Programming the Intel® Xeon Phi™ Coprocessor

Tim Cramer
cramer@rz.rwth-aachen.de

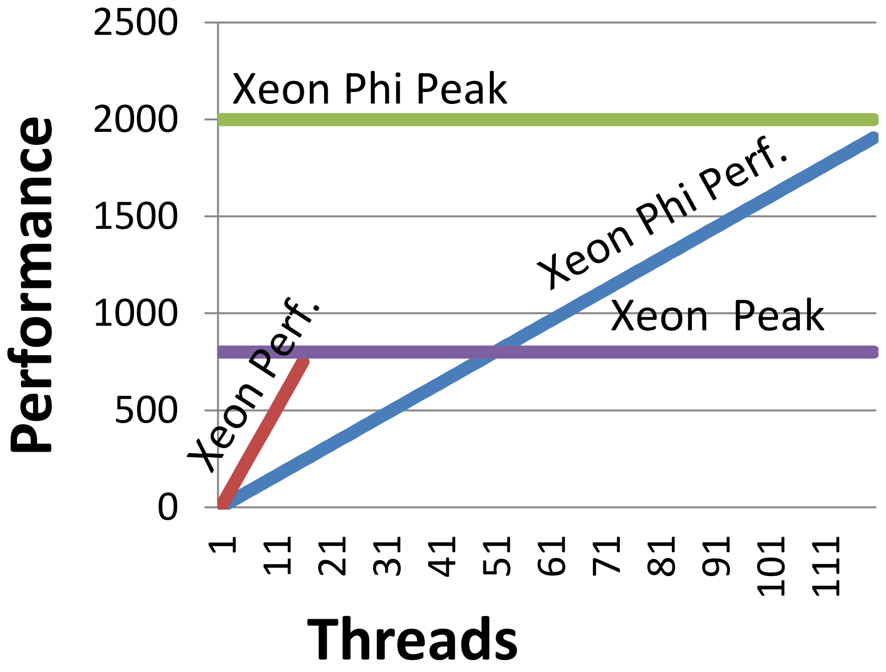
Rechen- und Kommunikationszentrum (RZ)

Agenda

- **Motivation**
- **Many Integrated Core (MIC) Architecture**
- **Programming Models**
  - → Native
  - → Language Extension for Offload (LEO)
  - → Symmetric (with Message Passing)
- **Debugging**
- **Optimization**
- **Case Study: CG Solver**
- **RWTH MIC Environment**

Motivation

- **Demand for more compute power**
- **Reach higher performance with more threads**
- **Power consumption: Better performance / watt ratio**
- **GPUs are one alternative, but: CUDA is hard to learn / program**
- **Intel Xeon Phi can be programmed with established programming paradigms like OpenMP, MPI, Pthreads**



Source: James Reinders, Intel

What is the Intel Xeon Phi?

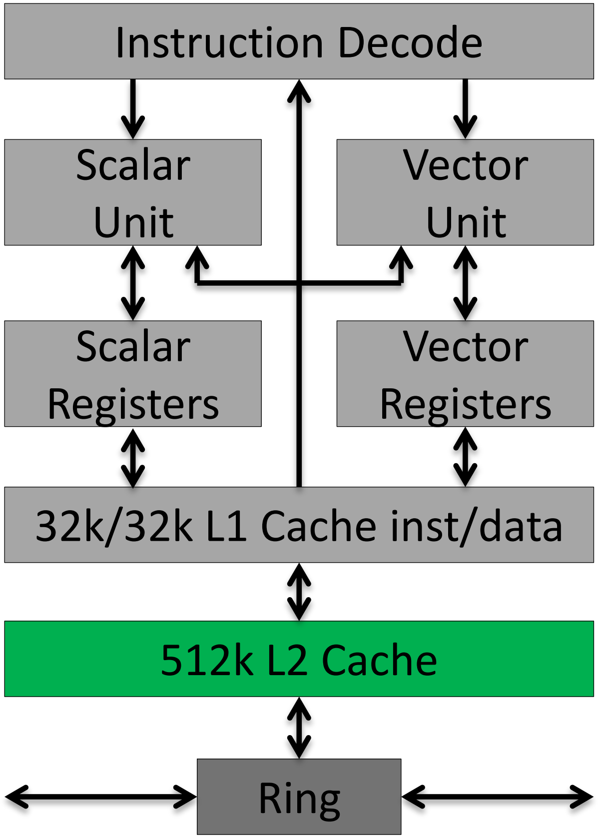
When to Use the Intel Xeon Phi?

- **Xeon Phi is not intended to replace Xeon -> choose the right vehicle**

Architecture (1/2)

Source: Intel

Intel Xeon Phi Coprocessor
- 1 x Intel Xeon Phi @ 1090 MHz
- 60 Cores (in-order)
- ~ 1 TFLOPS DP Peak
- 4 hardware threads per core
- 8 GB GDDR5 memory
- 512-bit SIMD vectors (32 registers)
- Fully-coherent L1 and L2 caches
- Plugged into PCI Express bus



Instruction Decode

Scalar Unit

Vector Unit

Scalar Registers

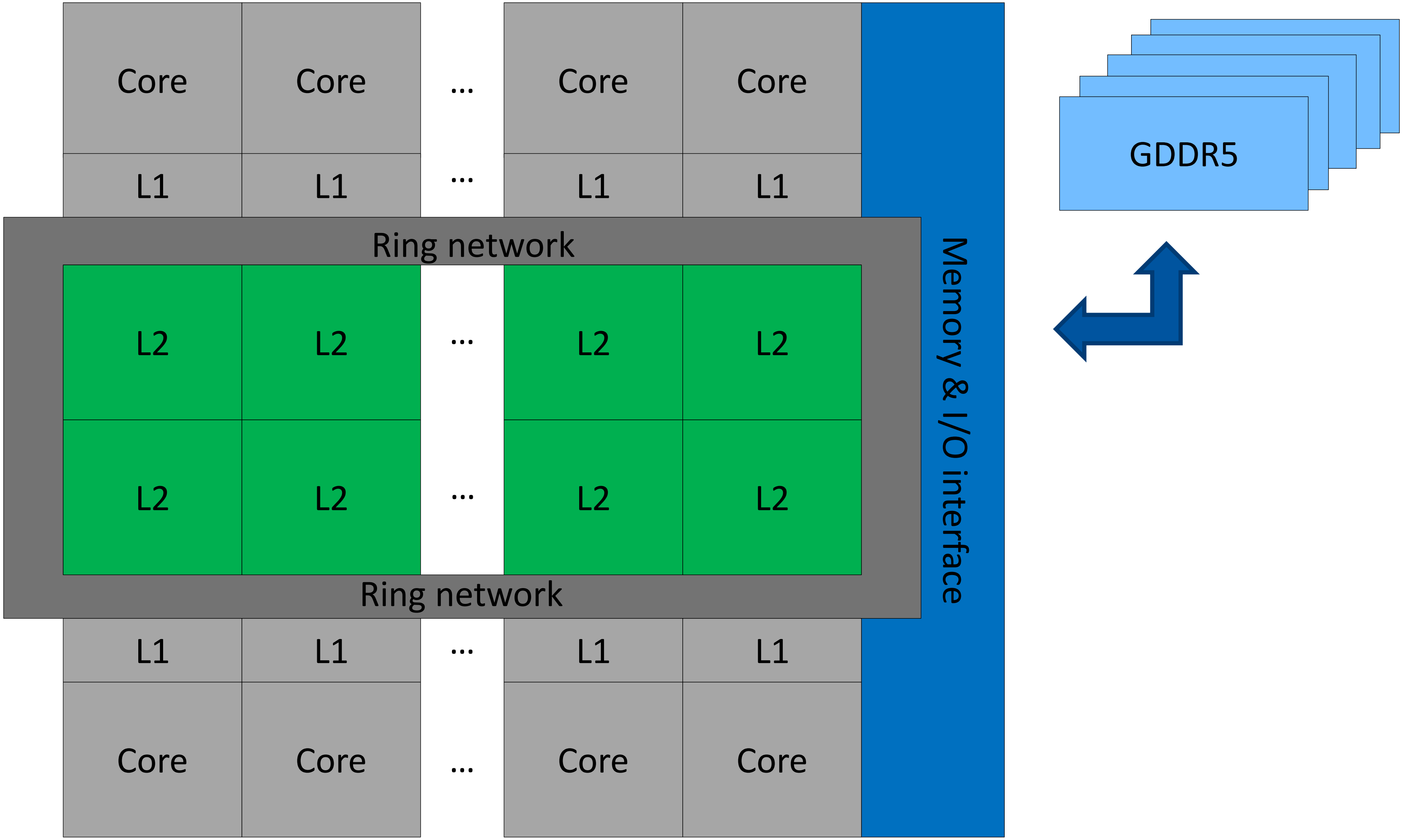
Vector Registers

32k/32k L1 Cache inst/data

512k L2 Cache

Ring

Architecture (2/2)



Core Core … Core Core

L1 L1 … L1 L1

Ring network

L2 L2 … L2 L2

L2 L2 … L2 L2

Ring network

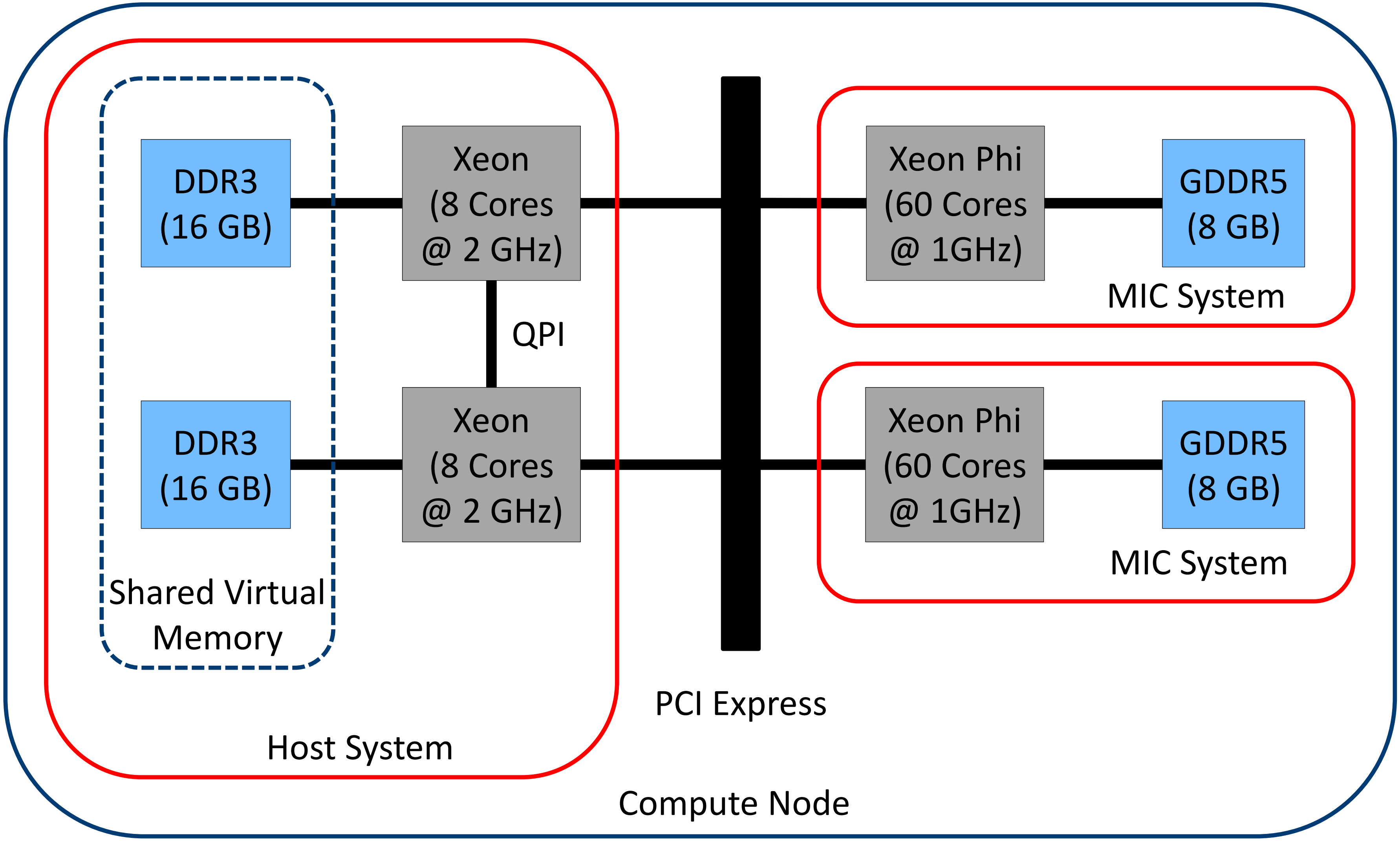
L1 L1 … L1 L1

Core Core … Core Core

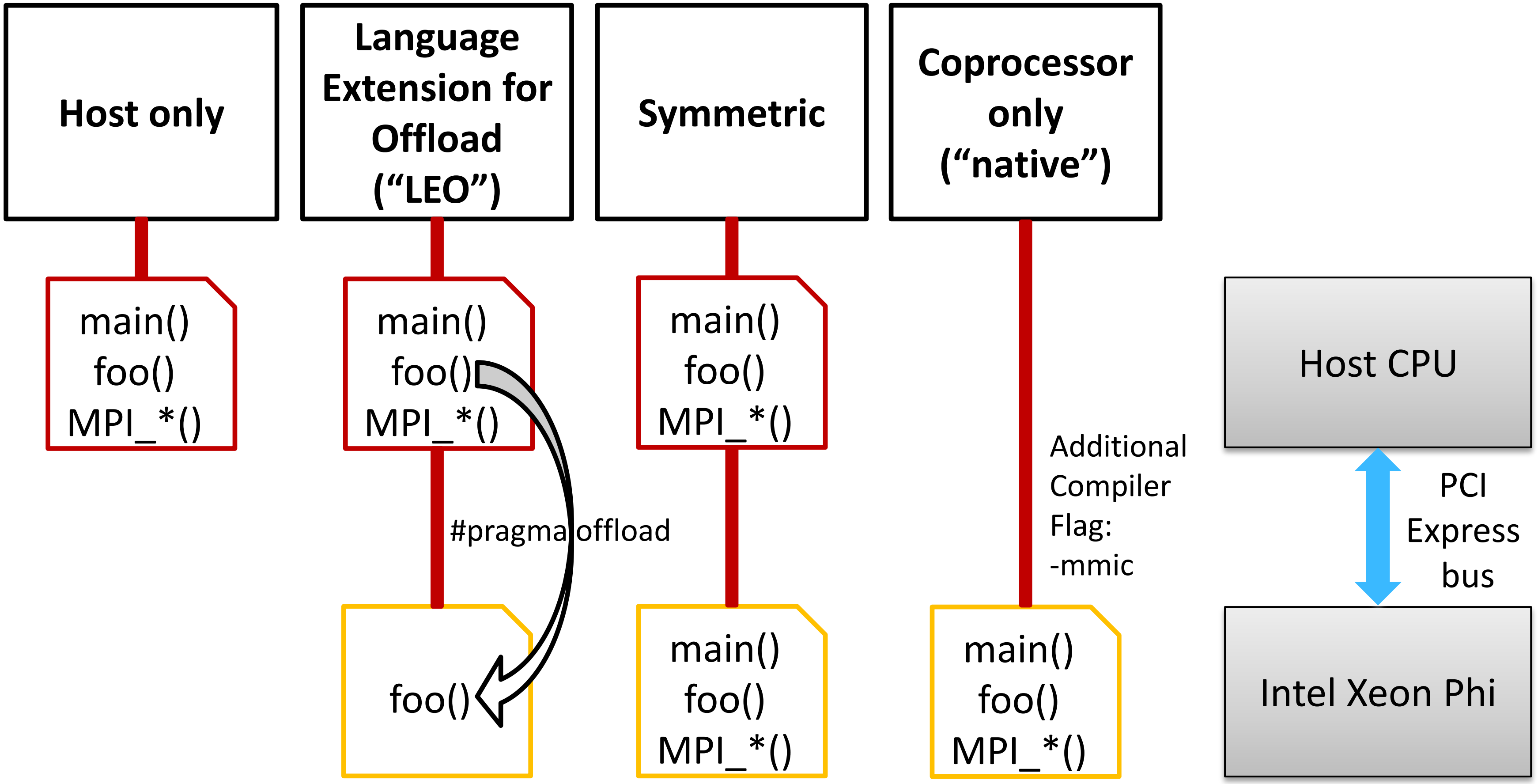
Memory & I/O interface

GDDR5

Xeon Phi Nodes at RWTH Aachen



Programming Models



Coprocessor only (“native”)

- **Cross-compile for the coprocessor**
  - → instruction set on the CPU and the coprocessor is similar, but identical
  - → easy (just add “-mmic”, login with ssh and execute)
  - → OpenMP, posix threads, OpenCL, MPI usable
  - → analyze benefit for hotspots
  - → very slow IO
  - → poor single thread performance
  - → host CPU will be bored
  - → only suitable for highly parallelized / scalable codes

Language Extension for Offload (LEO)

- **Add pragmas similar to OpenMP or OpenACC**
  - → C/C++: `#pragma offload target (mic:device_id)`
  - → Fortran: `!dir$ offload target (mic:device_id)`
  - → statements in this scope **can** be executed on the MIC (not guaranteed!)
- **Variable & function definitions**
  - → C/C++: `__attribute__ ((target(mic)))`
  - → Fortran: `!dir$ attributes offload:<MIC> :: <func-/varname>`
  - → compiles for, or allocates variable on, both the CPU and the MIC
  - → mark entire files or large blocks of code (C/C++ only)
    - → `#pragma offload_attribute(push, target(mic))`
    - → `#pragma offload_attribute(pop)`

LEO - Data Transfer

- **Host CPU and MIC do not share physical of virtual memory**
- **Implicit copy**
  - → Scalar variables
  - → Static arrays
- **Explicit copy**
  - → Programmer designates variables to be copied between host and card
    - →Specify `in`, `out`, `inout` or ~~`nocopy`~~ for the data
    - →`nocopy` is deprecated, use `in(data: length(0))` instead
  - → Data transfer **with** offload region
    - →C/C++: `#pragma offload target(mic) in(data:length(size))`
    - →Fortran: `!dir$ offload target(mic) in(data:length(size))`
  - → Data transfer **without** offload region
    - →C/C++: `#pragma offload_transfer target(mic) \`
      `in(data:length(size))`
    - →Fortran: `!dir$ offload_transfer target(mic) &`
      `in(data:length(size))`

LEO – First Simple Example

- **Data transfer for offload**

C/C++

```
#pragma offload target (mic) out(a:length(count)) \
                             in(b:length(count))
for (i=0; i<count; i++)
{
     a[i] = b[i] * c + d;
}
```

Fortran

```
!dir$ offload begin target (mic) out(a) in(b)
do i=1, count
      a(i) = b(i) * c +d
enddo
!dir$ end offload
```

LEO – Second Simple Example

- **Compile functions / subroutines for the CPU and the coprocessor**

C/C++

```
__attribute__ ((target(mic)))
void foo(){
    printf(“Hello MIC\n”);
}

int main(){
#pragma offload target (mic)
    foo();
return 0;
}
```

Fortran

```
!dir$ attributes &
!dir$ offload:mic :: hello
subroutine hello
    write(*,*) "Hello MIC“
end subroutine
program main
!dir$ attributes &
!dir$ offload:mic :: hello
!dir$ offload begin target (mic)
    call hello()
!dir$ end offload
end program
```

The directive is needed within both the calling routine's scope and the function definition/scope for the function itself.

LEO - Managing Memory Allocation (1/3)

- **Memory management for pointer variables on the**
  - → CPU is still up to the programmer
  - → coprocesser is done automatically in `in`, `inout` and `out` clauses
- **Input / Output Pointers**
  - → fresh memory allocation for each pointer variable by default (on coprocesser)
  - → de-allocation after offload region (on coprocesser)
  - → use `alloc_if` and `free_if` qualifiers to modify the allocation defaults
- **Data transfer in pre-allocated memory**
  - → Retain target memory by setting `free_if(0)` or `free_if(.false.)`
  - → Reuse data in subsequent offload by setting `alloc_if(0)` or `alloc_if(.false.)`
  - → important: always specify the target number on systems with multiple coprocessors: `#pragma offload target(mic:0)`

LEO - Managing Memory Allocation (2/3)

- **Example in C**

```
#Define macros to make modifiers more understandable
#define ALLOC   alloc_if(1)
#define FREE    free_if(1)
#define RETAIN  free_if(0)
#define REUSE   alloc_if(0)

#Allocate (default) the memory, but do not de-allocate
#pragma offload target(mic:0) in(a:length(8) ALLOC RETAIN)
…
#Do not allocate or de-allocate the memory
#pragma offload target(mic:0) in(a:length(8) REUSE RETAIN)
…
#Do not allocate the memory, but de-allocate (default)
#pragma offload target(mic:0) in(a:length(0) REUSE FREE)
…
```

LEO - Managing Memory Allocation (3/3)

- **Example in Fortran**

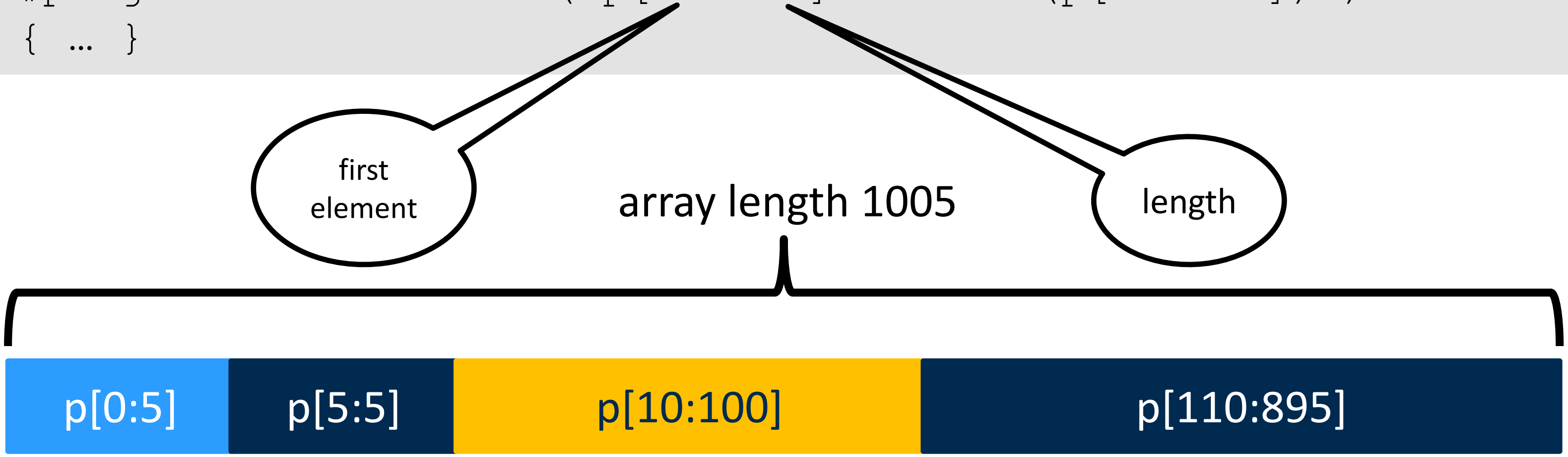
```
!Compiler allocated and frees data around the offload
real, dimension(8) :: a

!Allocate (default) the memory, but do not de-allocate
!dir$ offload target(mic:0) in(a:length(8) alloc_if(.true.) &
                              free_if(.false.))
…
!Do not allocate or de-allocate the memory
!dir$ offload target(mic:0) in(a:length(8) alloc_if(.false.) &
                              free_if(.false.))
…
!Do not allocate the memory, but de-allocate (default)
!dir$ offload target(mic:0) in(a:length(0) alloc_if(.false.) &
                               free_if(.true.))
…
```

LEO - Allocation for Parts of Arrays (1/3)

- **Allocation of array slices is possible**
  - → C/C++

```
int *p;
//  1000 elements allocated. Data transferred into p[10:100]
#pragma offload … in ( p[10:100] : alloc(p[5:1000]) )
{ … }
```

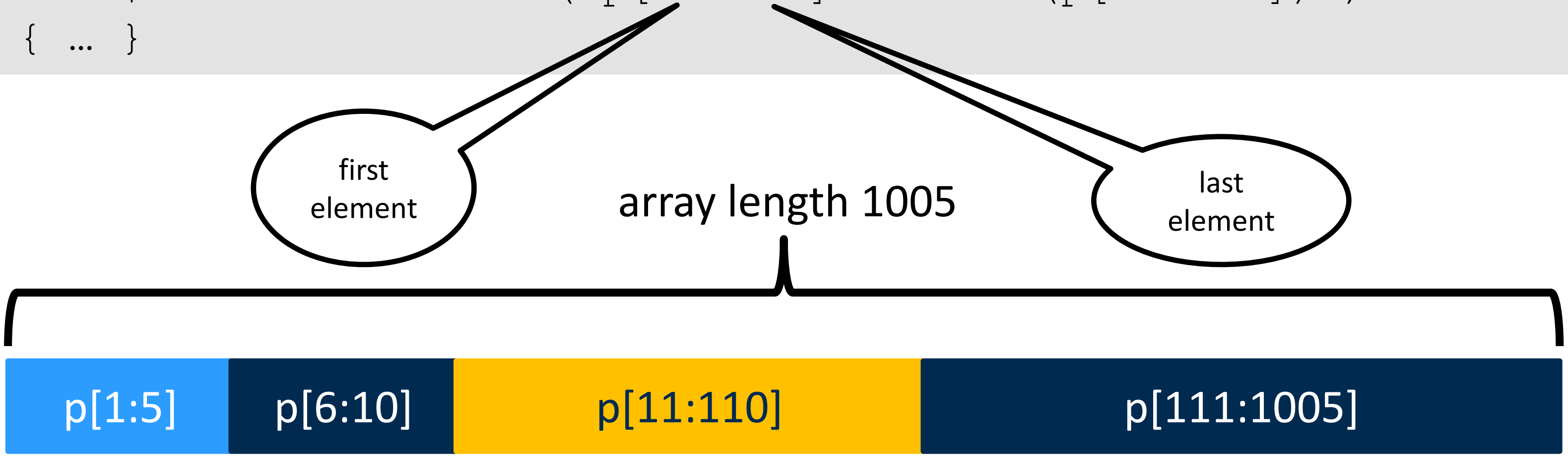


  - → `alloc(p[5:1000])` modifier allocate 1000 elements on coprocessor
  - → first useable element has index 5, last 1004 (dark blue + orange)
  - → `p[10:100]` specifies 100 elements to transfer (orange)

LEO - Allocation for Parts of Arrays (2/3)

- **Allocation of array slices is possible**
  - → Fortran

```
integer :: p (1005);
//  1000 elements allocated. Data transferred into p[11:110]
!dir$   offload … in ( p[11:110] : alloc(p[6:1005]) )
{ … }
```



- → `alloc(p[6:1005])` modifier allocate 1000 elements on coprocessor
- → first useable element has index 6, last 1005 (dark blue + orange)
- → `p[11:110]` specifies 100 elements to transfer (orange)

LEO - Allocation for Parts of Arrays (3/3)

- **Also possible for multi-dimensional arrays**
  - → C/C++ (Fortran analogous)

```
int[4][4] *p;
#pragma offload … in ( (*p)[2][:] : alloc(p[1:4][:]) )
{ … }
```

- → `alloc(p[1:4][:])` modifier allocates 16 elements on co-processor for a 5x4 shape (dark blue)
- → first row is not allocated (light blue)
- → only row 2 is transferred to the coprocessor (orange)

LEO – Partial Memory Copies

- **Moving data from one variable to another**
  - → copy data from the CPU to another array on the MIC is possible
  - → Overlapping copy is undefined

```
INTEGER :: P (1000), P1 (2000)
INTEGER :: RANK1 (1000), RANK2 (10, 100)
!          Partial copy
!DIR$ OFFLOAD … (P (1:500) : INTO ( P1 (501:1000) ) )
!          Overlapping copy; result undefined
!DIR$ OFFLOAD … IN (P (1:600) : INTO (P1 (1:600))) &
&               IN (P (601:1000) : INTO (P1 (100:499)))
```

LEO and OpenMP

- **Use OpenMP directives in an offload region**
- **Fortran**
  - → `omp` is optional
  - → When `omp` is present, the next line must be an OpenMP directive
  - → if not, the next line can also be a call or assignment statement
- **Number of processes**
  - → Set with environment variables
    `OMP_NUM_THREADS=16`
    `MIC_OMP_NUM_THREADS=120`
    `MIC_ENV_PREFIX=MIC`
  - → Set with API `omp_set_num_threads_target (TARGET_TYPE target_type, int target_number, int num_threads)`

```
#pragma offload target (mic)
#pragma omp parallel for
for (i=0; i<count; i++)
{
     a[i] = b[i] * c + d;
}
```

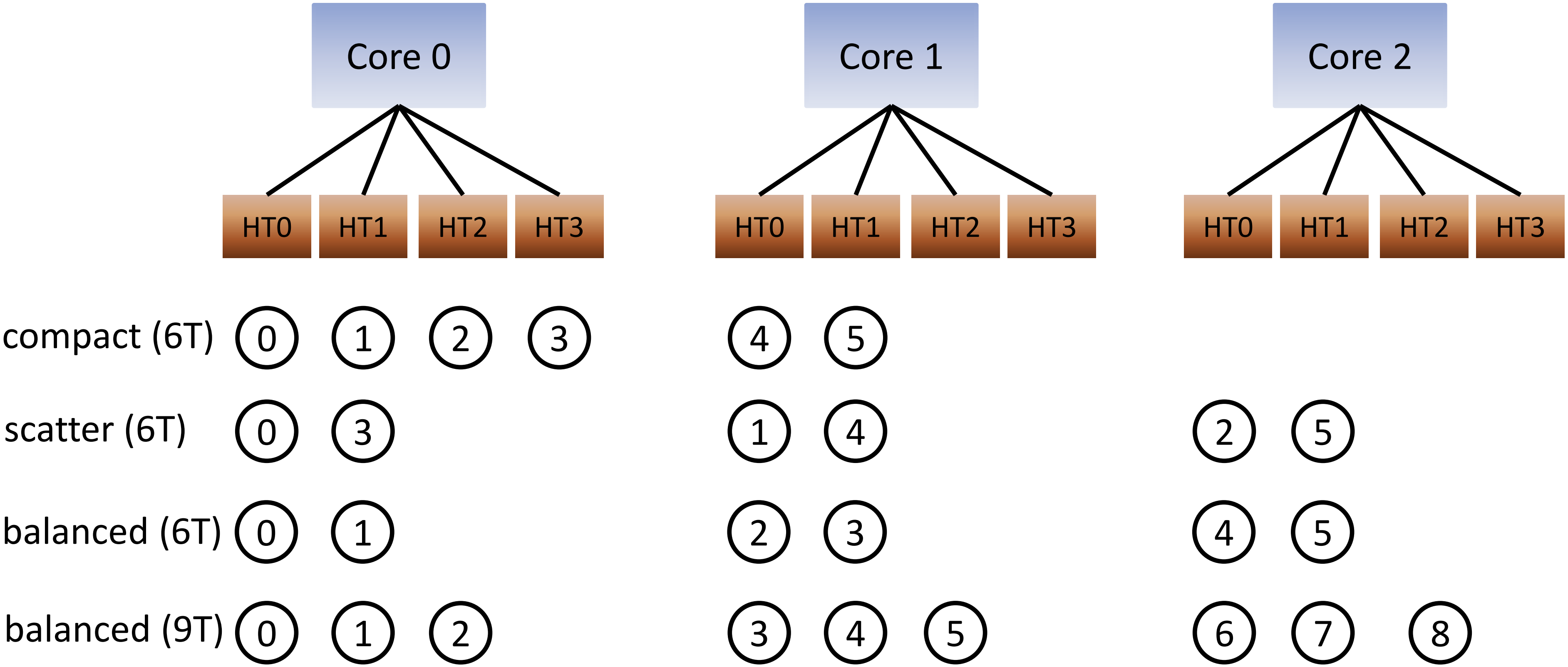
```
!dir omp offload target (mic)
!$omp parallel do
    do i=1, count
        A(i) = B(i) *c +d
    end do
!$omp end parallel
```

LEO – OpenMP Affinity (1/2)

- **The coprocessor has 4 hardware threads per core**
  - → to saturate the coprocessor at least 2 have to be utilized
- **Affinity strategies**
  - → `KMP_AFFINITY` propagates to the coprocessor
  - → new placing strategy for MIC: **`balanced`**
    - → runtime places threads on separate cores until all cores have at least one thread (similar to scatter)
    - → balanced ensures that threads are close to each other when multiple hardware threads per core are needed
    - → not supported for CPU, might produce a warning
    - → to avoid warning for CPU use `MIC_ENV_PREFIX=MIC` and then set `MIC_KMP_AFFINITY` for balanced placing strategy

LEO – OpenMP Affinity (2/2)

- **Example**
  - → 3-core system



	Core 0				Core 1				Core 2			
	HT0	HT1	HT2	HT3	HT0	HT1	HT2	HT3	HT0	HT1	HT2	HT3
compact (6T)	0	1	2	3	4	5						
scatter (6T)	0	3			1	4			2	5		
balanced (6T)	0	1			2	3			4	5		
balanced (9T)	0	1	2		3	4	5		6	7	8	

Using Native Libraries on MIC

- **Vendor / standard libraries**
  - → Standard libraries are available with no need to use special syntax or runtime features
  - → Some common libraries (MKL, OpenMP) are also available
- **Using native libraries in offload regions**
  - → To use shared libraries on MIC build it twice (for CPU and MIC using “`–mmic`”)
  - → Transfer native library to the device (e.g., using `ssh`)
  - → Use `MIC_LD_LIBRARY_PATH` to point to your own target library on the MIC

LEO - Static Libraries in Offloaded Code

- **Create own libraries**
  - → Use `xiar` or the equivalent `xild -lib` to create a static archive library containing routines with offload code
    - → Specify `-qoffload-build`, which causes `xiar` to create both a library for the CPU (`lib.a`), and for the coprocessor (`libMIC.a`)
    - → Use the same options available to `ar` to create the archive `xiar -qoffload-build` *`ar options archive [member...]`*
    - → Use the linker options `-Lpath` and `-llibname`, the compiler will automatically incorporate the corresponding coprocessor library (`libMIC.a`)
  - → Example
    - → Create `libsample.a` and `libsampleMIC.a:`

```
xiar -qoffload-build rcs libsample.a obj1.o obj2.o
! Linking:
ifort myprogram.f90 libsample.a
ifort myprogram.f90 -lsample
```

LEO Compiler – Command-line options

- **Offload-specific arguments for the Intel Compiler**
  - → Activate compiler reports for the offload

    ```
    -opt-report-phase:offload
    ```

  - → Deactivate offload support: `-no-offload`
  - → Build all functions and variables for the host and the MIC (might be very useful, especially for big code with many function calls)

    ```
    -offload-attribute-target=mic
    ```

  - → Set MIC specific option for the compiler or linker

    ```
    -offload-option,mic,tool,”option-list”
    ```

- **Example**

```
icc -g -O2 -lmiclib -xAVX -offload-option,mic,compiler,”-O3”
-offload-option,mic,ld:”-L/home/user/miclib” foo.c
```

LEO – Dealing with Multiple MICs

- **In source code**
  - → Include offload.h: `#include <offload.h>`
  - → determine the number of coprocessors in a system

    `_Offload_number_of_devices()`
  - → Determine the coprocessor on which a program is running

    `_Offload_get_device_number()`
- **At runtime**
  - → Restrict the devices that can be used for offload `OFFLOAD_DEVICES=1`
- **Memory management**
  - → If using `alloc_if` or `free_if` always specify the target device

LEO – MIC Specific Compiler Macros

- **Compiler defines macros for**
  - → Compiler independent code: `__INTEL_OFFLOAD`
  - → MIC specific code: `__MIC__`
- **Example**

```
#ifdef __INTEL_OFFLOAD
#include <offload.h>
#endif

#ifdef __INTEL_OFFLOAD
  printf("%d MICS available\n", _Offload_number_of_devices());
#endif

int main(){
#pragma offload target(mic)
  {
#ifdef __MIC__
  printf("Hello MIC number %d\n", _Offload_get_device_number());
#else
  printf("Hello HOST\n");
#endif
  }
}
```

LEO – I/O in Offloaded Regions

- **Stdout and stderr in offloaded codes**
  - → Writes (e.g., `printf`) performed in offloaded code may be buffered
  - → Output data may be lost when directed to a file

    ```
    $ ./a.out >log.txt
    ```

  - → I/O to a file requires an additional `fflush(0)` on the coprocessor
- **File I/O**
  - → All processes on the MIC will be started as `micuser` on the device
  - → At the moment no file I/O within offloaded regions possible

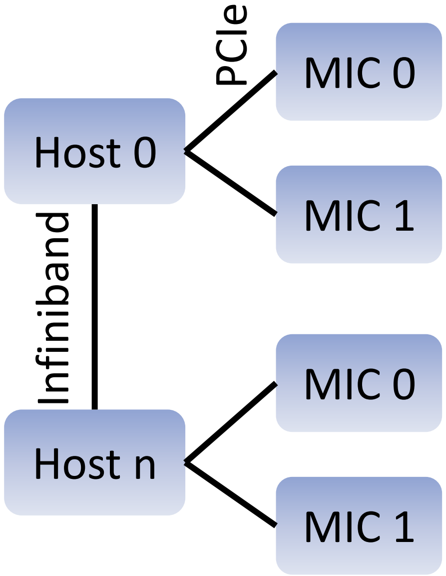
LEO – Asynchronous Computation

- **Default `offload` causes CPU thread to wait for completion**
  - → Asynchronous `offload` initiates the offload and continues immediately to the next statement
  - → Use `signal` clause to initiate
  - → Use `offload_wait` pragma to wait for completion
  - → These constructs always refer to a specific target
  - → Querying a signal before initiation will cause a runtime abort
- **Example**

```
char signal_var;
do {
      #pragma offload target (mic:0) signal(&signal_var)
      {
             long_running_mic_compute();
      }
      concurrent_cpu_activity();
      #pragma offload_wait target (mic:0) wait(&signal_var)
} while (1);
```

Symmetric Execution (1/2)

- **Using MPI on MIC in general**
  - → MPI is possible on MICs and host cpus
  - → host and cpu binaries are not compatible
  - → one for coprocessor, one for host cpu needed (e.g., `main` and `main.mic`)
  - → compile code twice, use "-`mmic`" for the coprocessor binary
  - → be aware of load-balancing
  - → hybrid codes using OpenMP and MPI might be very useful to avoid hundreds of processes on the relatively small coprocessor



Symmetric Execution (2/2)

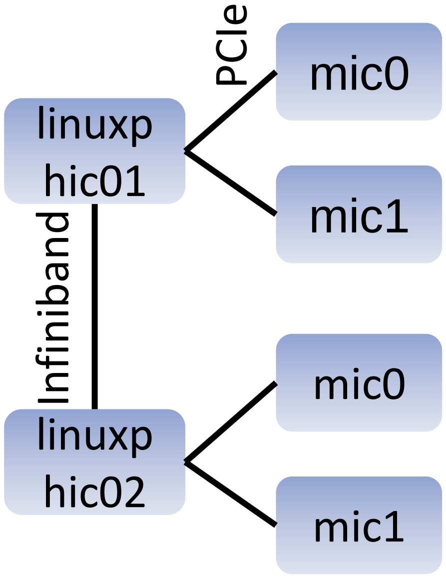
Using MPI on MIC on the RWTH Aachen Compute Cluster

- → At the moment we have a special module in our environment (`intelmpi/4.1mic` in `BETA`)
- → `ssh` is wrapped automatically
- → Environment variables are set

```
I_MPI_MIC=enabled
I_MPI_MIC_POSTFIX=.mic
```

- → Execution expects a `main` and `main.mic`

```
$ mpirun -machinefile=hostfile -n 512 ./main
```



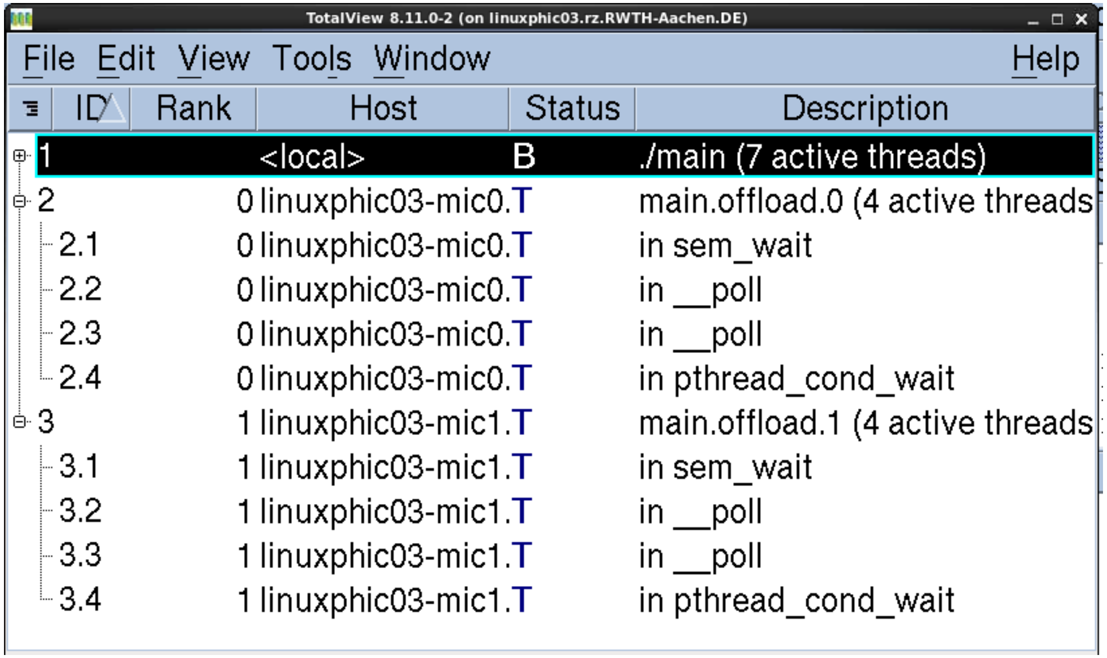
hostfile

```
linuxphic01:16
linuxphic02:16
linuxphic01-mic0:120
linuxphic01-mic1:120
linuxphic02-mic0:120
linuxphic02-mic1:120
```

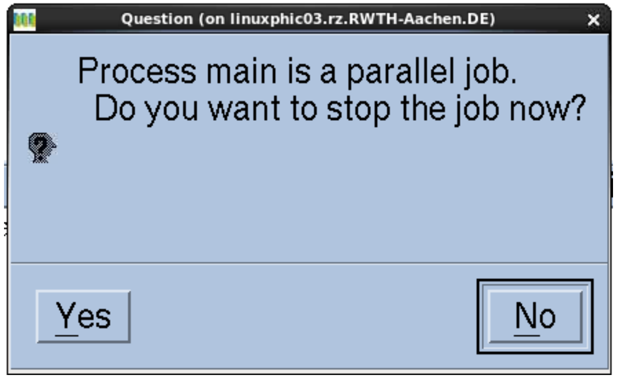
Debugging

- **Latest and greatest TotalView works (load `totalview/8.11.0-2`)**

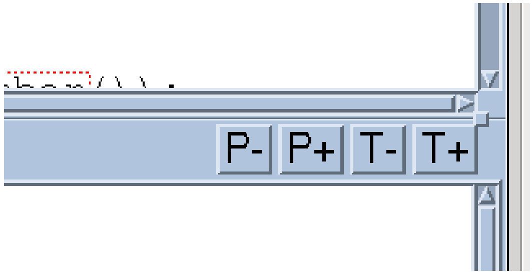
  → `icc -g -offload-option,mic,compiler,"-g" -o main main.c`

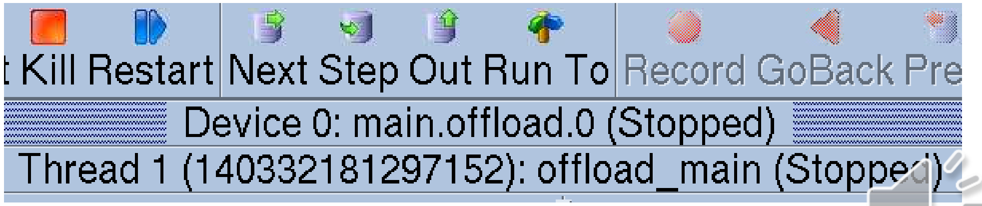


ID	Rank	Host	Status	Description
1		<local>	B	./main (7 active threads)
2	0	linuxphic03-mic0.	T	main.offload.0 (4 active threads
2.1	0	linuxphic03-mic0.	T	in sem_wait
2.2	0	linuxphic03-mic0.	T	in __poll
2.3	0	linuxphic03-mic0.	T	in __poll
2.4	0	linuxphic03-mic0.	T	in pthread_cond_wait
3	1	linuxphic03-mic1.	T	main.offload.1 (4 active threads
3.1	1	linuxphic03-mic1.	T	in sem_wait
3.2	1	linuxphic03-mic1.	T	in __poll
3.3	1	linuxphic03-mic1.	T	in __poll
3.4	1	linuxphic03-mic1.	T	in pthread_cond_wait



Process main is a parallel job.
Do you want to stop the job now?





Device 0: main.offload.0 (Stopped)

Thread 1 (140332181297152): offload_main (Stopped)

Optimizing for LEO

- **Measuring timing using the offload report**
  - → Set `OFFLOAD_REPORT` or use API `__Offload_report` to activate

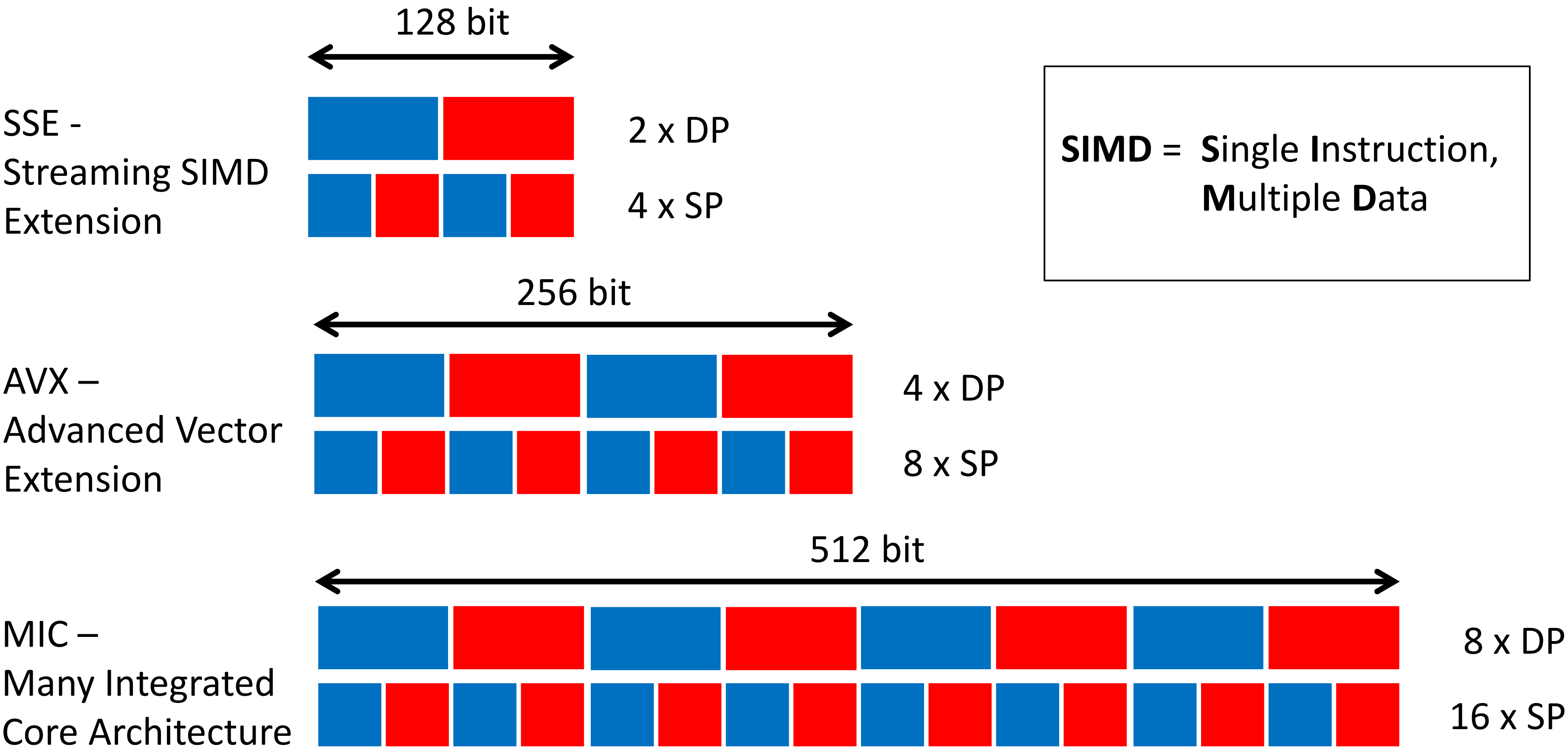
OFFLOAD_REPORT	Description
1	Report about time taken.
2	Adds the amount of data transferred between the CPU and the coprocessor
3	Additional details on offload activity

- **Conditional offload for LEO**
  - → `#pragma offload if (cond.)`
  - → Only offload if it is worth

```
#pragma offload target (mic) in (b:length(size)) \
             out (a:length(size)) if (size > 100)
for (i=0; i<count; i++)
{
      a[i] = b[i] * c + d;
}
```

Vectorization (1/3)

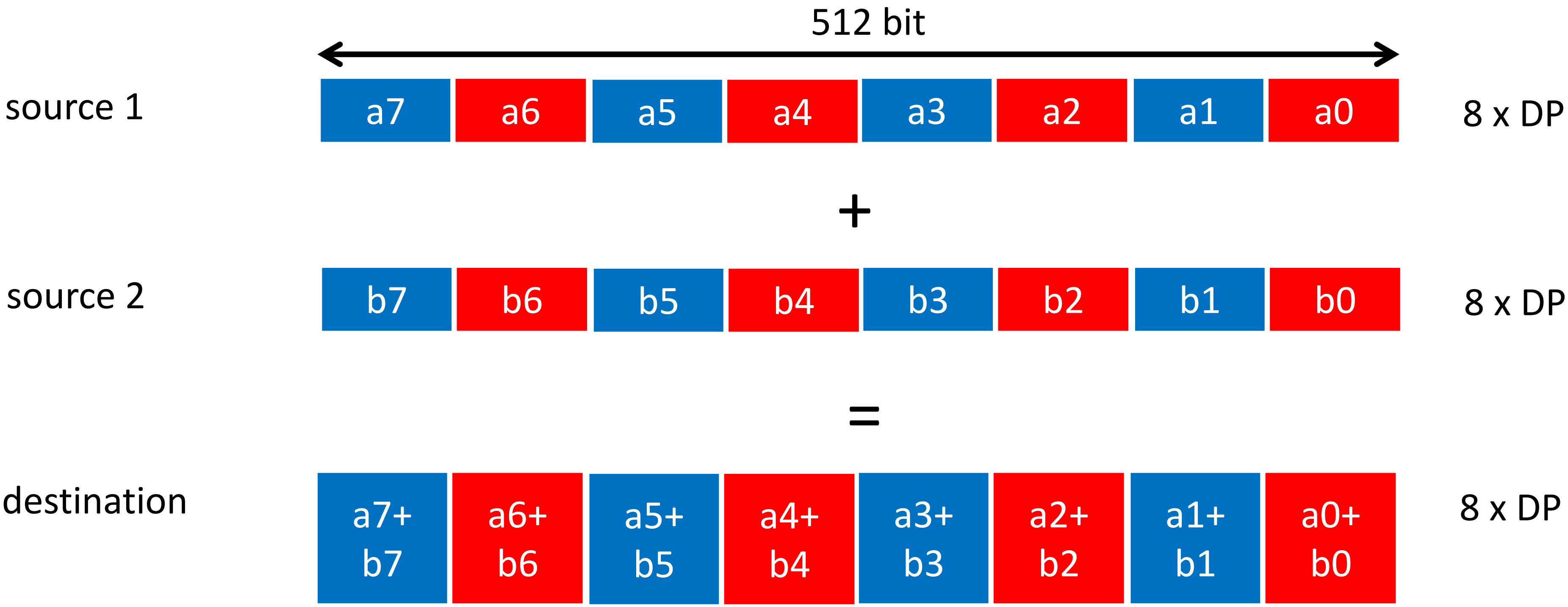
SIMD Vector Capabilities



SIMD = **S**ingle **I**nstruction,
Multiple **D**ata

Vectorization (2/3)

- **SIMD Vector Basic Arithmetic**

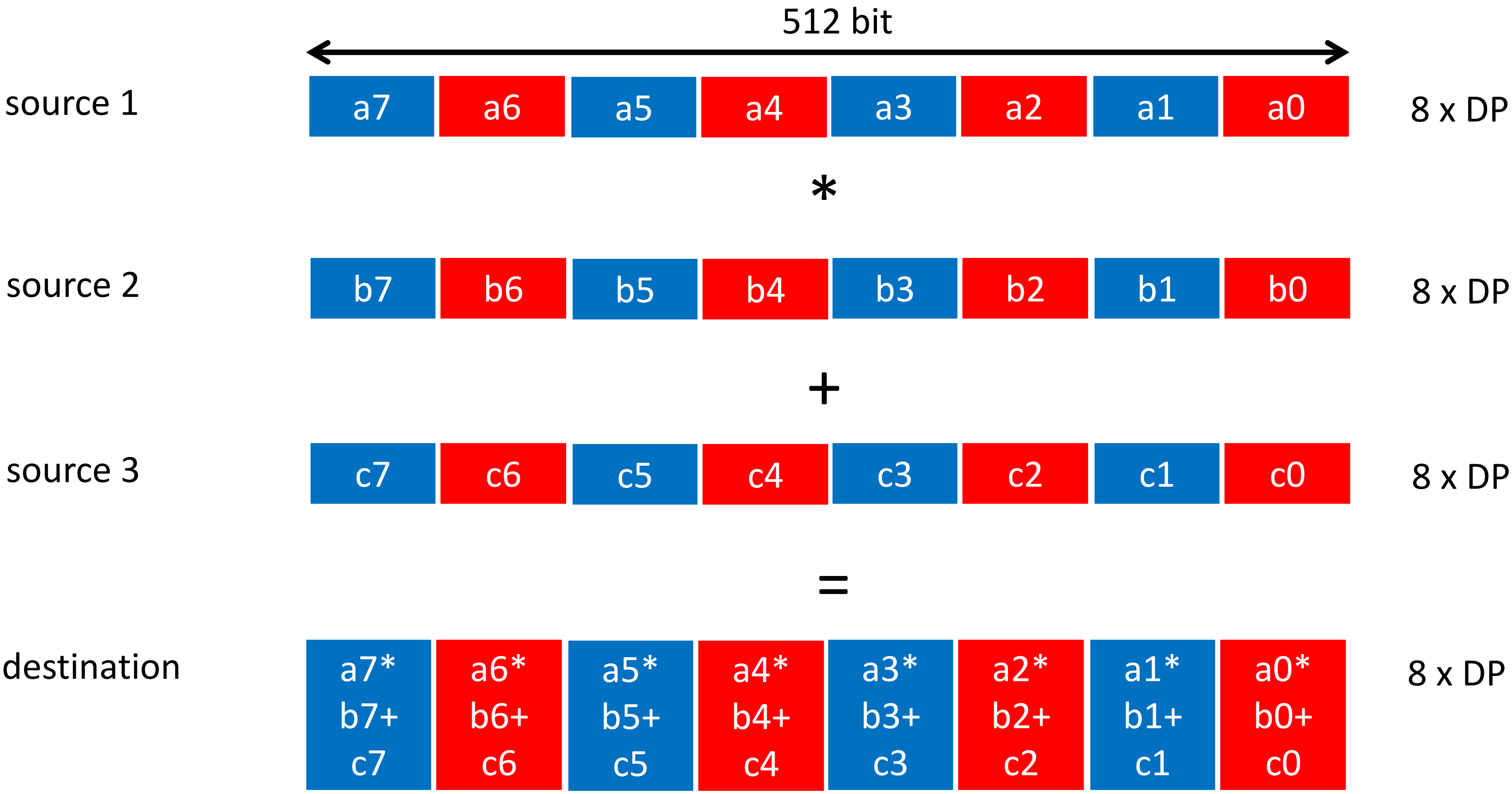


512 bit

source 1	a7	a6	a5	a4	a3	a2	a1	a0	8 x DP
					+				
source 2	b7	b6	b5	b4	b3	b2	b1	b0	8 x DP
					=				
destination	a7+ b7	a6+ b6	a5+ b5	a4+ b4	a3+ b3	a2+ b2	a1+ b1	a0+ b0	8 x DP

Vectorization (3/3)

- **SIMD Fused Multiply Add**



512 bit

source 1	a7	a6	a5	a4	a3	a2	a1	a0	8 x DP
					*				
source 2	b7	b6	b5	b4	b3	b2	b1	b0	8 x DP
					+				
source 3	c7	c6	c5	c4	c3	c2	c1	c0	8 x DP
					=				
destination	a7* b7+ c7	a6* b6+ c6	a5* b5+ c5	a4* b4+ c4	a3* b3+ c3	a2* b2+ c2	a1* b1+ c1	a0* b0+ c0	8 x DP

Compiler Support for SIMD Vectorization

- **Intel auto-vectorizer**
  - → Combination of loop unrolling and SIMD instructions to get vectorized loops
  - → No guarantee given, compiler might need some hints
- **Compiler feedback**
  - → Use `-vec-report [n]` to control the diagnostic information of the vectorizer
  - → `n` can be between 0-5 (recommended 3 or 5)
  - → concentrate on hotspots for optimization
- **C/C++ aliasing: Use `restricted` keyword**
- **Intel specific pragma**
  - → `#pragma vector` (Fortran: `!DIR$ VECTOR`)
    - → indicates to the compiler that the loop should be vectorized
  - → `#pragma simd` (Fortran: `!DIR$ SIMD`)
    - → enforces vectorization of the (innermost) loop
  - → SIMD support will be added in OpenMP 4.0
- **Refer to lab-exercises from Monday**

Alignment

- **Use efficient memory accesses**
  - → The MIC architecture requires all data accesses to be properly aligned according to their size
  - → For better performance align data to
    - →16 byte boundaries for SSE instructions
    - →32 byte boundaries for AVX instructions
    - →64 byte boundaries for MIC instructions
  - → Use `#pragma offload in(a:length(count) align(64))`

SoA vs. AoS

- use Structure of Arrays (SoA) instead of Array of Structures (AoS)
  - → Color structure

```
struct Color{ //AoS
    float r;
    float g;
    float b;
}
Color* c;
```

R	G	B

R	G	B	R	G	B	R	G	B

```
struct Colors{ //SoA
    float* r;
    float* g;
    float* b;
}
```

R	R	R	G	G	G	B	B	B

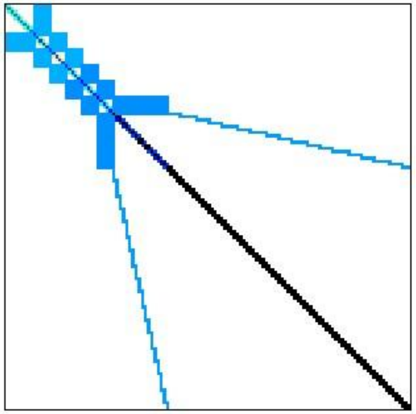
- **Detailed information: Intel Vectorization Guide**
  http://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers/

Performance Expectations

- **"WOW, 240 hardware threads on a single chip! My application will just rock!"**
  - → You really believe that?
  - → Remember the limitations!
    - →In-order cores
    - →limited hardware prefetching
    - →Running with 1GHz only
    - →Small Caches (2 levels)
    - →Poor single thread performance
    - →Small main memory
    - →PCIe as bottleneck + offload overhead

Case Study: CG solver

- **CG solver**
  - → Solves linear systems (`A*x=b`)
  - → dominated by sparse matrix vector mulitplication
  - → OpenMP
    - → first-touch
    - → optimal distribution (no static schedule)
- **Testcase**
  - → Fluorem/HV15R
  - → N=2,017,169, nnz=283,073,458
  - → 3.2 GB Memory footprint



- **Runtime one Xeon Phi and 16 (!) Nehalem EX**

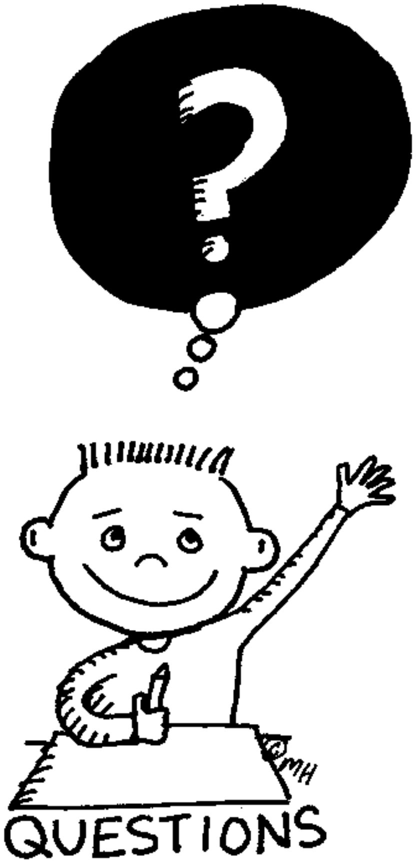
System	#Threads	Serial Time [s]	Parallel Time [s]	Speedup
Xeon Phi (61 cores)	244	2387.40	32.24	74
BCS (128 cores)	128	1176.81	18.10	65

- → As expected: BCS is faster, but results on Xeon Phi are pretty good
- → Experiences with other “real-world” applications are worse at the moment

RWTH MIC Environment

- **MIC cluster is brand-new and in beta stage**
  - → All configurations might change in future
- **Filesystem**
  - → `HOME` and `WORK` mounted `in /{home,work}/<timid>`
  - → Local filesystem in `/michome/<timid>`
  - → Software mounted in `/sw` on the device
- **Interactive usage**
  - → Linux is running on the device, but many features missing
  - → No modules available on the device
  - → Only one compiler version and one MPI version supported
- **MPI**
  - → Special module in BETA which wraps `ssh` and sets up specific environment
  - → Uses ssh keys in `$HOME/.ssh/mic`
  - → MPI will not be possible interactively in production stage (only in batch mode)

The End



Exercises

- Use `linuxphic{01,03,04,05,06}` (not linuxphic02!)

Front

Row 1	linuxphic01	linuxphic01
Row 2	linuxphic01	linuxphic03
Row 3	linuxphic03	linuxphic03
Row 4	linuxphic04	linuxphic04
Row 5	linuxphic05	linuxphic05
Row 6	linuxphic06	linuxphic06
Row 7	linuxphic06	