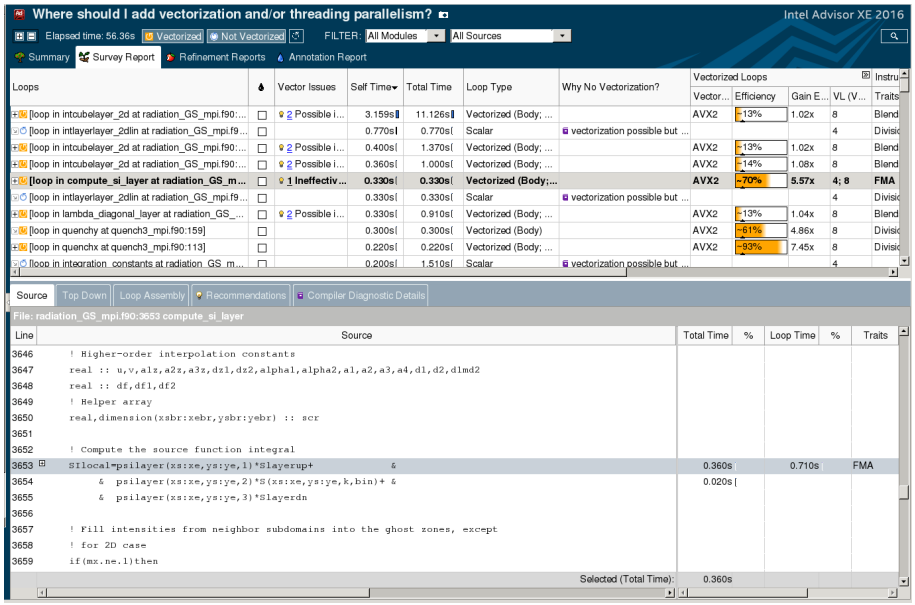
The Intel Vectorization Advisor is an analysis tool that lets you identify if loops utilize modern SIMD instructions or not, what prevents vectorization, what is performance efficiency and how to increase it. Vectorization Advisor shows compiler optimization reports in a user-friendly way, and extends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

Vectorization is very important. The vector units in the Haswell/Broadwell architecture are 256 bits wide. They can hence operate on 8 single precision (32 bits) or 4 double precision (64 bits) numbers simultaneously. A speedup factor of 4 or 8 compared to non-vectorized code is possible.

The Advisor can be used as a graphical X11 based tool (GUI) or it can display results as text using a command line tool. The GUI version provide a large range of different analysis tabs and windows. Intel provides a range of tutorials and videos on how to use this tool. Only some examples will be shown here, most focusing on command line tools and how to use them to collect and analyze.

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

Figure 4. Advisor GUI example



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

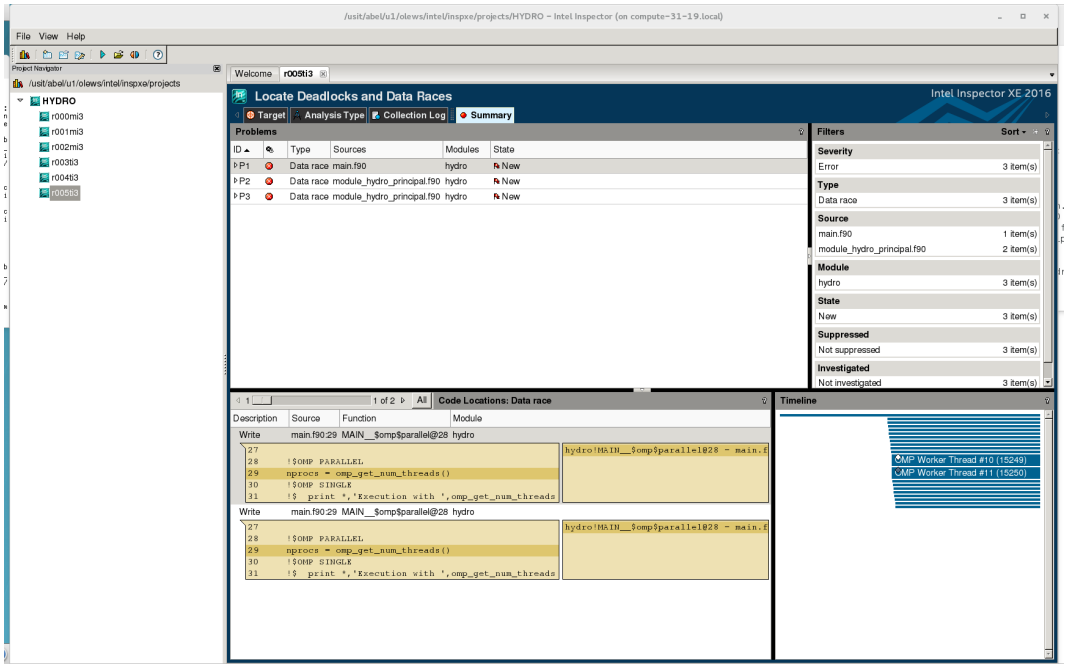
An official Intel FAQ about vectorization can be found here: vectorization-advisor-faq [https://software.intel.com/en-us/articles/vectorization-advisor-faq].

5.2.4. Intel XE-Inspector

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

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

Figure 5. Inspector GUI example



This tool is published by Intel, see: Intel Inspector [https://software.intel.com/en-us/intel-inspector-xe].

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

5.2.5. Intel VTune Amplifier

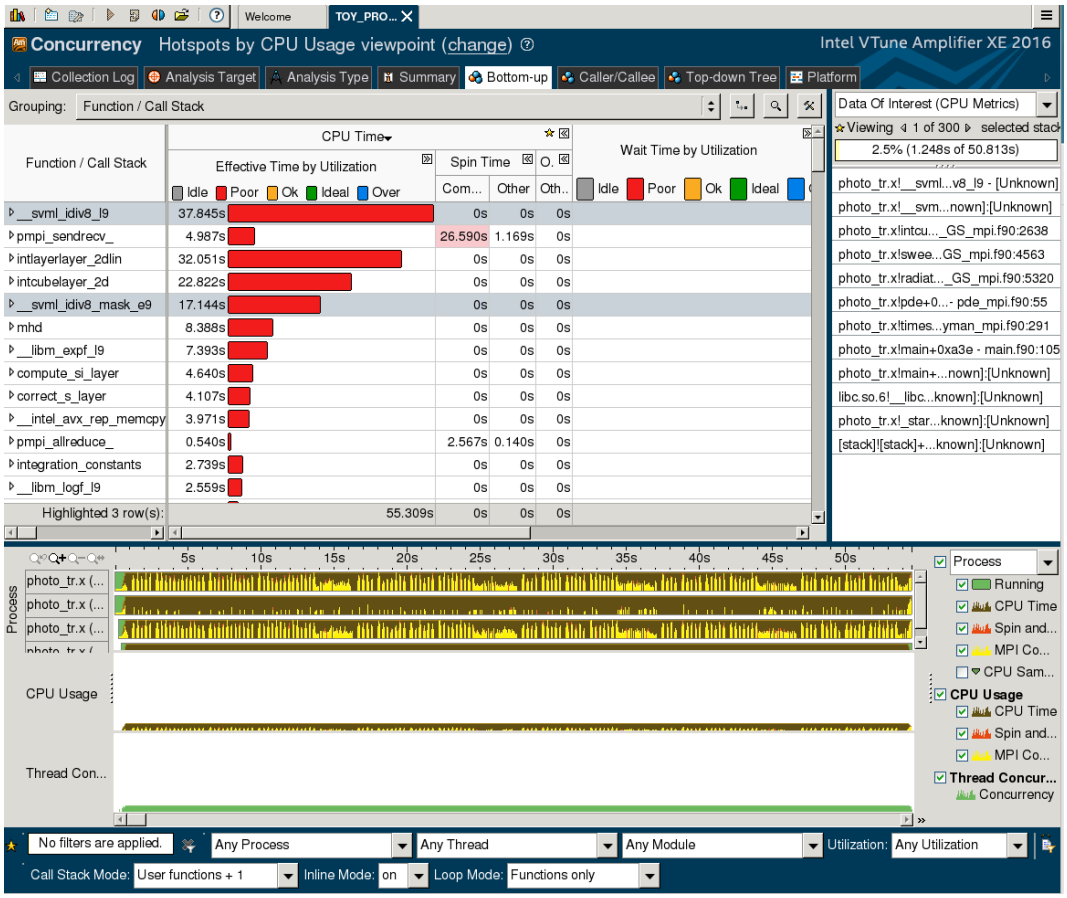
Intel VTune Amplifier provides a rich set of performance insight into CPU performance, threading performance and scalability, bandwidth, caching and much more. Originally it was a tool for processor development, hence the strong focus on hardware counters. The usage of hardware counters needs a kernel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

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

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

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

Figure 6. Amplifier GUI example



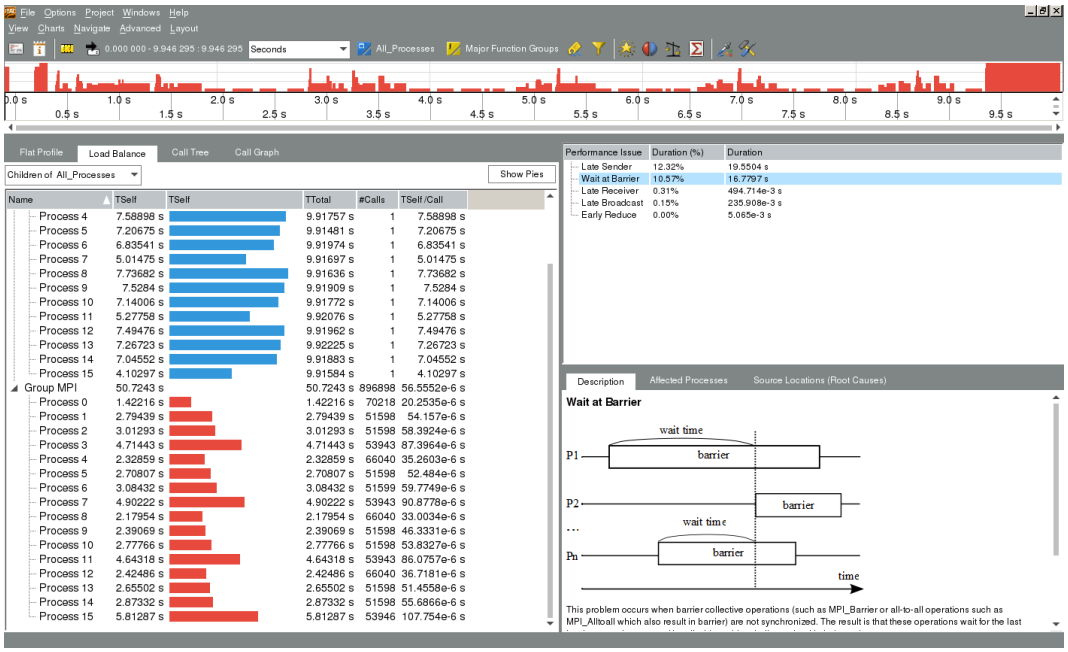
As for the other Intel performance tools this guide is not a guide about details of these tools. More information about the tools, how to obtain, install and use them must be found in the documentation. Official tutorials are found here: VTune tutorials [https://software.intel.com/en-us/articles/intel-vtune-amplifier-tutorials]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

5.2.6. Intel Trace Analyzer

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

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

Figure 7. Trace Analyzer GUI example



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

5.3. Single Core Optimization

There are two types of parallelism that can be exploited when using a single core. The first one is parallelism at the instruction level that is usually achieved by properly scheduling instructions such that it fills as much as possible the execution ports of the CPU. This is typically handled by the compiler. The other type is parallelism at the data level, and is achieved by exploiting the SIMD (single-instruction, multiple data) units present in modern processors. Sometimes this can be handled by the compiler, and we will show examples in the following sections. When the compiler cannot properly vectorize, one can either modify the code such that the compiler better understands the semantics, or provide some hints to the compiler instructing it to explicitly vectorize loops. More advanced scenarios require the use of intrinsics, or for getting as much performance as possible, assembly. Single core optimization also covers each individual MPI rank as the individual ranks are often single core executables. In this section we cover techniques that typically optimize the sequential part of the code. Vectorization is an important element of this section as the performance gain using the vector units are potentially quite high for modern CPUs like Intel Broadwell/Haswell and beyond. More vectorization details and examples are provided in the following sections.

5.3.1. Scalar Optimization

When focusing on single-core optimization, a developer should first begin by collecting software performance metrics and concentrating his/her effort on the parts of the code that could potentially provide the maximum speed-

up. Usually, only a small portion of the code bottlenecks the entire application, so those are the parts that should be addressed first. Hence, good profiling tools and an understanding of the architecture of the targetted system are essential. One of the low-hanging fruits that can give nice performance improvements is to use vendor-optimized libraries such as Intel MKL to offload the hotspots of the applications that were previously determined. If this is possible, it could yield the highest speed-up, as vendor libraries are typically tuned to use all the cores and all the vector capabilities of the target architecture. The second thing to try is to experiment with the various compilation flags, such that different sets of compiler optimizations are tested. Carefully reading the optimization and vectorization reports might also unveil some code change that can be done to allow the compiler to provide more efficient optimizations. The third optimization hint is to be careful about the data use and re-use of the determined hotspots. As Moore's law progresses, we see a widening gap between the memory speed of providing data to the CPU and the CPU's processing speed on those data items. Hardware solutions that alleviate this so-called memory wall are the multi-level cache/TLB hierarchies of modern processors. Caches are transparent to the programmer, and have the role to keep "hot" data close to the processing core. A single cache miss through the whole hierarchy makes the CPU wait for data from main memory, this often taking over 300 cycles. If data has to be fetched from disk this can take over 10M cycles. This would be enough time for the processor to execute hundreds, and up to millions of instructions. Thus, in order to get as close as possible to peak performance, one should be conscious about data use and re-use, and thus keep the data in caches as much as possible.

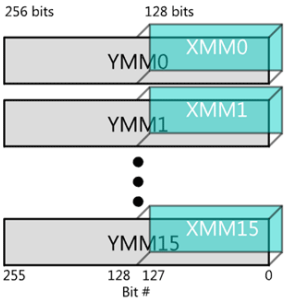
An important class of algorithmic changes involves blocking data structures to fit in cache. By organizing data memory accesses, one can load the cache with a small subset of a much larger data set. The idea is then to work on this block of data in cache. By using/reusing the data elements in cache we reduce the need to fetch data to memory (hence reducing the memory bandwidth pressure). Blocking is a well-known optimization technique that can help avoid memory bandwidth bottlenecks in a number of applications. The key idea behind blocking is to exploit the inherent data reuse available in the application by ensuring that data remains in cache across multiple uses. Blocking can be performed on 1-D, 2-D or 3-D spatial data structures. Some iterative applications can further benefit from blocking over multiple iterations (commonly called temporal blocking) to further mitigate bandwidth bottlenecks. In terms of code change, blocking typically involves a combination of loop splitting and interchange. In most application codes, blocking is best performed by the user by making the right source changes with some parameterization of the block-factors. Let's consider the different levels where data may reside. The closest point to execution units are the processor registers. Data in the registers may be acted upon immediately; incremented, multiplied, added, used in a comparison or boolean operation. Each core in a multicore processor typically has a private first level cache (called L1 cache). Data can be moved from the first level cache to a register very quickly. There may be several levels of cache, at minimum the last level of cache (called LLC) is typically shared among all cores in the processor. Intermediate levels of cache vary depending on the processor whether they are shared or private. On Intel platforms, the caches maintain coherency across a single platform even when there are multiple sockets. Data movement from the caches to the registers is faster than data fetches from main memory.

Another detail that sometimes limits the achieved speed-ups is that data structures might not be correctly aligned. This hurts both the memory subsystem (a 256-bit data structure can be split on 2 cache lines instead of only one), as well as the processing capabilities (the structure has to be read from both cache lines and gathered in a register before an operation can be applied). In general, the compiler will try to fulfill these alignment requirements for data elements whenever possible. In the case of the Intel C++ and Fortran compilers, you can enforce or disable natural alignment using the `-align` (C/C++, Fortran) compiler switch. For structures that generally contain data elements of different types, the compiler tries to maintain proper alignment of data elements by inserting unused memory between elements. This technique is known as "Padding". Also, the compiler aligns the entire structure to its most strictly aligned member. The compiler may also increase the size of structure, if necessary, to make it a multiple of the alignment by adding padding at the end of the structure. This is known as 'Tail Padding'.

5.3.2. Vectorization

We repeat it again: vectorization is very important. The vector units in the Haswell/Broadwell systems are 256 bits wide. They can hence operate on 8 single precision (32 bits) or 4 double precision (64 bits) numbers simultaneously, often in one clock cycle. Also, as described in the Instruction Set architecture section, they support the FMA extension, allowing both a multiplication and an addition in the same cycle. Thus, a maximum speedup of 16 or 8 compared to non-vectorized code is possible.

Figure 8. Vector registers on Haswell/Broadwell systems



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

Use:

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

Avoid:

- Function calls (other than math library calls).
- Non-vectorizable operations (either because the loop cannot be vectorized, or because an operation is emulated through a number of instructions).
- Mixing vectorizable types in the same loop (leads to lower resource utilization).
- Data-dependent loop exit conditions (leads to loss of vectorization).

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

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

5.3.3. Working example: Matrix-Matrix multiplication

In this section we will present several optimizations for a naive matrix-matrix multiplication algorithm. Note that we are only scratching the surface in terms of optimization here, and only give cache/NUMA behavior and vectorization optimizations.

The naive version of the algorithm is given below. It is implemented as a three-level nested for loop that reads in the elements of the b and c matrices, and computes the elements of the a matrix.

```
void basicmm(int n, int m, double a[n][m], double b[n][m], double c[n][m])
{
   int i, j, k ;
   for (i=0;i<n; i++)
      for (j = 0; j<n; j++)
         for (k=0;k<n; k++)
```

```
            a[i][j] += b[i][k]* c[k][j] ;
}
```

When running this naive version, with matrices of 2048x2048 elements, we achieve a wall-time of 67 seconds on a 24-core dual-socket Intel Haswell system. Also, checking the compilation report we see that none of the loops are vectorized. In the example below we introduce the `#pragma simd` directive, marking the innermost loop as good for vectorization.

```
void basicmm_simd(int n, int m, double a[n][m], double b[n][m],
                                                 double c[n][m])
{
 // pragmas added so compiler can generate better code
   int i, j, k ;
   for (i=0;i<n; i++)
      for (j = 0; j<n; j++)
         #pragma simd
         for (k=0;k<n; k++)
            a[i][j] += b[i][k]* c[k][j] ;
}
```

However, when running this version we don't gain much performance. This is because of the memory layout, that is suboptimal, leading to sub-optimal vectorization. The wall-clock time is 73 seconds in this case, on the 2-socket Intel Xeon CPU E5-2690 v3. By just interchanging the innermost two loops, we get over 10-fold performance increase. This step shows that data layout is extremely important for making efficient use of the memory subsystem. The code below executes in 5.8 seconds and illustrates this:

```
void basicmm_reorder(int n, int m, double a[n][m], double b[n][m],
                                                    double c[n][m])
{
   int i, j, k ;
   for (i=0;i<n; i++)
    for (k=0;k<n; k++)
      for (j = 0; j<n; j++)
            a[i][j] += b[i][k]* c[k][j] ;
}
```

By applying the `#pragma simd` directive on this version, vectorization is nicely applied and the execution time goes to 3.93 seconds.

```
void basicmm_reorder_simd(int n, int m, double a[n][m], double b[n][m],
                                                         double c[n][m])
{
 // pragmas added so compiler can generate better code
   int i, j, k ;
   for (i=0;i<n; i++)
    for (k=0;k<n; k++)
      #pragma simd
      for (j = 0; j<n; j++)
            a[i][j] += b[i][k]* c[k][j] ;
}
```

Another optimization that maximizes the CPU caches' efficiency is blocking the memory accesses, to increase data locality. The version below illustrates this:

```
void basicmm_reorder_simd_blocking(int n, int m, double a[n][m], double b[n][m],
```

```
                                    double c[n][m], int blockSize)
{
   int i, j, k, iInner, jInner, kInner ;
   #pragma vector aligned
   for (i = 0; i < n; i+=blockSize)
    for (k = 0 ; k < n; k+=blockSize)
       for (j=0; j<n ; j+= blockSize)
          for (iInner = i; iInner<i+blockSize; iInner++)
             for (kInner = k ; kInner<k+blockSize ; kInner++)
               #pragma vector aligned
               #pragma simd
               for (jInner = j ; jInner<j+blockSize; jInner++)
                 a[iInner][jInner] += b[iInner][kInner] * c[kInner][jInner];
}
```

The blocked matrix-matrix implementation is executed in 2.4 seconds, 24 times faster than the naive implementation. Since this is executed on a 24-core two-socket system, and we are now only making use of a single core, there is still room for performance improvements. Adding the `#pragma omp parallel for schedule(static)` like below allows the program to end 12 times faster, in 0.2 seconds. The speed-up factor is only 12 because of the (rather) small problem size. If we increase the matrix size to 4096x4096, the scaling factor goes to almost 20 on a 24-core NUMA system.

```
void basicmm_reorder_simd_blocking_openmp(int n, int m, double a[n][m],
                          double b[n][m], double c[n][m], int blockSize)
{
   int i, j, k, iInner, jInner, kInner ;
   #pragma omp parallel for schedule(static)
   #pragma vector aligned
   for (i = 0; i < n; i+=blockSize)
    for (k = 0 ; k < n; k+=blockSize)
       for (j=0; j<n ; j+= blockSize)
          #pragma omp parallel for schedule(static)
          for (iInner = i; iInner<i+blockSize; iInner++)
             for (kInner = k ; kInner<k+blockSize ; kInner++)
                #pragma vector aligned
                #pragma simd
                for (jInner = j ; jInner<j+blockSize; jInner++)
                  a[iInner][jInner] += b[iInner][kInner] * c[kInner][jInner];
}
```

The above code achieves this performance (337 times faster than the naive implementation) provided that the code is NUMA-aware. In order for the above-code to be NUMA-aware, the initialization of the arrays has to also be made in an `#omp parallel` loop, such that the memory "operated on" by each core is "owned" by that specific core. The initialization is outlined below:

```
void fillmat(int n, int m, double a[n][m])
{
   int i, j ;
   #pragma omp parallel for schedule(static)
   for (i = 0; i<n; i++)
      for (j = 0 ; j < m; j++)
        a[i][j] = (double)rand();
}
```

By omitting this type of parallel initialization, almost half of the performance is lost because the matrices are allocated in the memory that is local only to a particular socket, and remote to the other.

5.3.4. Compiler autovectorization

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

There are however several ways of helping the compiler to vectorize the code, the modern way of doing this in a portable way is to use OpenMP 4.0 SIMD directives as shown in the previous example. The older Intel specific directives work well on Intel compilers, but are not portable. See Table 22 for more information about reporting compiler options. For hints on using OpenMP SIMD instructions please refer to Section 5.5.

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

5.3.5. Interprocedural Optimization

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

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

5.3.5.1. IPO link time reporting

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

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

An example of a command line might look like this:

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

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

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

with furthermore inside the current loop:

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

```
     [square_gas_mpi.f90(666,8)]

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

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

or like this example:

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

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

5.3.6. Intel Advisor tool

5.3.6.1. Collecting performance data

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

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

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

5.3.6.2. Displaying performance data

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

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

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

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

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

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

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

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

provide valuable insight to what the compiler understands of the source code. The compiler tries to be on the safe side and often assumes dependencies where there might not be any - only the programmer can know this. Informing the compiler about this can in some cases suddenly flip a scalar code to a vectorized one with a potential gain in performance.

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

5.3.7. Intel VTune Amplifier tool

5.3.7.1. Collecting performance data

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

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

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

5.3.7.2. Displaying performance data

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

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

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

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

5.4. Threaded performance tuning

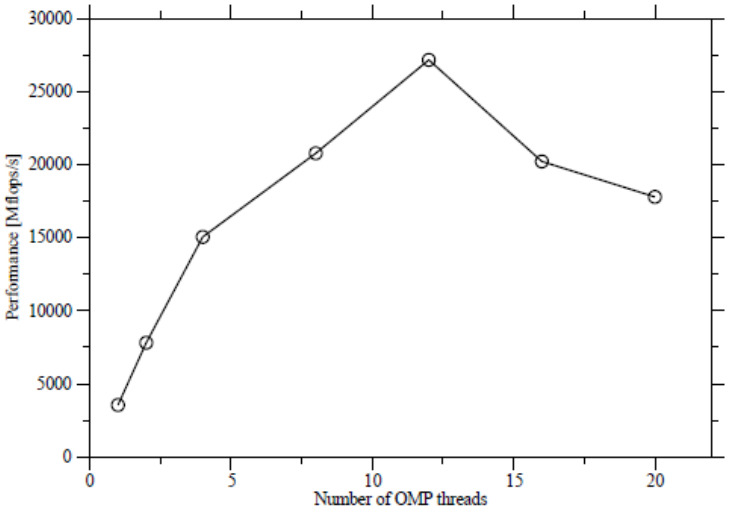
5.4.1. Shared memory / single node

Shared memory applications make up a significant part of the total application suite. Their performance relies on effective vectorization and threading usage. Tuning of OpenMP is covered in Section 5.5.

5.4.2. Core count and scaling

As always one of the fist steps is to assess the performance increase as the number of cores and threads increase. Plotting speedup versus core count is always a good start. Not all threaded applications scale well to a core count of 64 which can be found on some of today's systems. Hence assessment of scaling is important.

Figure 9. Scaling example, one thread per core.



The above scaling example illustrates the point that in many cases scaling is limited and this should always be checked. In this example it is counterproductive to use more than about 12 threads with one thread per core.

One simple way to overcome this scaling problem is to increase the problem size. The example is taken from the well known benchmark mg (rewritten with threads) from NPB. The size is class D which scales to about 12 cores, but the smaller class C only scales to a fraction of this core count.

5.4.3. False sharing

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

A relatively simple example might illustrate this:

```
!$omp parallel private(i,j) shared(a,s)
       do i=1,m
```

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

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

5.4.4. Intel XE-Inspector tool

5.4.4.1. Collecting performance data

Collecting data using the command line tool is quite straightforward.

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

5.4.4.2. Analyzing performance data

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

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

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

The output can look like this:

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

The command line tool is quite powerful and will provide all the data needed to analyze the code using user scripts.

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

5.5. Advanced OpenMP Usage

OpenMP version 4.0 specifies both thread parallelism and SIMD vector parallelism. There is also the concept of workshare which is mostly another form of threading. See the OpenMP web site [http://openmp.org] for more information. A SIMD OpenMP tutorial is beyond the scope of this guide. Only examples how to use the SIMD OpenMP features and tools to do so will be covered. As vectorization is very important it's vital to use the techniques listed in Section 5.3.2 to tune the vectorization.

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

5.5.1. SIMD vectorization

Sometimes the compiler cannot autovectorize the code and the programmer needs to help or instruct the compiler to vectorize the code. Here is where the OpenMP SIMD directives becomes handy.

Table 24. Intel OpenMP SIMD flags

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

OpenMP SIMD directives are used to instruct the compiler to use the SIMD vector unit as the programmer 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, far more powerful than the Intel specific directives.

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

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

A lot of teaching materials have been produced. An Intel presentation about vectorization using SIMD directives can be found here [https://software.intel.com/en-us/videos/vectorizing-fortran-using-openmp-4x-filling-the-simd-lanes]. The presentation contains links to more information from Intel.

5.5.2. Thread parallel

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

Table 25. Intel OpenMP flags

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

Default flag	Description
	• compat - use the GNU compatible thread local storage • legacy - use the Intel compatible implementation (DEFAULT)

5.5.3. Tuning / Environment Variables

The Intel specific environment variables:

- KMP_PLACE_THREADS
- KMP_AFFINITY

allow you to control how the OpenMP runtime uses the hardware threads on the processors. The KMP_AFFINITY variable controls how the OpenMP threads are bound to the hardware resources allocated by the KMP_PLACE_THREADS variable. See table below.

5.5.4. Thread Affinity

For controlling thread allocation see https://software.intel.com/en-us/node/581389 [https://software.intel.com/en-us/node/581389].

The following are the recommended affinity types to use to run your OpenMP threads on the processor:

- compact: sequentially distribute the threads among the cores that share the same cache.
- scatter: distribute the threads among the cores without regard to the cache.

The following table shows how the threads are bound to the cores when you want to use three threads per core on two cores by specifying KMP_PLACE_THREADS=2c,3t:

Table 26. Affinity settings

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

5.6. Advanced MPI Usage

5.6.1. Intel Advisor tool

5.6.1.1. Collecting performance data

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

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

or collecting from all ranks:

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

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

Another example where rank 2 and 3 are run with collection and rank 0,1 and 4 through 27 are run without collection is shown here:

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

5.6.1.2. Displaying performance data

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

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

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

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

To get help how to interpret the advisor output please consult the Inspector documentation that comes with the software.

5.6.2. Intel VTune Amplifier tool

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

5.6.2.1. Collecting performance data

Collecting data with a MPI job is done using some extra options to `mpirun`. Below an example is shown how to collect data from MPI ranks 0 and 2 and place the results into the current directory:

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

or collecting from all ranks:

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

There is a range of differnt metrics to collect, the example shows hotspots which is one of the more common. Other metrices are e.g. advanced-hotspots, memory-access etc. To use these hardware based samples a special module must be loaded. Below two examples are shown how to collect data using the hardware based sampling. The examples show how to collect data from all ranks or only from specified MPI ranks.

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

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

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

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

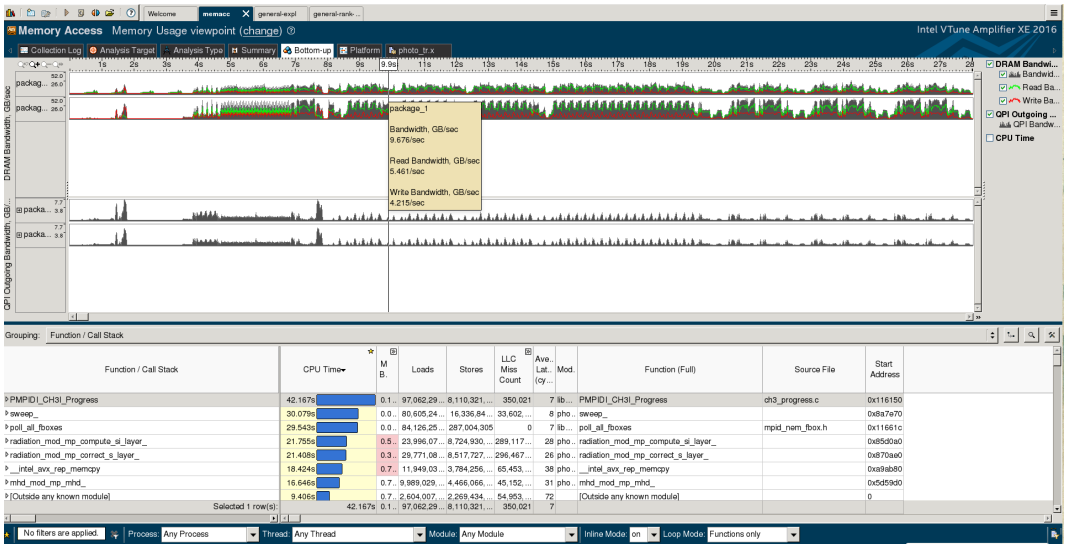
5.6.2.2. Displaying performance data

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

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

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

Figure 10. Example of Amplifier GUI with HW based sampling MPI collections



5.6.3. Intel Trace Analyzer tool

5.6.3.1. Collecting MPI trace data

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

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

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

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

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

An example taken from a real test is given below:

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

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

5.6.3.2. Displaying and analysing MPI trace data

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

Figure 11. Trace Analyzer GUI summary example

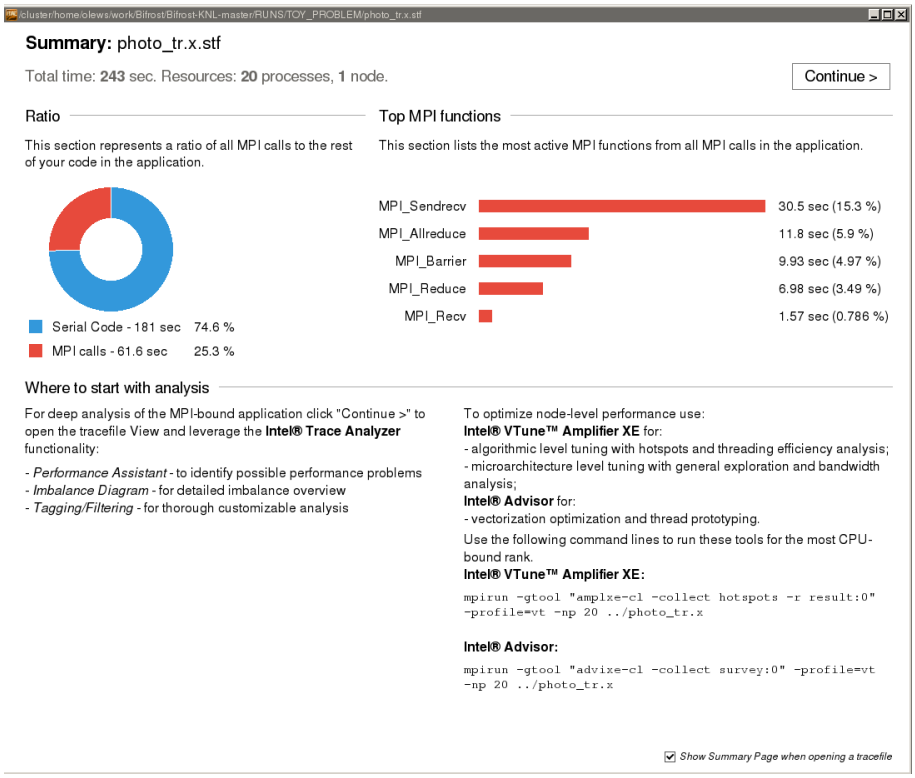
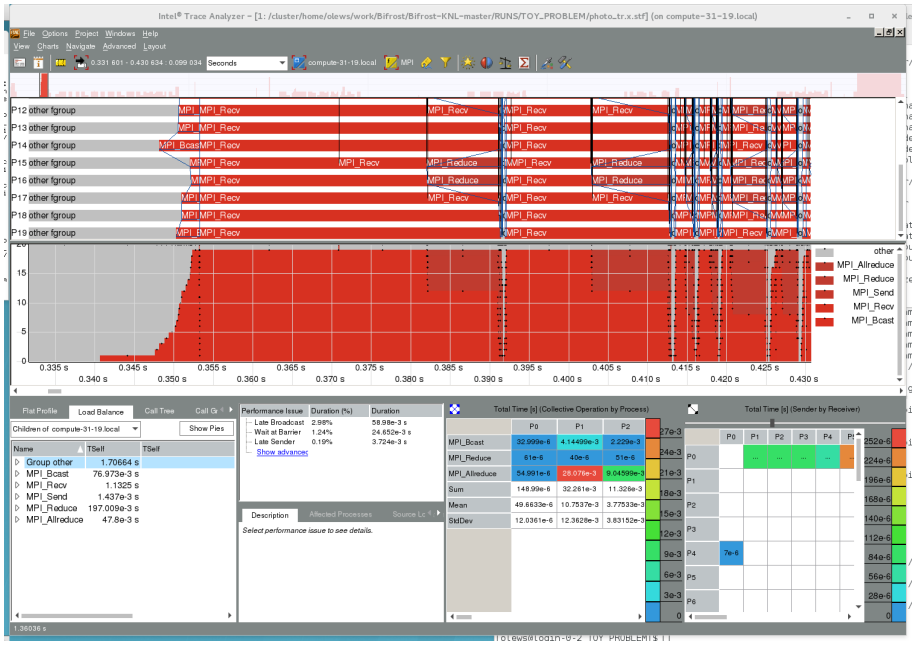


Figure 12. Trace Analyzer GUI complex example



Another option is to analyze the traces with scripts using the ASCII data provided with the tools.

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

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

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

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

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

# COLLOP_PROFILE  <type>:<root>:<communicator>:<index of process>:<count>:
<min_byte_sent>:<max_byte_sent>:<sum_byte_sent>:<min_byte_recv>
:<max_byte_recv>:<sum_byte_recv>:<min_rate>:<max_rate>:<avg_rate>:<min_duration
>:<max_duration>:<sum_duration>
INFO COLLOP_PROFILE 1:0:1:0:690:0:0:0:0:0:0:0.000000:0.000000:0.000000:8192000
                                                         :139264000:14647296000
INFO COLLOP_PROFILE 1:0:1:1:690:0:0:0:0:0:0:0.000000:0.000000:0.000000:40960000
                                                      :339976192000:4353564672000

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

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

6. Debugging

6.1. Available Debuggers

In this section, we shortly describe the ubiquitous gdb and the parallel debugger Totalview.

- GDB

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

- TotalView debugger

  TotalView is a graphical portable powerful debugger from Rogue Wave Software designed for HPC environments. It also includes MemoryScape and ReverseEngine. It can debug one or many processes and/or threads. It is compatible with MPI, OpenMP, Intel Xeon Phi and CUDA.

6.2. Compiler Flags

In this section we present the compiler flags needed for profiling and/or debugging applications. The table below shows common flags for most compilers:

Table 27. Compiler Flags

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

7. European Haswell-based systems

7.1. Hazel Hen @ HLRS

The chapter on “Hazel Hen” is partially based on materials from courses given at HLRS. [1] [2]

Figure 13. Hazel Hen

“Hazel Hen” is another name for the hazel grouse, a bird whose range extends from Europe to Asia.

7.1.1. System Architecture / Configuration

7.1.1.1. Overview on documentations

Starting points to find and get documentations on the Cray XC40 "Hazel Hen" at HLRS are:

General HLRS Documentation: [https://wickie.hlrs.de/platforms/index.php/Platforms]

By HLRS regarding Cray XC40: [https://wickie.hlrs.de/platforms/index.php/Cray_XC40]

CRAY original documents sitemap: [http://docs.cray.com/cgi-bin/craydoc.cgi?mode=SiteMap;f=xc_sitemap]

7.1.1.2. Hardware Architecture

The Cray XC40 Hazel Hen is based on the Intel Haswell Processor and the Cray Aries network. An overview on the techncal data of Hazel Hen and links to more detailed documentation are given at [http://www.hlrs.de/systems/cray-xc40-hazel-hen/].

7.1.1.2.1. Technical Data

Most important technical data are:

[1] Stefan Andersson, Mandes Schönherr (Cray): *Introduction to the Cray XC40 HPC System at HLRS*, October 25, 2016.
[2] Stefan Andersson, Mandes Schönherr (Cray): *Cray XC40 Optimization and Scaling Workshop*, October 26-28, 2016.

Peak performance:	7420 TFlops
Number of compute nodes:	7712
Number of compute cores:	185088 (per node 2 sockets with 12 cores each, therefore 24 cores/node.)
Processor type in compute nodes:	Intel® Xeon® CPU E5-2680 v3 (30M Cache, 2.50 GHz)
Memory/node:	128 GB
Disk capacity:	~10 PB
Node-node interconnect:	Aries

A detailed description of Hardware and Architecture of the Cray XC40 "Hazel Hen" at HLRS can be found at [https://wickie.hlrs.de/platforms/index.php/CRAY_XC40_Hardware_and_Architecture].

7.1.1.2.2. Processor Architecture

Each Intel® Xeon® Processor E5-2680 v3 has 12 cores, 30 MB Cache and a base frequency of 2.5 GHz. For a more detailed description of the processor see [http://ark.intel.com/products/81908/Intel-Xeon-Processor-E5-2680-v3-30M-Cache-2_50-GHz].

7.1.1.2.3. Building Block Architecture

4 nodes are located on one blade; these 4 nodes share one single Aries network chip. 16 Blades are mounted in a chassis and all-to-all connected via a backplane. 3 of these chassis are mounted in a cabinet (rack) and 2 cabinets (6 chassis) build a Cabinet Group. Connections within these groups are realized using copper cables. The cabinet groups are interconnected via optical cables.

7.1.1.2.4. Memory Architecture

Each processor (containing 12 cores) is directly connected to 64 GB of memory, building a NUMA (Non-uniform memory access) domain. Two NUMA domains are located within one node, each node therefore contains 128 GB of memory totally. The two processors within one node are connected via a "QuickPath Interconnect" (QPI) to each other.

Data transfer between cores and memory is much faster within a NUMA domain than between cores and memory located elsewhere. Therefore data used by a certain processor should be placed on the 64 MB memory within the same NUMA domain if possible.

One of the processors has a connection to the Aries network, used for access to the memory on other nodes also.

7.1.1.2.5. Node Interconnect

The communication on the Cray XC runs over the Cray Interconnect, which is implemented via the Aries network chip. The communication via this network is decribed at [https://wickie.hlrs.de/platforms/index.php/Communication_on_Cray_XC40_Aries_network], a more detailed description of the communication network within the Cray XC series is given in [http://www.cray.com/Assets/PDF/products/xc/CrayXC30Networking.pdf].

The network is organized in several levels:

Blades : 4 nodes are located on one blade.

Chassis : A *Rank 1 Network* connects 16 blades, therefore 64 nodes, via a chassis backplane.

Group : A *Rank 2 Network* connects 6 chassis (which are located in 2 cabinets) via copper cables. So 384 nodes are located in each group.

System : A *Rank 3 Network* connects hundreds of cabinets to the whole system.

Adaptive Routing is used to optimize, depending on the load, the paths between the nodes within a group.

7.1.1.3. Home, Scratch, Mid Time Storage

An overview on the resources for storage of data is given at [https://wickie.hlrs.de/platforms/index.php/CRAY_XC40_Disk_Storage].

7.1.1.3.1. File system policy

There is no backup done of any user data located on HLRS Cluster systems. The only protection of user data is the redundant disk subsystem. This RAID system is able to handle a failure of one component. There is no way to recover inadvertently removed data. Users have to backup critical data on their local site.

For data which should be available longer than the workspace time limit allows, and for very important data storage, please use the High Performance Storage System HPSS.

7.1.1.3.2. HOME Directories

Users' HOME directories are located on a shared RAID system and are mounted via NFS on all login (frontend) and compute nodes. The path to the HOME directories is consistent across all nodes. The filesystem space on HOME is limited by a small quota.

Due to the limited network performance, the HOME filesystem is not intended for use in any compute jobs! Do not read or write files within any compute job, as this will cause trouble for all users. For compute jobs please use the workspace mechanism which uses the Lustre based scratch directories.

7.1.1.3.3. SCRATCH directories

For large files and fast I/O Lustre based scratch directories are available. The path to the directories within the *Lustre filesystem (LFS)* is consistent across all nodes too.

The Lustre filesystems are connected via a high speed network infrastructure to the compute nodes. Each Lustre filesystem consists of several *Object Storage Targets (OST)*. This leads to a highly parallel storage system.

At HLRS a special mechanism for scratch directories, named "Workspace mechanism", is implemented to keep data outside of home directories not only during a run, but also after a run: Disk space can be allocated for a number of days. A name is given to each workspace, this name allows to identify and to distinguish several workspaces. Scratch directories are available on all compute and login (frontend) nodes via this workspace mechanism.

Workspaces have some restrictions: There is a time limit for each workspace (currently maximum 60 days), after which they will be deleted automatically. A quota is set on these file systems for each project group; the amount might depend on the project. A detailed description how to use the workspaces is given at [https://wickie.hlrs.de/platforms/index.php/Workspace_mechanism].

7.1.1.3.4. Mid term storage

The High Performance Storage System (HPSS) is designed to manage petabytes of data stored on disks and in tape libraries. An Introduction onto HPSS can be found at [https://wickie.hlrs.de/platforms/index.php/HPSS_Introduction].

The User Access on HPSS is decribed at [https://wickie.hlrs.de/platforms/index.php/HPSS_User_Access].

Because HPSS is a Hierarchical Storage Management System with tape storage included, certain characteristics differ from a disk usage. When transferring a file to the system the file is stored on the disk cache as first step. Later, the data is migrated to two copies on tape. When retrieving a file, it may happen that the file is not on the disk cache anymore and has to be recalled from tape storage. This can take a few minutes.

Some recommendations:

- Do not store large numbers of files.
- Do not recursively store directory structures.

- Collect small files to one larger archive with `tar`. This can also be used to keep the directory structures.
- HPSS is not meant as a backup system.
- Parallel FTP is preferred (instead of `ftp`) for access to the HPSS-server.

On Hazel Hen, there is a Parallel FTP client available. The advantage of Parallel FTP is a file transfer which makes use of several parallel network connections or at least several parallel I/O streams.

On Hazel Hen the module "`hpss`" has to be loaded first: `module load hpss`

To start the client, you should specify the number of parallel streams. We recommend a setting of 4. Call the Parallel FTP client by: `pftp_client -w 4 hpsscore 4021`

On request provide your username and password. The password is your general hww password (as you have it for Hazel Hen). For technical reasons a password change is valid on HPSS with a one day delay. Use the parallel `pput` and `pget` commands instead of the normal `put` and `get`.

More details of access to HPSS can be found in the official User's guide: [http://www.hpss-collaboration.org/user_doc.shtml].

7.1.2. System Access

7.1.2.1. Application for an account

A description of solutions and services at HLRS, including the prerequisites for access to the systems, is given at [http://www.hlrs.de/solutions-services/].

[http://www.hlrs.de/solutions-services/service-portfolio/] gives an overview on general services; the opportunities to get access are described

- for academic users at [http://www.hlrs.de/solutions-services/academic-users/],
- for enterprises at [http://www.hlrs.de/solutions-services/enterprises-sme/].

7.1.2.2. User Regulations

The *User Regulations for Digital Information Processing and Communication Equipment (IaC) at the University of Stuttgart* can be read in English at [http://www.uni-stuttgart.de/zv/bekanntmachungen/bekanntm_179-engl.html] and in the (legally binding) German Version at [http://www.tik.uni-stuttgart.de/dienste/formales/rus-ordnungen/Benutzungsordnung_IuK-Systeme-2006-12-18.pdf].

7.1.2.3. How to Reach the System

General informations on access to Hazel Hen are given at [https://wickie.hlrs.de/platforms/index.php/CRAY_XC40_access]

7.1.2.3.1. Firewall at HLRS

The systems at HLRS are connceted to the outer networks via a firewall. Therefore the IP-adresses of the workstations used to get access to HLRS systems from outside need to be registered there. If this is not done please contact your HLRS contact person (adviser/Betreuer), or submit a bug-report via the Trouble Ticket Submission Form [http://www.hlrs.de/trouble-ticket-submission-form/] to "Accounting", providing your username and the static IP Address, from which you want access.

7.1.2.3.2. Secure Shell (ssh)

Access to the login nodes of Hazel Hen from outside of HLRS is possible via ssh only. There are several login nodes for users available, `hazelhen.hww.de` uses a DNS round robin for load balancing. At [https://wickie.hlrs.de/platforms/index.php/Secure_Shell_ssh] is decribed how to set up ssh.

If you need to configure a firewall at your site please run the command `host hazelhen.hww.de` to get the IP adresses your network administrator needs for the configuration.

7.1.2.3.3. Login Nodes: single point of access

The login nodes are intended as single point of access to the entire cluster. Here you can set your environment, move your data, edit and compile your programs and create batch scripts. Usage which leads to a high load on CPUs or memory, like running your own program, is not allowed. Such programs should run on the compute nodes instead. The compute nodes for running parallel jobs are available through the Batch system only.

7.1.2.3.4. Access to software repositories

Due to security policies from the inside of the HLRS network general internet access is not possible. To get access to software repositories (svn servers, git servers, etc.), an ssh tunnel could be useful. Details are described at [https://wickie.hlrs.de/platforms/index.php/Secure_Shell_ssh#ssh_tunnel].

7.1.2.3.5. Data Transfer with GridFTP

The usual FTP protocol can not utilize high bandwidth channels, as required to transfer large amounts of data. For this task, an extension has been definied: GridFTP supports parallel TCP streams and multi-node transfers to achieve a high data rate via high bandwidth connections.

GridFTP has a typical client/server architecture. If you want to transfer files between your local computer and HLRS, it may be necessary to install a GridFTP client at your computer outside of HLRS. A simple GridFTP client (`globus-url-copy`) is provided by the Globus Toolkit. Please see [https://wickie.hlrs.de/platforms/index.php/Data_Transfer_with_GridFTP] for further details.

7.1.3. Production Environment

7.1.3.1. Description of nodes

7.1.3.1.1. Login nodes

The login nodes are intended for job preparation and submission like editing files, compiling code, submitting jobs to the batch queue and other interactive tasks. They are shared resources, that may be used concurrently by multiple users, and therefore are not intended for usages which lead to a high load.

7.1.3.1.2. Service nodes

Some nodes of Hazel Hen are service nodes, used for different services, like input, output and network connections. These nodes are managing running jobs, but can be accessed using an interactive session too. With `qsub` a batch job is sent to a MOM node ("*Machine Oriented Mini-server*", a special kind of service node), where the batch script is executed. The submitted job scripts (batch scripts) will be executed on these.

The service nodes are shared resources, thus no computational or memory intensive tasks should be performed on this nodes. Only parallel tasks started with `aprun` will be offloaded to the compute nodes. During an interactive session the user's shell will be also located on one of the service nodes similar to an executed batch script. Thus the same procedures and rules have to be respected.

7.1.3.1.3. Graphics and Pre-/Postprocessing nodes

For graphical and other pre- and post-processing purposes several smp nodes equipped with 128 GB, 256 GB resp. 512 GB of main memory have been integrated into the external nodes of Hazelhen. Some other multi user smp nodes are equipped with 1.5 TB of memory. Access to these systems is descried at [https://wickie.hlrs.de/platforms/index.php/CRAY_XC40_Graphic_Environment], some additional information is given at [https://wickie.hlrs.de/platforms/index.php/CRAY_XC40_PrePostprocessing_Environment].

7.1.3.1.4. Compute nodes

Most nodes of Hazel Hen are compute nodes, intended for the compute-intensive usages via the batch system. The compute nodes are exclusively used by one user at a time. They run Compute Node Linux, a version of the

OS optimised for running batch workloads. They can only be accessed in conjunction with a batch system, by starting jobs with `aprun`.

7.1.3.2. Module Environment

HLRS systems use environment modules, packages to manage user environments. These modules allow an easy access to the software packages and program libraries installed at the system. Via the modules each user can configure an individual environment avoiding conflicts between the packages available on the system. A short overview is given at [https://wickie.hlrs.de/platforms/index.php/CRAY_XC40_Environment].

For the dynamic modification of a user's environment module files are provided at the system. Each modulefile contains the information needed to configure the shell for an application. The Cray XC system uses modules in the user environment to support multiple software versions and to create integrated software packages. As new versions of the supported software and associated man pages become available, they are added automatically to the Programming Environment as a new version, while earlier versions are retained to support legacy applications. A default version of an application can be used, or another version by using Modules system commands.

The module tool takes care for environment variables like `PATH`, `MANPATH`, `LD_LIBRARY_PATH`, `LM_LICENSE_FILE` and takes care of compiler and linker arguments of loaded products, including paths, linker paths etc. Metamodules bundle multiple modules. One can create own (meta)modules.

The modules can be activated by the user via the `module` command. This command is decribed at [https://wickie.hlrs.de/platforms/index.php/Module_command]. A description how to prepare own module files is given there also.

At HLRS about 1000 modules are available, regarding libraries (e.g. communication, parallel IO, FFTW, BLAS), tools (e.g. compilers, debuggers, profiler, visualization, scripting) and applications (e.g. Ansys, StarCCM, OpenFOAM, CP2K, Gromacs).

Most important `module` commands are:

- `module avail`: The `avail` option displays all the modules that are available on the system.
- `module avail prod / module avail -S prod`: List all the modules starting with `prod`. If the `-S` option is added, modules containing `prod` are listed also.
- `module list`: The `list` option displays all the modules that are currently loaded into your user environment.
- `module add / module load [modulename]`: The `add` option and the `load` option have the same function - to load the specified module into your user environment.
- `module rm / module unload [modulename]`: The `rm` option and the `unload` option have the same function - to unload the specified module from your user environment. Before loading a module that replaces another version of the same package, you should always unload the module that is to be replaced or use the `module switch` command (described below) alternatively.
- `module switch [modulename] [modulename]/[newversion]`: The `switch` option replaces a currently loaded module by a different one. `/[newversion]` can be ommitted for the default version. When the new module is loaded, the man page for the specified software will be updated also.
- `module display/show [modulename]`: The `display` option shows the changes that the specified module will make in your environment, for example, what will be added to the `PATH` and `MANPATH` environment variables.
- `module whatis/help [modulename]`: Prints the modules (short) description.
- `module load use.own`: add `$HOME/privatemodules` to the list of directories that the module command will search for modules.

`PrgEnv-[X]` are “meta”-modules, loading several modules, including the compiler, the corresponding mathematical libs, MPI and the system environment needed for the compiler wrappers. `PrgEnv-cray` is the default.

7.1.3.3. Batch System

An overview how to run an application via the batch system is given at [https://wickie.hlrs.de/platforms/index.php/CRAY_XC40_Using_the_Batch_System]. More detailed informations can be found in the *Cray Programming Environment User's Guide* [http://docs.cray.com/books/S-2529-116/] and in the *Workload Management and Application Placement for the Cray Linux Environment* Document [http://docs.cray.com/books/S-2496-5202/].

Jobs on the compute nodes of Hazel Hen can be started via the batch system only. The batch system is based on the resource management system `torque` and the scheduler `moab`. The user applications are always launched on the compute nodes using the application launcher, `aprun`, which submits applications to the *Application Level Placement Scheduler* (`ALPS`) for placement and execution. `ALPS` is always used to schedule a job on the compute nodes. A few definitions:

PE: *Processing Element*, basically a Unix 'Process', can be an MPI Task, CAF image, UPC thread, ...

numa_node: *Non-uniform memory access nodes*, the cores and memory on a node with 'flat' memory access, on Hazel Hen the 12 cores on a single socket and the directly attached memory.

Thread: A *thread* is contained inside a process. Multiple threads can exist within the same process and share resources such as memory, while different PEs do not share these resources. Mostly used are OpenMP threads.

`qsub` is the torque submission command for batch job scripts. `aprun` must be used to run applications on the compute nodes both interactively or in a batch job. With `qsub` a batch job is sent to a MOM node ("*Machine Oriented Mini-server*", a special kind of service node), where the batch script is executed. `aprun` starts from there the application on the compute nodes. **If `aprun` is not used, the application is launched on the MOM node and will most likely fail!** `aprun` is decribed in more details later in this document.

A batch script is the usual way to submit a job to the batch system. Interaction with the batch system can happen in two ways: through options specified in job submission scripts or by using `torque` or `moab` commands on the login nodes. There are three key commands used to interact with torque: `qsub`, `qstat`, `qdel`. More advanced commands and options can be seen via `man pbs` (Portable Batch System).

7.1.3.3.1. Requesting resources using the batch system

7.1.3.3.1.1. Batch Mode

Production jobs are typically run in batch mode via shell scripts. The number of required nodes, cores, wall time and more can be set by the parameters in the job script header with "`#PBS`":

```
#!/bin/bash
#PBS -N job_name
#PBS -l nodes=2:ppn=24
#PBS -l walltime=00:20:00

# Change to the directory where the job was submitted from
cd $PBS_O_WORKDIR

# Launch the parallel job to the allocated compute nodes

aprun -n 48 -N 24 ./my_mpi_executable arg1 arg2
```

The job is submitted by the qsub command

```
qsub my_job_script.pbs
```

All script head parameters `#PBS` can also be adjusted directly by `qsub` command options. Setting `qsub` options on the command line will overwrite the settings given in the batch script:

```
qsub -N other_name -l nodes=2:ppn=24,walltime=00:20:00 my_job_script.pbs
```

Resources are not granted to the batch job immediately; the job might wait in the queue of pending jobs before its resources become available.

The example listed above will run the executable `my_mpi_executable` in 48 parallel MPI processes. Torque will allocate 2 nodes to your job for a maximum time of 20 minutes and place 24 processes on each node (one process per core). The batch system allocates nodes exclusively for one job only. After the walltime limit is exceeded, the batch system will terminate your job. The job launcher for the XC40 parallel jobs `aprun` needs to be started from your workspace (a subdirectory of `/mnt/lustre_server`).

The `aprun` example above will start the parallel executable `my_mpi_executable` with the arguments `arg1` and `arg2`. The job will be started using 48 MPI processes with 24 processes on each of your allocated nodes (remember that a node consists of 24 cores on Hazel Hen). You need to have nodes allocated by the batch system (`qsub`) before starting `aprun`. The maximum size of a job is determined by the resources requested when the batch session is launched with the `qsub` command.

To see further options of `aprun`, please use `man aprun` and `aprun -h`.

7.1.3.3.1.2. Interactive batch Mode

The interactive mode is typically used for debugging or optimizing of code, but not for production runs. To start an interactive session, use the `qsub -I` command:

```
qsub -I -l nodes=2:ppn=24,walltime=00:30:00
```

If the requested resources are available (in the example above: 2 nodes/24 cores for 30 minutes) you get a session on the MOM node ("*Machine Oriented Mini-server*", a special kind of service node). Now you have to use the `aprun` command to launch your application to the allocated compute nodes. To finish enter `logout` to exit the batch system and return to the normal command line.

7.1.3.3.1.3. Runnning a job on another Account ID

There are Unix groups associated to the project account ID (`Acid`). To run a job on a non-default project budget (associated to a secondary group), the groupname of this project has to be passed in the group_list:

```
qsub -W group_list=groupname ...
```

This is neither applicable nor necessary for the default project (associated to the primary group), printed with `id -gn`. To get your available groups use `id -Gn`.

7.1.3.3.1.4. Usage of a Reservation

For nodes which are reserved for special groups or users, you need to specify an additional option for this reservation: E.g. a reservation named `john.1` will be used with the command `qsub -W x=FLAGS:ADVRES:john.1 ...`.

7.1.3.3.1.5. Status Information

Status informations can be seen by commands described at [https://wickie.hlrs.de/platforms/index.php/CRAY_XC40_Using_the_Batch_System#Status_Information].

The status of jobs is shown by the commands `qstat`, `qstat -a` and `showq`; the status of qeues (in two different formats) by `qstat -q` or `qstat -Q`.

The status of job scheduling is shown by `checkjob [jobID]` .

`showbf` can help to build small jobs that can be backfilled immediately before the resources to become available for larger jobs.

The status of Nodes/System is shown by the commands `xtnodestat` and `apstat`.

For further details type on the login node:

```
man qstat
man apstat
man xtnodestat
showbf -h
showq -h
checkjob -h
```

All these commands show the state of the system at the moment when the command is issued. The starting time of jobs for instance also depends on other events like jobs submitted in the future, which may fit better into the scheduling of the machine, on the shape of the hardware, other queues and reservations.

7.1.3.3.1.6. Deleting a Batch Job

The commands `qdel` *jobID* and `canceljob` *jobID* allow to remove jobs from the job queue. If the job is running, `qdel` will abort it. The Job ID is given in the output of the `qsub` command used to start your job, but can also be obtained from the output of `qstat` later.

7.1.3.3.1.7. Limitations

An introduction on the Batch System Layout and the System Limits (Job limits and queues) is given at: [https://wickie.hlrs.de/platforms/index.php/CRAY_XC40_Batch_System_Layout_and_Limits].

7.1.3.4. Accounting

To each project a "*project account ID*" (`Acid`) is associated. This account ID is an integer number, usually used as Unix group id (`gid`) for the project also. All activities done under the same account ID are accounted to the same project.

A limited amount of computing time, named "*Resource Time*" (measured in hours, abbreviated "*RTh*"), is given to each project. At HLRS usually the granted resource time is given in core hours. Nevertheless, because on Hazel Hen complete nodes are allocated to a user, accounting on this machine is done by "*node hours*". Each Hazel Hen node contains 24 cores, therefore the accounted node hours (duration of allocation * number of allocated nodes) should be multiplied by 24 to get a comparable value to the granted resource time in core hours.

To get an overview on consumed accounting units HLRS provides a statistic interface for the customers, typically to be used by the project managers. The statistics include all accounting data up to the day before. Three types of statistics are provided by now:

- monthly sums of the consumed *Resource Time* based on user and resource,
- job detailed statistics to view sums of the consumed *Resource Time*, based on single jobs within a given time-frame,
- job detailed statistics to view the consumed *Resource Time*, based on single jobs and node types within a given timeframe.

This interface can be reached via [https://java.hlrs.de/hpc-projects/ProjectManager/]. Login data are the *Acid* and a password, given to the project leader by HLRS.

7.1.3.5. Cray Documentations

The *Cray Programming Environment User's Guide* describes the software environment and tools used to develop, debug, and run applications on Cray XT, Cray XE, Cray XK, and Cray XC40 systems. It is intended as a general overview and introduction to the Cray system for new users and application programmers. It might be a good idea to read at least chapters 1-4: [http://docs.cray.com/books/S-2529-116/].

The *CLE User Application Placement Guide* should be used in conjunctio and describes how to launch and execute applications using the Cray Linux Environment (CLE) with the Application Level Placement Scheduler (ALPS) and `aprun` command in considerably greater detail: [http://docs.cray.com/books/S-2496-5204/].

7.1.3.6. Further User Documentations

A general starting point to get CRAY customer documentation is [http://docs.cray.com/]. Some manuals, which might be helpfull, are:

- *Workload Management and Application Placement for the Cray Linux Environment* [http://docs.cray.com/books/S-2496-5202/]
- *Cray Linux Environment (CLE) Software Release Overview* [http://docs.cray.com/books/S-2425-52xx/]
- *Cray C and C++ Reference Manual* [http://docs.cray.com/books/S-2179-83/]
- *Cray Fortran Reference Manual* [http://docs.cray.com/books/S-3901-83/]

7.1.3.7. Training

An overview on Training courses and workshops at HLRS (including usage of Hazel Hen, but on oher subjects also) is given at [http://www.hlrs.de/solutions-services/service-portfolio/training/].

7.1.3.8. Frequently Asked Questions (FAQs)

A FAQ section specific to XC40 related information, which will be extended continuously, is located at [https://wickie.hlrs.de/platforms/index.php/CRAY_XC40_FAQ].

7.1.3.9. Trouble Ticket Submission Form

A form for trouble ticket submission is found at [http://www.hlrs.de/solutions-services/service-portfolio/trouble-ticket-submission-form/].

7.1.3.10. Troubleshooting Support (contact/staff)

Informations how to reach the staff for further support is given at [https://wickie.hlrs.de/platforms/index.php/CRAY_XC40_Support].

7.1.3.11. Mailing list regarding system news

HLRS has established a mailing list to provide users automatically and on time with important informations about HLRS-Systems, their status, problems, maintenance schedules etc. To subscribe resp. unsubscribe from this list please go to [https://listserv.uni-stuttgart.de/mailman/listinfo/hwwsysnews/].

7.1.4. Programming Environment

7.1.4.1. Available Compilers

On Hazel Hen four compiler environments are available:

Vendor	**Module name**
Cray	`PrgEnv-cray` (default)
Intel	`PrgEnv-intel`
GNU	`PrgEnv-gnu`
PGI	`PrgEnv-pgi`

All compilers are accessed through wrappers named `ftn`, `cc` and `CC`. To change a compiler or a version use `module swap`. It depends on the application, and even on the input at runtime, which compiler generates the fastest executable. The wrappers are scripts, which choose the required compiler version, target architecture options, scientific libraries and include files from the module environment. [https://wickie.hlrs.de/platforms/index.php/CRAY_XC40_compiler_wrapper] describes some more details.

The wrapper scripts are prepared for cross compilation, they create a highly optimized executable tuned for the compute nodes. This executable may fail on login nodes and pre/postprocessing nodes. If compiling for the login nodes, the original direct compiler commands, e.g. `ifort`, `pgcc`, `crayftn`, `gcc`, ... should be used. All libraries have to be linked in manually then. Alternatively the compiler wrappers could be used with a `-target-cpu=` option. The `x86_64` is the most compatible, but also less specific one.

For libraries and include files being triggered by module files, nothing should be added to the Makefile. No additional MPI flags, and no `-I`, `-l` or `–L` flags for the Cray provided libraries are needed, because they are included by wrappers. If the Makefile or Cmake requires an input for `–L` to work correctly, try using '`.`'. If a specific path really is needed, checking `module show <X>` for environment variables may help.

Currently, static linking is default. To decide how to link, either set `CRAYPE_LINK_TYPE` to `static` resp. `dynamic`, or pass the `–static` resp. `-dynamic` option to the wrapper. The `–shared` option is used to create shared libraries `*.so`.

Dynamic linking delivers smaller executable and makes automatic use of new libraries. A longer startup time to load and find the libraries might be needed. The environment (loaded modules) should be the same between compiler setup and batch script (eg. `PrgEnv-intel`). To hardcode the `rpath` into the executable, set `CRAY_ADD_RPATH=yes` during compilation. This will always load the same version of the library when running, independent of the version loaded by modules. Static linking allows a faster startup, the application will run the same code every time it runs (independent of the environment).

OpenMP is supported by all of the PrgEnvs. CCE (`PrgEnv-cray`) recognizes and interprets OpenMP directives by default. If there are OpenMP directives in the application, which should not be used, they can be disabled by `–hnoomp`. The options to enable OpenMP are:

```
PrgEnv-cray   -homp

PrgEnv-intel  -openmp

PrgEnv-gnu    -fopenmp

PrgEnv-pgi    -mp
```

More information on individual compilers is given in the `man`-pages:

PrgEnv	**C**	**C++**	**Fortran**
`PrgEnv-cray`	`man craycc`	`man crayCC`	`man crayftn`
`PrgEnv-intel`	`man icc`	`man icpc`	`man ifort`
`PrgEnv-gnu`	`man gcc`	`man g++`	`man gfortran`
`PrgEnv-pgi`	`man pgcc`	`man pgCC`	`man pgf90`
Wrappers	`man cc`	`man CC`	`man ftn`

The version of a compiler can be checked with the `-V` option on a `cc`, `CC`, or `ftn` command with PGI, Intel and Cray compilers, with the `--version` option with GNU compilers.

7.1.4.1.1. Recommended compiler optimization levels

`PrgEnv-cray` The default optimization level is equivalent to `–O3` of most other compilers. `-hfp3` gives additional floating point optimizations. In case of precision errors, try a lower `–hfp<number>` (`-hfp1` first, `-hfp0` only if absolutely necessary).

`PrgEnv-intel` The default optimization level (equal to `-O2`) is safe. Try with `–O3`. If that works, try with `–Ofast -fp-model fast=2`.

`PrgEnv-gnu` Almost all HPC applications compile correctly with using `-O3`, so use that instead of the cautious default. `-ffast-math` may give some extra performance.

7.1.4.1.2. Inlining and inter-procedural optimization

`PrgEnv-cray` Inlining within a file is enabled by default. The command line options `–OipaN` (`ftn`) and `–hipaN` (`cc`/`CC`) with `N=0..4` provide a set of choices for inlining behavior: `0` disables inlining, `3` is the default, `4` is even more elaborate. `–Oipafrom=` (`ftn`) or `–hipafrom=` (`cc`/`CC`) instructs the compiler to look for inlining candidates from other source files, or a directory of source files. The `-hwp` combined with `-h pl=...` enables whole program automatic inlining.

`PrgEnv-intel` Inlining within a file is enabled by default. Multi-file inlining is enabled by the flag `-ipo`.

`PrgEnv-gnu` Quite elaborate inlining is enabled by `–O3`.

7.1.4.1.3. Loop transformations

`PrgEnv-cray` Most useful techniques are in their aggressive state already by default. Loop restructurization might be improved by `–h vector3`.

`PrgEnv-intel` Loop unrolling is enabled with `-funroll-loops` or `-unroll-aggressive`.

`PrgEnv-gnu` Loop blocking is enabled by `-floop-block`, loop unrolling by `-funroll-loops` or `-funroll-all-loops`.

7.1.4.1.4. Directives for the Cray Compiler

On the Cray Compiler directives can be used to control blocking, unrolling and vectorizing of loops. Examples:

```
!dir$ concurrent
!dir$ ivdep
!dir$ interchange
!dir$ unroll
!dir$ loop_info [max_trips] [cache_na]
!dir$ blockable
```

More information is given in `man directives` and `man loop_info`.

7.1.4.2. Software Development Tools and Libraries

[https://wickie.hlrs.de/platforms/index.php/Software_Development_Tools,_Compilers_%26_Libraries] gives an overview on software development tools, compilers and libraries installed at HLRS.

The *SuperLU Users Guide* describes a collection of three ANSI C subroutine libraries for solving sparse linear systems of equations AX = B . The guide is found at [http://docs.cray.com/books/S-6532-30/].

7.1.4.3. Application software packages

At [https://wickie.hlrs.de/platforms/index.php/Application_software_packages] is an overview on application software packages at HLRS given.

7.1.5. Performance Analysis

7.1.5.1. Available Performance Analysis Tools

7.1.5.1.1. CrayPat-lite

CrayPat-lite is an easy-to-use version of the Cray Performance Measurement and Analysis Tool set. Example of its usage: First recompile and build an instrumented binary:

```
module load perftools-base
```

```
module load perftools-lite
make clean; make
```

Then run this instrumented binary:

```
module load perftools-base
module load perftools-lite
aprun –n 24 app.exe >& job.out
```

Running the instrumented binary creates a `*.rpt` and a `*.ap2` file, the report is additionally printed to `stdout`. Observations, e.g. real time in functions and IO observations, and suggestions, e.g. for rank reordering, are printed.

The example above generates a **sampling profile**. To generate a **tracing profile** instead, replace `perftools-lite` by `perftools-lite-events`. A tracing profile is comparable to a sampling profile, but now the functions are really traced from the beginning to the end. A **loop profile** is generated by setting `perftools-lite-loops` at the same locations instead.

7.1.5.1.2. Further Cray Performance Measurement and Analysis Tools

A detailed description of Cray Performance Measurement and Analysis Tools is given at [http://docs.cray.com/books/S-2376-622/].

7.1.6. Tuning

7.1.6.1. Programming How-To's, Tips & Tricks

[https://wickie.hlrs.de/platforms/index.php/Programming_How-To's,_Tips_%26_Tricks] is an entry point for informations on optimization and other tips regarding the HLRS systems.

7.1.6.2. Usage of Huge Pages

Modern computer architectures use virtual memory pages. If an array in memory is accessed, the system will have to access the page table first to find out where the page is stored and then in a second transaction access the data. If hugepages are used, the size of the actual page is changed from small pages (4k) to a larger size. By increasing the size of memory pages more physical memory is mapped within one virtual page. This can improve the application performance. See [https://wickie.hlrs.de/platforms/index.php/HugePages] for further details.

7.1.6.3. Short Reads for I/O Optimisation

The POSIX standard allows the read to return with less data than actually requested. This is called a short read. The read command will return the length of the record read. A description is located at [https://wickie.hlrs.de/platforms/index.php/Lustre_short_read#lustre_read].

7.1.6.4. Optimization of I/O by striping

To get the best performance out of the Lustre filesystem it may be necessary to optimize the striping.

`lfs` is the Lustre utility for setting the stripe properties of new files, or displaying the striping patterns of existing ones. The most used options are:

`setstripe`	Set striping properties of a directory or new file
`getstripe`	Return information on current striping settings
`df`	Show disk usage of this file system
`help`	Show an overview of options
`lfs help [option]`	Show a more detailed description of this option

`lfs df [filesystem]` lists all OSTs (and some more information) of a file system. The number of OSTs in a filesystem is shown e.g. by `lfs df [filesystem] | grep 'OST:' | wc -l`.

`lfs setstripe [--stripe-size|-s size] [--stripe-count|-c count] [file|dir]` sets the stripe for a file or a directory. The parameters are:

`size` Number of bytes on each OST (0: filesystem default)

`count` Number of OSTs to stripe over (0: default, -1: all)

The stripe values of a file are set when the file is created. It is not possible to change them afterwards. `lfs` can create an empty file with the stripes wanted (like the touch command). `lfs` can apply striping settings to a directory, any children will inherit parent's stripe settings on creation.

Selecting the striping values will have a large impact on the I/O performance of the application. Try to use all OSTs:

`#files > #OSTs :` Set `stripe_count = 1`. You will reduce the lustre contention and OST file locking this way and gain performance.

`#files == 1 :` Set `stripe_count = #OSTs`.

`#files < #OSTs :` Select `stripe_count` so that you use all OSTs. Example: If you have 8 OSTs and write 4 files at the same time, then select `stripe_count = 2`.

Conclusions: Lustre is a high performance, high bandwidth parallel file system. It requires many multiple writers to multiple stripes to achieve best performance. There is large amount of I/O bandwidth available to applications that make use of it. However users need to match the size and number of Lustre stripes to the way files are accessed:

- Large stripes and counts for big files
- Small stripes and counts for smaller files

Use Lustre for what it is designed for: Lustre aggregates multiple storage devices providing scalable I/O for very large systems. Sweet-spot is writing of large files. Lustre is designed to provide a consistent (POSIX) view of the filesystem and this requires extra work to maintain So don't use Lustre for local `TMPDIR`. This can be particularly problematic for large compilations.

Some expensive metadata operations: The `stat` operations return information on file ownerships, permissions, size, update times etc. To obtain the file size requires a lookup on the meta data and an enquiry for file size on each OST owning a stripe. Therefore

- Avoid `ls -l` (and coloured `ls`)
- Avoid file name completion in shells
- Use `Open` and check for failure instead of `stat`/`INQUIRE`
- Don't stripe small files (you may have to check every OST that might own a part of the file)

7.1.7. MPI

The implementation of MPI on Hazel Hen is based on MPICH3 from the Argonne National Laboratory. It includes many improved algorithms and tweaks for Cray hardware:

- Improved algorithms for many collectives
- Asynchronous progress engine, that allows overlap of computation and communication
- Customizable collective buffering when using MPI-IO
- Optimized Remote Memory Access (one-sided), including passive RMA

MPI-3 support is implemented with minor exceptions, support for Fortran 2008 bindings is included.

Module `cray-mpich` must be loaded to use MPI; this should be done by default. `module list` should show this module, followed by the version number, e.g. `cray-mpich/7.4.4`.

7.1.7.1. Some basic MPI-Job examples

Single node, single task - Run a job on one task on one node with full memory:

```
#PBS –l nodes=1:ppn=24
...
aprun –n 1 ./<exe>
```

Single node, Multiple Ranks - Run a pure MPI job with 24 ranks or less on one node:

```
#PBS –l nodes=1:ppn=24
...
aprun –n 24 ./<exe>
#aprun –n 8 ./<exe>
```

Multiple nodes, Multiple Ranks - Run a pure MPI job on 4 nodes with 24 MPI ranks or less on each node:

```
#PBS –l nodes=4:ppn=24
...
aprun –n 96 –N 24 ./<exe>
#aprun –n 32 –N 8 ./<exe>
```

Pure OpenMP Job - Using 4 threads on a single node:

```
#PBS –l nodes=1:ppn=24
...
export OMP_NUM_THREADS=4
echo "OMP_NUM_THREADS: $OMP_NUM_THREADS"
aprun –n 1 –d $OMP_NUM_THREADS ./<omp_exe>
```

Hybrid MPI/OpenMP job - 3 nodes with 12 MPI ranks per node, 4 threads for each rank, using Hyperthreads:

```
#PBS –l nodes=3:ppn=24
...
export OMP_NUM_THREADS=4
echo "OMP_NUM_THREADS: $OMP_NUM_THREADS"
aprun –n 36 –N 12 –d $OMP_NUM_THREADS –j 2 ./<hybrid_exe>
```

7.1.7.2. Run-time optimization for MPI

7.1.7.2.1. Performance analysis

For a performance analysis profiling tools, at least CrayPat-lite, should be used. Usually it will be reasonable to select a test case, e.g. with a smaller simulation time as the intended production runs, for the performance analysis.

Comparisions of packed and unpacked CPU-usage, as in the lines

```
aprun --p-state 2500000 –N 12 –S 12
aprun --p-state 2500000 –N 12 –S 6
```

show memory bandwidth limitations.

Runs at different clock speeds, e.g. by

```
aprun --p-state 2500000
aprun --p-state 2300000
```

show core/compute limitations.

Runs at different sizes (number of PEs) at the last good scaling point versus the first bad scaling point show scaling issues.

Usually good Cray MPI environment settings for performance analysis are

```
# General setup information :
MPICH_VERSION_DISPLAY=1
MPICH_ENV_DISPLAY=1
MPICH_CPUMASK_DISPLAY=1 # uncomment if output to large.
MPICH_RANK_REORDER_DISPLAY=1 #
# If using MPI-IO (parallel NetCDF or parallel HDF5) :
MPICH_MPIIO_AGGREGATOR_PLACEMENT_DISPLAY=1
MPICH_MPIIO_HINTS_DISPLAY=1
MPICH_MPIIO_STATS=1 # or 2
```

7.1.7.2.2. Load Imbalance Analysis

The **imbalance time** (abbreviated as `Imb. Time` in the output of CrayPat) is a metric based on the execution time. Its definition depends on the type of activity:

For user functions: `Imb. Time = Max. time – Average time`.

For synchronization (collective communication and barriers): `Imb. Time = Average time – Min. time`.

The imbalance time identifies computational code regions and synchronization calls that could benefit most from load balance optimization. It estimates how much overall program time could be saved if the corresponding section of code had a perfect balance. It represents an upper bound on potential savings. It assumes other processes are waiting, not doing useful work while the slowest member finishes.

The imbalance time **percentage** represents the percentage of resources available for parallelism that is "wasted". It corresponds to the percentage of time that the rest of team is not engaged in useful work on the given function. Perfectly balanced code segments have an imbalance of 0 %, serial code segments have an imbalance of 100 %.

`MPI Sync time` measures load imbalance in programs which are instrumented to trace MPI functions. This helps to determine if MPI ranks arrive at collectives together and to separate load imbalance from data transfer. `MPI Sync time` is reported by default, but **only for tracing experiments**, if MPI functions are traced. If desired, `PAT_RT_MPI_SYNC=0` deactivates this feature.

7.1.7.2.3. Setting `aprun`

Some options of `aprun`:

`-n` Total number of PEs used by the application

`-N` Number of PEs per compute node

`-d` depth, the number of cores for the threads of each PE; in other words, the "stride" between 2 PEs on a node. For OpenMP applications, set both the `OMP_NUM_THREADS` environment variable to specify the number of threads and the `aprun -d` option to specify the number of CPUs hosting the threads.

`-j` Number of threads to run on each core

`-cc <arg>` Binds PE and threads to cores

All options are described at `man aprun`, it contains several useful examples too.

Descriptions of `aprun` can also be found at [https://wickie.hlrs.de/platforms/index.php/CRAY_XC40_Using_the_Batch_System#Understanding_aprun] and [http://docs.cray.com/cgi-bin/craydoc.cgi?mode=Show;q=2496;f=/books/S-2496-5001/html-S-2496-5001/cnl_apps.html].

By default `aprun` will bind each PE to a single core for the duration of the run. This prevents PEs from moving between cores. PEs are assigned to CPUs on the node in increasing order. The number of threads created by each PE should be given to `aprun` by the `–d` (depth) flag. `aprun` does not create threads, but the master PE only. PEs are bound to cores spaced by the depth argument. Each child process/thread created subsequently by a PE is bound by the OS to the next core (modulo by the depth argument).

`aprun` cannot prevent PEs from spawning more threads than requested. In such cases threads will start to “wrap around” and be assigned to earlier cores.

`aprun` can be prevented from binding PEs and their children to cores by specifying `–cc none`. All PEs and their child processes and threads are allowed to migrate across cores as determined by the standard Linux process scheduler. This is useful where PEs spawn many short lived children (e.g. compilation scripts) or over-subscribe the node.

7.1.7.2.3.1. Hyperthreading

On Hazel Hen users can choose to run with one or two PEs or threads per core. The default is to run with one.

`aprun –n### -j 1 …` : Single Stream mode, one rank per core (default)

`aprun –n### -j 2 …` : Dual Stream mode, two ranks per core (Hyperthreading)

The numbering of the ranks in single stream mode is 0-11 for die 0 and 12-23 for die 1. With dual stream mode the numbering of the first 24 ranks stays the same, ranks 24-35 are on die 0 and 36-47 on die 1. This makes the numbering of the ranks in hypterthread mode not contiguous:

Mode	**ranks on die 0**	**ranks on die 1**
Single Stream	0-11	12-23
Dual Stream	0-11, 24-35	12-23, 36-47

The ranks are assigned consecutive, which means in hyperthread mode: 0,24,1,25,...,11,35,12,36,...,23,47.

7.1.7.2.3.2. aprun CPU Affinity control

CLE can dynamically distribute work by allowing PEs and threads to migrate from one CPU to another within a node. In some cases, moving PEs or threads from CPU to CPU increases cache and translation lookaside buffer (TLB) misses and therefore reduces performance.

The CPU affinity options enable to bind a PE or thread to a particular CPU or a subset of CPUs on a node.

`-cc cpu`: default setting, PEs are bound a to specific core.

`-cc numa_node`: binding PEs to a specific numa node, but not to a specific core therein.

`-cc none`: no binding.

`-cc 0,4,3,2,1,16,18,31,9,....`: own binding.

7.1.7.2.3.3. aprun Memory Affinity control

Even if your PE and threads are bound to a specific numa_node, the memory used does not have to be ‘local’. By setting `–ss` a PE can allocate the memory local to its assigned NUMA node only. If this is not possible, your application will crash.

7.1.7.2.3.4. Some `aprun` examples

7.1.7.2.3.4.1. Pure MPI application, using all the available cores in a node

```
aprun -n 48 -N 48 -j2 ./a.out
```

7.1.7.2.3.4.2. Pure MPI application, using only 1 core per node

24 MPI tasks, 24 nodes with 24*24 cores allocated - to increase the memory available for the MPI tasks:

```
aprun -n 24 -N 1 -d 24 ./a.out
```

7.1.7.2.3.4.3. Hybrid MPI/OpenMP application, 4 MPI ranks per node

24 MPI tasks, 12 OpenMP threads each - need to set `OMP_NUM_THREADS`:

```
export OMP_NUM_THREADS=12
aprun -n 24 -N 4 -d $OMP_NUM_THREADS -j2
```

7.1.7.2.3.4.4. Multiple Programs, Multiple Data (MPMD)

`aprun` supports MPMD (*Multiple programs, multiple data streams*). Several executables which are part of the same `MPI_COMM_WORLD` are launched by

```
aprun –n 96 exe1 : -n 48 exe2 : -n 48 exe3
```

Each executable needs a dedicated node, `exe1` and `exe2` cannot share the same node. Therefore the command

```
aprun –n 1 exe1 : -n 1 exe2 : -n 1 exe3
```

needs 3 nodes, althought only one core of each node will be used.

Usage of a script to start several serial jobs on a node:

```
aprun –a xt –n 1 –d 24 –cc none script.sh
```

with file `script.sh`:

```
./exe1&
./exe2&
./exe3&
wait
```

7.1.7.2.3.4.5. `cpu_lists` for each PE

Separating `cpu_lists` by colons (`:`) allows the user to specify the cores used by processing elements and their child processes or threads. This provides the user more granularity to specify `cpu_lists` for each processing element. Here an example with 3 threads:

```
aprun -n 4 -N 4 -cc 1,3,5:7,9,11:13,15,17:19,21,23
```

7.1.7.2.4. Rank Reordering

The default ordering of the rank placement can be changed by `export MPICH_RANK_REORDER_METHOD=N`. These are the different values of `N` possible (the examples are for 8 tasks on 4 nodes, 2 tasks per node):

`N=0` Round-robin placement – Sequential ranks are placed on the next node, e.g. 0, 1, 2, 3, 0, 1, 2, 3.

`N=1` (**Default**) SMP-style- (block-) placement, e.g. 0, 0, 1, 1, 2, 2, 3, 3.

`N=2` Folded rank placement, e.g. 0, 1, 2, 3, 3, 2, 1, 0.

`N=3` Custom ordering. The ordering is specified in a file named `MPICH_RANK_ORDER`.

Rank reordering is usefull

- if Point-to-point communication consumes a significant fraction of runtime and a load imbalance is detected,
- for efficent usage of collectives (`alltoall`) on subcommunicators,
- to spread out I/O servers across nodes.

7.1.7.2.5. MPI and OpenMP with Intel RTE

Note: The Intel Run Time Environment (Intel RTE) is not effective by default, but only if module `PrgEnv-intel` is loaded.

The Intel RTE creates one extra thread when spawning the worker threads, `$OMP_NUM_THREADS+1` threads are started in total. This makes the efficient pinning more difficult for `aprun`. In the default setting the threads are scheduled round robin, the extra thread on the second core, while at the end two application threads (first and last one) are both placed on the first core. This results in a significant performance degradation. But this extra thread usually has no significant workload; this extra thread does not influence the performance of an application thread, when it is located on the same core.

Thus, we suggest adding the `-cc depth` option. Then all threads can migrate with respect to the specified cpumask. Additionally the Intel-specific method of binding to cores (`KMP_AFFINITY`) should be disabled on Hazel Hen. For example by using

```
export KMP_AFFINITY=disabled
export OMP_NUM_THREADS=$omps
aprun -n $npes -N $ppn -d $OMP_NUM_THREADS -cc depth a.out
```

all `$omps` application threads will be located each on a single core and the extra thread on one of these.

7.1.7.2.6. Core Specialisation

Occasionally it is necessary to run additional processes (like OS, MPI progress engines, daemons) on some cores. If all cores are in use, the OS must swap a user process out to execute the other process. Normally this introduces only a small overhead to the application. However in some cases this causes greater delays, e.g. if there are frequent synchronisations between nodes (e.g. collectives). Core specialisation reserves some cores for the OS/system/daemon tasks and improves the overall performance then. The reserved cores are automatically chosen from unused cores on Compute Units (e.g. spare Hyperthreads), even if `-j1` has been selected. How many free cores/cpus are used can be specified using the `-r` option to reserve them. Additionally `MPICH_NEMESIS_ASYNC_PROGRESS=MC` and `MPICH_MAX_THREAD_SAFETY=multiple` must be set. See `man aprun` and `man mpi` for details.

7.1.7.3. Further documents on MPI

The output of `man mpi` is an introduction on MPI in the Cray Linux Environment. It is also found at [http://docs.cray.com/cgi-bin/craydoc.cgi?mode=Show;q=;f=man/xe_mptm/72/cat3/intro_mpi.3.html].

A Manual "Getting Started on MPI I/O" is placed at [https://fs.hlrs.de/projects/craydoc/docs_merged/books/S-2490-40/html-S-2490-40/index.html] and also at [http://docs.cray.com/books/S-2490-40/].

Best practices for I/O, Parallel I/O and MPI-IO at HLRS are described at [https://wickie.hlrs.de/platforms/index.php/MPI-IO].

7.1.8. Debugging

An overview on debuggers is given at [https://wickie.hlrs.de/platforms/index.php/Debugging_On_XC40].

ATP	In case of segmentation faults, CRAYs Abnormal Termination Processing (ATP) provides an application stack trace at the moment of the error of the related process.
STAT	In case of a hanging application, CRAYs Stack Trace Analysis Tool (STAT) can help identifying dead locks.
Allinea DDT	More complex issues or wrong results can be investigated with the parallel debugger Allinea DDT to monitor the applications behaviour. A DDT Users Guide is placed at [http://content.allinea.com/downloads/userguide-forge.pdf].

7.2. MinoTauro @ BSC

7.2.1. System Architecture / Configuration

This section gives an overview of the MinoTauro system. More detailed documentation about MinoTauro can be found online under https://www.bsc.es/innovation-and-services/supercomputers-and-facilities/minotauro [hhttps://www.bsc.es/innovation-and-services/supercomputers-and-facilities/minotauro] and https://www.bsc.es/user-support/mt.php.

MinoTauro is a heterogeneous cluster with 2 configurations.

On one hand, it has 61 Bull B505 nodes, with 2 Intel E5649 (6-Core) processors (Westmere-EP) at 2.53 GHz and 2 M2090 NVIDIA GPU cards for each node. Additionally, these nodes have 24 GB of main memory and 250 GB SSD as local storage. The peak performance of these nodes is 88,60 Tflops.

On the other hand, MinoTauro has 39 bullx R421-E4 servers with 2 Intel Xeon E5–2630 v3 (Haswell) 8-core processors (each core at 2.4 GHz, and with 20 MB L3 cache) and 2 K80 NVIDIA GPU cards for each server. Additionally, these servers have 128 GB of main memory (distributed in 8 DIMMs of 16 GB DDR4 @ 2133 MHz) and 120 GB SSD as local storage. The peak performance of these servers is 250,94 Tflops.

The operating system is RedHat Linux 6.7 for both configurations.

7.2.1.1. Processor Architecture / MCM Architecture

MinoTauro is equipped with an Infiniband interconnect for the MPI communication and Ethernet network for management and GPFS. Each K80 node has 1 PCIe 3.0 x8 8 GT/s, Mellanox ConnectX®–3FDR 56 Gbit and 4 Gigabit Ethernet ports.

7.2.1.2. Interconnect

Intel Xeon E5–2630 v3 processors have 8 cores each running at a clock speed of 2,4GHz. Each core has a 32 KB 8-way set associative L1 instruction cache, 32 KB 8-way set associative L1 data cache and 256 KB 8-way set associative L2 cache. Each processor has 20MB L3 shared cache.

7.2.1.3. I/O Subsystem Architecture

The I/O subsystem consists of a collection of IBM GPFS file systems. The IBM General Parallel File System (GPFS) is a high-performance shared-disk file system providing fast, reliable data access from all nodes of the cluster to a global filesystem. GPFS allows parallel applications simultaneous access to a set of files (even a single file) from any node that has the GPFS file system mounted while providing a high level of control over all file system operations. In addition, GPFS can read or write large blocks of data in a single I/O operation, thereby minimizing overhead.

7.2.1.4. Available File Systems

Every node has a local SSD hard drive that can be used as a local scratch space to store temporary files during executions of the jobs. This space is mounted on the `/scratch/` directory and pointed to by the $TMPDIR environment variable. The amount of space within the `/scratch` filesystem varies depending on the configuration. The M2090 configuration has about 200 GB while the K80 configuration has about 100 GB.

7.2.1.4.1. Home, Scratch, Long Time Storage

The home file systems, scratch file systems, project file system and the long time storage (Active Archive) use IBM GPFS technology. All file systems, except Archive are shared by all nodes. An incremental backup is performed daily only for home file system and projects file system.

The home file system (`/gpfs/home`) contains the home directories of all the users. The home file systems are intended as a general repository of user resources: source codes, binaries, libraries...

As implied by the name, the scratch file systems are intended as a “scratch space” (store temporary files) during the job executions. Each user has a directory under `/gpfs/scratch`.

The projects file system (`/gpfs/projects`) is intended to store data that needs to be shared between the users of the same group or project. It is the project’s manager responsibility to determine and coordinate the better use of this space, and how it is distributed or shared between their users.

Active Archive (AA) is a mid-long term storage filesystem that provides more than 3 PB of total space. It is accesible from the data transfer machine (`dt01.bsc.es` and `dt02.bsc.es`) under `/gpfs/archive/[group]`. There is no backup of this filesystem. The user is responsible for adequately managing the data stored in it. To move or copy from/to AA the user have to use special commands: `dtcp, dtmv, dtrsync, dttar`. These commands submit a job into a special class performing the selected command. Their syntax is the same than the shell command without ‘dt’ prefix (`cp, mv, rsync, tar`).

Additionally, MinoTauro has an apps file system (`/apps`) where the applications and libraries that have already been installed on the machine.

7.2.1.4.2. Performance of File Systems

The performance results of the file systems during the past month are:

Figure 14. Performance of the home file system

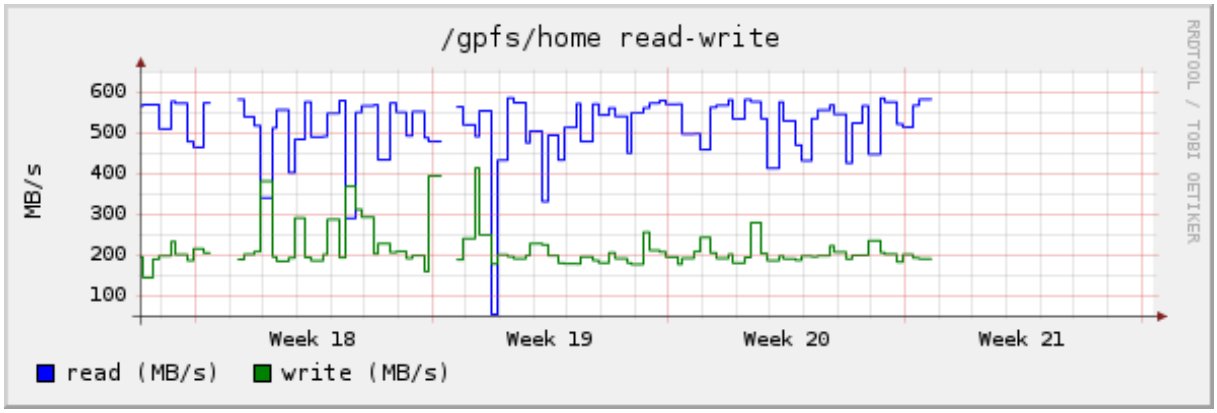


Figure 15. Performance of the scratch file system

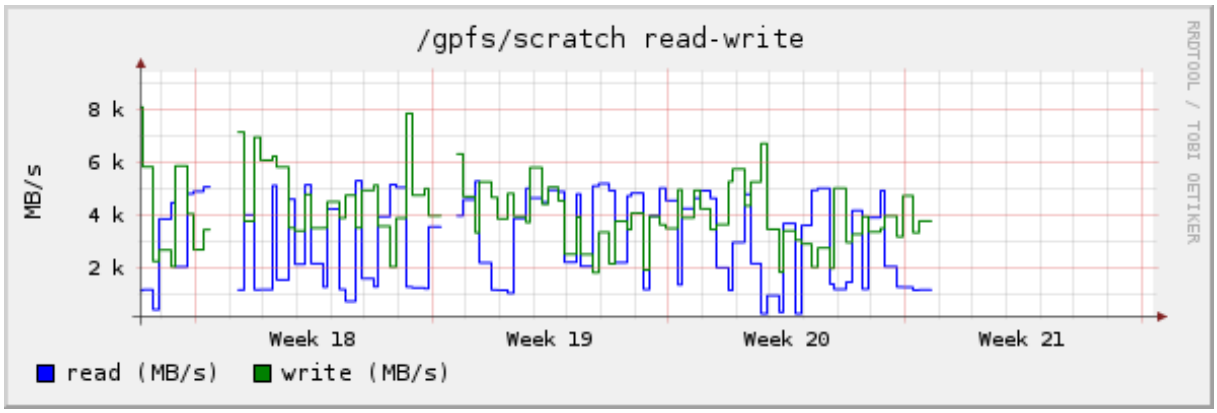
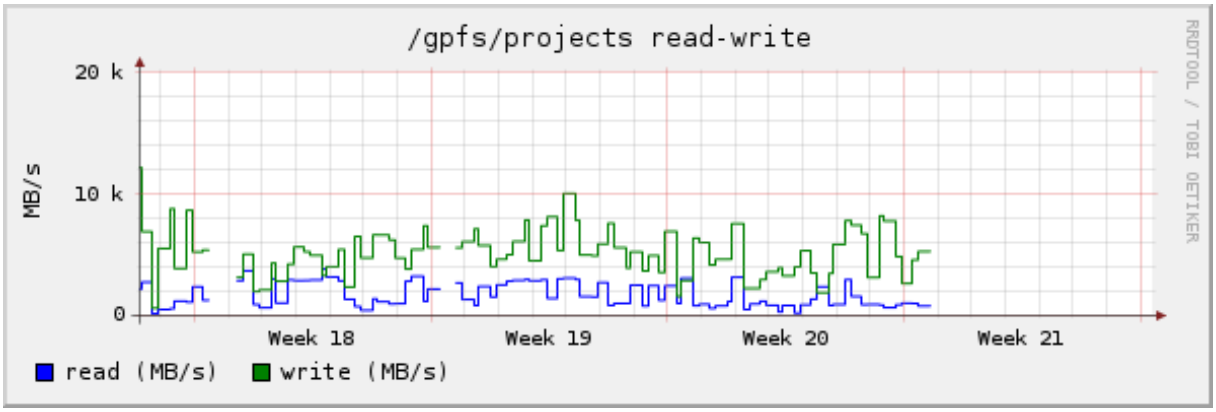


Figure 16. Performance of the projects file system



7.2.2. System Access

The users must use Secure Shell (ssh) tools to login into the cluster or transfer files to it. Incoming connections from commands `like telnet, ftp, rlogin, rcp`, or `rsh` are not allowed. Once the users have a username and its associated password, they can get into the cluster through one of the login nodes (`mt1.bsc.es` and `mt2.bsc.es`).

There are two ways to copy files from/to the cluster, doing a direct `scp` or `sftp` to the login nodes or using a data transfer machine which shares all the GPFS file system for transferring large files. All `scp` and `sftp` commands have to be executed from the user local machines and never from the cluster, because connections from inside the cluster to the outside world are not allowed. On a Windows system, most of the secure shell clients come with a tool to ensure secure copies or secure ftp's.

We provide special machines for file transfer (required for large amounts of data). These machines are dedicated to data transfer and are accessible through ssh with the same account credentials as the cluster. The login nodes for this machine are (`dt01.bsc.es` and `dt02.bsc.es`). These machines share the GPFS filesystem with all other BSC HPC machines. PRACE users can use the 10 Gbps PRACE network for moving large data among PRACE site with the data transfer tool Globus/GridFTP available on `dt02.bsc.es`.

7.2.3. Production Environment

7.2.3.1. Module Environment

The Environment Modules package (http://modules.sourceforge.net [http://modules.sourceforge.net/]) provides a dynamic modification of a user's environment via modulefiles. Each modulefile contains the information needed to configure the shell for an application or a compilation. Modules can be loaded and unloaded dynamically, in a clean fashion. All popular shells are supported, including `bash, ksh, zsh, sh, csh, tcsh`, as well as some scripting languages such as `perl`.

Installed software packages are divided into different categories. The environment category have modulefiles dedicated to prepare the environment, for example, get all necessary variables to use openmpi to compile or run programs. The tools category is for useful tools which can be used at any time (`php`, `perl`, …). The applications category is for High Performance Computers programs (GROMACS, …). The libraries category is for those modules that are typically loaded at compilation time, they load into the environment the correct compiler and linker flags (FFTW, LAPACK, …). Finally, the compilers category is for compiler suites available for the system (intel, gcc, …).

7.2.3.2. Batch System

Slurm is the utility used for batch processing support, so all jobs must be run through it.

7.2.3.3. Accounting

On MinoTauro, the system usage is measured in CPU hours.

7.2.4. Programming Environment

7.2.4.1. Available Compilers

7.2.4.1.1. Intel Compilers

Intel compilers are optimized for computer systems using processors that support Intel architectures. They are designed to minimize stalls and to produce code that executes in the fewest possible number of cycles. On the cluster you can find these Intel Compilers:

- icc - Intel C Compiler
- icpc - Intel C++ Compiler
- ifort - Intel Fortran Compilers

7.2.4.1.2. PGI Compilers

The Portland Group (PGI) was a company that produced a set of commercially available Fortran, C and C++ compilers for high-performance computing systems. Today, the PGI compilers and tools are made available through NVIDIA under "The Portland Group Compilers and Tools" brand name. On the cluster you can find these PGI compilers:

- pgcc - PGI C Compiler
- pgc++ - PGI C++ Compiler
- pgfortran - PGI Fortran Compilers

7.2.4.1.3. GCC

The GNU Compiler Collection (GCC) is a compiler system produced by the GNU Project supporting various programming languages. On the cluster you can find these GNU Compilers:

- gcc - GNU C Compiler
- g++ - GNU C++ Compiler
- gfortran - GNU Fortran Compiler

7.2.4.2. Available (Vendor Optimized) Numerical Libraries

The vendor optimized numerical library for this platform is MKL.

7.2.4.3. Available MPI Implementations

The distributed memory parallelism can be easily exploited by your C/C++ or Fortran codes by means of the MPI (Message Passing Interface) library. The available MPI implementations for MinoTauro are Bull MPI, Intel MPI and OpenMPI. These implementations can be loaded via:

```
module load [implementation]/[version]
```

where `implementation` should be `bullxmpi, impi` or `openmpi`.

To compile MPI programs it is recommended to use the following handy wrappers: `mpicc, mpicxx` for C and C++ source code. The users must choose the parallel environment first: `module load [implementation]`. These wrappers include all the necessary libraries to build MPI applications without having to specify all the details by hand.

```
mpicc a.c -o a.exe
```

```
mpicxx a.C -o a.exe
```

7.2.4.4. OpenMP

OpenMP provides an easy method for SMP-style parallelization of discrete, small sections of code, such as a do loop. OpenMP can only be used among the processors of a single node. For use with production scale, multi-node codes, OpenMP threads must be combined with MPI processes.

7.2.4.4.1. Compiler Flags

OpenMP directives are fully supported by the Intel, PGI and GNU compilers. To use it with GNU or Intel, the flag `-?openmp` must be added to the compile line:

```
icc -qopenmp -o exename filename.c
icpc -qopenmp -o exename filename.C
ifort -qopenmp
gcc -fopenmp -o exename filename.c
g++ -fopenmp -o exename filename.C
gfortran -fopenmp
```

To use it with PGI, the flag `-mp` must be added to the compile line:

```
pgcc -mp
pgc++ -mp
pgfortran -mp
```

It is also possible to mix MPI and OpenMP code with the MPI wrappers mentioned above.

7.2.4.5. Batch System / Job Command Language

There are two supported methods for submitting jobs. The first one is to use a wrapper maintained by the Operations Team at BSC that provides a standard syntax regardless of the underlying batch system (`mnsubmit`). The other one is to use the SLURM `sbatch` directives directly. The second option is recommended for advanced users only.

A job is the execution unit for SLURM. A job is defined by a text file containing a set of directives describing the job's requirements, and the commands to execute. In order to ensure the proper scheduling of jobs, there are execution limitations in the number of nodes and cpus that can be used at the same time by a group. Since MinoTauro is a cluster where more than 90% of the computing power comes from the GPUs, jobs that do not use them have a lower priority than those that are GPU-ready.

These are the basic directives to submit jobs with `mnsubmit` or `sbatch`:

Table 28. SLURM directives

Command Description	`mnsubmit`	`sbatch`
Submit a "job script" to the queue system	`mnsubmit job_script`	`sbatch job_script`
Show all the submitted jobs	`mnq`	`squeue`
Remove the job from the queue system, canceling the execution of the processes, if they are still running	`mncancel job_id`	`scancel job_id`
Allocate an interactive session in the debug partition	`mnsh -k80`	`mnsh -k80`

A job must contain a series of directives to inform the batch system about the characteristics of the job. These directives appear as comments in the job script and have to conform to either the `mnsubmit` or the `sbatch` syntaxes. Using `mnsubmit` syntax with `sbatch` or the other way around will result in failure.

Table 29. Job script syntax

	`mnsubmit`	`sbatch`
Default syntax form	# @ directive = value	#SBATCH --directive=value
	# @ partition = debug	#SBATCH --partition=debug
	# @ class = debug	#SBATCH --qos=debug
The limit of wall clock time	# @ wall_clock_limit = HH:MM:SS	#SBATCH --time=HH:MM:SS
The working directory of your job	# @ initialdir = pathname	#SBATCH --cwd=pathname
The name of the file to collect the standard error output (stderr) of the job	# @ error = file	#SBATCH --error=file
The name of the file to collect the standard output (stdout) of the job	# @ output = file	#SBATCH --output=file
The number of processes to start	# @ total_tasks = number	#SBATCH --ntasks=number
How many threads each process would open (optional)	# @ cpus_per_task = number	#SBATCH --cpus-per-task=number
The number of tasks assigned to a node	# @ tasks_per_node = number	#SBATCH --ntasks-per-node=number
The number of GPU cards assigned to the job	# @ gpus_per_node = number	#SBATCH --gres gpu:number
Select which configuration to run your job on [m2090 or k90]	# @ features = (config)	#SBATCH --constraint=(config)
Request a specific network topology in order to run his job. If is not possible after timout minutes, Slurm will schedule by default	# @ switches = "number@timeout"	#SBATCH --switches=number@timeout
In order to use the CPU device to run OpenC	# @ intel_opencl = 1	no sbatch equivalent
Handle the job as graphical	# @ X11 = 1	no sbatch equivalent

There are also a few SLURM environment variables users can use in the scripts:

Table 30. SLURM environment variables

Variable	Meaning
SLURM_JOBID	Specifies the job ID of the executing job
SLURM_NPROCS	Specifies the total number of processes in the job
SLURM_NNODES	Is the actual number of nodes assigned to run your job
SLURM_PROCID	Specifies the MPI rank (or relative process ID) for the current process. The range is from 0-(SLURM_NPROCS–1)
SLURM_NODEID	Specifies relative node ID of the current job. The range is from 0-(SLURM_NNODES–1)

SLURM_LOCALID	Specifies the node-local task ID for the process within a job

7.2.4.5.1. Job script examples (`mnsubmit`)

Example for a sequential job:

```
                                #!/bin/bash
 # @ job_name= test_serial
 # @ initialdir= .
 # @ output= serial_%j.out
 # @ error= serial_%j.err
 # @ total_tasks= 1
 # @ wall_clock_limit = 00:02:00
 ./serial_binary> serial.out
```

The job can be submitted using:

```
mnsubmit ptest.cmd
```

Example for a job using the new K80 GPUs:

```
       #!/bin/bash
# @ job_name= test_k80
# @ initialdir= .
# @ output= k80_%j.out
# @ error= k80_%j.err
# @ total_tasks= 16
# @ gpus_per_node= 4
# @ cpus_per_task= 1
# @ features = k80
# @ wall_clock_limit = 00:02:00
srun ./parallel_binary_k80> parallel.output
```

7.2.4.5.2. Job script examples (`sbatch`)

Example for a sequential job:

```
#!/bin/bash
#SBATCH --name="test_serial"
#SBATCH --cwd=.
#SBATCH --output=serial_%j.out
#SBATCH --error=serial_%j.err
#SBATCH --ntasks=1
#SBATCH --time=00:02:00
./serial_binary> serial.out
```

The job would be submitted using:

```
sbatch ptest.cmd
```

Example for a job using the new K80 GPUs:

```
#!/bin/bash
```

```
#SBATCH --name=test_k80
#SBATCH --cwd=.
#SBATCH --output= k80_%j.out
#SBATCH --error= k80_%j.err
#SBATCH --ntasks=16
#SBATCH --gres: gpu:4
#SBATCH --cpus-per-task=1
#SBATCH --constraint=k80
#SBATCH --time=00:02:00
srun ./parallel_binary_k80> parallel.output
```

7.2.5. Performance Analysis

7.2.5.1. Available Performance Analysis Tools

7.2.5.1.1. Add instrumentation instructions using Extrae

The most recent stable version of Extrae is always located at:

`/apps/CEPBATOOLS/extrae/latest/default/64`

This package is compatible with the default MPI runtime in MinoTauro (Bull MPI). Packages corresponding to older versions and enabling compatibility with other MPI runtimes (OpenMPI, MVAPICH) can be respectively found under this directory structure:

`/apps/CEPBATOOLS/extrae/(choose-version)/(choose-runtime)/64`

7.2.6. Debugging

7.2.6.1. Available Debuggers

7.2.6.1.1. Total View

TotalView is a graphical portable powerful debugger from Rogue Wave Software designed for HPC environments. It also includes MemoryScape and ReverseEngine. It can debug one or many processes and/or threads. It is compatible with MPI, OpenMP, Intel Xeon Phi and CUDA. Users can access to the latest version of TotalView 8.13 installed in:

`/apps/TOTALVIEW/totalview`

Since TotalView uses a single window control the users should access with `ssh -X` to the cluster and submit the jobs to the x11 queue.

There is a Quick View of TotalView available for new users. Further documentation and tutorials can be found on their website or on the cluster at:

`/apps/TOTALVIEW/totalview/doc/pdf`

7.2.6.2. Compiler flags

In this section we summarise the compiler flags needed for profiling and/or debugging applications. The table below shows common flags for most compilers:

Table 31. Compiler Flags

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

-pg	Generate extra code to write profile information suitable for the analysis program gprof. You must use this option when compiling the source files you want data about, and you must also use it when linking.

7.3. Salomon @ IT4Innovations

7.3.1. System Architecture / Configuration

This section gives an overview of the Salomon system. More detailed documentation about Salomon can be found online under https://docs.it4i.cz/ [https://docs.it4i.cz/].

The Salomon cluster consists of 1008 compute nodes, totaling 24192 compute cores with 129 TB RAM and providing over 2 Pflop/s theoretical peak performance. Each node is a powerful x86-64 computer, equipped with 24 cores and at least 128 GB RAM. Nodes are interconnected by a 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 accelerators.

The cluster runs the CentOS Linux operating system, which is compatible with the RedHat Linux family.

Figure 17. Salomon Racks

Table 32. 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, 2.5 GHz, 12 cores
RAM	128 GB, 5.3 GB 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 33. Compute nodes

Node	Count	Processor	Cores	Memory	Accelerator
w/o accelerator	576	2x Intel Xeon E5-2680v3, 2.5 GHz	24	128 GB	-
MIC accelerated	432	2x Intel Xeon E5-2680v3, 2.5 GHz	24	128 GB	2x Intel Xeon Phi 7120P, 61 cores, 16 GB RAM

For remote visualization two nodes with NICE DCV software are available each configured:

Table 34. Remote visualization nodes

Node	Count	Processor	Cores	Memory	GPU Accelerator
visualization	2	2x Intel Xeon E5-2695v3, 2.3 GHz	28	512 GB NVIDIA QUADRO K5000, 4 GB RAM	

For large memory computations a special SMP/NUMA SGI UV 2000 server is available:

Table 35. SGI UV 2000

Node	Count	Processor	Cores	Memory	Extra HW
UV2000	1	14x Intel Xeon E5-4627v2, 3.3 GHz, 8 cores	112	3328 GB DDR3@1866 MHz	2x 400 GB local SSD 1x NVIDIA GM200 (GeForce GTX TITAN X), 12 GB RAM

7.3.2. System Access

7.3.2.1. 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 welcome to apply via the Director's 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

the 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.

7.3.2.2. Shell access and data transfer

The Salomon cluster is accessed by the SSH protocol via the 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.

7.3.3. Production Environment

7.3.3.1. Application Modules

In order to configure a shell for running a particular application on Salomon the module package interface is used.

Application modules on the 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.

To check the 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 an active shell such that everything is prepared to run the OpenMPI software.

To check the loaded modules use:

```
$ module list
```

To unload a module, for example the OpenMPI module use:

```
$ module unload OpenMPI
```

7.3.4. Programming Environment

7.3.4.1. EasyBuild Toolchains

As mentioned earlier, EasyBuild is used 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 for providing 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 EasyBuild 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, currently the following toolchains are 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

7.4. SuperMUC @ LRZ

7.4.1. Introduction

Figure 18. SuperMUC Phase 1 (left) and Phase 2 (right)

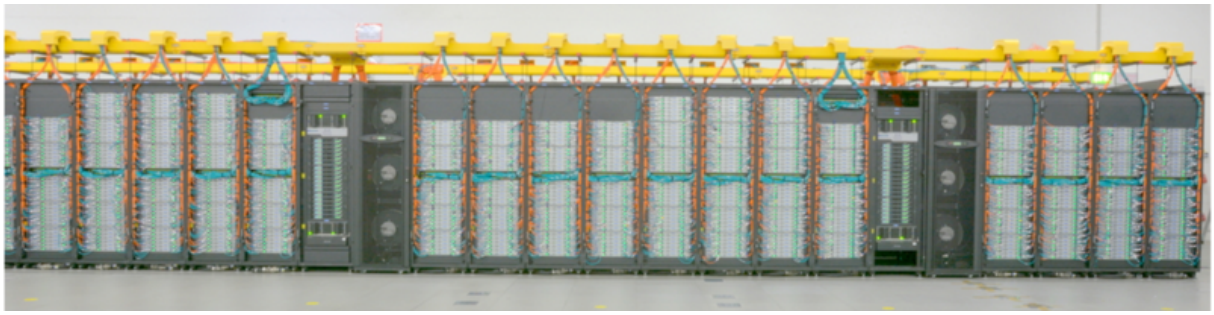
SuperMUC is the name of the high-end supercomputer at the Leibniz Supercomputing Centre in Garching near Munich. With more than 241,000 cores and a combined peak performance of the two installation phases of more than 6.8 Petaflop/s, it is one of the fastest supercomputers in Europe.

The first installation phase, SuperMUC Phase 1, was inaugurated in July 2012 and consists of 18 Thin Node Islands based on Intel Sandy Bridge-EP processor technology with 147.456 cores in total and one Fat Node Island based on Intel Westmere-EX processor technology with 8.200 cores in total. A Best Practice Guide describing SuperMUC Phase 1 was published within PRACE-2IP in May 2013 and is available on the PRACE webpage under http://www.prace-ri.eu/best-practice-guides/. It describes SuperMUC as of 2013. Most up-to-date documentation about SuperMUC is available online under https://www.lrz.de/services/compute/supermuc/.

In addition, SuperMIC, a cluster of 32 Intel Ivy Bridge-EP nodes each having two Intel Xeon Phi (Knights Corner) accelerator cards installed, is also part of the SuperMUC system. SuperMIC started operation in 2013 and is described in detail in the update of the Intel Xeon Phi Best Practice guide written within PRACE-4IP and published in January 2017.

In this section we focus on SuperMUC Phase 2, which was inaugurated end of June 2015 and consists of 6 Thin Node Islands based on Intel Haswell-EP processor technology with 86,016 cores in total. SuperMUC Phase 2 is shown exclusively in the following picture:

Figure 19. SuperMUC Phase 2

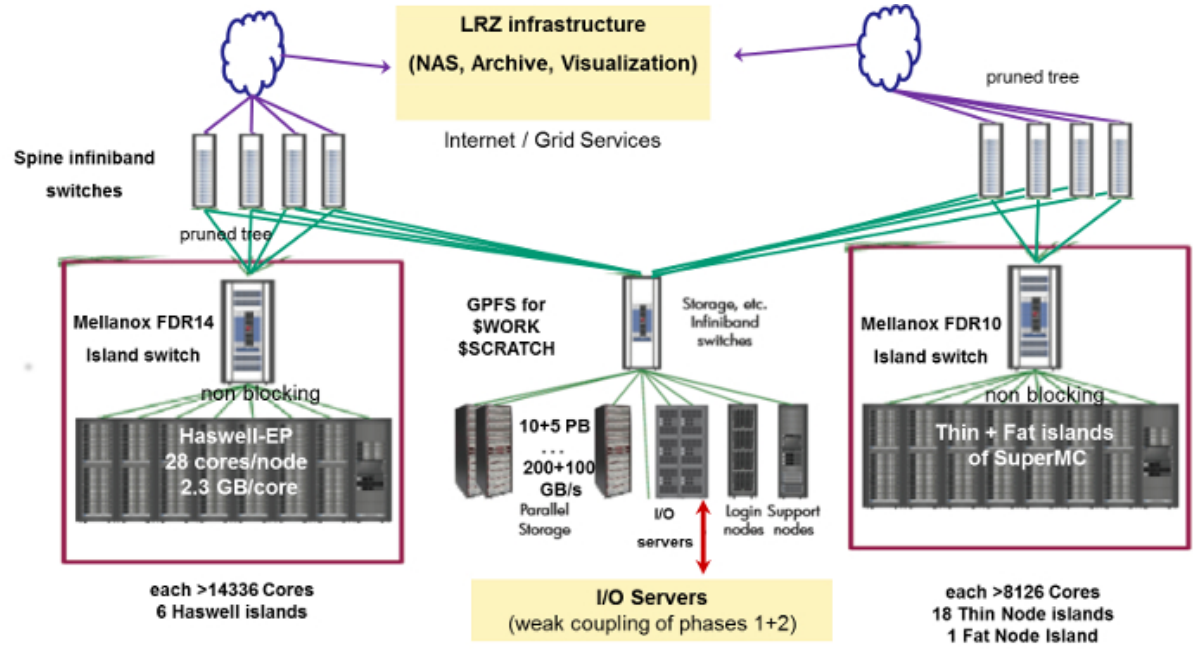


SuperMUC uses a new, revolutionary form of warm water cooling developed by IBM. Active components like processors and memory are directly cooled with water that can have an inlet temperature of up to 40 degrees Celsius. This "High Temperature Liquid Cooling" together with very innovative system software cuts the energy consumption of the system up to 40%. It is easily possible to provide water having up to 40 degrees Celsius using simple "free-cooling" equipment as outside temperatures in Germany hardly ever exceed 35 degrees Celsius. At the same time the outlet water can be made quite hot (up to 70 degrees Celsius) and re-used in other technical processes - for example to heat buildings.

7.4.2. System Architecture / Configuration

The following picture shows an overview of SuperMUC Phase 1 and 2. The Haswell-based SuperMUC Phase 2 consists of 6 Islands, each equipped with 512 nodes per Island.

Figure 20. SuperMUC Architecture



Each node has 2 Haswell Xeon Processor E5-2697 v3 processors with 14 cores with a nominal frequency of 2.6 GHz. See http://ark.intel.com/products/81059/Intel-Xeon-Processor-E5-2697-v3-35M-Cache-2_60-GHz [http://ark.intel.com/products/81059/Intel-Xeon-Processor-E5-2697-v3-35M-Cache-2_60-GHz] for the specification of the processor.

Details on the system configuration of SuperMUC comparing Phase 1 and Phase 2 are shown in the following table:

Installation Phase	**Phase 1**			**Phase 2**
Installation Date	2011	2012	2013	2015
Islandtype	Fat Nodes	Thin Nodes	Many Cores Nodes	Haswell Nodes
System	BladeCenter HX5	IBM System x iDataPlex dx360M4	IBM System x iDataPlex dx360M4	Lenovo NeXtScale nx360M5 WCT
Processor Type	Westmere-EX Xeon E7-4870 10C	Sandy Bridge-EP Xeon E5-26808C	Ivy-Bridge (IvyB) and Xeon Phi 5110P	Haswell Xeon Processor E5-2697 v3
Nominal Frequency [GHz]	2.4	2.7	1.05	2.6
Performance per core	4 DP Flops/cycle = 9.6 DP Flop/s; 2-wide SSE2 add + 2-wide SSE2 mult	8 DP Flops/cycle = 21.6 DP Flops/s; 4-wide AVX add + 4-wide AVX mult	16 DP Flops/cycle = 16.64 DP Flops/s; 8-wide fused multiply-adds every cycle using 4 threads	16 DP Flops/cycle = 41.6 DP Flops/s; two 4-wide fused multiply-adds
Total Number of nodes	205	9216	32	3072
Total Number of cores	8,200	147,456	3,840 (Phi)	86,016

Total Peak Performance [PFlop/s]	0.078	3.2	0.064 (Phi)	3.58
Total Linpack Performance [PFlop/s]	0.065	2.897	n.a.	2.814
Total size of memory [TByte]	52	288	2.56	194
Total Number of Islands	1	18	1	6
Nodes per Island	205	512	32	512
Processors per Node	4	2	2 (IvyB) 2.6 GHz+ 2 Phi 5110P	2
Cores per Processor	10	8	8 (IvyB) + 60 (Phi)	14
Cores per Node	40	16	16 (host)+ 120 (Phi)	28
Logical CPUs per Node (Hyperthreading)	80	32	32 (host) + 480 (Phi)	56
System Software				
Operating System	Suse Linux Enterprise Server (SLES)			
Batchsystem	IBM Loadleveler			
Parallel Filesystem for SCRATCH and WORK	IBM GPFS			
File System for HOME	NetApp NAS			
Archive and Backup Software	IBM TSM			
System Management	xCat from IBM>			
Monitoring	Icinga, Splunk			

Further details can be found under https://www.lrz.de/services/compute/supermuc/systemdescription/.

7.4.3. Memory Architecture

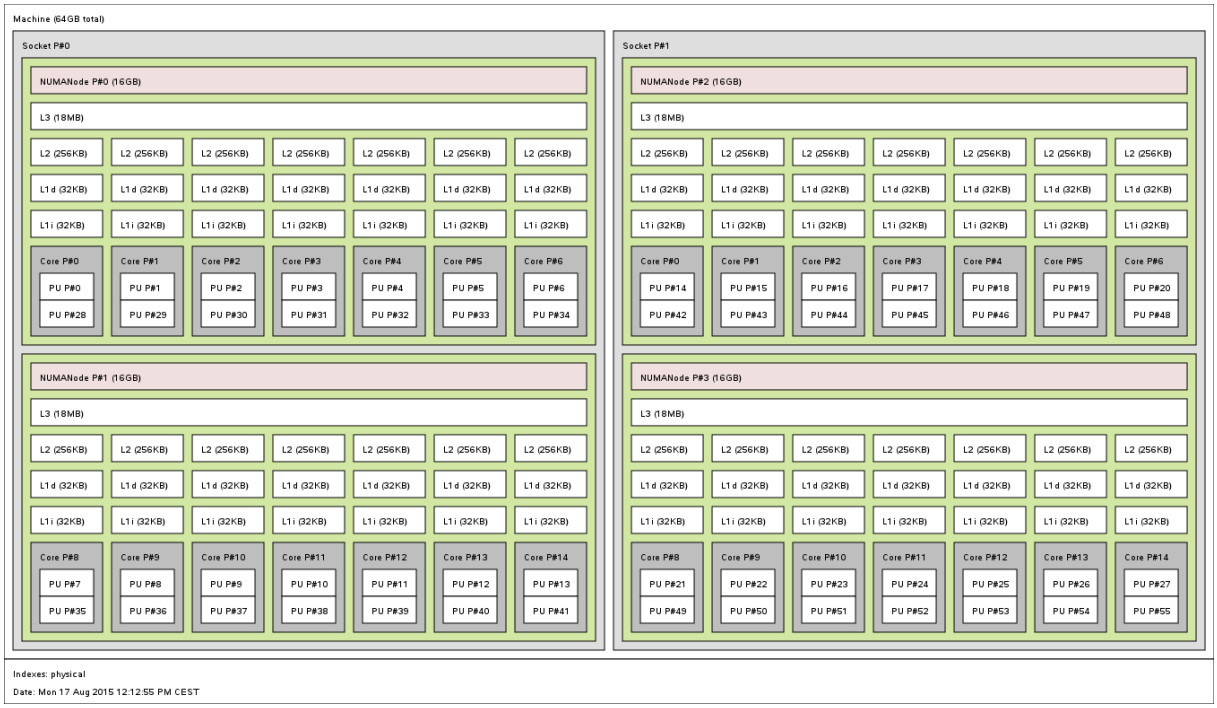
Details about the memory and cache sizes are summarised in the following table:

Installation Phase	**Phase 1**			**Phase 2**
Installation Date	2011	2012	2013	2015
Islandtype	Fat Nodes	Thin Nodes	Many Cores Nodes	Haswell Nodes
Memory per Core [GByte](typically available for applications)	6.4 (~6.0)	2 (~1.5)	4 (host) + 2 x 0.13 (Phi)	2.3 (2.1)
Size of shared Memory per node [GByte]	256	32	64 (host) + 2 x 8 (Phi)	64 (8 nodes in job class big: 256)
Bandwidth to Memory per node [Gbyte/s]	136.4	102.4	Phi: 384	137
Level 3 Cache Size (shared) [Mbyte]	4x30	2x20		4x18

Level 2 Cache Size per core [kByte]	256	256	Phi: 512	256
Level 1 Cache Size [kByte]	32	32	32	32
Latency Access Memory		~ 160		~ 200
Level 3 Latency [cycles]		~ 30		36
Level 2 Latency [cycles]		12		12
Level 1 Latency [cycles]	4	4		4
Latency Access Memory		~ 160		~ 200

A graphical representation of the topology of the Haswell processors used in SuperMUC Phase 2 is shown in the following picture:

Figure 21. Topology of the SuperMUC Phase 2 Haswell processor.



7.4.4. Interconnect

All compute nodes within an individual Island are connected via a fully non-blocking Infiniband network: FDR14 for the Haswell nodes of Phase 2 (in contrast to FDR10 for the Thin Nodes and QDR for the Fat Nodes of Phase 1). Above the Island level, the pruned interconnect enables a bi-directional bi-section bandwidth ratio of 4:1 (intra-Island / inter-Island).

Details are given in the following table:

Installation Phase	**Phase 1**			**Phase 2**
Installation Date	2011	2012	2013	2015
Islandtype	Fat Nodes	Thin Nodes	Many Cores Nodes	Haswell Nodes

Technology	Infiniband QDR	Infiniband FDR10	Infiniband FDR10	Infiniband FDR14
Intra-Island Topology	non-blocking Tree			non-blocking Tree
Inter-Island Topology	Pruned Tree 4:1		n.a.	Pruned Tree 4:1
Bisection bandwidth of Interconnect [TByte/s]	12.5		n.a.	5.1

7.4.5. Available Filesystems / Storage Systems

SuperMUC has a powerful I/O-Subsystem which helps to process large amounts of data generated by simulations. An overview is given in the following table:

Size of parallel storage (SCRATCH/WORK) [Pbyte]	15
Size of NAS storage (HOME) [PByte]	3.5 (+ 3.5 for replication)
Aggregated bandwidth to/from parallel storage [GByte/s]	250
Aggregated bandwidth to/from NAS storage [GByte/s]	12
Capacity of Archive and Backup Storage [PByte]	> 30

7.4.5.1. Home file systems

Permanent storage for data and programs is provided by a 16-node NAS cluster from NetApp. This primary cluster has a capacity of 3.5 Petabytes and has demonstrated an aggregated throughput of more than 12 GB/s using NFSv3. Netapp's Ontap 8 "Cluster-mode" provides a single namespace for several hundred project volumes on the system. Users can access multiple snapshots of data in their home directories.

Data is regularly replicated to a separate 4-node Netapp cluster with another 3.5 PB of storage for recovery purposes. Replication uses Snapmirror-technology and runs with up to 2 GB/s in this setup.

The storage hardware consists of > 3,400 SATA-Disks with 2 TB each, protected by double-parity RAID and integrated checksums.

7.4.5.2. Work and Scratch areas

For high-performance I/O, IBM's General Parallel File System (GPFS) with 12 PB of capacity and an aggregated throughput of 250 GB/s is available.

7.4.5.3. Tape backup and archives

LRZ's tape backup and archive systems based on TSM (Tivoli Storage Manager) from IBM are used for or archiving and backup. They have been extended to provide more than 30 Petabytes of capacity to the users of SuperMUC. Digital long-term archives help to preserve results of scientific work on SuperMUC. User archives are also transferred to a disaster recovery site.

7.4.5.4. Overview

An overview of all available file systems is shown in the following table.

File system	Environment Variable for Access	Purpose	Implementation, Overall size, Bandwidth	Backup and Snapshots	Intended Lifetime and Cleanup Strategy	Quota size (per project)
NAS-based shared file system						

/home/hpc	$HOME	store the user's source, input data, and small and important result files. Gobally accessible from login and compute nodes.	NAS-Filer, 1.5 PB, 10 GB/s	YES: backup to tape and Snapshots	Project duration	default: 100 GB per project
High-performance parallel file system on Phase 1 and Phase 2						
/gss/scratch	$SCRATCH	temporary huge files (restart files, files to be pre-/ post-processed). Gobally accessible from login and compute nodes.	GPFS, 5.2 PB, up to 150 GB/s	NO	Automatic deletion will happen.	no quota, but high water mark deletion if necessary
/gpfs/work	$WORK	huge result files. Gobally accessible from login and compute nodes.	GPFS, 10 PB (shared with old scratch area) up to 200 GB/s	NO	Project duration (beware of technical problems, archive important data to tape or other safe place!)	default: 1 TB per project
Different on Thin/Fat Nodes and on Compute/Login Nodes						
various	$TMPDIR	temporary filesystem for system command use. different on login and compute nodes.	setting may vary	NO	May be deleted after job end or logout	
/tmp	direct use of /tmp is strongly discouraged on compute nodes (may impact system usability) !					

7.4.6. System Access

From the UNIX command line the login to an LRZ account xxyyyzz can be performed via:

System part	Login	Architecture	Number of login nodes behind theround-robin address
SuperMUC / Phase 1 Thin Nodes	ssh -Y sb.supermuc.lrz.de -l xxyyyzz	Intel Sandy Bridge EP	5
SuperMUC / Phase 1 Fat Nodes	ssh -Y wm.supermuc.lrz.de -l xxyyyzz	Intel Westmere EX	2
SuperMUC / Phase 2 Haswell Nodes	**ssh -Y hw.supermuc.lrz.de -l xxyyyzz**	**Intel Haswell EP**	**3**

Nodes on SuperMUC Phase 1 with connection to the archive system	ssh -Y sb-tsm.supermuc.lrz.de -l xxyyyzz	Intel Sandy Bridge EP	2
Nodes on SuperMUC Phase 2 with connection to the archive system	**ssh -Y hw-tsm.supermuc.lrz.de -l xxyyyzz**	**Intel Haswell EP**	**2**
Xeon Phi Nodes (SuperMIC)	ssh -Y supermic.smuc.lrz.de -l xxyyyzz	Intel Ivy Bridge & Xeon Phi	1
Nodes with connection to the dedicated PRACE network. The access is restricted to a limited number of machines.	ssh -Y sb.supermuc-prace.lrz.de -l xxyyyzz; **ssh -Y hw.supermuc-prace.lrz.de -l xxyyyzz**; ssh -Y sb-tsm.supermuc-prace.lrz.de -l xxyyyzz; **ssh -Y hw-tsm.supermuc-prace.lrz.de -l xxyyyzz**	Intel Sandy Bridge EP/ **Intel Haswell EP**	1

It is also possible to login via grid services using GSI-SSH. More details about Logging in to SuperMUC can be found under https://www.lrz.de/services/compute/supermuc/access_and_login/.

Information about specific calls for computing time on SuperMUC or additional funding of HPC projects can be found under https://www.lrz.de/services/compute/supermuc/calls/ .

7.4.7. Production Environment

SuperMUC Phase 1 and Phase 2 are only loosely coupled through the GPFS and NAS File systems, used by both Phase 1 and Phase 2. It is not possible to run one single job across Phase 1 and Phase 2. The scheduling and job classes of Phase 1 and Phase 2 are different. However, Phase 1 and Phase 2 share the same programming environment.

7.4.7.1. Module environment

LRZ uses the environment module approach to manage the user environment for different software, library or compiler versions. The distinct advantage of the modules approach is that the user is no longer required to explicitly specify paths for different executable versions, and try to keep the MANPATH and related environment variables coordinated. With the modules approach, users simply "load" and "unload" modules to control their environment. Type

```
module avail
```

to list all the modules which are available to be loaded. Notice that most of them indicate associated version numbers. Modules make it easy to switch between versions of a package. Specifying a module name without a version number will select the default production version.

Many modules contain the URL of the LRZ documentation for the specific software package. Type

```
module show <modulename>
```

and look for the variable which ends with `_WWW`.

More information on LRZ specific tools for SuperMUC (lrztools) can be found under https://www.lrz.de/services/compute/supermuc/software/.

7.4.7.2. Batch System and Accounting

IBM LoadLeveler is used as batch system. Please mind the following on SuperMUC Phase 2:

1. It is not possible to run one single job across Phase 1 and Phase 2. The scheduling and job classes of Phase 1 and Phase 2 are different. However, Phase 1 and Phase 2 share the same programming environment.

2. It's not possible to submit jobs to SuperMUC Phase 2 from login nodes of Phase 1.

3. Only complete nodes are provided to a given job for dedicated use.

4. Accounting is performed by using: AllocatedNodes*Walltime*(number_of_core_in_node).

5. Core hours of Phase 1 and Phase 2 are accounted equally (1 core-hour of Phase 1 = 1 core-hour of Phase 2).

6. Running large jobs (> 512 nodes) requires permission for the job class "special". User must pass their requests for "special" through the LRZ service desk.

The following table shows the job classes available on the SuperMUC Haswell Nodes (Phase 2)

ClassName	Purpose	Remarks	Max. Island Count (min,max)	min - max nodes	Wall Clock-Limit	Memory per Node (usable)	Run limit per user
test	Test and interactive use		1	1 - 20	30 min	64 (~58)	1
micro	Small jobs, pre- and postprocessing runs (internally restricted to run only on some specific islands)		1	1 - 20	48 h	64 (~58)	8
general	Medium-sized production runs fitting into a single island	Requires a minimum number of nodes (via #@ island_count).	1 or 1,2	21 - 512	48 h	64 (~58)	8
big	Big memory per node jobs (pre- and postprocessing runs, on some Haswell Nodes equipped with 256 GB/node)		1	1 - 8	6 h	256 (~250)	1

More information on submitting Batch Jobs via IBM Load Leveler can be found online under https://www.lrz.de/services/compute/supermuc/loadleveler/.

An example jobscript for a pure MPI only job using IBM MPI is shown here:

```
MPI only Job (IBM MPI)

#!/bin/bash
# DO NOT USE environment = COPY_ALL
```

```
##
## optional: energy policy tags
##
#@ job_type = parallel
#@ class = general
#@ node = 100
###@ island_count= not needed for
#### class general
#@ total_tasks= 2800
## other example
##@ tasks_per_node = 28
#@ wall_clock_limit = 1:20:30
##                    1 h 20 min 30 secs
#@ job_name = mytest
#@ network.MPI = sn_all,not_shared,us
#@ initialdir = $(home)/mydir
#@ output = job.$(schedd_host).$(jobid).out
#@ error =  job.$(schedd_host).$(jobid).err
#@ notification=always
#@ notify_user=youremail_at_yoursite.xx
#@ queue
. /etc/profile
. /etc/profile.d/modules.sh
poe ./myprog.exe
```

More SuperMUC Phase 2 specific examples can be found under https://www.lrz.de/services/compute/supermuc/loadleveler/examples_haswell_nodes/.

7.4.8. Programming Environment

General information about the programming environment can be found on https://www.lrz.de/services/compute/supermuc/programming/.

7.4.8.1. Available Compilers

A complete list of all compilers and parallel programming libraries available on SuperMUC can be obtained using the following module command

```
module avail -c compilers
module avail -c parallel
```

which lists all packages installed in LRZ's module classes compilers and parallel.

Since SuperMUC is based on Intel technology, LRZ recommends the usage of the **Intel Compilers and Performance Libraries** such as Intel MKL as a first choice. Licensing and support agreements with Intel ensure that bug-fixes should be available within reasonably short order.

Recommended Intel compiler Flags:

```
icc -O3 -xCORE-AVX2 program.c
icpc -c program.cpp -xHost -O3 -xCORE-AVX2
ifort program.f90  -O3 -xCORE-AVX2
```

Use the **GCC compilers** only if strict compatibility to gcc/gfortran is needed.

Recommended GNU environment:

```
module unload intel
module load gcc
```

```
gcc -c program.c -march=core-avx2
```

Some commercial packages still require the availability of the **Portland Group compiler suite**. High Performance Fortran is also supported by the PGI Fortran compiler. Hence, this package is available on SuperMUC. In order to use the PGI compiler on the login-nodes of SuperMUC, please load the compiler modules ccomp/pgi and fortran/pgi. Anyway, support for the PGI compilers is limited: the LRZ HPC team will report bugs to PGI, but this is kept at low priority. Documentation for the PGI compilers is available from the PGI web site [http://www.pgroup.com/resources/docs.htm].

7.4.8.2. Available MPI Implementations

The fully supported MPI environments are listed in the following table.

Hardware Interface	supported Compiler	MPI flavour	Environment Module Name	Compiler Wrappers	Command for Starting Executable
Infiniband and shared memory	Intel compilers(others are possible)	IBM MPI	mpi.ibm	mpif90, mpicc, mpiCC	poe, mpiexec
Infiniband and shared memory	Intel compilers(others are possible)	Intel MPI	mpi.intel	mpif90, mpicc, mpiCC	poe, mpiexec

The MPI implementation by IBM is the default MPI environment on SuperMUC. Specific hints on the usage of this MPI variant can be found under https://www.lrz.de/services/software/parallel/mpi/ibmmpi/

Futhermore Intel's MPI implementation is also fully supported. You need to unload the default MPI environment before loading the Intel MPI module:

```
module unload mpi.ibm
module load mpi.intel
```

Details about Intel MPI on SuperMUC can be found under https://www.lrz.de/services/software/parallel/mpi/intelmpi/.

7.4.8.3. OpenMP

Intel’s OpenMP implementation is the recommended implementation on SuperMUC.

7.4.9. Performance Analysis

Since SuperMUC is Intel based we recommend the Intel tools described in detail in this guide for performance analysis. Other tools available on SuperMUC can be found under https://www.lrz.de/services/compute/supermuc/tuning/

7.4.10. Debugging

The following 2 debuggers with graphical Interface (GUI) are available:

1. DDT: Distributed Debugging Tool: a commercial product by Allinea Software (`module load ddt`).

2. Totalview: A commercial product by Etnus. (`module load totalview`).

The GUI driven debuggers offer a graphical user interface; simple debugging sessions can therefore be handled without intensive, prior study of man-pages and manuals. DDT and Totalview are advanced tools for more complex debugging, especially when it comes to debugging parallel codes (MPI, OpenMP). They allow to inspect data structures in the different threads of a parallel program, set global breakpoints, set breakpoints in individual threads, etc. DDT is the preferred debugger on SuperMUC, and the largest number of licences is available. Totalview can also be used in CLI mode, whereas DDT is a pure GUI tool.

More information is available in `($DDT_DOC)` and `($TOTALVIEW_DOC)`. These environment variables are set by the module command on the LRZ HPC systems after loading the module of the respective debugger.

Other debuggers like gdb are also available, but they can hardly be used for parallel programs.