Manycore Architectures

Paweł Czarnul

Dept. of Computer Architecture
Faculty of Electronics, Telecommunications and Informatics
Gdansk University of Technology

November 3, 2015

Plan of the Lecture

Focus on environments supporting multi- and many- core processing

- Passing criteria, grade criteria
  - minimum 50% out of lab
  - minimum 50% out of written exam
  - the exam and the lab each contribute 50% to the final grade
- landscape of today's HPC processing including multicore CPUs, GPUs, Intel Xeon Phi

Types of HPC systems today

Shared memory systems:

1. multicore CPUs
2. accelerators and coprocessors such as:
   - Intel Xeon Phi
   - GPU

Distributed memory systems:

1. computational clusters – many nodes intrerconnected with a fast network such as Infiniband
   - many CPUs per node
   - accelerators/coprocessors in nodes → see the TOP500 list
2. volunteer computing – volunteers join projects with their own resources

Multi- vs Many- core systems

1. multicore – CPUs
2. manycore – Intel Xeon Phi
3. clusters:
   - CPU only
   - CPU + accelerator/coprocessor

Modern CPUs, accelerators and coprocessors

- multicore CPUs such as:
  Intel® Xeon® Processor E7-8890 v3 (45M Cache, 2,50 GHz, 3,3 GHz turbo), LLC 45 MB, 18 cores (36 threads), TDP 165 W
- accelerators and coprocessors such as:
  - Intel Xeon Phi
    Intel® Xeon Phi™ Coprocessor 7120A (16GB, 1,238 GHz) 61 cores (244 threads), 30,5 MB L2, TDP 300 W
  - GPU
    Tesla K80 (2x Kepler GK210), 24 GB GDDR5, 4992 CUDA cores

Use of multi- and many- core compute devices in TOP500 clusters

From the June 2015 edition of the TOP500 list (`http://www.top500.org`)

1. Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P
   location: China
   3 120 000 cores, 33,8 PFlop/s, 17,8 MW
2. Titan – Cray XK7, Opteron 6274 16C 2,2 GHz, Cray Gemini interconnect, NVIDIA K20x
   location: U.S.A.
   560 640 cores, 17,59 PFlop/s, 8,2 MW
3. Sequoia - BlueGene/Q, Power BQC 16C 1,60 GHz
   location: U.S.A.
   1 572 864 cores, 17,1 PFlop/s, 7,9 MW

Intel Xeon Phi - characteristics

Facts relevant to Intel Xeon Phi:

- a coprocessor that can be installed in a computer in a PCI-E slot
  $\rightarrow$ communication through PCI-E which can be a bottleneck
- memory between 6 and 16 GBs now
- around 60 physical cores (from 57 to 61 depending on the model)
- up to 244 threads to be run efficiently (4 per 1 physical core)
- computational power of around 1-1.2 TFlop/s depending on the model
- TDP up to 300 W
- each Intel Xeon Phi card runs Linux

Current models of Intel Xeon Phi

Intel® Many Integrated Core Architecture (Intel® MIC Architecture) combines many cores into a single chip and allows using standard programming tools and methods[1].
Intel Xeon Phi is the name of products made available by Intel.
There are 7000, 5000 and 3000 series coprocessors available – see details on Intel webpages.
Product codes:

1. Knights Ferry (prototype) 2010
2. Knights Corner 2011
3. Knights Landing – see another slide.

[1] http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html

Intel Xeon Phi Cores

- based on Intel Pentium core (with 64-bit support)
- in-order execution
- supports four hardware threads per core (use at least two threads per core in order to utilize the hardware)
- a Xeon Phi core is less powerful than a Xeon CPU core
- clocked around 1GHz
- includes L1 cache (see the architecture diagram)
- vectorized execution is possible using 512 bits
- two pipelines – two instructions per cycle can be executed (V- and U- pipelines)

Intel Xeon Phi Architecture

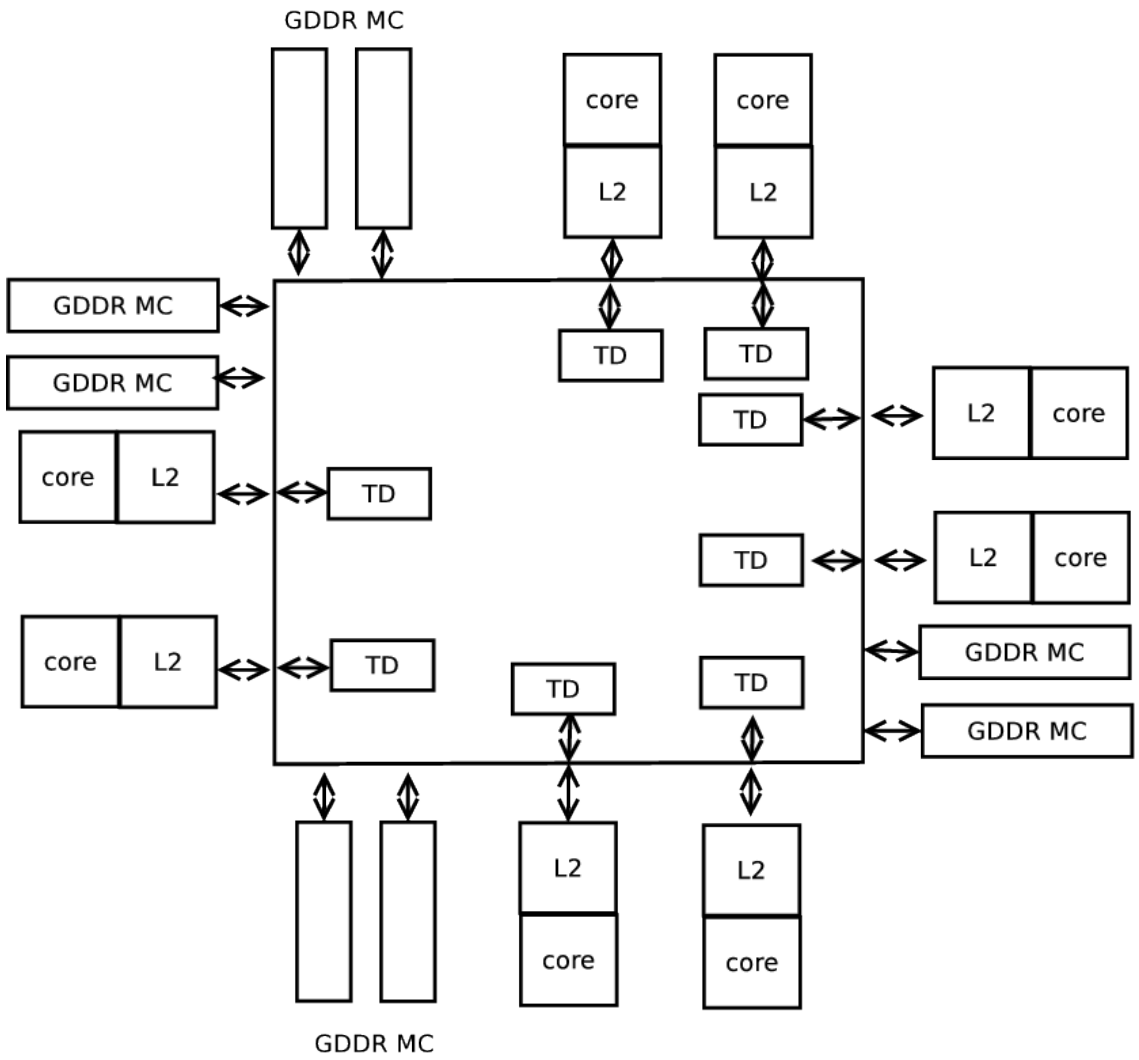

Figure: Xeon Phi Architecture

Core components within Intel Xeon Phi

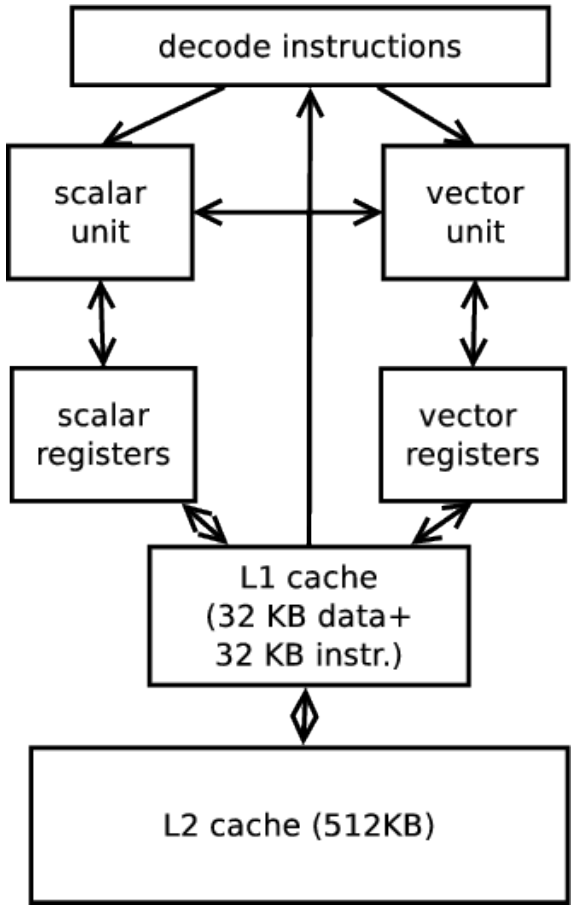

Figure: Xeon Phi Core

Vector Processing in Intel Xeon Phi

- Xeon Phi supports 512 bit wide SIMD processing which results in:
  - 16 single precision elements
  - 8 double precision elements
- there are vector registers available
- memory operations (vector): load/store, gather/scatter
- supports Fused Multiply-Add operations

Architecture of a node with Intel Xeon Phi cards

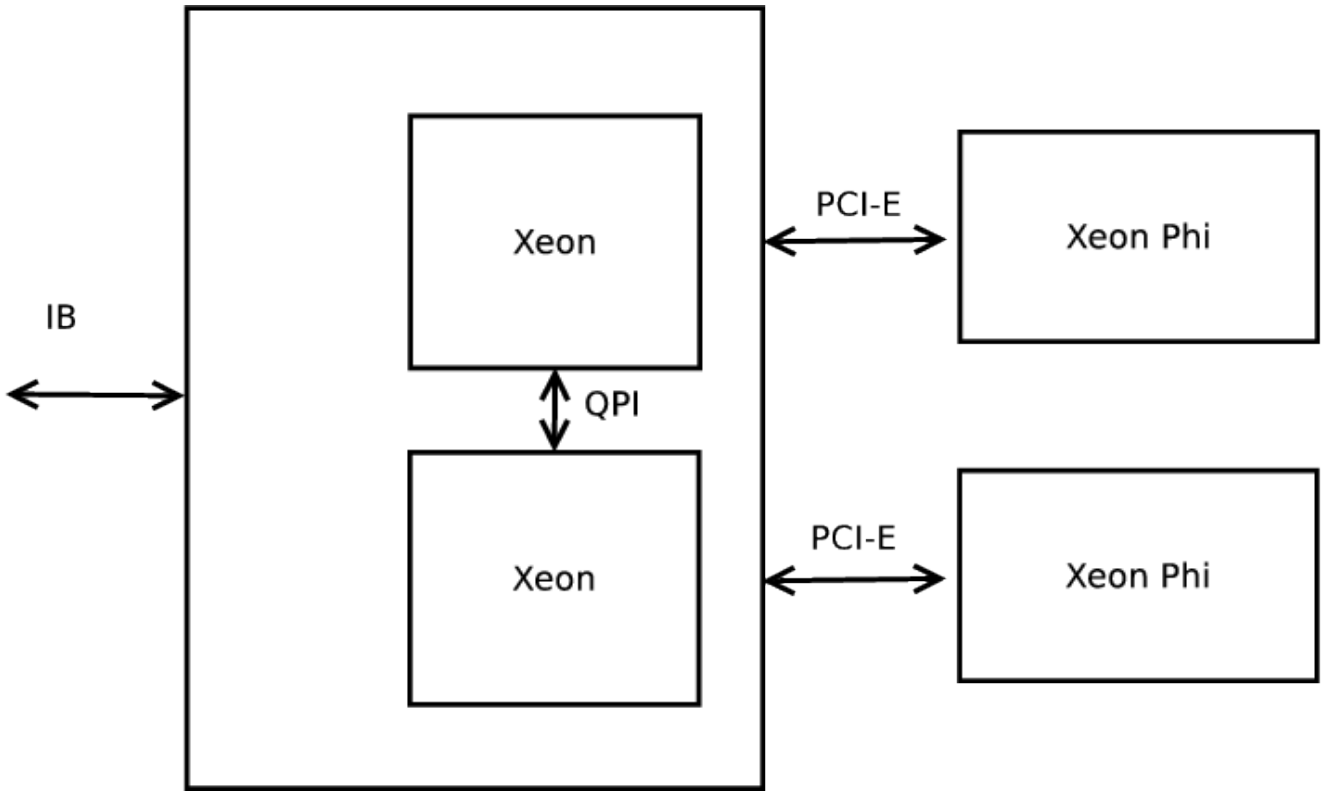

Figure: Xeon Phi Core

Programming Intel Xeon Phi – APIs

Intel Xeon Phi is programmer's friendly as it supports the following popular APIs for parallel programming:

1. OpenMP
2. OpenCL
3. MPI

and more such as Intel Threading Building Blocks or Cilk.
Xeon Phi requires a high level of parallelism for efficient execution. Per thread performance will be worse than on an Xeon CPU.

Programming Intel Xeon Phi – Programming Models

There are three typical programming modes to be used with Intel Xeon Phi:

1. native – log in to the Xeon Phi and run an application there utilizing its cores
2. offload – run an application on a CPU (can be more than one with multiple cores) and offload computations on an Xeon Phi (such as with OpenMP directives)
3. symmetric – run an application on cores of a CPU (Xeon) and Xeon Phi – such as with MPI

Future of Intel Xeon Phi

Knights Landing

1. more cores such as 72
2. up to 284 GB DDR4 memory
3. 2D mesh interconnect
4. 3 TFlop/s
5. out-of-order
6. AVX-512
7. 3 versions: KNL Coprocessor (PCI-E), host processor, host processor with integrated fabric

Preparation of the environment from user's point of view

```
pczarnul@wolf:~$ ssh apl12.eti.pg.gda.pl
pczarnul@apl12.eti.pg.gda.pl's password:
[pczarnul@apl12 ~]$ ifconfig
eth0      Link encap:Ethernet  HWaddr **************
          inet addr:153.19.50.44  Bcast:153.19.50.63  Mask:255.

...

mic0      Link encap:Ethernet  HWaddr **************
          inet addr:172.31.1.254  Bcast:172.31.1.255  Mask:255.

...

mic1      Link encap:Ethernet  HWaddr **************
          inet addr:172.31.2.254  Bcast:172.31.2.255  Mask:255.
          inet6 addr: fe80::4e79:baff:fe34:1879/64 Scope:Link
```

Preparation of the environment from user's point of view

Compiling and running an application on the **host** – multicore CPU(s)

```
[pczarnul@apl12 ~]$ source /opt/intel/composer_xe_2013_sp1.3.
174/bin/compilervars.sh intel64
[pczarnul@apl12 ~]$ icc
icc: command line error: no files specified; for help type
"icc -help"
[pczarnul@apl12 ~]$ icc test-openmp.c -fopenmp
[pczarnul@apl12 ~]$ ./a.out
```

Preparation of the environment from user's point of view

Compiling and running an application on the **host** but with cross compiling for the Xeon Phi architecture – **-mmic** flag

```
pczarnul@wolf:~$ ssh student01@apl12.eti.pg.gda.pl
student01@apl12.eti.pg.gda.pl's password:
[student01@apl12 ~]$
[student01@apl12 ~]$ source /opt/intel/composer_xe_2013_sp1.3.
174/bin/compilervars.sh intel64
[student01@apl12 ~]$ icc -openmp -mmic -O3 example.c -o exampl
[student01@apl12 ~]$ scp /opt/intel/composer_xe_2013_sp1.3.174/
compiler/lib/mic/libiomp5.so mic0:~
libiomp5.so                                   100% 1116KB
1.1MB/s   00:00
```

Preparation of the environment from user's point of view

Compiling and running an application on the **host** but with cross compiling for the Xeon Phi architecture – **-mmic** flag

```
[student01@apl12 ~]$ scp example mic0:~
example                                       100%  110KB
110.3KB/s   00:00
[student01@apl12 ~]$ ssh mic0
[student01@apl12-mic0 ~]$ ls
example      libiomp5.so
[student01@apl12-mic0 ~]$
[student01@apl12-mic0 ~]$ ./example
./example: error while loading shared libraries: libiomp5.so:
cannot open shared object file: No such file or directory
[student01@apl12-mic0 ~]$ export LD_LIBRARY_PATH=$LD_LIBRARY_PA
[student01@apl12-mic0 ~]$ ./example
```

Hints for optimization

- use alignment to 64 bytes for the Xeon Phi
- make sure that vectorization is used if possible for the code
  - check if vectorization was successful for loops by using -vecreport3 during compilation
- by default hardware prefetching from the L2 cache is enabled
- software prefetching is used with -O2 and higher flags during compilation

Useful #pragma

- `#pragma ivdep` – information given to the compiler to ignore any pointer dependencies i.e. when the programmer is certain non-overlapping areas will be referenced
- `#pragma vector aligned` – informs the compiler that array data within a loop is aligned
- `#pragma simd` or `#pragma vector always` – force the compiler to vectorize the loop

Prefetching on the MIC architecture

_mm_prefetch()) or pragmas (#pragma prefetch var).

Offloading computations to the MIC

Offloading is possible by using proper Intel's pragma. For example,

```
#pragma offload target(mic) in(tab0,tab1:length(k*k)) \
inout(tab2:length(k*k))
{

}
```

asks to offload computations in a block and specifies data transfer as follows:

- transfer of arrays tab0 and tab1 to the mic
- transfer of array tab2 from the mic to the host
- note that since tab0, tab1 and tab2 are pointers then length (the number of elements) is specified for each transfer

Offloading

- In case there are more coprocessors on a node (2+) it is possible to specify the target using num
  `#pragma offload target(mic:num)`
- by default, offloading is enable. It can be turned off by using `-no-offload` at compile time
- reporting for offloads can be turned on using `-opt-report-phase:offload`
- it is possible to call a function from within an offloaded block. In such a case one needs to declare such a function with `__attribute__((target(mic)))`

Offloading

Mark a larger part of code to be available on the Xeon Phi

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

Offloading

It is possible to offload data without the offload block

```
#pragma offload_transfer target(mic) in(tab0,tab1:length(k*k))
```

Clauses for definition of data transfer

Data transfer when offloading can generally be defined as follows:
Syntax: clause(variables modifiers)
Clauses can be as follows:

- in – to device
- out – from device
- inout – to and from device
- nocopy – on device

Allowed modifiers

The following modifiers are allowed for the aforementioned clauses:

- length(k) – copy k elements (type of the pointer
- alloc_if (bool) – allocate space on the coprocessor (true is default)
- free_if (bool) – release space on the coprocessor after the offload (true is default)
- align (k bytes) – specifies alignment on the coprocessor

Persistent data on Xeon Phi

Using proper modifiers in successive invocations it is possible to reuse the data allocated previously – it saves the PCIe bandwidth and increases performance:

1. allocate space for an array but do not release it after the offload block has finished `#pragma offload target(mic) in (tab0:length(k) alloc_if(1) free_if(0))`
2. reuse the space for an array from a previous offload but do not release it after the offload block has finished `#pragma offload target(mic) in (tab0:length(k) alloc_if(0) free_if(0))`

Working in parallel on the host and on Xeon Phi

It is obviously possible to start computations on an Xeon Phi in an offload mode and also start computations on the host.
One of possible ways of doing it would be to start computations in parallel e.g. in parallel section blocks - directive:
`#pragma omp sections` followed by two or more `#pragma omp section` blocks

Affinity settings

What is affinity?
Thread affinity defines how threads of an application are mapped to particular cores of a computing device.
Firstly, the number of threads in OpenMP can be specified in a few ways:

1. `OMP_NUM_THREADS` environment variable
2. `void omp_set_num_threads(int num_threads);` function
3. `num_threads(x)` clause in omp blocks

Then, there are some environment variables that instruct how to assign threads to cores.

Affinity settings

`KMP_AFFINITY` allows to specify:

1. affinity
   - compact – a following thread will be places as close as possible to a previous thread (thus filling in physical cores on the Xeon Phi)
   - scatter – threads will be scattered (spread) across available computing cores (physical cores first on the Xeon Phi)
   - balanced (Xeon Phi specific, not available on the host) – it is a mix of scatter and compact i.e. groups of threads will be placed next to each other first within a granularity context, then on following cores
2. granularity
   - fine – each thread is bound to a single HW thread
   - core – groups of four threads are bound to a core (floating)

Affinity settings

verbose can be added so that some output can be seen – how threads are assigned
e.g. `export KMP_AFFINITY=verbose,balanced`
Newer versions also accept setting `KMP_PLACE_THREADS` for setting details using:

- C – denotes cores
- T – denotes threads
- O – denotes offset starting from core 0, the default value here is 0O

Example:
10C,4T,10O – 10 cores used with 4 threads per core starting from core 10

Affinity settings – two environments

If there are two environment to be used (host + mic or host + more mics) then it is possible to set these values for the environments (for offload). Specifically, it can be done as follows:

```
export MIC_ENV_PREFIX=PHI
export PHI_KMP_AFFINITY=balanced
export PHI_KMP_PLACE_THREADS=60c,2t
export PHI_OMP_NUM_THREADS=120
```

If there are 61 cores in the Xeon Phi system, such assignment leaves 1 core for running OS threads

Setting the number of threads on the host programmatically

Setting the number of threads on the host programmatically is possible using
`omp_set_num_threads_target (TARGET_TYPE target_type, int target_number, int num_threads)`
Reference available at: https://software.intel.com/en-us/node/524665

Offloading – asynchronous transfers

It is possible to perform computations on an Xeon Phi and on the host at the same time:

```
#pragma offload target (mic:0) signal(&signal_handle)
{
computations_on_xeon_phi();
}
computations_on_cpu();
#pragma offload_wait target (mic:0) wait(&signal_handle)
```

Conditional offloading

It is possible to add a condition to an offload clause such as
`if (data_size > 350)`

Performance of various constructs

In some cases it is possible to implement a certain scenario in various ways. This refers especially to synchronizing among threads.
In many applications successive iterations (a big loop) are performed in which processes perform computations and need to synchronize at the end of an iteration.
There are at least two ways of implementing it in OpenMP:

1. Running a big loop outside with iterations each of which starts a parallel region in which threads perform computations. Note that the region synchronizes threads at the end of the iteration by default
2. Starting a parallel region first and running a loop inside. A barrier can then be used for synchronization of threads

Which is faster?

False sharing

Each core of the system has a cache. Cache needs to be coherent in the system (among the cores). So-called cache lines reflect the main memory. The cache line on an Intel Xeon Phi is 64 bytes.
When a memory is updated, cache needs to be coherent within the system. This may mean a considerable overhead, especially if many cores are involved – such as in an Intel Xeon Phi system.
This generally means that false sharing may occur if threads modify memory locations close to each other – even if there is no real overlap.
Ring bus on Intel Xeon Phi can affect performance if false sharing occurs.

Ways of dealing with false sharing

If there is a shared array that is constantly updated by threads in a parallel region there is a risk of false sharing. This can happen several times due to many updates in a loop executed several times.
One way of dealing with this is to use private memory in each of the threads. Specifically, perform local operations within threads and then update local variables.
Only synchronize when merging results at the end of computations
However, try to use memory sparingly – it is a scarce resource given the number of cores and consequently the number of threads. One could replicate arrays in order to decrease the risk of false sharing but it would result in too much memory usage – simple variables in each thread may be enough!

Ways of dealing with false sharing

To summarize, note the following possibilities while dealing with false sharing:

1. padding a structure to the end of cache line
2. use local copies of data
3. make sure that few cache lines are shared among threads – this also means that if you have a code with many iterations updating successive data (in arrays) one can assign many iterations per thread – such as with static and specification of a particular chunk size!

Inspect the code for arrays, dynamically allocated data used by many threads.

Level of parallelization

One should think at which level of an algorithm (especially if nested loops are involved) parallelization should be performed (#pragma omp ....)
Often, parallelization is performed at entries to outer loops (note a previous slide on synchronization using a parallel block and barrier).
In some cases, it is possible to enable nested parallelism for parallel execution of inner parallel regions. This will involve some overhead though.

Typical errors in OpenMP

Study:
Michael Suss and Claudia Leopold. Common Mistakes in OpenMP and How To Avoid Them. A Collection of Best Practices
Reference:
`http://wwwi10.lrr.in.tum.de/~gerndt/home/Teaching/EfficientHPC`

Options for parallelized loops – scheduling of iterations

In the `#pragma omp parallel for` it is possible to specify how iterations will be scheduled among threads. Specifically, there is the schedule(kind,chunksize) clause that can be added. kind can have the following values:

- `static` – round robin distribution of chunksize iterations
- `dynamic` – allocation to threads that fetch and process iterations of chunksize
- `guided` – size of a chunk decreases in time; chunks are proportional to the number of iterations remaining / the number of threads; not smaller than chunksize
- `runtime` – policy determined at runtime by the `OMP_SCHEDULE` environment variable

Performance considerations

Study the following in your application:

1. parallelization – is the implementation highly parallel?
2. cache utilization
3. false sharing
4. memory utilization per thread
5. computations vs synchronization overhead
6. synchronization ways and possible implementations

Synchronization

Note differences between:

- `#pragma omp critical` – specifies a block which will be executed by one thread at a time – a critical section; there can be two types of critical sections:
  - named
  - unnamed
- `#pragma omp atomic` – specifies a memory update instruction (such as incrementation) which will be executed atomically – faster then `critical`

Application use cases

Parallel computations of similarities of a large number of multidimensional vectors
Describe and compare various versions of code.
Note the following issues:

- how parallelization is exposed – at what level
- memory usage – where is the input data and where results are stored – location in memory
- potential elimination of false sharing
- synchronization ways among threads
- computations/communication ratio

Application use cases

Parallelization of divide-and-conquer applications

- A dynamic implementation
- How to control the number of active threads
- Performance and limitations
- Thread affinities vs performance

P. Czarnul. Parallelization of divide-and-conquer applications on Intel Xeon Phi with an OpenMP based framework. ISAT 2015. Springer. Advances in Intelligent Systems and Computing, in press.
`http://isat.pwr.wroc.pl/`

MPI – Fabric

It is possible to define fabric for both:

- intranode
- internode

The syntax is as follows:
`export I_MPI_FABRICS=<intranode fabric>:<internode fabric>`
with the following options:

- shm Shared-memory
- tcp TCP/IP-capable network fabric
- ofa OFA-capable network fabric e.g. InfiniBand
- dapl DAPL network fabric e.g. InfiniBand, Dolphin

MPI – preparation of the environment

Root should execute the following before compilation and running:

```
service mpss start
scp /opt/intel/impi/4.1.2.040/mic/bin/mpiexec.hydra
root@mic0:/bin
scp /opt/intel/impi/4.1.2.040/mic/bin/pmi_proxy
root@mic0:/bin
scp /opt/icc_xe2013/composer_xe_2013.5.192/compiler/
lib/mic/libiomp5.so mic0:/lib64
scp /opt/icc_xe2013/composer_xe_2013.5.192/compiler/
lib/mic/libimf.so mic0:/lib64
scp /opt/icc_xe2013/composer_xe_2013.5.192/compiler/
lib/mic/libsvml.so mic0:/lib64
scp /opt/icc_xe2013/composer_xe_2013.5.192/compiler/
lib/mic/libirng.so mic0:/lib64
scp /opt/icc_xe2013/composer_xe_2013.5.192/compiler/
lib/mic/libintlc.so mic0:/lib64
```

MPI – preparation of the environment

Root should execute the following before compilation and running:

```
scp /opt/intel/impi/4.1.2.040/mic/lib/libmpigf.so.4
mic0:/lib64
scp /opt/intel/impi/4.1.2.040/mic/lib/libmpi.so.4
mic0:/lib64
scp /opt/intel/impi/4.1.2.040/mic/lib/libmpi_mt.so.4
mic0:/lib64
```

MPI – preparation of the environment

A programmer should prepare the environment including:

```
source /opt/icc_xe2013/bin/compilervars.sh intel64
source /opt/intel/impi/4.1.2/intel64/bin/mpivars.sh
export I_MPI_MIC=on
export I_MPI_FABRICS=shm:tcp
```

MPI – communication

Improving MPI communication:
https://software.intel.com/en-us/blogs/2014/12/09/improving-mpi-communication-between-the-intel-xeon-host-and-intel-xeon-phi
Some info regarding OFED:
https://www.openfabrics.org/index.php/openfabrics-software.html

Compilation (note that paths may be different):

```
source /opt/icc_xe2013/bin/compilervars.sh intel64
source /opt/intel/impi/4.1.2/intel64/bin/mpivars.sh
mpiicc -O3 -o program3.mic program3.c -mmic
scp program3.mic mic0:~
```

MPI – Running

Running an MPI application on a mic from the host e.g. using a script such as:

```
or pcount in 1 2 4 8 16 32 64 128 192 228 456 912
do
echo Running simulation on $pcount processes
time mpirun -host mic0 -n $pcount ~/program3.mic
done
```

MPI – compilation for the host and a mic

In case an application is to be run on both the host and a mic:

```
source /opt/icc_xe2013/bin/compilervars.sh intel64
source /opt/intel/impi/4.1.2/intel64/bin/mpivars.sh
mpiicc -O3 -o program3.mic program3.c -mmic
mpiicc -O3 -o program3.XEON program3.c
scp program3.mic mic0:~
cp program3.XEON ~
```

MPI – running the host and a mic

In case an application is to be run on both the host and a mic:

```
export I_MPI_MIC=on
export I_MPI_FABRICS=shm:tcp
time mpirun -host mic0 -n <proc_count_on_mic>
~/program3.mic : -host localhost -n
<proc_count_on_CPU> ~/program3.XEON
```

Additional optimization concerns

- note subexpressions in loops, this may also be optimized by the compiler
- note how many times you refer to e.g. functions or expressions in macro:

```
#define min(a, b) ( (a) < (b) ? (a) : (b) )
x = min(func(x),func(y));
```

  How many times? How can you optimize this?
- use proper data types – do you need single precision or double precision computations?

Additional optimization concerns

- note that there are several versions of mathematical operations depending on the data type:

```
#include <math.h>

double cos(double x);
float cosf(float x);
long double cosl(long double x);
```

Precision

Flags in the Intel compiler:
`-fp-model`
See details at:
`https://software.intel.com/en-us/node/525262` in particular the following modes:

- precise
- fast[=1|2]
- strict
- source

Also:
`https://wiki.scinet.utoronto.ca/wiki/images/f/f2/FP_Consistency`

Additional optimizations

Static and dynamic scheduling of loops can have impact on performance e.g. when an internal loop (across space) of a big time loop is to be parallelized.
dynamic would yield better results in such cases

Additional optimizations

- in case of two loops (nested) note the order – which is an outer and which is the inner loop (indexes)
- maximize data locality
- in some cases the compiler can handle this
- tiling of loops can improve locality

Benchmarking an application using Intel VTune

How to run the tool?

```
ssh -X student01@apl12.eti.pg.gda.pl
```

```
source /opt/intel/vtune_amplifier_xe/amplxe-vars.sh
```

Now it is possible to launch an application on an Xeon Phi. Let us prepare a script for launching the application first:

```
[student01@apl12 ~]$ ssh mic0
```

```
[student01@apl12-mic0 ~]$ cat r
export  LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
export  KMP_AFFINITY=balanced
export OMP_NUM_THREADS=240
./s21   10000   10000   100
```

Benchmarking an application using Intel VTune

Now it is possible to run tests from the host in the following way:

```
[student01@apl12 ~]$ amplxe-gui
```

Then perform the following steps:

1. Start a new project – select a name
2. Application – ssh
3. Application parameters – mic0 ./r
4. Select OK

Benchmarking an application using Intel VTune

1. Select New analysis
2. In the tree on the left side select Knights Corner Platfom
3. Select Hotspots or General Exploration, or Bandwidth
4. Click Start
5. Wait for results and analyze

Benchmarking an application using Intel VTune

Important counters according to Intel documentation:

- Cycles Per Instruction (CPI), the best CPI per core is 1 with one thread per core, 0.5 with 2, 3 and 4 threads per core
  - check if CPI per thread is > 4
  - check if CPI per core > 1
  - try to reduce latency
- L1 hit rate – check if $< 95\%$
- vectorization – check if < 8 (double precision), < 16 (single precision)

Benchmarking an application using Intel VTune

Important counters according to Intel documentation:

- bandwidth – check if $<$ 80GB/s – peak should be 140GB/s
- Estimated Latency Impact – check if >145
- L1 TLB miss ratio – check if $>$ 1%
- L2 TLB miss ratio – check if $>$ .1%

General optimization notes

Use the following to improve performance on Xeon Phi:

- prefer SoA to AoS
  https://software.intel.com/sites/default/files/article/392271/aos-to-soa-optimizations-using-iterative-closest-point-mini-app.pdf
- if beneficial use software prefetching
- use tiling for cache use
- align data
- use array notation (to pointer)

Bibliography I

[1] Avoiding and identifying false sharing among threads.
Intel Developer Zone, November 2011.
https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads.

[2] Michaela Barth, Mikko Byckling, Nevena Ilieva, Sami Saarinen, Michael Schliephake, and Volker Weinberg.
Best practice guide – intel xeon phi, February 2014.
Partnership for Advanced Computing in Europe
http://www.prace-ri.eu/best-practice-guide-intel-xeon-phi-html/.

[3] Shannon Cepeda.
Optimization and performance tuning for intel® xeon phi™ coprocessors, part 2: Understanding and using hardware events, November 2012.

Bibliography II

[4] Pawel Czarnul.
Parallel programming on intel xeon phi. compute intensive parallel applications using openmp and mpi, 2014.
http://enauczanie.pg.gda.pl/moodle/pluginfile.php/58545/mod_resou
manual-PCzarnul-v1.pdf.

[5] James Jeffers and James Reinders.
Intel Xeon Phi Coprocessor High Performance Programming.
Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

Bibliography III

[6] David Mackay.
Optimization and performance tuning for intel® xeon phi™ coprocessors - part 1: Optimization essentials, November 2012.
https://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-1-optimization.

[7] Hans Pabst.
Debugging and profiling on intel® xeon phi™, July 2013.
PRACE Summer School, CINECA, http://www.training.prace-ri.eu/uploads/tx_pracetmo/intel_mic_tools.pdf.

[8] Robert Reed.
Performance tuning for intel® xeon phi™ coprocessors, 2014.
https://www.rcac.purdue.edu/tutorials/phi/PerformanceTuningXeonI
Tullos.pdf.

Bibliography IV

[9] James Reinders.
An overview of programming for intel xeon processors and intel xeon phi coprocessors.
https://software.intel.com/sites/default/files/article/330164/an-overview-of-programming-for-intel-xeon-processors-and-intel-xeon-phi-coprocessors_1.pdf.

[10] Andrey Vladimirov, Ryo Asai, and Vadim Karpusenko.
Parallel Programming and Optimization with Intel Xeon Phi Coprocessors.
Colfax International, May 2015.
ISBN 978-0-9885234-0-1.