Introduction to Intel Xeon Phi programming techniques

Fabio Affinito

CINECA SCAI
SuperComputing Applications and Innovation

Outline

- High level overview of the Intel Xeon Phi hardware and software stack
- Intel Xeon Phi programming paradigms: offload and native
- Performance and thread parallelism
- Using MPI
- Tracing and profiling
- Conclusions

Preliminaries

- Wrong: Intel Xeon PHI.

  Correct: Intel Xeon Phi

Preliminaries

- Wrong: Intel Xeon PHI.

  Correct: Intel Xeon Phi
- Intel MIC is the name of the architecture, Intel Knights Corner is the name of the first model of the MIC architecture, Intel Xeon Phi is the commercial name the product...

Preliminaries

- Wrong: Intel Xeon PHI.

  Correct: Intel Xeon Phi
- Intel MIC is the name of the architecture, Intel Knights Corner is the name of the first model of the MIC architecture, Intel Xeon Phi is the commercial name the product...
- The Intel Xeon Phi IS NOT an accelerator

Preliminaries

- Wrong: Intel Xeon PHI.

  Correct: Intel Xeon Phi
- Intel MIC is the name of the architecture, Intel Knights Corner is the name of the first model of the MIC architecture, Intel Xeon Phi is the commercial name the product...
- The Intel Xeon Phi IS NOT an accelerator

  Ok, but it can behave very similarly to an accelerator

Preliminaries

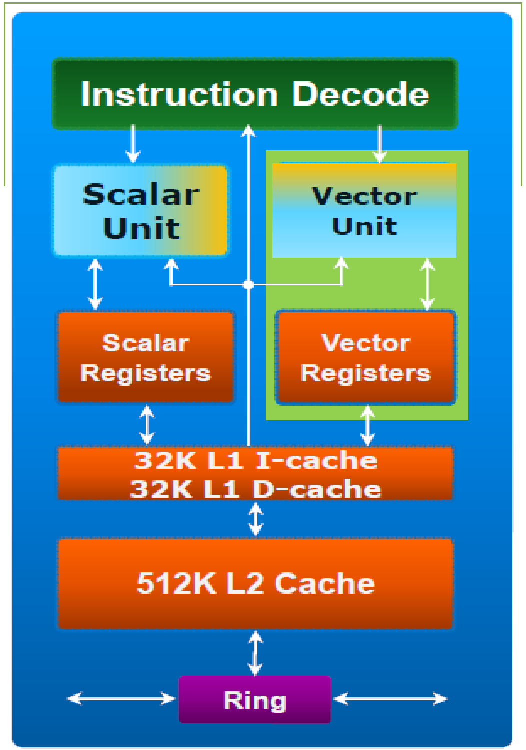
Yeah, they look pretty similar...

Outline

- High level overview of the Intel Xeon Phi hardware and software stack
- Intel Xeon Phi programming paradigms: offload and native
- Performance and thread parallelism
- Using MPI
- Tracing and profiling
- Conclusions

Intel Xeon Phi overview

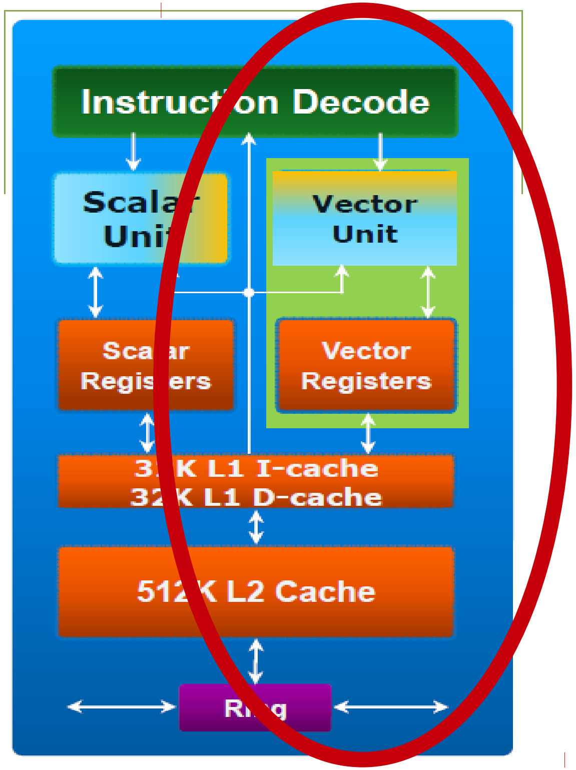
Each Intel Xeon Phi is a multithread execution unit



- > 50 in-order cores
- ring network
- 64-bit architecture
- scalar unit based on Intel Pentium processor family
  - two pipelines
    - dual issue with scalar instructions
  - one-per-clock scalar pipeline througput
    - 4 clock latency from issue to resolution
- 4 hardware threads per core

Intel Xeon Phi overview

Each Intel Xeon Phi is a multithread execution unit

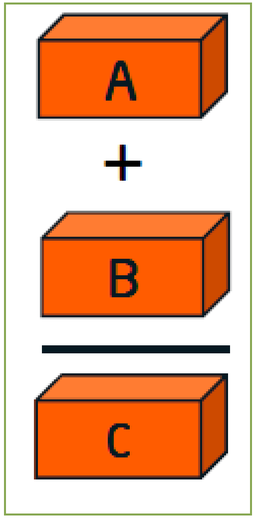


- New vector unit
  - 512-bit SIMD Instructions
    - not Intel SSE or Intel AVX
  - 32 512-bit wide vector registers
    - can contain 16 singles or 8 doubles per register
- Fully coherent L1 and L2 caches

Intel Xeon Phi overview

Vectorization: what is it?

```
for (i=0;i<=MAX;i++)
    c[i]=a[i]+b[i];
```



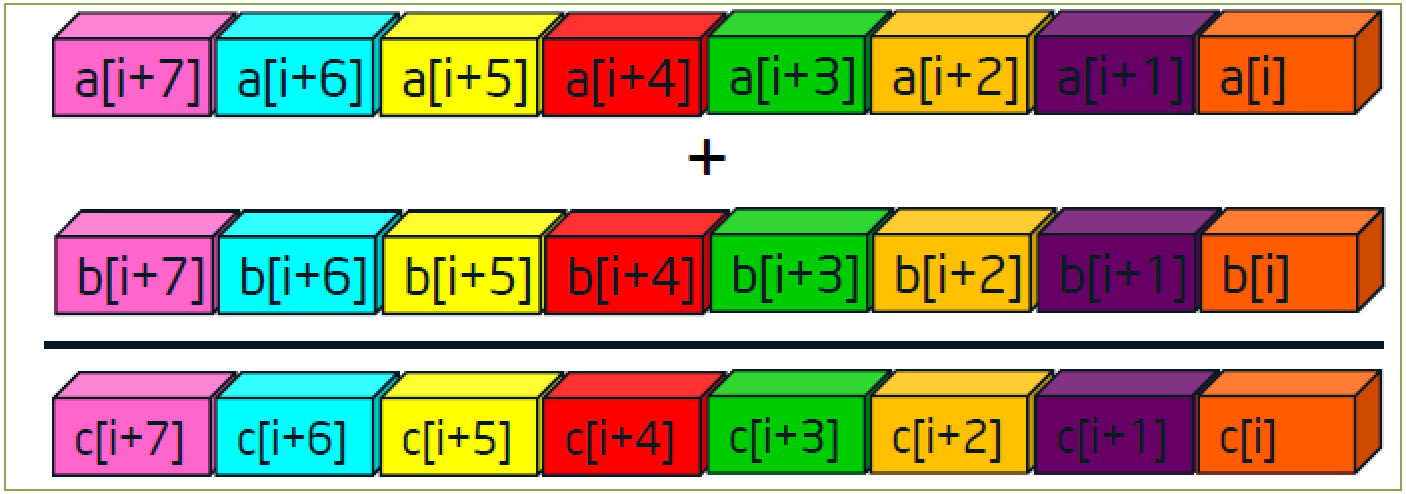
Scalar:

one instruction per cycle
one mathematical operation per cycle

Intel Xeon Phi overview

Vectorization: what is it?

```
for (i=0;i<=MAX;i++)
    c[i]=a[i]+b[i];
```

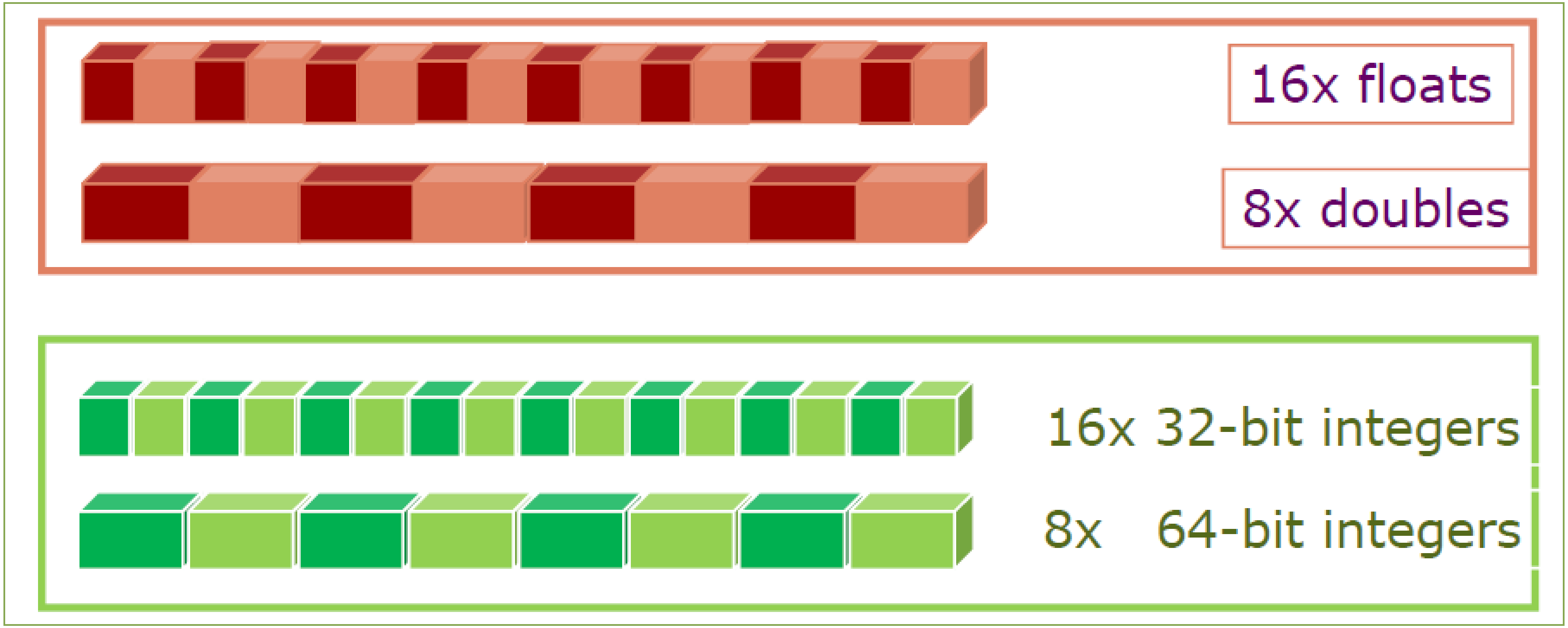


Vector:

one instruction per cycle
eight mathematical operation per cycle

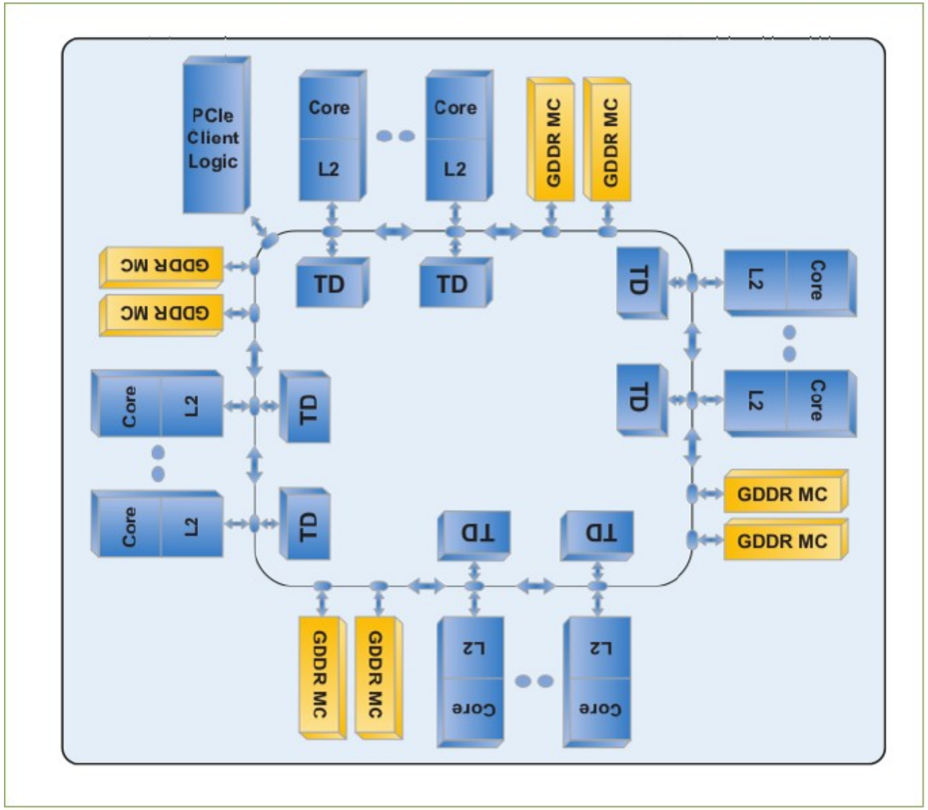
Intel Xeon Phi overview

Vectorization is crucial



Intel Xeon Phi overview

Caches and internal network



- bidirectional ring 115 GB/s
- GDDR5 memory
  - 16 memory channels
  - up to 5.5 Gb/s
  - 8 to 16 GB
- L1 32 K cache per core
  - 3 cycle access
  - up to 8 concurrent accesses
- L2 512 K cache per core
  - 11 cycle best access
  - up to 32 concurrent accesses

Intel Xeon Phi family

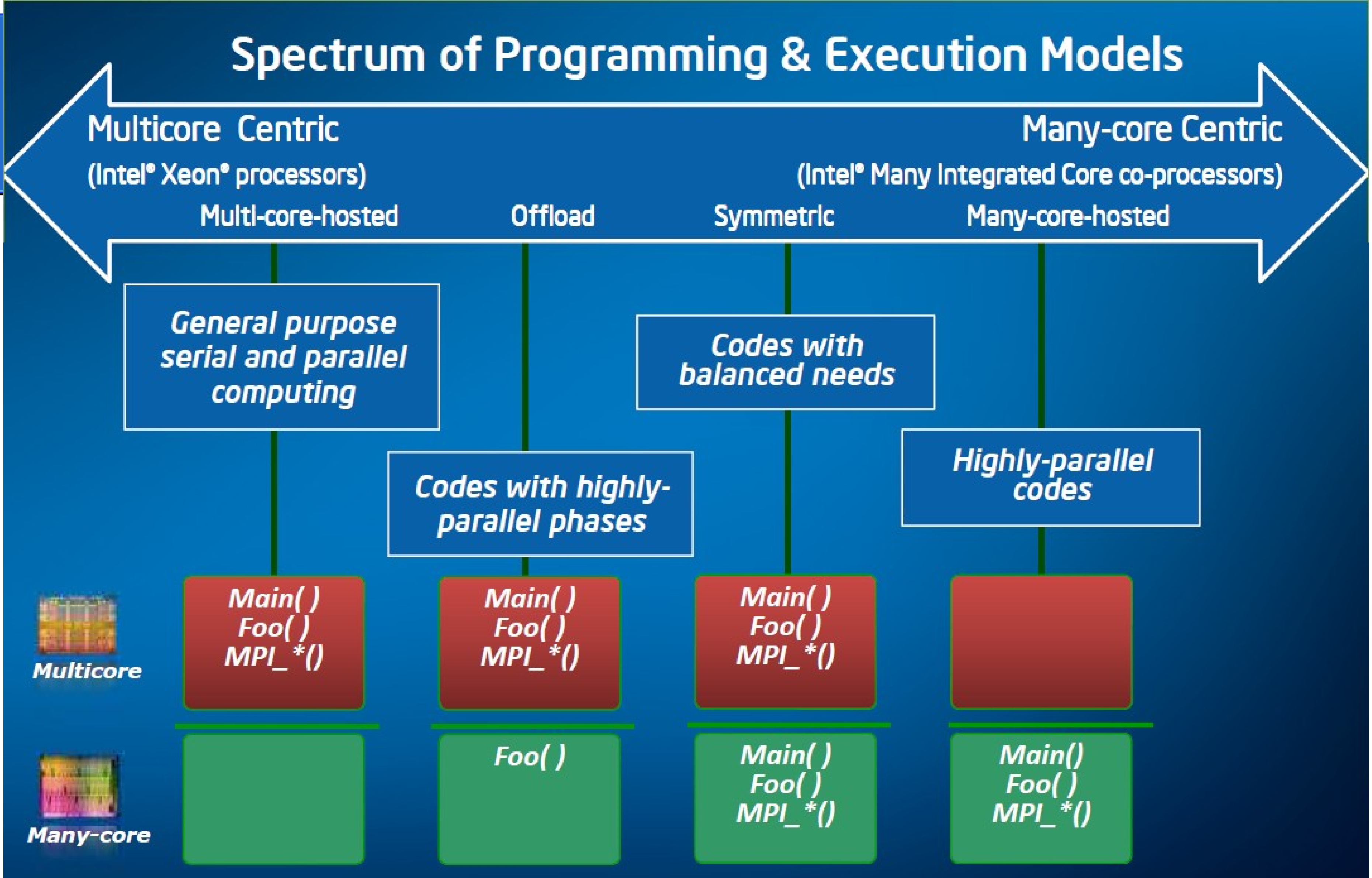
Processor Brand Name	Codename	SKU #	Form Factor, Thermal	Board TDP (Watts)	Max # of Cores	Clock Speed (GHz)	Peak Double Precision (GFLOP)	GDDR5 Memory Speeds (GT/s)	Peak Memory BW	Memory Capacity (GB)	Total Cache (MB)	Enabled Turbo	Turbo Clock Speed (GHz)
Intel® Xeon Phi™ Coprocessor x100	Knights Corner	7120P	PCIe Card, Passively Cooled	300	61	1.238	1208	5.5	352	16	30.5	Y	1.333
		7120X	PCIe Card, No Thermal Solution	300	61	1.238	1208	5.5	352	16	30.5	Y	1.333
		5120D	PCIe Dense Form Factor, No Thermal Solution	245	60	1.053	1011	5.5	352	8	30	N	N/A
		3120P	PCIe Card, Passively Cooled	300	57	1.1	1003	5.0	240	6	28.5	N	N/A
		3120A	PCIe Card, Actively Cooled	300	57	1.1	1003	5.0	240	6	28.5	N	N/A
		Previously Launched and Disclosed											
		5110P*	PCIe Card, Passively Cooled	225	60	1.053	1011	5.0	320	8	30	N	N/A

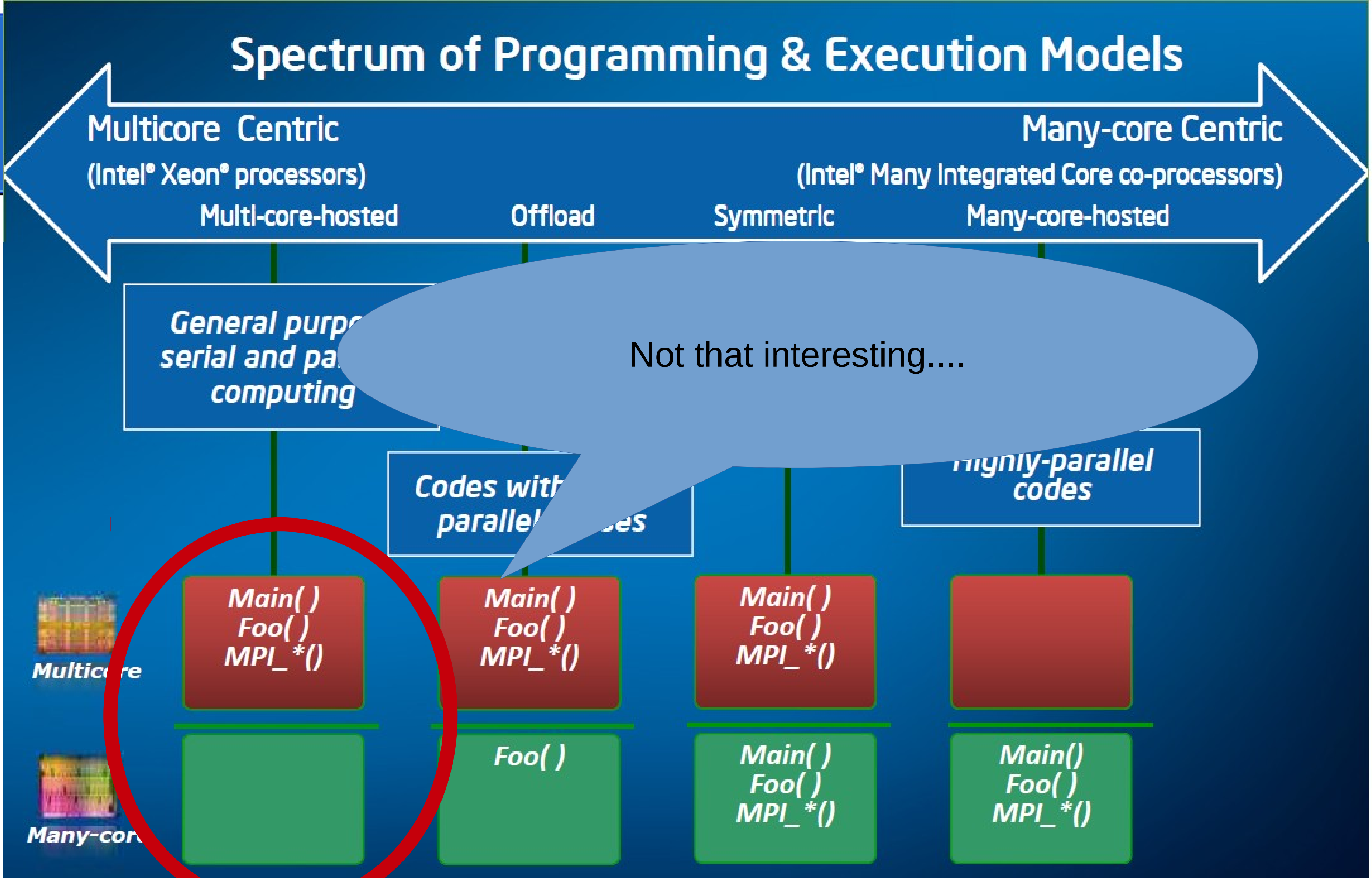
Intel Xeon Phi software

- Relying on the same architecture of the Pentium family, the Intel Xeon Phi platform can uses all the tools and software stack used by the Xeon product line:
  - Intel Composer XE (compilers)
  - Intel Vtune Amplifier XE, Advisor XE, Trace Analyzer (profiling and traces)
  - Intel MPI
  - Intel MKL libraries

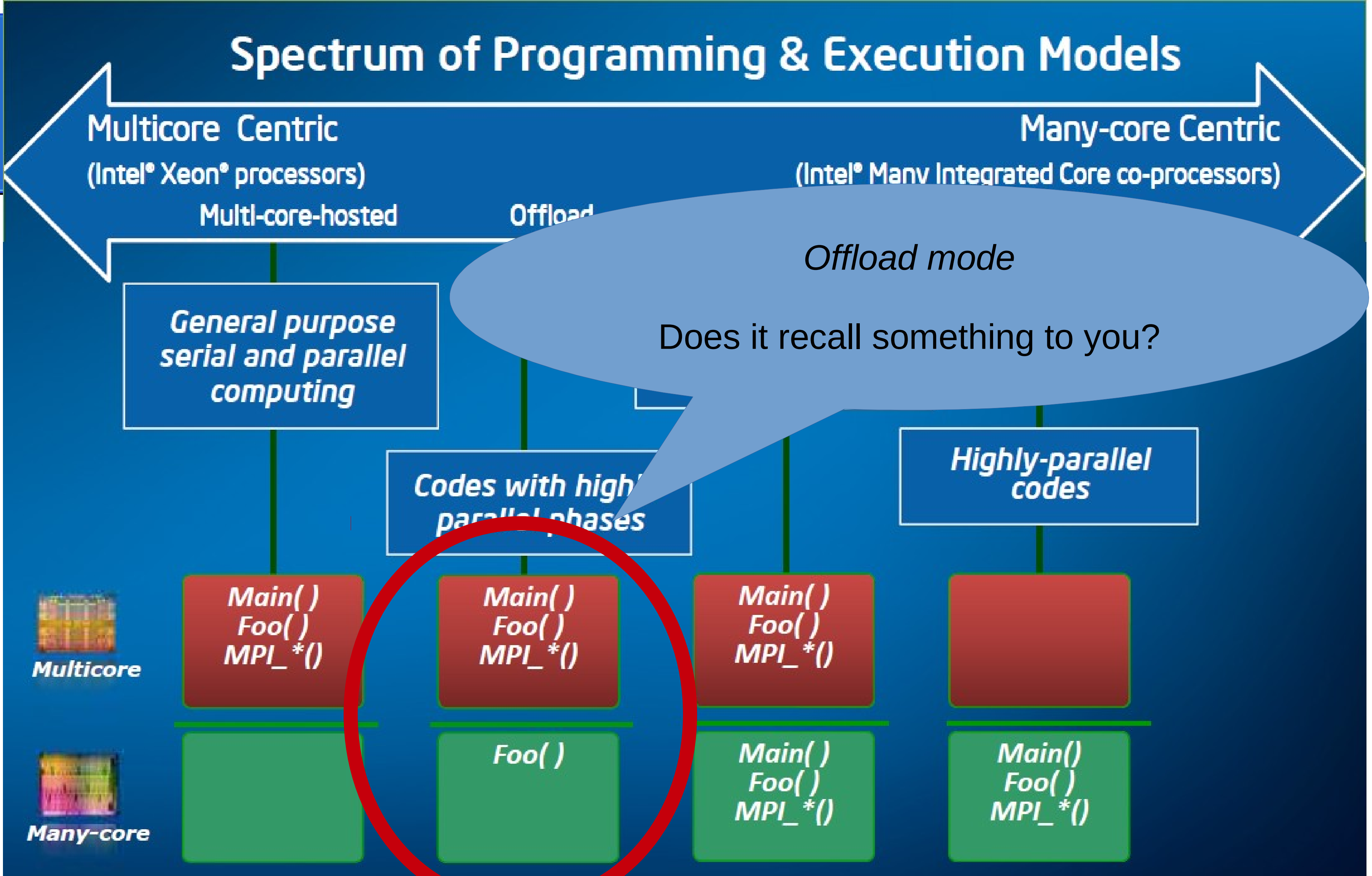
Introduction

- High level overview of the Intel Xeon Phi hardware and software stack
- Intel Xeon Phi programming paradigms: offload and native
- Performance and thread parallelism
- Using MPI
- Tracing and profiling
- Conclusions


Spectrum of Programming & Execution Models
Multicore Centric
(Intel® Xeon® processors)
Many-core Centric
(Intel® Many Integrated Core co-processors)
Multi-core-hosted
Offload
Symmetric
Many-core-hosted
General purpose serial and parallel computing
Codes with highly-parallel phases
Codes with balanced needs
Highly-parallel codes
Multicore
Many-core
Main( )
Foo( )
MPI_*()
Main( )
Foo( )
MPI_*()
Main( )
Foo( )
MPI_*()
Foo( )
Main( )
Foo( )
MPI_*()
Main()
Foo( )
MPI_*()

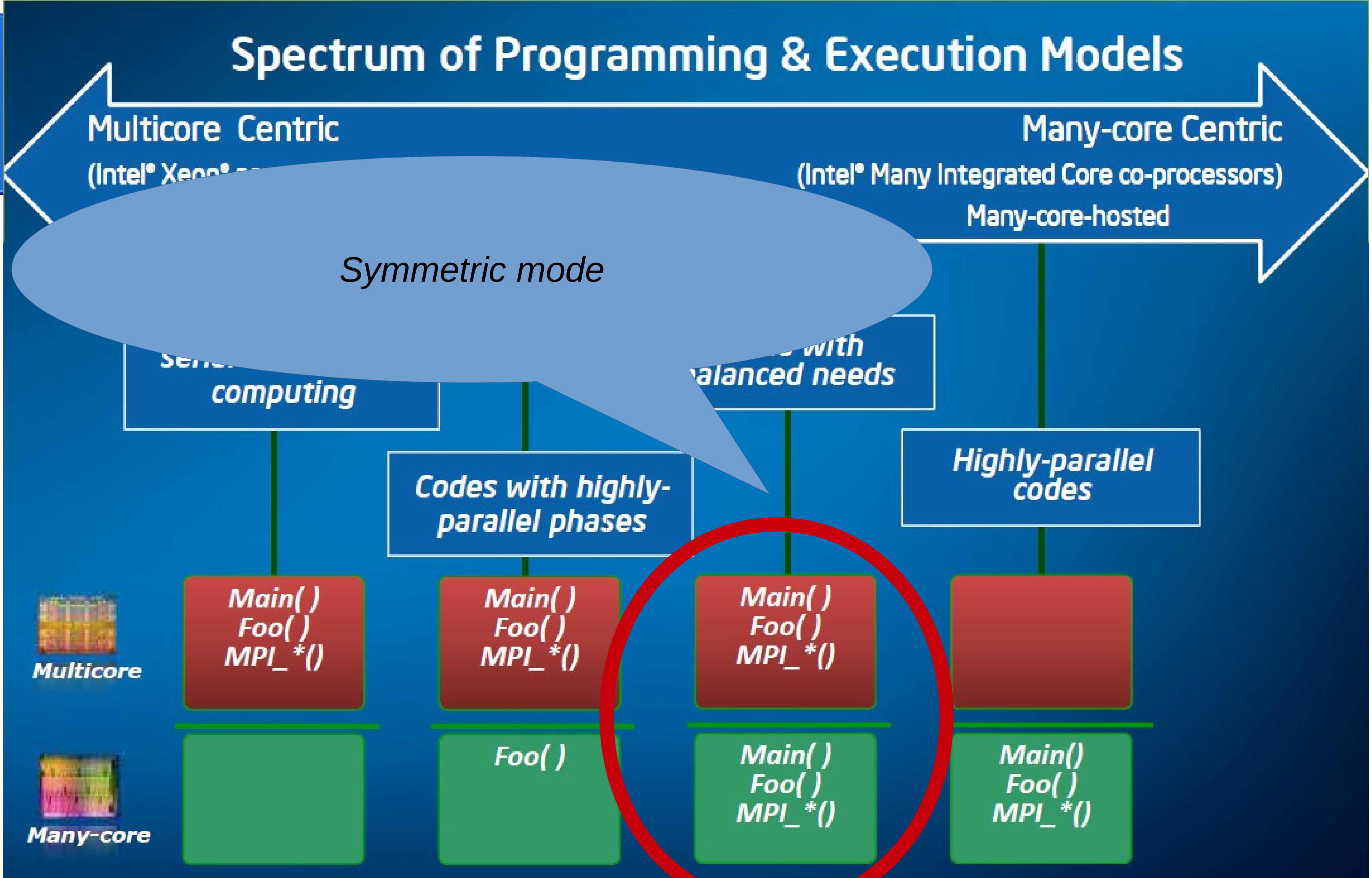


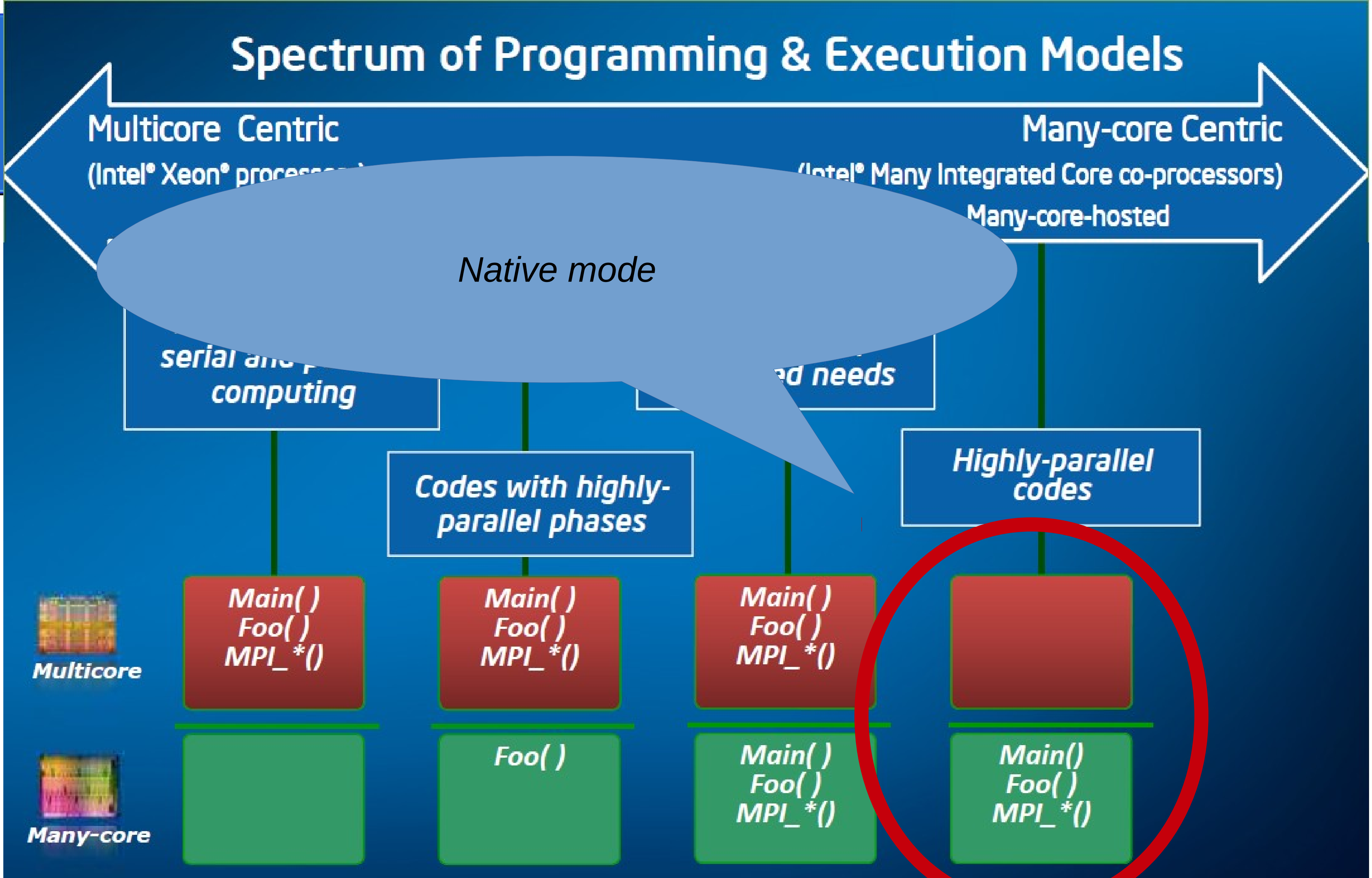
Spectrum of Programming & Execution Models
Multicore Centric
(Intel® Xeon® processors)
Many-core Centric
(Intel® Many Integrated Core co-processors)
Multi-core-hosted
Offload
Symmetric
Many-core-hosted
computing
codes
Not that interesting....
Main( )
Foo( )
MPI_*()
Main( )
Foo( )
MPI_*()
Main( )
Foo( )
MPI_*()
Foo( )
Main( )
Foo( )
MPI_*()
Main()
Foo( )
MPI_*()



Offload mode

Does it recall something to you?


Spectrum of Programming & Execution Models
Multicore Centric
Many-core Centric
(Intel® Many Integrated Core co-processors)
Many-core-hosted
Symmetric mode
computing
Codes with highly-parallel phases
Highly-parallel codes
Multicore
Many-core
Main( )
Foo( )
MPI_*()
Main( )
Foo( )
MPI_*()
Main( )
Foo( )
MPI_*()
Foo( )
Main( )
Foo( )
MPI_*()
Main()
Foo( )
MPI_*()



Spectrum of Programming & Execution Models
Multicore Centric
(Intel® Xeon® processors)
Many-core Centric
(Intel® Many Integrated Core co-processors)
Many-core-hosted
Native mode
serial and p
computing
ed needs
Codes with highly-parallel phases
Highly-parallel codes
Multicore
Many-core
Main( )
Foo( )
MPI_*()
Main( )
Foo( )
MPI_*()
Foo( )
Main( )
Foo( )
MPI_*()
Main( )
Foo( )
MPI_*()
Main()
Foo( )
MPI_*()

Intel Xeon Phi double nature

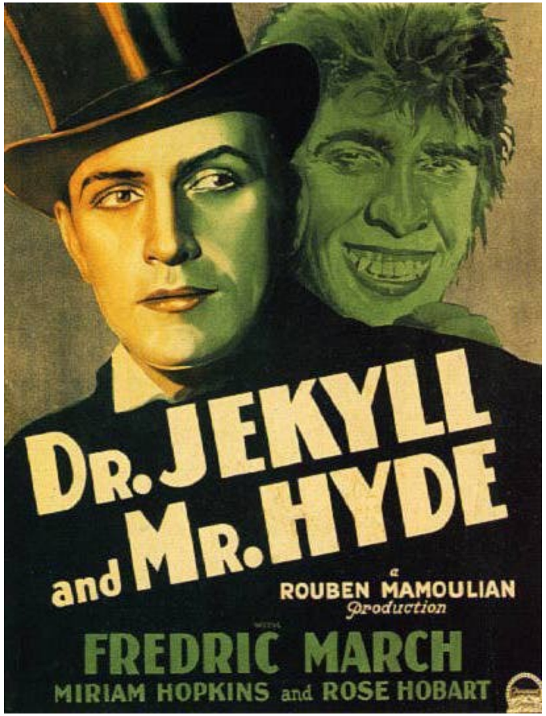
- Since it is built on a x86 architecture, the Intel Xeon Phi can behave...

Intel Xeon Phi double nature

- Since it is built on a x86 architecture, the Intel Xeon Phi can behave...

as an accelerator,
using the offload model

as an many-core platform,
using the native or symmetric model



Intel Xeon Phi as an accelerator

- The host can offload on the Xeon Phi the computation of hotspots or highly parallel kernels
- Also libraries can be offloaded (for example MKL)
- Advantages:
  - More memory available
  - Better file access
  - Host can better manage serial part of the code
  - Better use of resources

Intel Xeon Phi as a many core node

- The Intel Xeon Phi can behave as co-processor aside the the Xeon cpu, or alone as a single stand-alone node
- Advantages:
  - Simpler model (no directives)
  - Easier to port
  - Good kernel test
- Use only:
  - Not serial
  - Modest memory footprint
  - Complex code
  - No singular hotspots

Intel Xeon Phi as a many core node

- The Intel Manycore Software Stack (MPSS) provides a striped version of Linux on the coprocessor
- Intel MPSS also provides a virtual FS on the Xeon Phi
  - You can mount on the Xeon Phi the host FS using NFS
- The architecture is not exactly the same of the host
  - cross compiling is needed to build executables for the MIC architecture:

```
icc -O3 -g -mmic nativeMIC myNativeProgram.o
```

Using the offload with Intel Xeon Phi

- Intel provides a set of directives (Intel LEO: Language Extensions for Offload) in order to manage explicitly the offload.
- These directives implemented in the Intel Composer compile objects for both the host and the coprocessor and manage the data transfer between them

```
#pragma offload target(mic) inout(A:length(2000))        C/C++
!DIR$ OFFLOAD TARGET(MIC) INOUT(A: LENGTH(2000))         Fortran
```

Offload programming model

Variable and function definitions

C/C++

```
__attribute__ ((target(mic)))
```

Fortran

```
!dir$ attributes offload:mic :: <function/var name>
```

It compiles (allocates) variables on both the host and device

For entire files or large blocks of code (C/C++ only)

```
#pragma offload_attribute (push, target(mic))
#pragma offload_attribute (pop)
```

Offload programming model

Since host and device don't have physical or virtual shared memory, variable must be copied in an explicit or in an implicit way.

Implicit copy is assumed for
- scalar variables
- static arrays

Explicit copy must be managed by the programmer using clauses defined in the LEO

Offload programming model

Programmer clauses for explicit copy:
`in, out, inout, nocopy`

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))
```

Offload programming model

C/C++

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

Fortran

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

Offload programming model

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
```

Offload programming model

Memory allocation

- CPU is managed as usual
- on coprocessor is defined by in,out and inout clauses

Input/Output pointers

- by default on coprocessor “new” allocation is performed for each pointer
- by default de-allocation is performed after offload region
- defaults can be modified with alloc_if and free_if qualifiers

Offload programming model

Using memory qualifiers

free_if(0)
free_if(.false.) retain target memory

alloc_if(0)
alloc_if(.false.) reuse data in subsequent offload

alloc_if(1)
alloc_if(.true.) allocate new memory

free_if(1)
free_if(.true.) deallocate memory

Offload programming model

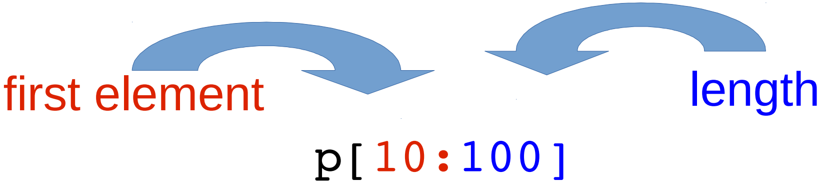
```
#define ALLOC alloc_if(1)
#define FREE free_if(1)
#define RETAIN free_if(0)
#define REUSE alloc_if(0)

#allocate the memory but don't de-allocate
#pragma offload target(mic:0) in(a:length(8)) ALLOC RETAIN)
...

#don't allocate or deallocate the memory
#pragma offload target(mic:0) in(a:length(8)) REUSE RETAIN)

#don't allocate the memory but de-allocate
#pragma offload target(mic:0) in(a:length(8)) REUSE FREE)
```

Offload programming model

Partial offload of arrays

```
int *p;
#pragma offload ... in (p[10:100] : alloc(p(5:1000))
{...}
```

It allocates 1000 elements on coprocessor; first usable element has index 5, last has index 1004; only 100 elements are tranferred,
starting from index 10.



Offload programming model

Copy from a variable to another one

It permits to copy data from the host to a different array allocated on the device

```
integer :: p(1000), p1(2000)
integer :: rank1(1000), rank2(10,100)

!dir$ offload ... (p(1:500) : into (p1(501:1000)))
```

Offload programming model

optional, if defined, it must be immediately followed by a openmp directive

Using OpenMP in an offload region:

C/C++

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

Fortran

```
!dir$ omp offload target (mic)
!$omp parallel do
do i=1,n
    A(i)=B(i)*C+D
end do
!$omp end parallel
```

Offload programming model

Asynchronous computation

By default, offload forces the host to wait for completion

- Asynchronous offload starts the offload and continues on the next statement just after the offload region
- Use the signal clause to synchronize with a offload_wait statement

Offload programming model

Example

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

Offload programming model

Reporting

Use OFFLOAD_REPORT with a verbosity from 1 to 3.
OFFLOAD_REPORT=1 only provides timing

Conditional offload

Only offload if it is worth

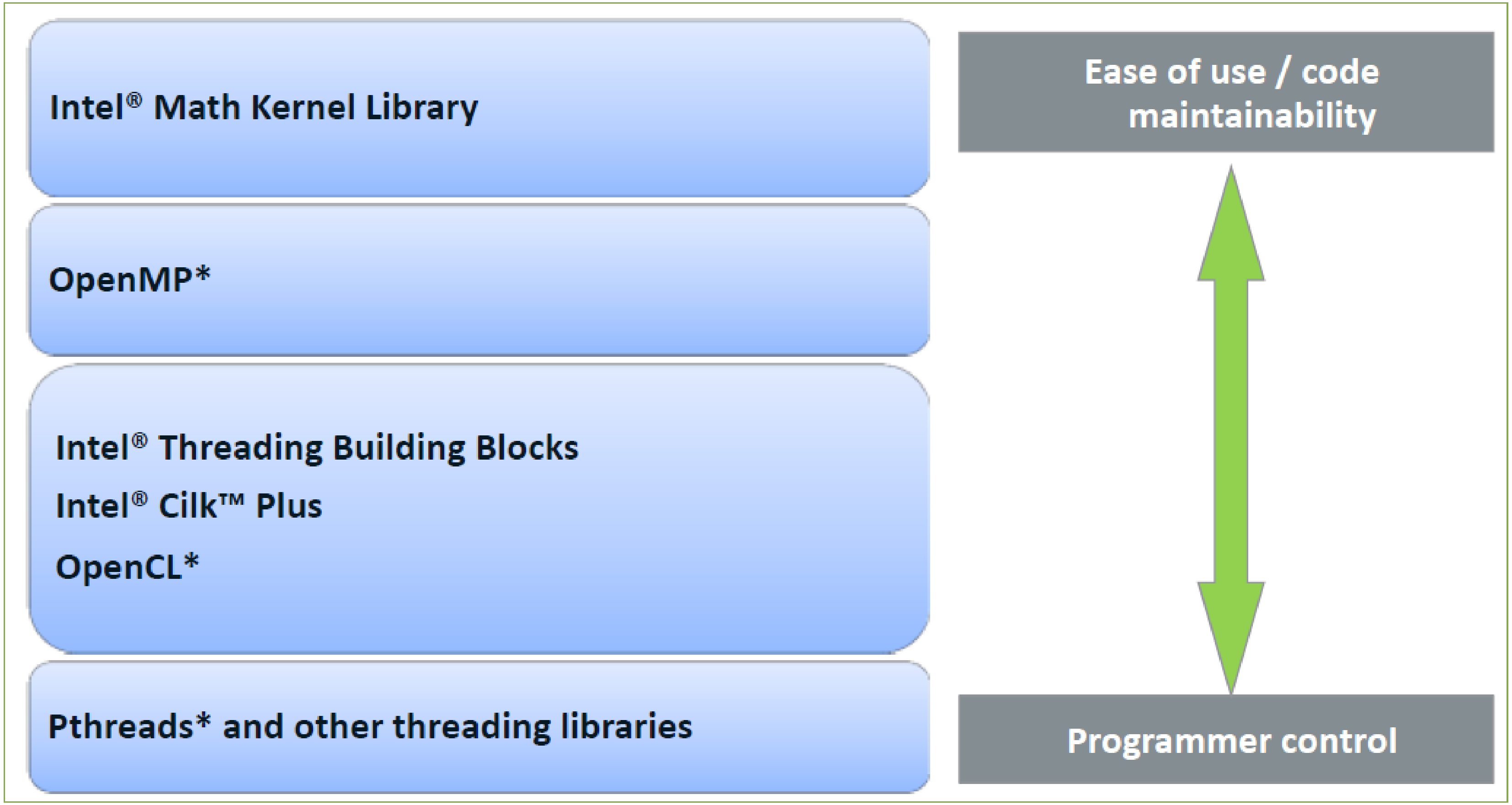
```
#pragma offload target (mic) in (b:length(size)) \
                      out (a:length(size) \
                      if(size>100)
```

	C/C++ Syntax
Offload pragma	#pragma offload <clauses> <statement> Allow next statement to execute on coprocessor or host CPU
Variable/function offload properties	__attribute__((target(mic))) Compile function for, or allocate variable on, both host CPU and coprocessor
Entire blocks of data/code defs	#pragma offload_attribute(push, target(mic)) #pragma offload_attribute(pop) Mark entire files or large blocks of code to compile for both
	Fortran Syntax
Offload directive	!dir$ omp offload <clauses> <statement> Execute OpenMP* parallel block on coprocessor
	!dir$ offload <clauses> <statement> Execute next statement or function on coproc.
Variable/function offload properties	!dir$ attributes offload:<mic> :: <ret-name> OR <var1,var2,…> Compile function or variable for CPU and coprocessor
Entire code blocks	!dir$ offload begin <clauses> !dir$ end offload

Clauses	Syntax	Semantics
Multiple coprocessors	`target(mic[:unit] )`	Select specific coprocessors
Conditional offload	`if (condition) / manadatory`	Select coprocessor or host compute
Inputs	`in(var-list modifiers`$_{opt}$`)`	Copy from host to coprocessor
Outputs	`out(var-list modifiers`$_{opt}$`)`	Copy from coprocessor to host
Inputs & outputs	`inout(var-list modifiers`$_{opt}$`)`	Copy host to coprocessor and back when offload completes
Non-copied data	`nocopy(var-list modifiers`$_{opt}$`)`	Data is local to target
Modifiers		
Specify copy length	`length(N)`	Copy N elements of pointer's type
Coprocessor memory allocation	`alloc_if ( bool )`	Allocate coprocessor space on this offload (default: TRUE)
Coprocessor memory release	`free_if ( bool )`	Free coprocessor space at the end of this offload (default: TRUE)
Control target data alignment	`align ( N bytes )`	Specify minimum memory alignment on coprocessor
Array partial allocation & variable relocation	`alloc ( array-slice )` `into ( var-expr )`	Enables partial array allocation and data copy into other vars & ranges

- High level overview of the Intel Xeon Phi hardware and software stack
- Intel Xeon Phi programming paradigms: offload and native
- Performance and thread parallelism
- Using MPI
- Tracing and profiling
- Conclusions

Thread parallelism

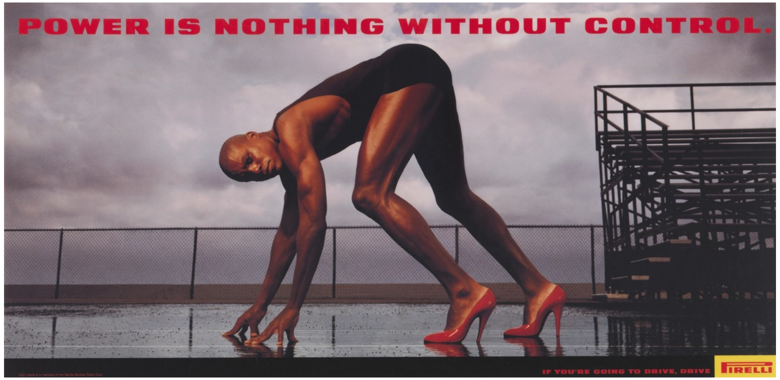


OpenMP on the Intel Xeon Phi

- Basically, it works just like for the Intel Xeon cpu
- But this is essential to obtain good performances both in offload and native modes
- There are 4 hardware threads per core
  - at least 2 x no_of_cores threads for good performances
  - for all except the most memory-bound workload
  - only sometimes 3x or 4x can be effective
  - use always the KMP_AFFINITY to control the thread binding

OpenMP on the Intel Xeon Phi

- What are the default values?
  - 1 per core on the host (if hyperthreading is disabled)
  - 4 per core on native coprocessor executions
  - 4 per (core-1) for offload executions
- It's a good rule to manually set up all the values using environment variables because...



OpenMP on the Intel Xeon Phi

- Define environment variables for the Xeon Phi:

  MIC_ENV_PREFIX=MIC

- Define Xeon Phi specific values:

  MIC_OMP_NUM_THREADS=120

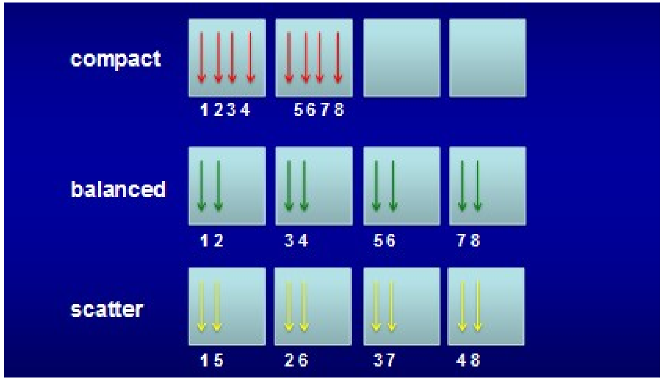
  MIC_2_OMP_NUM_THREADS=120

  MIC_3_OMP_NUM_THREADS=”240|KMP_AFFINITY=balanced”

Threads affinity

- Setting the threads affinity on the Xeon Phi is really important, because it helps to optimize the access to memory or cache
- Particularly important if all available h/w threads are not used (it prevents migration and overload)

KMP_AFFINITY = ...



Using MKL libraries

- MKL is the Intel specific math library. It covers:
  - Linear algebra (BLAS, LAPACK, ScaLAPACK)
  - Fast Fourier transform (up to 7D, FFTW interface)
  - Vector math
  - Random number generators
  - Statistics
  - Data fitting

Using MKL libraries

```
[cin0644a@terminus lib]$ pwd
/opt/intel/composer_xe_2015.0.090/mkl/lib
[cin0644a@terminus lib]$
[cin0644a@terminus lib]$
[cin0644a@terminus lib]$ ls -lart
total 20
drwxr-xr-x 10 root root 4096 Jul 25  2014 ..
drwxr-xr-x  5 root root 4096 Jul 25  2014 .
drwxr-xr-x  3 root root 4096 Sep 23  2014 intel64
drwxr-xr-x  3 root root 4096 Sep 23  2014 mic
drwxr-xr-x  3 root root 4096 Sep 23  2014 ia32
[cin0644a@terminus lib]$
```

Using MKL libraries

Three different usage models

- Automatic offload
  - no codes changes are required
  - it uses automatically host and coprocessor
  - transparent data movement and execution management
  - not available for every MKL function
- Compiler assisted offload
  - It uses the offload directives to offload MKL functions
  - It can be used together with the automatic offload
- Native execution
  - It uses the coprocessor as independent node
  - It is implemented in a different library linkable by the the native executable

MKL: Controlling the automatic offload

- Several API functions or env variables are provided to manage and control the automatic offload.

  MKL_MIC_0_WORKDIVISION=0.5

  for example, offload 50% of the computation only to the first Xeon Phi card

MKL: Compiler assisted offload

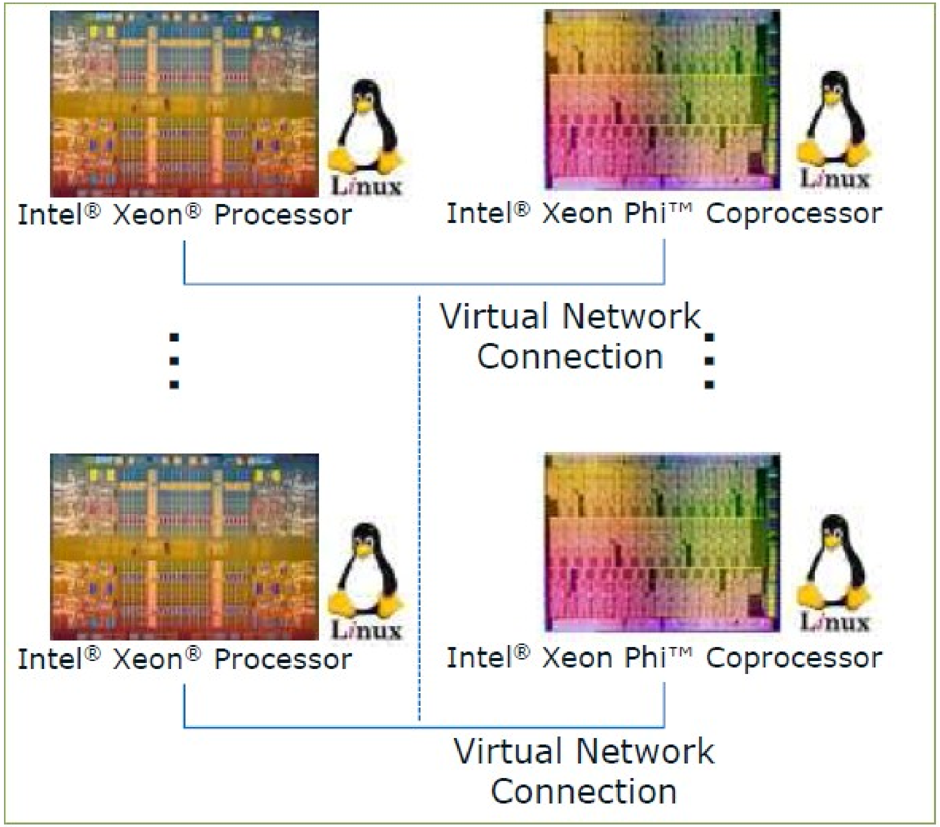
- You can use the offload directives applied to any MKL function to offload the computation to the coprocessor

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

- High level overview of the Intel Xeon Phi hardware and software stack
- Intel Xeon Phi programming paradigms: offload and native
- Performance and thread parallelism
- Using MPI
- Tracing and profiling
- Conclusions

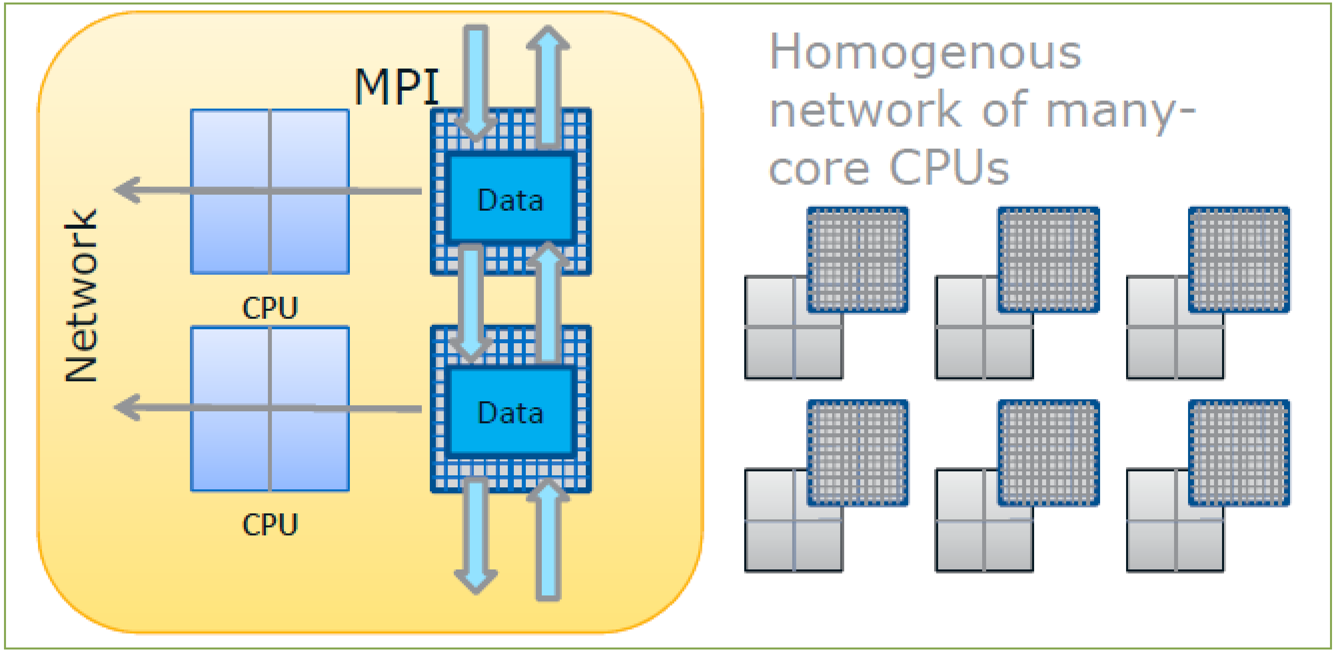
Intel Xeon Phi as a network node

- Each Xeon Phi has a network IP
- Xeon Phi can participate to a MPI communicator



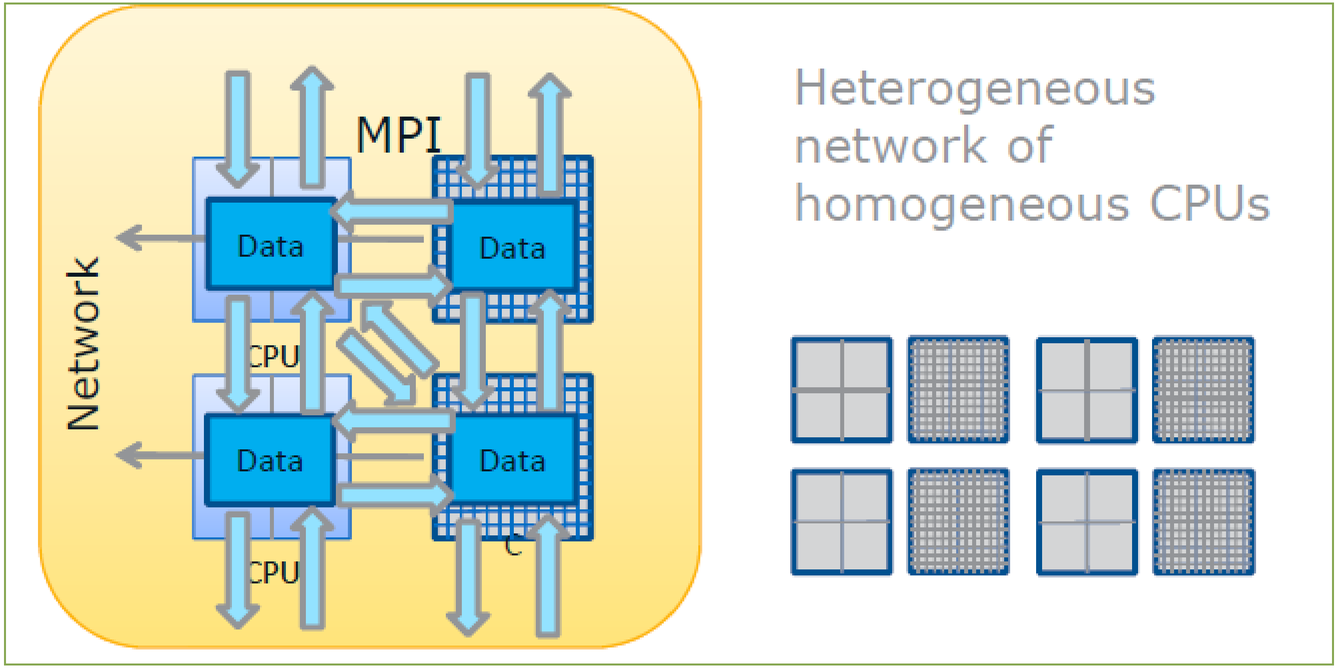
Coprocessor only programming model

- MPI ranks only on Intel Xeon Phi coprocessor



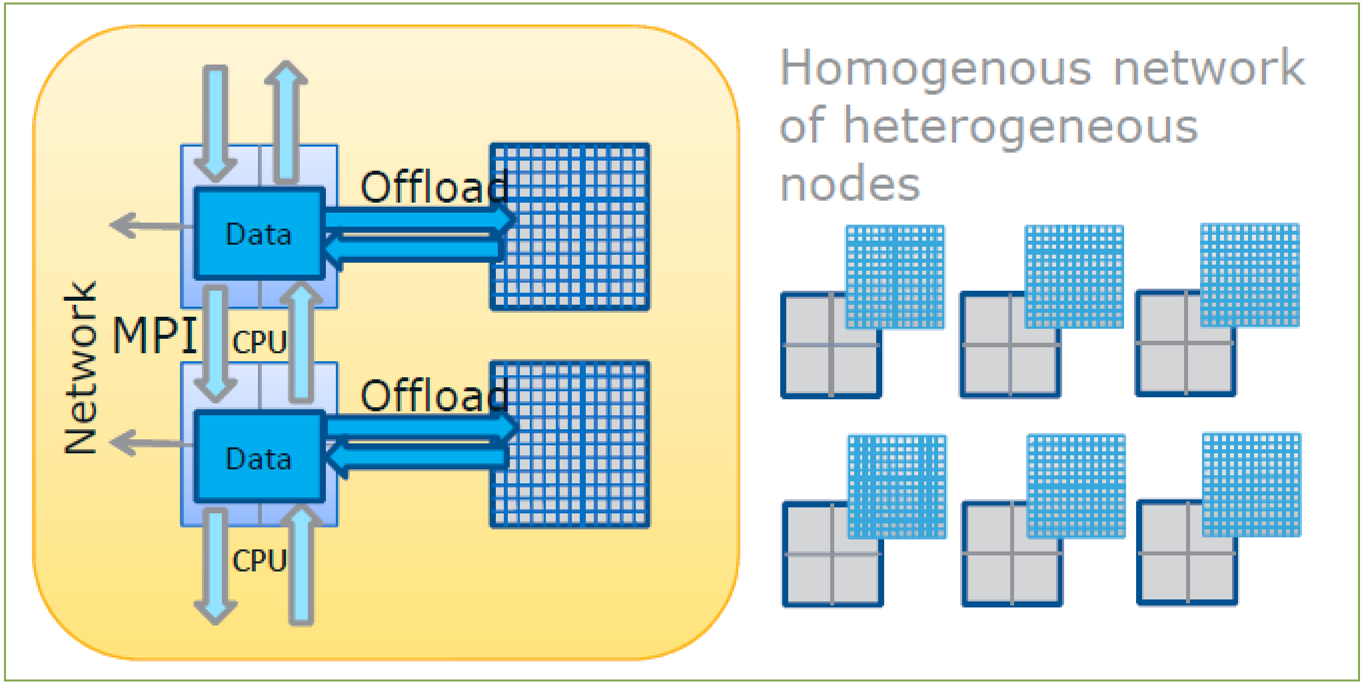
Symmetric programming model

- MPI ranks are both on Intel Xeon Phi and on host CPUs



MPI+Offload programming model

- MPI ranks are on Intel Xeon processor only. Intel Xeon Phi are used in offload mode

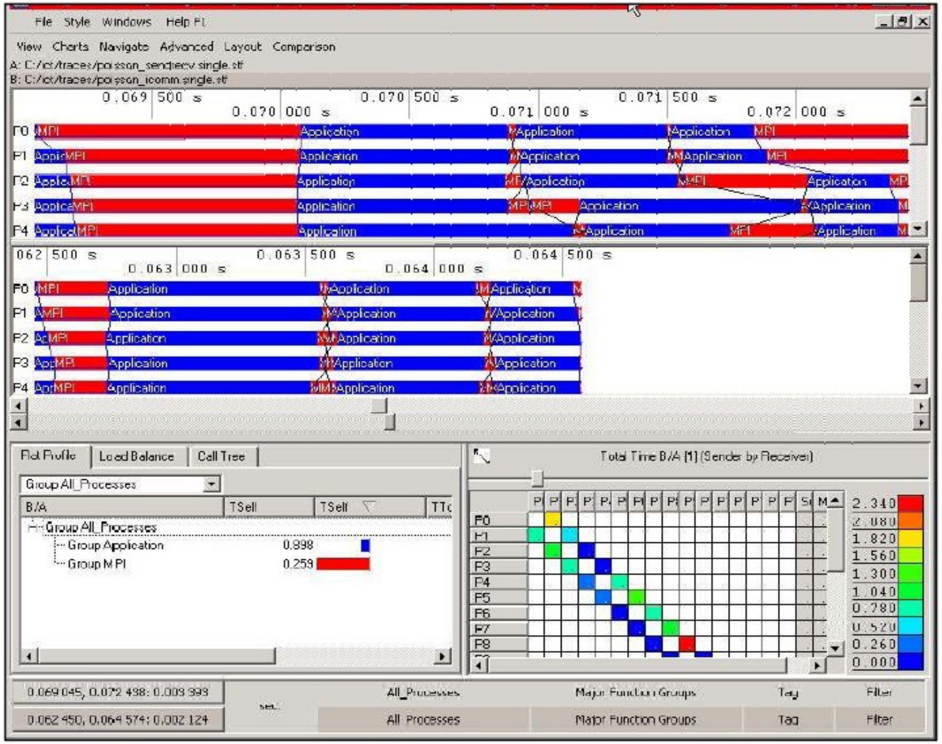


- High level overview of the Intel Xeon Phi hardware and software stack
- Intel Xeon Phi programming paradigms: offload and native
- Performance and thread parallelism
- Using MPI
- Tracing and profiling
- Conclusions

Tracing and Profiling tools

- In addition to free tools, there are severals tools from Intel designed to obtain traces and profiles of applications running on Intel Xeon Phi
- Intel Trace Analyzer and Collector (ITAC) permits to analyze the event timeline of the application, distinguishing computation and communication
- Intel Vtune Amplifier permits an in-depth profiling, also accessing hardware counters

Intel Trace Analyzer and Collector

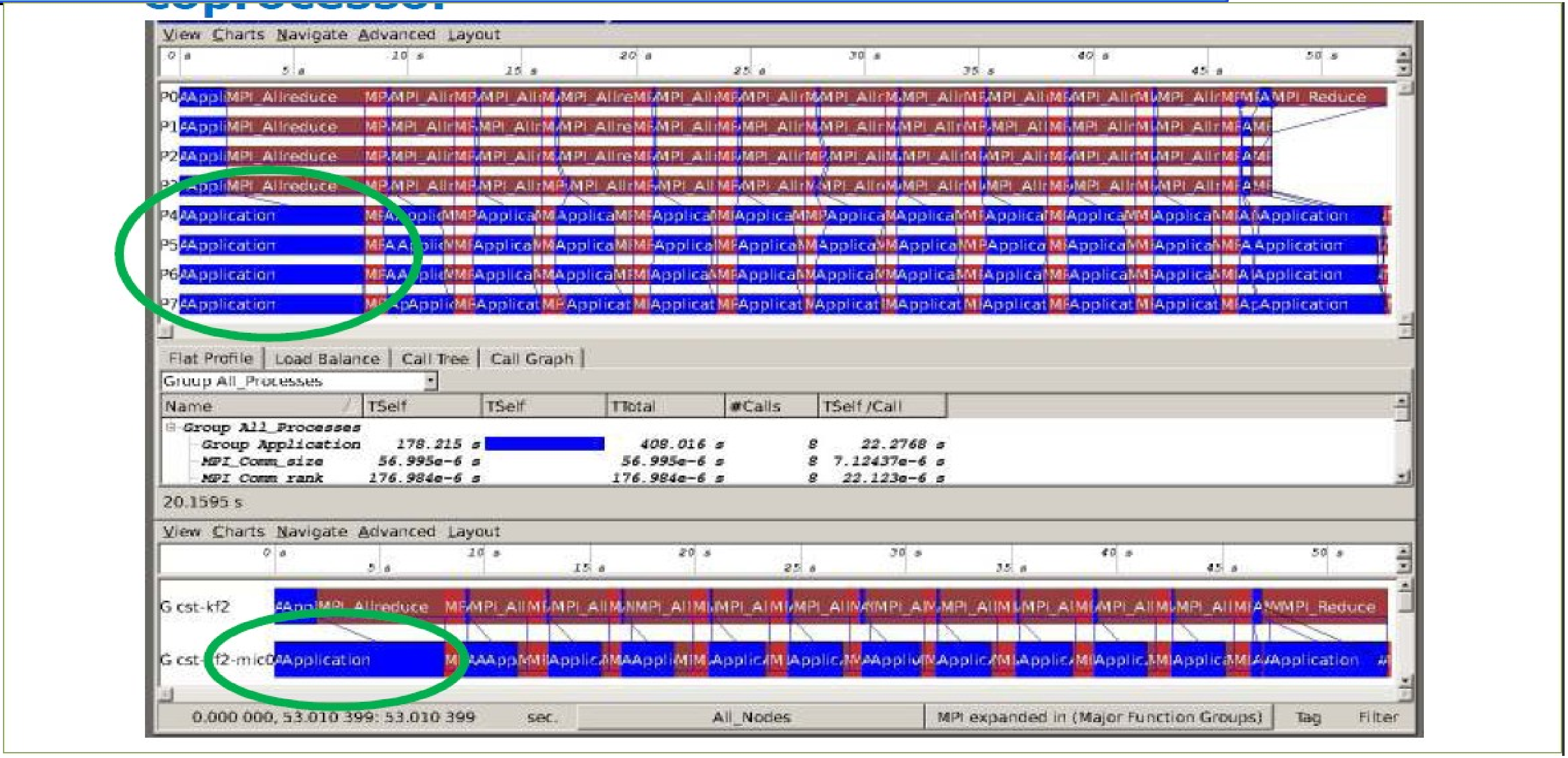


Compare the event timelines of two communication profiles

Blue = computation
Red = communication

Chart showing how the MPI processes interact

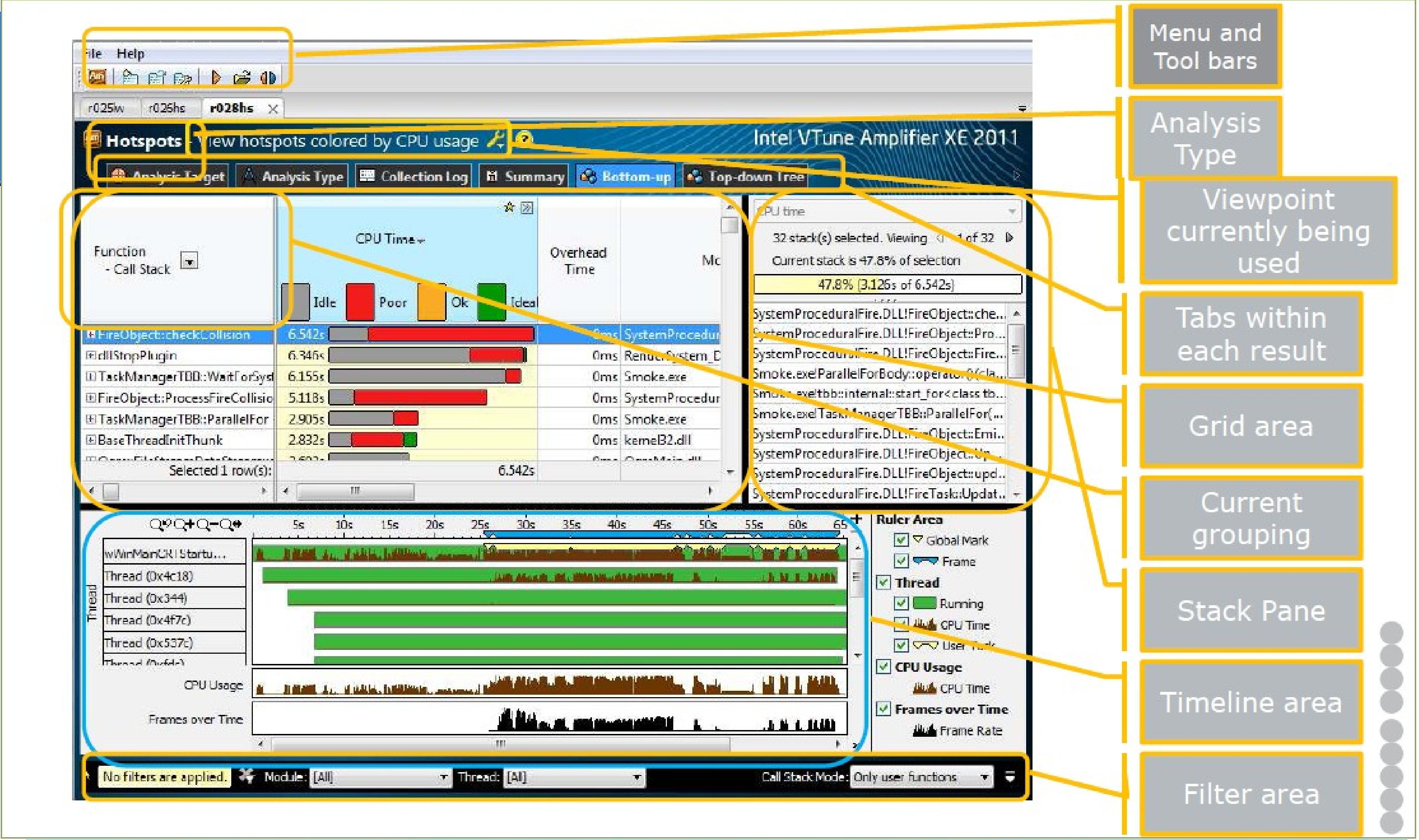
Intel Trace Analyzer and Collector on Intel Xeon Phi



Profiling with hardware data

- Vtune permits to analyze data from hardware counters
  - 2 counters in core, most thread specific
  - 4 outside the core that get no core or thread details
- Vtune can use CL or GUI.
  - Use CL to collect data
  - Use GUI to analyze data

```
amplxe-cl -collect knc_general_exploration -- mpirun -host mic0 -n 10 -env OMP_NUM_THREADS=6 -env KMP_AFFINITY=granularity=fine,balanced -env LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/intel/composerxe/lib/mic/:/opt/intel/composer_xe_2015/mkl/lib/mic/  ~/yambo-native -F ./INPUTS/02_QP_PPA -J TEST_L_29
```


Menu and Tool bars
Analysis Type
Viewpoint currently being used
Tabs within each result
Grid area
Current grouping
Stack Pane
Timeline area
Filter area
Hotspots - View hotspots colored by CPU usage
Intel VTune Amplifier XE 2011
Analysis Target
Analysis Type
Collection Log
Summary
Bottom-up
Top-down Tree
Function - Call Stack
CPU Time
Overhead Time
No filters are applied.
Module: [All]
Thread: [All]
Call Stack Mode: Only user functions

Adjust Data Grouping

- Function - Call Stack
- Module - Function - Call Stack
- Source File - Function - Call Stack
- Thread - Function - Call Stack

… (Partial list shown)

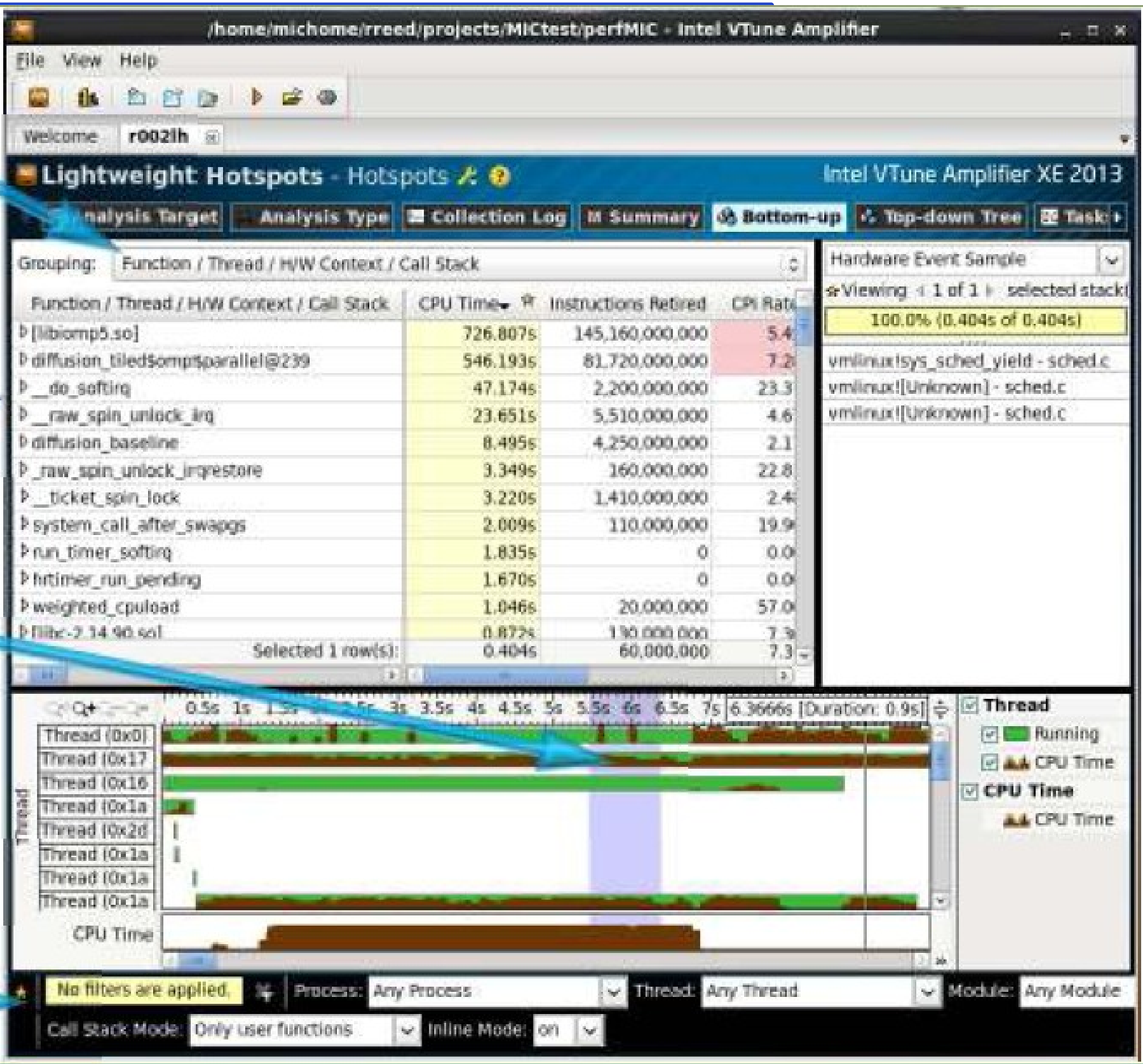
No Call Stacks Yet

Double Click Function to View Source

Filter by Timeline Selection (or by Grid Selection)

- Zoom In And Filter On Selection
- Filter In by Selection
- Remove All Filters

Filter by Module & Other Controls



- High level overview of the Intel Xeon Phi hardware and software stack
- Intel Xeon Phi programming paradigms: offload and native
- Performance and thread parallelism
- Using MPI
- Tracing and profiling
- Conclusions

Conclusions

- Intel Xeon Phi is a manycore platform that can be used both as coprocessor and as an accelerator
- Intel development environment is available:
  - Compiler
  - IntelMPI
  - Performance libraries: MKL
  - Profiling tools (ITAC, VTUNE)
- Standard techniques are available: MPI+OpenMP
- Offload permits to use Xeon Phi as an accelerator
- Three different usage models: offload, native, symmetric

Resources

- https://software.intel.com/mic-developer

Resources - books

- J. Jeffers, J. Reinders. Intel Xeon Phi Coprocessor High-Performance programming
- J. Jeffers, J. Reinders, High Performance Parallelism Pearls
- R. Rahman, Intel Xeon Phi Coprocessor Architecture and Tools, Apress (FREE)