HPCS 2014

Optimization hints for Intel Xeon Phi

F. Salvadore - Cineca

Performance and parallelism

- In principle the main advantage of using Intel MIC technology with respect to other coprocessors is the simplicity of the porting
    - Programmers may compile their source codes based on common HPC languages (Fortran/ C / C++) specifying MIC as the target architecture (*native mode*)
- Is it enough to achieve good performances? By the way, why offload?
- Usually not, parallel programming is not easy
    - A general need is to **expose parallelism**

GPU vs MIC

- GPU paradigms (e.g. CUDA):
  - Despite the sometimes significant effort required to port the codes...
  - ...are designed to force the programmer to expose (or even create if needed) parallelism
- Programming Intel MIC
  - The optimization techniques are not far from those devised for the common CPUs
  - As in that case, achieving optimal performance is far from being straightforward
- What about device maturity?

CINECA

Xeon Phi very basic features

- Let us recall 3 basic features of current Intel Xeon Phi
- Peak performance originates from “many slow but vectorizable cores”

```
clock frequency x n. cores x n. lanes x 2 FMA Flops/cycle
1.091 GHz x 61 cores x 16 lanes x 2 = 2129.6 Gflops/cycle
1.091 GHz x 61 cores x 8 lanes x 2 = 1064.8 Gflops/cycle
```

- Bandwidth is (of course) limited, caches and alignment matter
- The card is not a replacement for the host processor. It is a coprocessor providing optimal power efficiency

Optimization key points

- In general terms, an application must fulfill three requirements to efficiently run on a MIC

  (1) Highly vectorizable, the cores must be able to exploit the vector units. The penalty when the code cannot be vectorized is very high

  (2) high scalability, to exploit all MIC multi-threaded cores: scalability up to 240 processors (processes/threads) running on a single MIC, and even higher running on multiple MIC

  (3) ability of hiding I/O communications with the host processors and, in general, with other hosts or coprocessors

Vectorization: auto

- In recent Intel compilers, vectorization is enabled by default
  - May be turned off by explicit options
  - The compiler must be able to detect the possibility to do that
- The essential requirement is the possibility to unroll the loop having the different iterations performed simultaneously
- Some critical conditions
  - If the loop is part of a loop nest, it must be the inner loop unless it is completely unrolled or interchange occurs (use -O3)
  - Straight-line code: no jumps or branches but masked assignment allowed
  - Countable loop: number of iterations must be known when starting (even if not at compile time)
  - No loop dependencies: iterations must be performed in parallel

Vectorization: arrays and restrict

- Writing “clean” code is a good starting point to have the code vectorized
  - Prefer array indexing instead of explicit pointer arithmetic
  - Use *restrict* keyword to tell the compiler that there is no array aliasing
- Excerpt from a real code the compiler manages to vectorize:

```
REAL * __restrict__ anspx=an+spxoff;
REAL * __restrict__ ansmx=an+smxoff;
...
for(ix=istart; ix<iend; ix++) {
    as = anspx[ix]*JpxWO[ix] + anspy[ix]*JpyWO[ix] +
         anspz[ix]*JpzWO[ix] + ansmx[ix]*JmxWO[ix] +
         ansmy[ix]*JmyWO[ix] + ansmz[ix]*JmzWO[ix] +
    ...
}
```

Vectorization: array notation

- Using array notation is a good way to guarantee the compiler that the iterations are independent
  - In Fortran this is consistent with the language array syntax

```
a(1:N) = b(1:N) + c(1:N)
```

  - In C the array notation is provided by Intel Cilk Plus

```
a[1:N] = b[1:N] + c[1:N]
```

- Beware:
  - The first value represents the lower bound for both languages
  - But the second value is the upper bound in Fortran whereas it is the length in C
  - An optional third value is the stride both in Fortran and in C
  - Multidimensional arrays supported, too

Vectorization: directives

- Another opportunity is forcing vectorization by means of directives
  - The programmer guarantees the possibility to vectorize
  - Until a few years ago, only compiler dependent directives available

```
#pragma ivdep
```

- Instructs the compiler to ignore assumed vector dependencies (proven dependencies area not ignored)

```
#pragma vector always
```

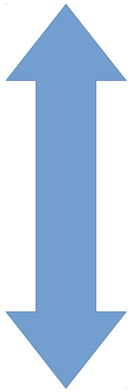
- Instructs the compiler to override any efficiency heuristic during the decision to vectorize or not

Vectorization: OpenMP 4.0 *simd*

- Intel took leadership in defining OpenMP 4.0 SIMD extensions
  - Several tuning options available

Ease of use

Thread Level Parallelism	SIMD parallelism
Auto Parallel	Auto vectorization
OpenMP threading	OpenMP 4.0 simd
Posix threads	Vectorization intrinsics



Programmer control

```
#pragma omp simd
```

- Applied to a loop

```
#pragma omp declared simd
```

- Applied to a function to enable the creation of a version that can process arguments using SIMD instructions from a single invocation from a SIMD loop

Vectorization: Phi intrinsics

- IMCI intrinsics
  - The coding become hard
  - And the code is no more portable to common CPUs

```
for(i=0; i<N; i++)
     A[i] = A[i] + B[i];
```

```
for(i=0; i<N; i+=16) {
    __mm512 Avec = mm512load_ps(A+i);
    __mm512 Bvec = mm512load_ps(B+i);
    Avec = mm512add_ps(Avec, Bvec);
    _mm512_store_ps(A+i,Avec);
}
```

- The arrays float A[N] and float B[N] are aligned on a 64-byte boundary
- Variables Avec and Bvec are 512=16 x sizeof(float) bits

Exploiting cores

- MPI and OpenMP are the most common choices
  - Up to 60 MPI processes are reasonable for a single MIC
  - And 1 MPI process per MIC may be an interesting choice
  - The optimal choice between MPI and OpenMP depends on the application
- MPI Programming models, basically three configurations
  - Co-processor only (native mode)
  - MPI+Offload
  - Symmetric

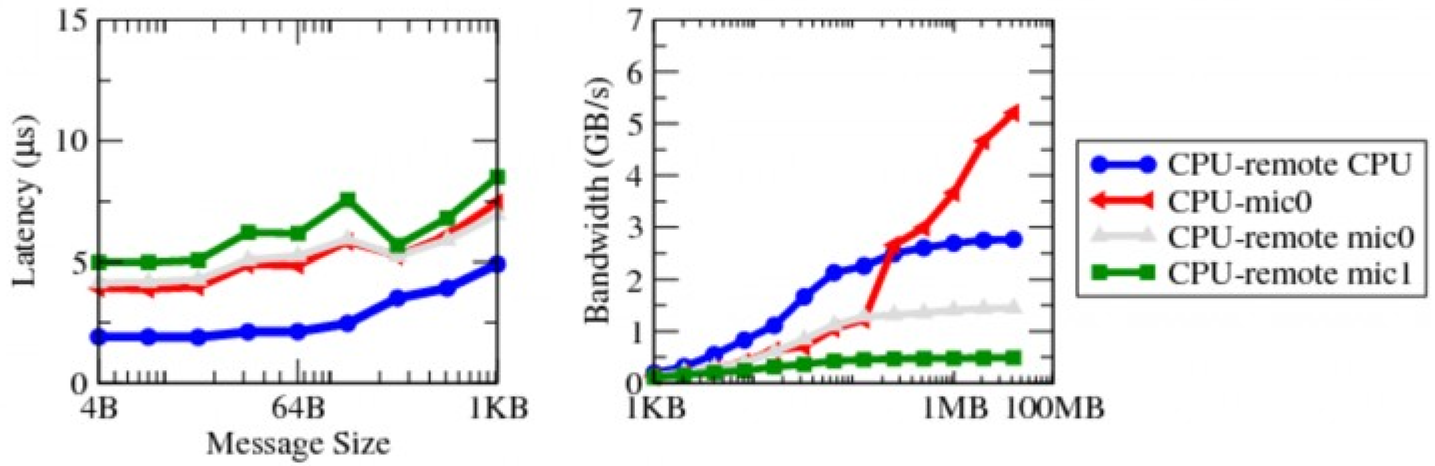
Experimenting heterogeneity

- MPI communications are heterogeneous. Performances strongly vary!
- From some tests on the Eurora cluster at Cineca

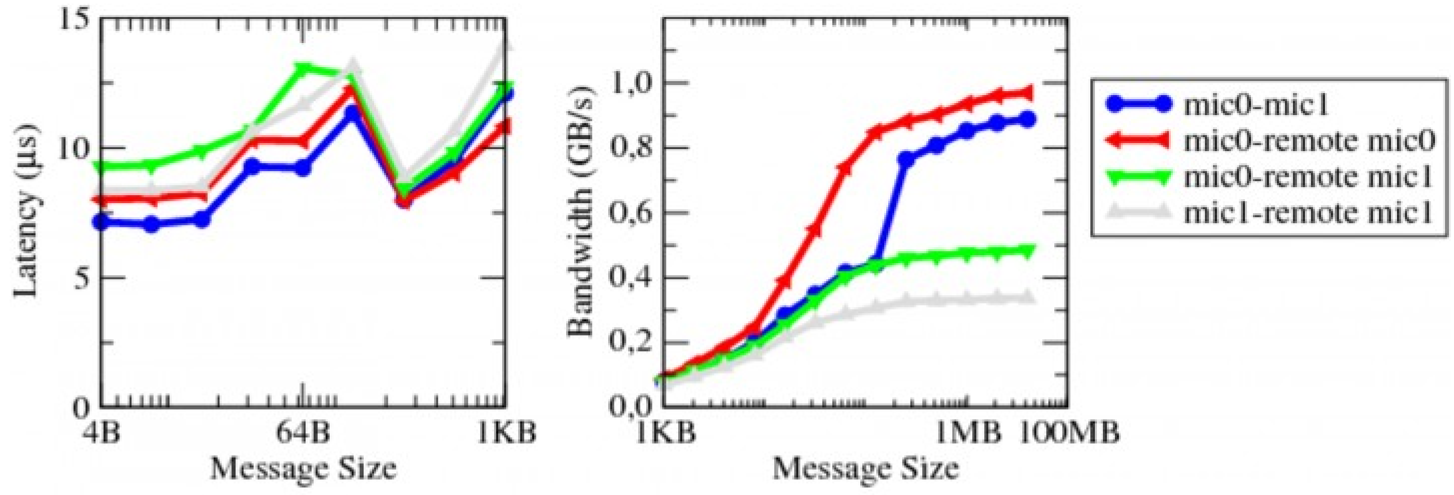
	PingPong	SendRecv
CPU-CPU same node	5-11	5-22
CPU-CPU diff node	2.9	5
MIC-MIC same node	0.9	1.8
MIC-MIC diff node	0.9	1.6
CPU-MIC same node	5.9	11
CPU-MIC diff node	1.45	1.65

HPCS 2014

Experimenting heterogeneity/2



Intel MPI Benchmark, PingPong test over the Infiniband fabric (I_MPI_FABRICS=dapl)
for communications originating in a CPU host



Intel MPI Benchmark, PingPong test over the InfiniBand fabric (I_MPI_FABRICS=dapl)
for communications originating in a MIC host

Symmetric mode: load balancing

- When running in symmetric mode, load balancing is a critical issue
  - Usual MPI decompositions assume homogeneous computing units
- Mixing MPI and OpenMP may help
  - Assign a different number of MPI processes to host and coprocessor
  - Exploit the full machine potential by means of OpenMP threads
  - E.g.
    - Host: 4 MPI ranks + 4 OpenMP threads
    - MIC: 8 MPI ranks + 30 OpenMP threads

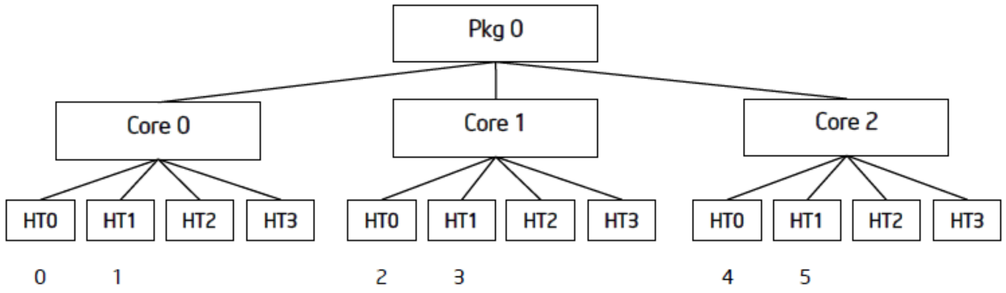
Threading models

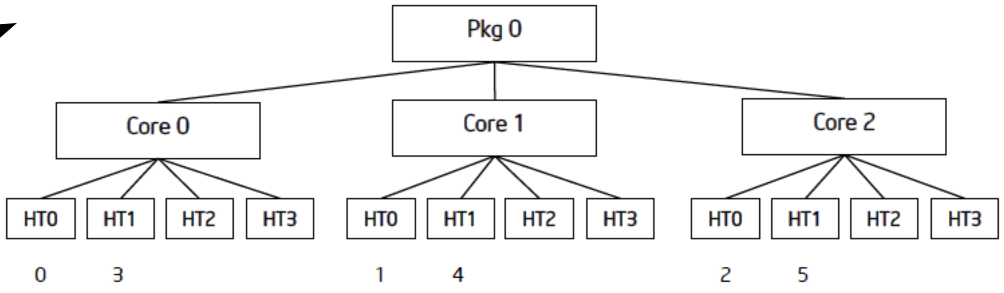
- Several threading models available
  - OpenMP
  - Fortran (2008) DO concurrent
  - Intel Cilk Plus
  - Intel Threading Building Block
  - Intel Math Kernel Library

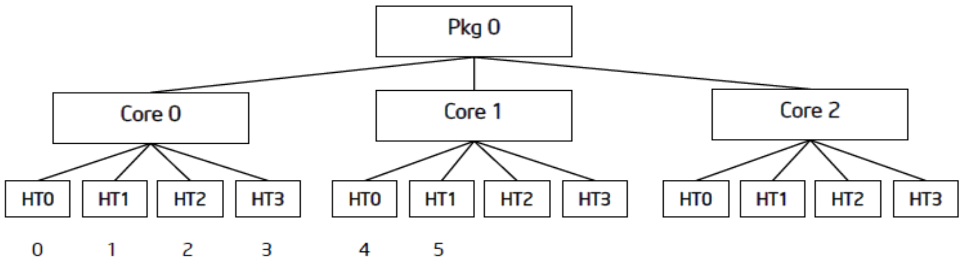
- OpenMP has clear advantages wrt portability
- In offload mode, it is possible (required) to tune both the host and coprocessor parameters (e.g. number of threads)

Thread Affinity

- Placement of threads on MIC cores and hardware threads
  - The basic configuration is controlled by the variable KMP_AFFINITY
  - Additional advanced settings are possible too
  - Scatter
  - Balanced
  - Compact

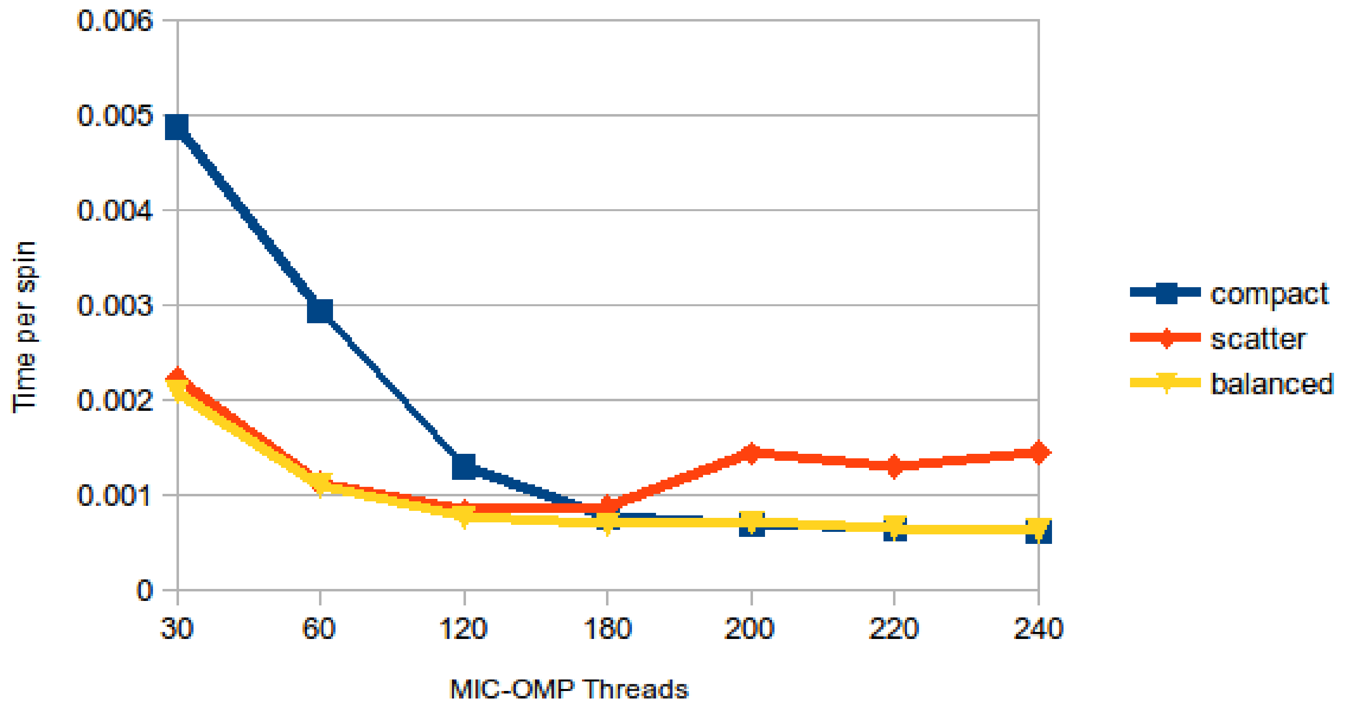






Affinity and performances

- The impact of affinity on performance may be very significant
- From a realworld example (3d-stencil code)



Collapse loops

- As recalled, the number of threads for each MPI process may become large (up to 240)
  - From different tests, it turns out that collapsing OpenMP loops results in improved performances
- From a realworld example (3d reacting Navier-Stokes equations)

MIC OMP threads	no-collapse	collapse
1	108.7	109.26
16	7.67	7.52
30	5.24	4.51
60	3.08	2.51
120	2.60	1.87
180	1.89	1.77
240	2.20	1.67

CINECA

Tiling

- “Dividing a loop into a set of parallel tasks of a suitable granularity. In general, tiling consists of applying multiple steps on a small part of a problem instead of running each step on the whole problem one after the other. The purpose of tiling is to increase reuse of data in caches”

```
#pragma omp for collapse(2)
for (int z = 0; z < nz; z++) {
   for (int y = 0; y < ny; y++) {
      for (int x = 0; x < nx; x++) {

#define YBF 16
#pragma omp for collapse(2)
for (int yy = 0; yy < ny; yy += YBF) {
   for (int z = 0; z < nz; z++) {
       int ymax = yy + YBF;
       if (ymax >= ny) ymax = ny;
       for (int y = yy; y < ymax; y++) {
```

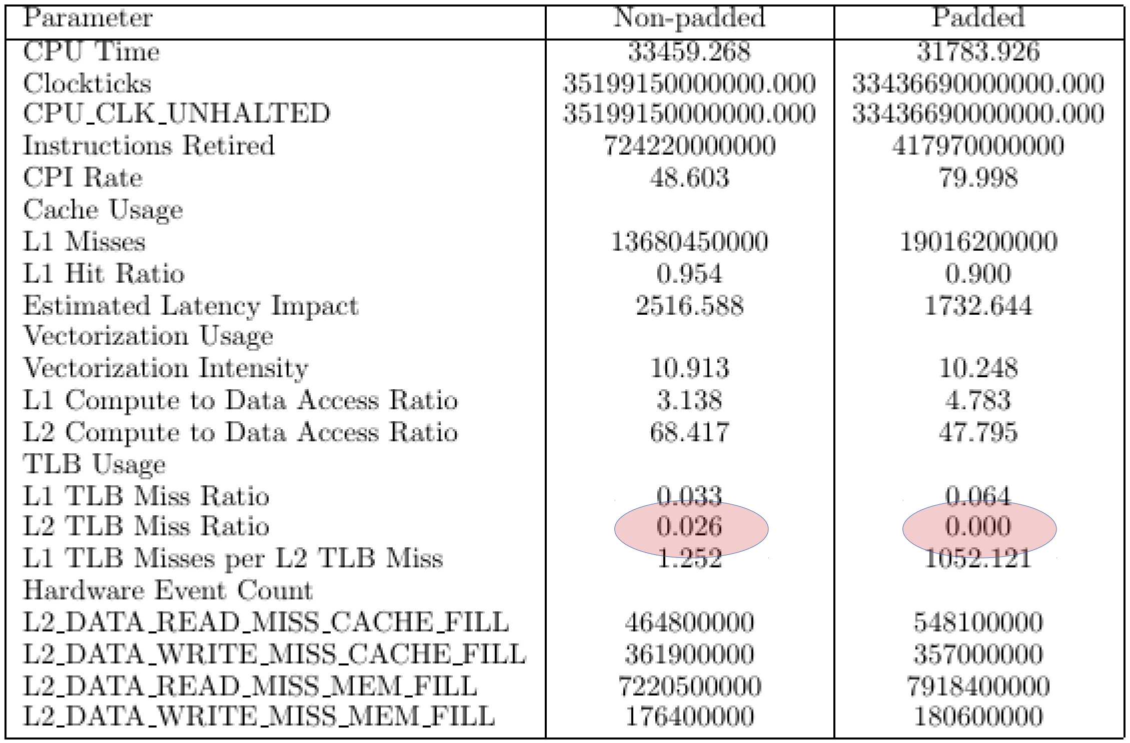
TLB cache thrashing

- "Depending on the memory patterns, possible TLB cache thrashing must be considered with care
    - Padding between allocated arrays may be a good solution
    - The problem may be difficult to analyze for non-HPC experts
- From a spin glass simulation code, the spin updating time has been measured against the padding pages between arrays

Padding pages	Time per spin
0	1.458
1	0.737
4	0.764
8	1.222
16	1.537
32	1.543

Intel VTune

- When getting unexpected performance results or whenever there is the need to have a deep understanding of the measured times, using Intel Vtune profiler is a good idea
- From the previous TLB thrashing example



Parameter	Non-padded	Padded
CPU Time	33459.268	31783.926
Clockticks	35199150000000.000	33436690000000.000
CPU_CLK_UNHALTED	35199150000000.000	33436690000000.000
Instructions Retired	724220000000	417970000000
CPI Rate	48.603	79.998
Cache Usage		
L1 Misses	13680450000	19016200000
L1 Hit Ratio	0.954	0.900
Estimated Latency Impact	2516.588	1732.644
Vectorization Usage		
Vectorization Intensity	10.913	10.248
L1 Compute to Data Access Ratio	3.138	4.783
L2 Compute to Data Access Ratio	68.417	47.795
TLB Usage		
L1 TLB Miss Ratio	0.033	0.064
L2 TLB Miss Ratio	0.026	0.000
L1 TLB Misses per L2 TLB Miss	1.252	1052.121
Hardware Event Count		
L2_DATA_READ_MISS_CACHE_FILL	464800000	548100000
L2_DATA_WRITE_MISS_CACHE_FILL	361900000	357000000
L2_DATA_READ_MISS_MEM_FILL	7220500000	7918400000
L2_DATA_WRITE_MISS_MEM_FILL	176400000	180600000

Data alignment/1

- DA is a method to force the compiler to create data objects in memory on specific byte boundaries. This is done to increase efficiency of data loads and stores to and from the processor.
  - For MIC memory movement is optimal when the data starting address lies on 64 byte boundaries
- Two steps are needed
- (1) Align the data

```
float A[1000] __attribute__((aligned(64)));
buf = (char*) _mm_malloc(bufsizes[i], 64);
real, allocatable :: a(:)
!dir$ attributes align:64 :: a
```

Data alignment/2

- (2) Use pragma/directives and clauses to tell the compiler that the accesses are aligned
  - For an i-loop that has a memory access of the form a[i+n1], the loop has to be structured in such a way that the starting-indices have good alignment properties.

```
__assume_aligned(a, 64);
__assume(n1%16==0);
__assume(n2%16==0);
for(i=0;i<n;i++) {
 // Compiler vectorizes loop with all aligned accesses
  X[i] += a[i] + a[i+n1] + a[i-n1]+ a[i+n2] + a[i-n2];
}
```

Streaming store and prefetch

- Starting with Composer XE 2013 Update 1 compiler, streaming stores instructions are generated under certain conditions
  - Instructions intended to speed up the performance in the case of vector-aligned unmasked stores in streaming kernels where we want to avoid wasting memory bandwidth by being forced to read the original content of an entire cache line from memory when we overwrite their whole content completely
- Heuristics may be not sufficient: user can provide hints to the compiler, e.g.

```
#pragma vector nontemporal A
```

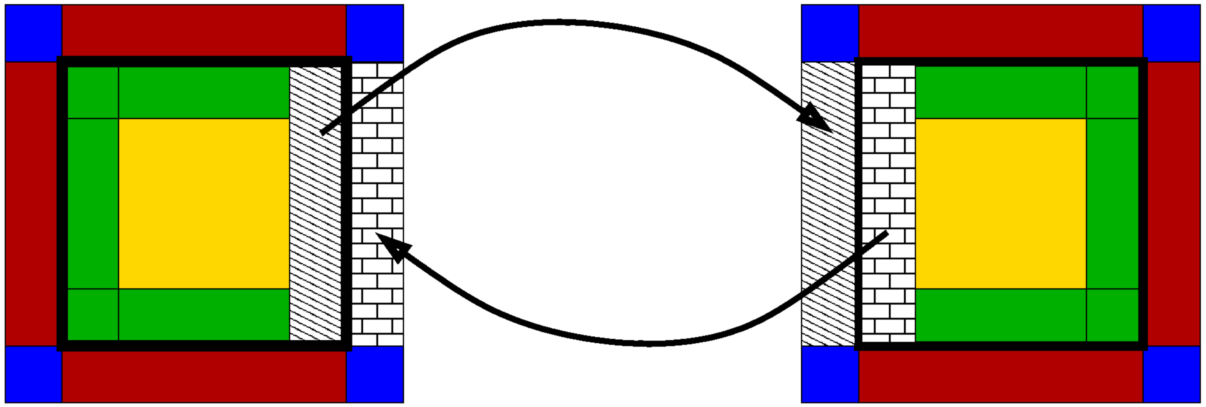
where A[i]=... is the store inside the loop

Native *vs* Offload

- Why offload mode?
- Cons
  - The porting is much more complex than to native mode
  - And the programmer must take care of host-coprocessors data exchanges which may be disastrous wrt performances
  - The symmetric mode allows to use both host and MIC at the same time
- Pros
  - it is also reasonable to assume that, the host being in charge of MPI calls (as it happens in offload mode), the MIC is free to execute, at its best, the computing intensive part of the code without wasting time in managing the communications

MPI optimizations: FDTD

- Consider a finite difference time domain code parallelized by standard domain decomposition. At each step:
  - (a) update boundary and bulk values
  - (b) exchange ghosts with neighboring processes



MPI optimizations: FDTD/2

- MPI patterns allow to overlap computations with communications (hiding the communication cost)
- Standard CPU pattern using MPI non blocking functions (available for MIC native mode as well)
  - Update boundary
  - Exchange ghost – MPI non blocking
  - Update bulk
  - Wait exchanges – MPI wait
- To achieve full overlapping, the bulk updating time must be larger than the communication time
- Using MIC (native), sometimes the final performances are far from optimal

MPI optimizations: FDTD/3

- MIC-Offload pattern (similar to multi-GPU approach)
  - Update boundary
  - Update bulk – asynchronous (non blocking)
  - Exchange ghost – MPI blocking
  - Wait bulk update

```
#pragma offload target(mic:0) … async(a)
{
<code to be offloaded>
}
CPU operations (e.g. MPI calls)
#pragma offload_wait(a)
```

MPI optimizations: HSG

HPCS 2014

CINECA

- Scaling results from Heisenberg Spin Glass code
- Strong scaling for native/offload and sync/async versions

#MICS	Native-Sync	Native-Async	Offload-Sync	Offload-Async
1	0.709	0.717	1.049	1.078
2	0.484	0.431	0.558	0.527
4	0.445	0.325	0.335	0.281
8	0.376	0.246	0.219	0.167
16	0.343	0.197	0.154	0.113

- Weak scaling comparison with other architectures

#Procs	Size	CPU	GPU	MIC-n	MIC-o
1	256	3.73	0.67	0.78	1.34
8	512	0.48	0.068	0.25	0.17
Efficiency		96.2%	123%	39.9%	100%