HPC Code Optimization Workshop

Dr. Fabio Baruffa
fabio.baruffa@lrz.de

Dr. Luigi Iapichino
luigi.iapichino@lrz.de

Outline

- Part 1:
  - Introduction and Motivation
  - Modern Computer Architecture
  - Cache and Memory System
  - Roofline model
- Part 2:
  - Optimization Process
  - Nbody Example
  - Enable vectorization: SIMD
- Part 3:
  - Data layout
  - Data alignment
  - Enable OpenMP
- Part 4:
  - Profiling tools
  - Intel® Advisor XE
- Concluding remarks:
  - Intel PCC @ LRZ
  - Intel® MIC Architecture

Data Layout

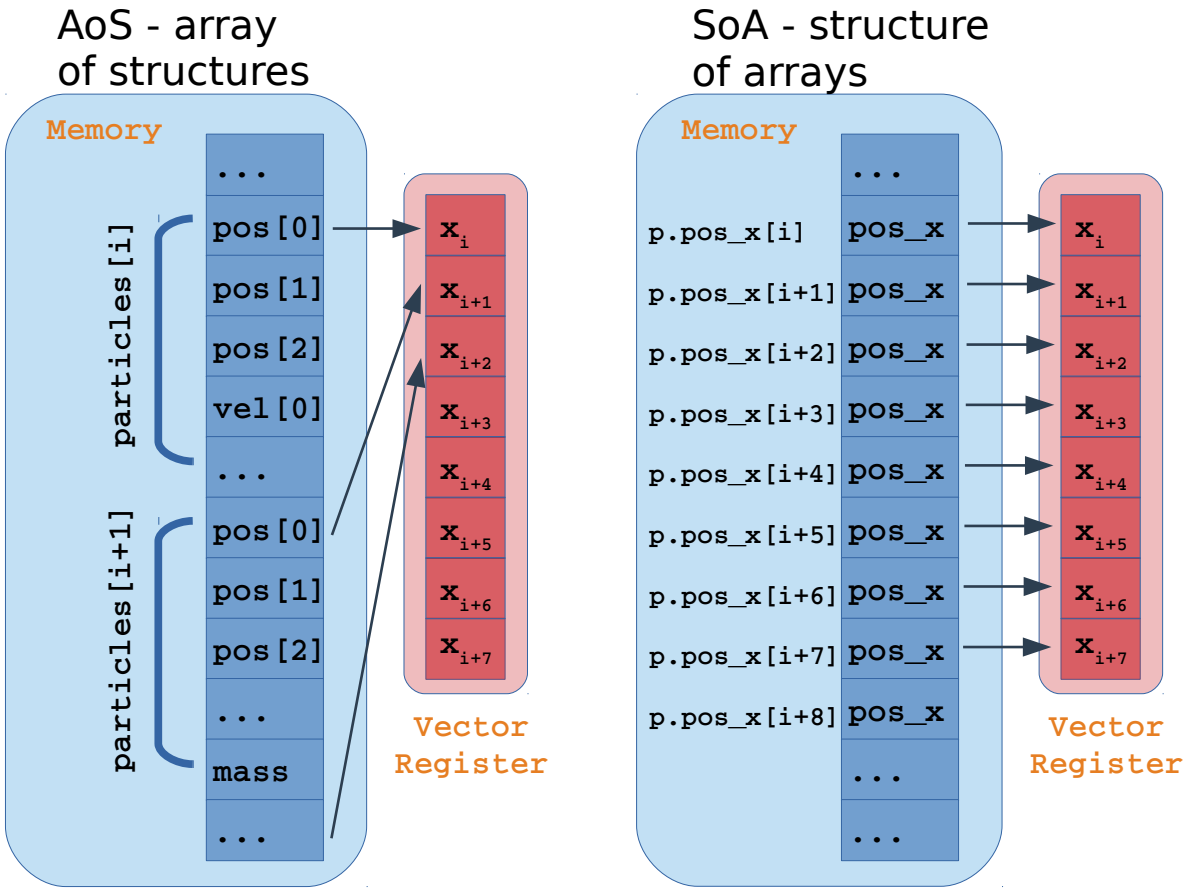
Compiler report: ver1

```
LOOP BEGIN at GSimulation.cpp(132,20)
         remark #15335: loop was not vectorized:  inner loop was already vectorized

LOOP BEGIN at GSimulation.cpp(137,7)
      remark #25444: Loopnest Interchanged: ( 1 2 ) --> ( 2 1 )
      remark #15542: loop was not vectorized: inner loop was already vectorized
[ GSimulation.cpp(137,7) ]

LOOP BEGIN at GSimulation.cpp(135,5)
      <Predicate Optimized v1>
         remark #25423: Condition at line 139 hoisted out of this loop

         remark #15328: vectorization support: gather was emulated for the variable dx:
masked, strided by 10   [ GSimulation.cpp(145,31) ]
         remark #15328: vectorization support: gather was emulated for the variable dx:
masked, strided by 10   [ GSimulation.cpp(146,31) ]
         remark #15328: vectorization support: gather was emulated for the variable dx:
masked, strided by 10   [ GSimulation.cpp(147,31) ]
         remark #15329: vectorization support: scatter was emulated for the variable dx:
masked, strided by 10   [ GSimulation.cpp(152,31) ]
...
         remark #15305: vectorization support: vector length 8
         …
```

Vectorization report: gather/scatter

A loop that has been automatically vectorized contains loads from memory locations which are **not contiguous** in memory → non-unit stride load
The compiler has issued a hardware gather/scatter instructions.

```
struct Particle
{
  public:
    ...
    real_type pos[3];
    real_type vel[3];
    real_type acc[3];
    real_type mass;
};


struct ParticleSoA
{
  public:
    ...
    real_type *pos_x,*pos_y,*pos_z;
    real_type *vel_x,*vel_y,*vel_z;
    real_type *acc_x,*acc_y;*acc_z
    real_type *mass;
};
```



SoA: unit stride access

- The Particle structure has **strided** access: the distance between 2 consecutive position for different particles is **10** elements.

```
struct Particle
{
  public:
    ...
    real_type pos[3];
    real_type vel[3];
    real_type acc[3];
    real_type mass;
};
Particle *particles;
```

```
void GSimulation :: start()
{
  //allocate particles
  particles = new Particle[get_npart()];

  init_pos();
  init_vel();
  init_acc();
  init_mass();
  ...
```

- The ParticleSoA structure has **unit-stride** access: the distance between 2 consecutive position for different particles is **1** element.

```
struct ParticleSoA
{
  public:
    ...
    real_type *pos_x,*pos_y,*pos_z;
    real_type *vel_x,*vel_y,*vel_z;
    real_type *acc_x,*acc_y;*acc_z
    real_type *mass;
};
ParticleSoA *particles;
```

```
void GSimulation :: start()
{
  //allocate particles
  particles = new ParticleSoA[get_npart()];

  particles->pos_x = new real_type[get_npart()];
  particles->pos_y = new real_type[get_npart()];
  particles->pos_z = new real_type[get_npart()];
  particles->vel_x = new real_type[get_npart()];
  particles->vel_y = new real_type[get_npart()];
...
```

Hands-on session

Let’s start a different approach for the vectorization

- Go to the folder **code/nbody/ver1**
- Run **make clean** to remove old files
- Always use the compiler flags to generate the report:
  -qopt-report=5 -qopt-report-filter="GSimulation.cpp,130-204"
  -qopt-report-phase=vec
- Change the *Array of Structures* in *Structure of Arrays*:
  i. Look in the Particle.hpp file and define a new C structure named ParticleSoA similar to the structure Particle but now **using pointers**
  ii. Change the GSimulation.cpp file accordingly (new/delete)
  iii. Do not forget to change GSimulation.hpp

Hands-on session: solution

Let’s start a different approach for the vectorization

- Solution in the folder **code/nbody/ver3**
- Look at the files: Particle.hpp and GSimulation.hpp
- Run **make clean** to remove old files and **recompile**
- Is the new version automatically vectorized?
- Let’s have a look at the compiler report

Final results of the Nbody example

Version	Optimization / Comments	Performance	% of the Peak E5-2650 v2
base	-O2 / 1 thread	2.28 GFs	5.4% (0.6 %)
ver1	-O3 -xAVX / scalar optimization / 1 thread	6.07 GFs	14.4% (1.8 %)
ver2	#pragma simd / 1 thread	10.7 GFs	25.6% (3.2 %)
ver3	AoS → SoA (no auto-vector) / 1 thread	2.08 GFs	5.0% (0.6%)

Data dependencies

```
LOOP BEGIN at GSimulation.cpp(143,20)
   remark #15541: outer loop was not auto-vectorized: consider using SIMD directive

   LOOP BEGIN at GSimulation.cpp(146,4)
      remark #15344: loop was not vectorized: vector dependence prevents vectorization
      remark #15346: vector dependence: assumed ANTI dependence between dx line 156 and this
line 165
      remark #15346: vector dependence: assumed FLOW dependence between this line 165 and dx
line 156

      LOOP BEGIN at GSimulation.cpp(148,4)
         remark #15344: loop was not vectorized: vector dependence prevents vectorization
         remark #15346: vector dependence: assumed ANTI dependence between dx line 156 and this
line 165
         remark #15346: vector dependence: assumed FLOW dependence between this line 165 and dx
line 156
      LOOP END
   LOOP END

   LOOP BEGIN at GSimulation.cpp(171,4)
      remark #15344: loop was not vectorized: vector dependence prevents vectorization
      remark #15346: vector dependence: assumed FLOW dependence between this line 173 and this
line 185
      remark #15346: vector dependence: assumed ANTI dependence between this line 185 and this
line 173
   LOOP END
LOOP END
```

Data dependencies

Vectorization changes the order of the operation inside a loop, since each **SIMD** instruction operates on several data at once.
Vectorization is only possible if this does not change the results.

ANTI dependence: write-after-read (WAR). Statement **i** precedes **j**, and **i** uses a value that **j** computes: 2 → 3

FLOW (true) dependence: read-after-write (RAW). Statement **i** precedes **j**, and **i** uses a value that **j** computes: 1 → 2, 2 → 4

```
1: x = 1;
2: y = x + 2;
3: x = z - w;
…
4: x = y / z;
```

```
for(i=0; i<N-1; i++)
  a[i] = a[i+i] + b[i];

for(i=0; i<N; i++)
  a[i] = a[i-i] + b[i];
```

Hands-on session

Let’s start a different approach for the vectorization

- Go to the folder **code/nbody/ver3**
- Run **make clean** to remove old files
- Try to remove the data dependencies defining the appropriate variables before the inner loop, which is the candidate for vectorization

Hands-on session

Let’s start a different approach for the vectorization

- Go to the folder **code/nbody/ver3**
- Run **make clean** to remove old files
- Try to remove the data dependencies defining the appropriate variables before the inner loop, which is the candidate for vectorization
- Solution in the folder: **code/nbody/ver3_1**

Final results of the Nbody example

Version	Optimization / Comments	Performance	% of the Peak E5-2650 v2
base	-O2 / 1 thread	2.28 GFs	5.4% (0.6 %)
ver1	-O3 -xAVX / scalar optimization / 1 thread	6.07 GFs	14.4% (1.8 %)
ver2	#pragma simd / 1 thread	10.7 GFs	25.6% (3.2 %)
ver3	AoS → SoA (no auto-vector) / 1 thread	2.08 GFs	5.0% (0.6%)
ver3_1	removed vector dependence / 1 thread	17.5 GFs	42.1% (5.2%)

Data Alignment

Unaligned access

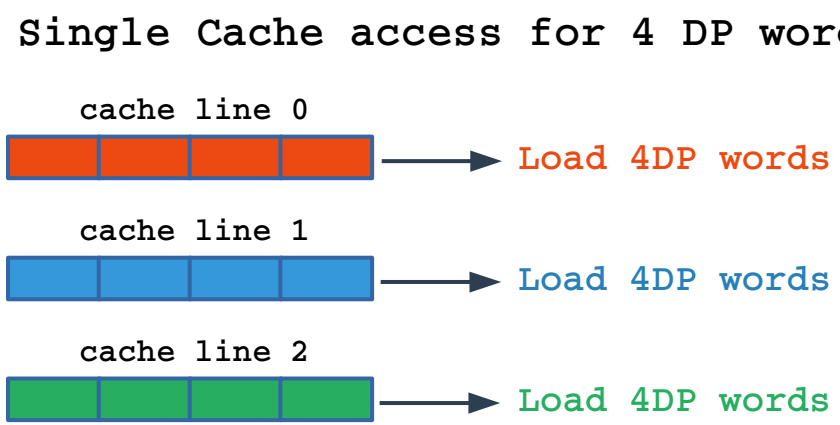
```
LOOP BEGIN at GSimulation.cpp(159,2)
      <Peeled loop for vectorization>
LOOP END

LOOP BEGIN at GSimulation-nodep.cpp(159,2)
   remark #15389: vectorization support: reference px_i has unaligned access [Gsimul...(167,6)]
   remark #15389: vectorization support: reference py_i has unaligned access [Gsimul...(168,6)]
   remark #15388: vectorization support: reference pz_i has aligned access   [Gsimul...(169,6)]
   remark #15389: vectorization support: reference ax_i has unaligned access [Gsimul...(174,6)]
   remark #15389: vectorization support: reference ax_i has unaligned access [Gsimul...(175,6)]
   remark #15389: vectorization support: reference ax_i has unaligned access [Gsimul...(176,6)]

   remark #15381: vectorization support: unaligned access used inside loop body
   remark #15305: vectorization support: vector length 8
   remark #15309: vectorization support: normalized vectorization overhead 1.026
   remark #15300: LOOP WAS VECTORIZED
   remark #15442: entire loop may be executed in remainder
   remark #15454: masked aligned unit stride loads: 1
   remark #15456: masked unaligned unit stride loads: 5
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 189
   remark #15477: vector loop cost: 29.370
   remark #15478: estimated potential speedup: 5.160
   remark #15488: --- end vector loop cost summary ---
LOOP END

LOOP BEGIN at GSimulation.cpp(159,2)
      <Reminder loop for vectorization>
LOOP END
```

Data alignment

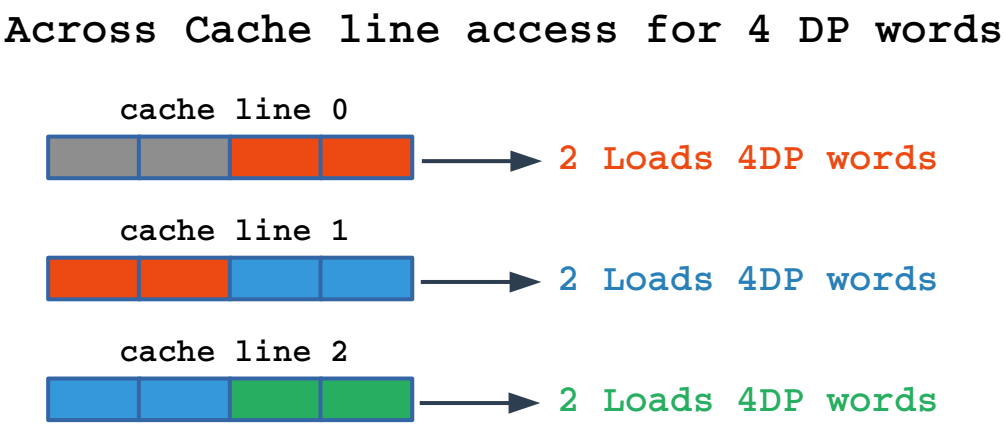
The compiler cannot know if your data is aligned to a multiple of the vector register width. This could effect the performance.

A pointer **p** is aligned to a memory location on a n-byte boundary if:
((size_t)p%n==0)

For AVX, alignment to 32byte boundaries (4 DP words) allows a single reference to a cache line for moving 4 DP words into the registers.



32byte Aligned



Non-Aligned

Data alignment

On the Stack: for declared variables the Intel® C/C++ compiler aligned the data naturally:

```
float f; //4-byte aligned          double d; //8-byte aligned
```

For array data an attribute is necessary:

```
float array[N] __attribute__((aligned(32)); //32-byte aligned
```

On the Heap: the array can be allocated/deallocate with special functions:

```
#include <malloc.h>
…
float *array = (float*) _mm_malloc(N*sizeof(float), 32);
…
_mm_free(array);
```

Data alignment

SSE: works better with 16 bytes alignment

Why?: the XMM registers are 16 bytes (i.e. 128 bits)

Penalties:

Unaligned access vs aligned access (but still in the same cache line)
40% worse.

Unaligned access vs aligned access (but split over cache line)
500% worse.

Rule of thumb: Try to align to the SIMD register size
MMX: 8 Bytes; SSE2: 16 Bytes; AVX: 32 Bytes; AVX512/MIC: 64 Bytes.

Also try to align blocks of data to **cacheline** size – i.e. 64 Bytes.

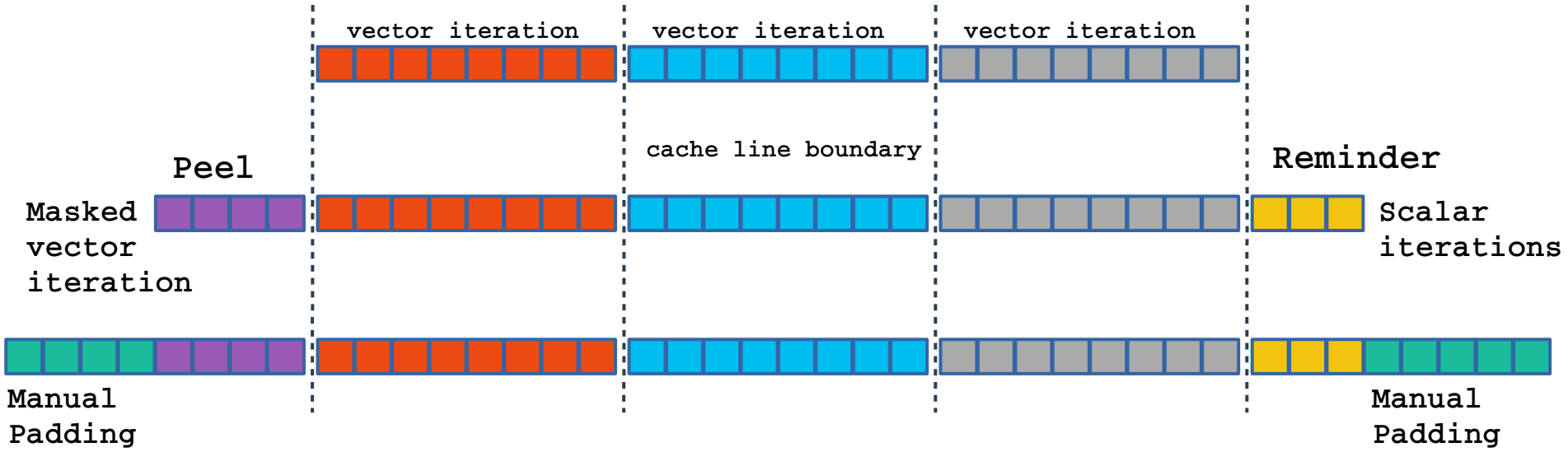
Peel and reminder loop

The compiler can generate a **Peel** and **Reminder** loop in case where:

- The loop trip count is known only during runtime
- The alignment is not known during compilation

Then the compiler generates a check in code at the beginning of the loop to verify its assumptions. This could cause inefficiency, since every time enters the loops, it does these checks.

```
for(j = 0; j < N; j++) array[j]= ...
```



Hands-on session

Let’s start a different approach for the vectorization

- Go to the folder **code/nbody/ver3_1**
- Run **make clean** to remove old files
- Replace the **new/delete** statements with the memory alignment functions
- Does the compiler report say what you expect?

Hands-on session

Let’s start a different approach for the vectorization

- Go to the folder **code/nbody/ver3_1**
- Run **make clean** to remove old files
- Replace the **new/delete** statements with the memory alignment functions
- Does the compiler report say what you expect?
- Solution for the alignment in the folder **code/nbody/ver3_2**:
  i. cp GSimulation.cpp GSimulation.cpp_bkp
  ii.patch GSimulation.cpp < align.patch

Final results of the Nbody example

Version	Optimization / Comments	Performance	% of the Peak E5-2650 v2
base	-O2 / 1 thread	2.28 GFs	5.4% (0.6 %)
ver1	-O3 -xAVX / scalar optimization / 1 thread	6.07 GFs	14.4% (1.8 %)
ver2	#pragma simd / 1 thread	10.7 GFs	25.6% (3.2 %)
ver3	AoS → SoA (no auto-vector) / 1 thread	2.08 GFs	5.0% (0.6%)
ver3_1	removed vector dependence / 1 thread	17.5 GFs	42.1% (5.2%)
ver3_2	alignment allocation / 1 thread	17.8 GFs	42.8% (5.3%)
ver3_2p	__assume_align(…) / 1 thread	20.4 GFs	61.4% (7.6%)

Final remarks on vectorization

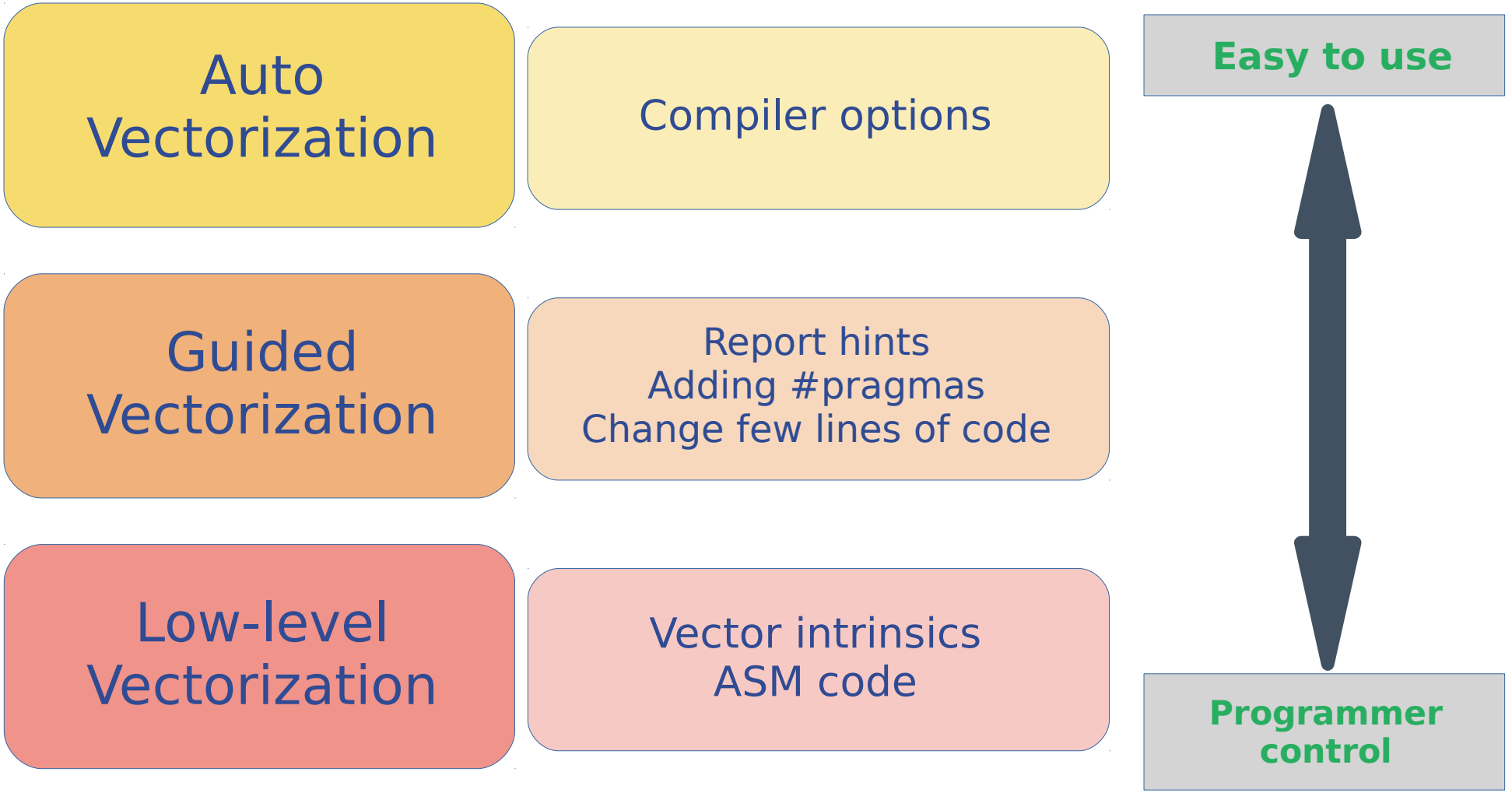
- Not always the compiler does what we want, we need to give suggestions: `__assume_aligned(…)`.
  Without this step, the compiler will not detect the optimal alignment for accesses using such arrays
  Alignment is generally unknown at compile time.

- We have changed not too much  in the code.

- We can give hints if the compiler does not vectorize as we expect. (`#pragma vector, #pragma simd`)

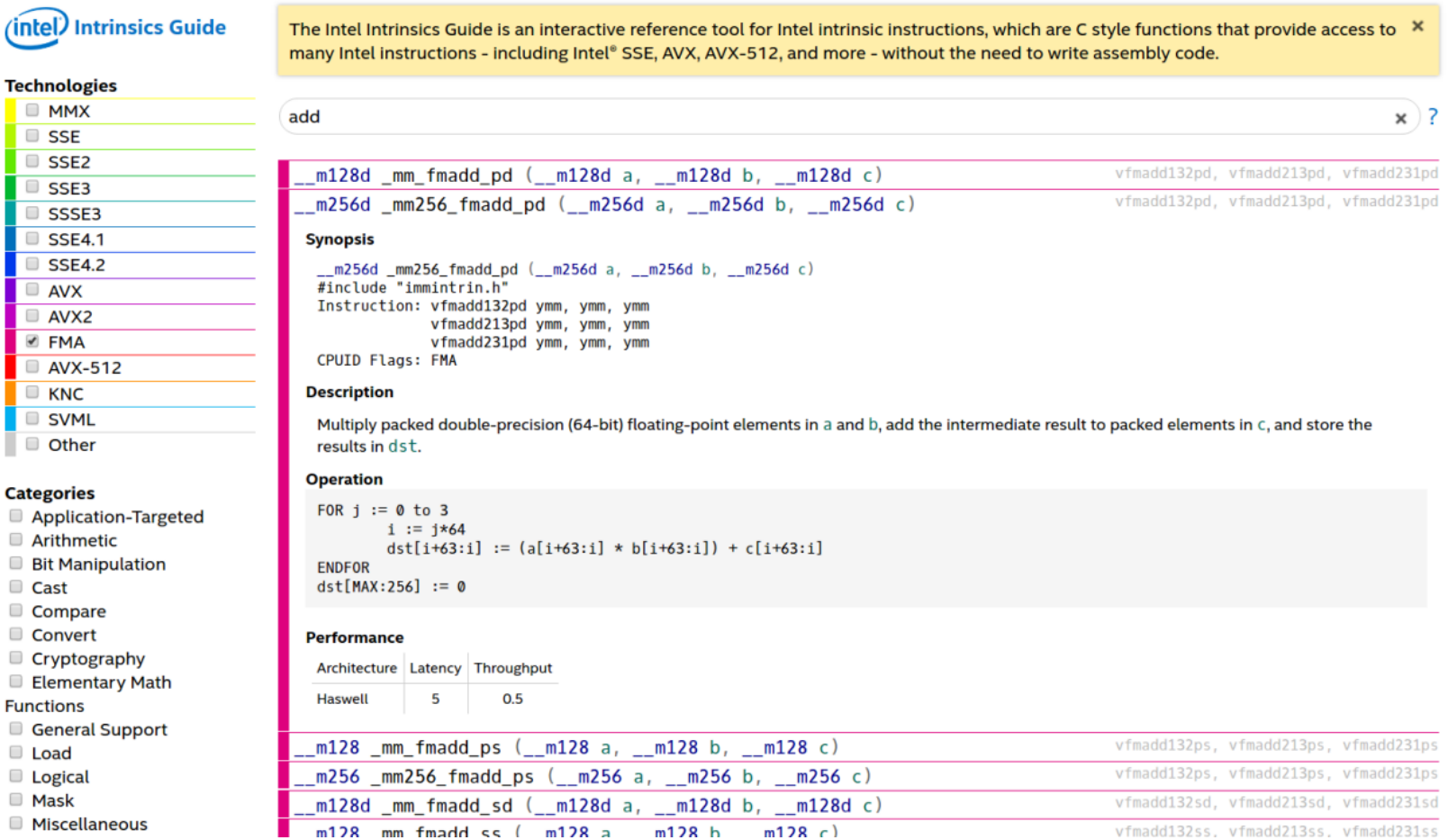
- Very good speedup: ~8.9x

Enabling vectorization



Intel® intrinsic instructions

Intrinsics are like library functions, but directly understood by the compiler. They are almost translated into assembly code and are **hardware specific**.



Array notation using Intel® Cilk™ Plus

Intel Cilk Plus includes extensions to C and C++ that allows for parallel operations on arrays. The intent is to allow users to express high-level **vector parallel** array operations. This helps the compiler to effectively vectorize the code. Array notation can be used for both static and dynamic arrays.

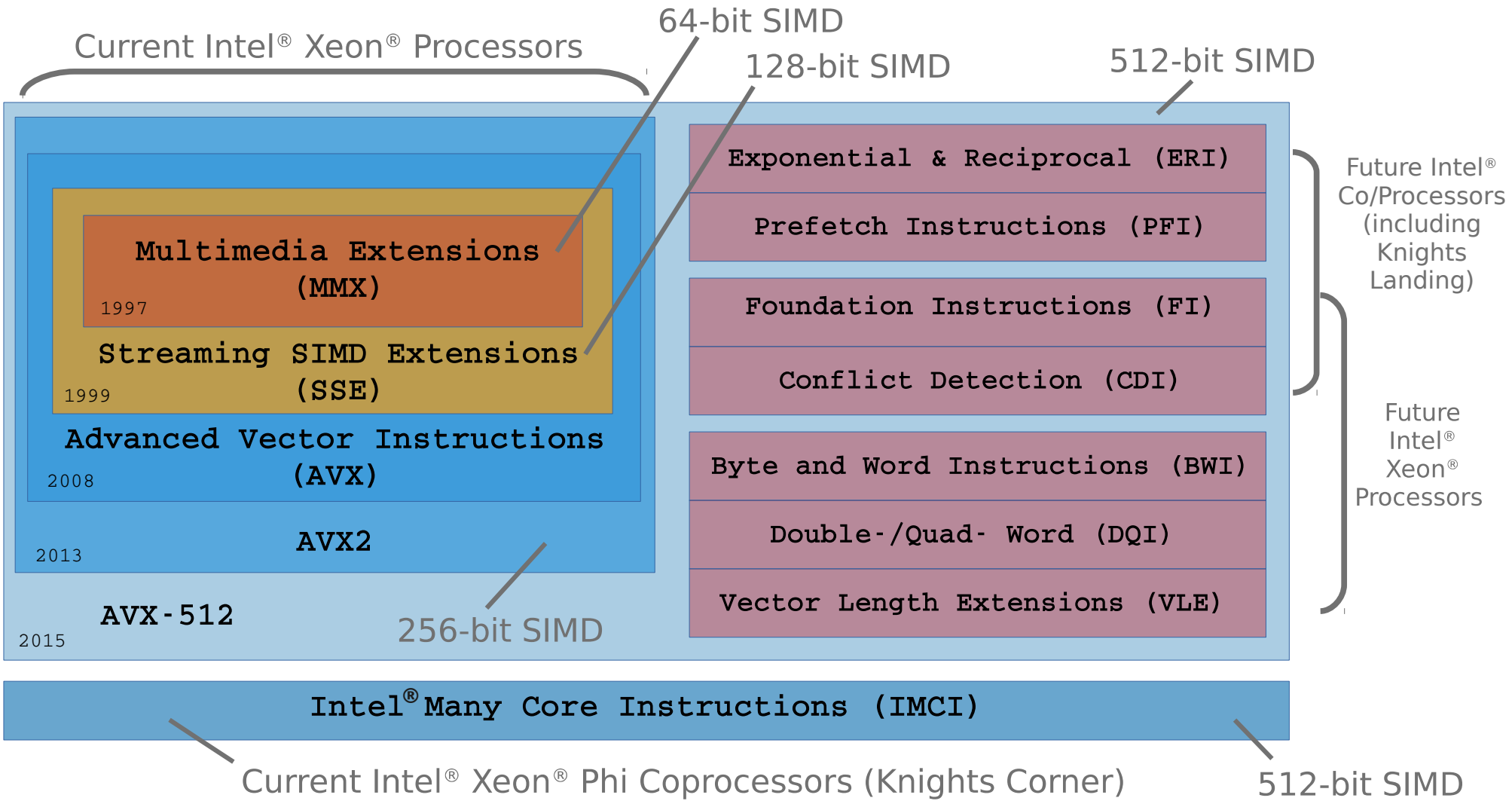
It is supported in C/C++ Intel compiler and GCC 4.9.

The vectorization become explicit: *array-expression[lower-bound : length : stride]*

```
int main(int argc, char **argv)
{
    cost int array_size = 10;
    int a[array_size];
    int b[array_size];

    // Initialize array using for loop
    for (int i = 0; i < array_size; i++)
        a[i] = 5;

    // Initialize the array using Array Notation.  Since the array is
    // statically allocated, we can use default values for the start index (0)
    // and number of elements (all of them).
    b[:] = 5;
...
```

Future Intel® Co/Processors (including Knights Landing)

Past, present and future of Intel® SIMD

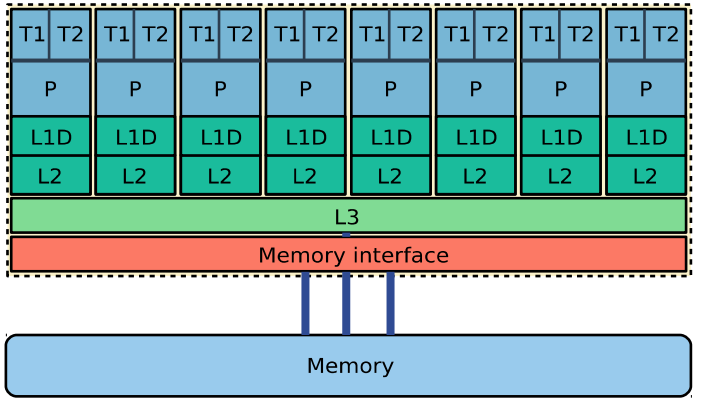


Enable OpenMP

Multi-threading: OpenMP introduction

Shared memory system: the RAM can be accessed by several different CPUs.

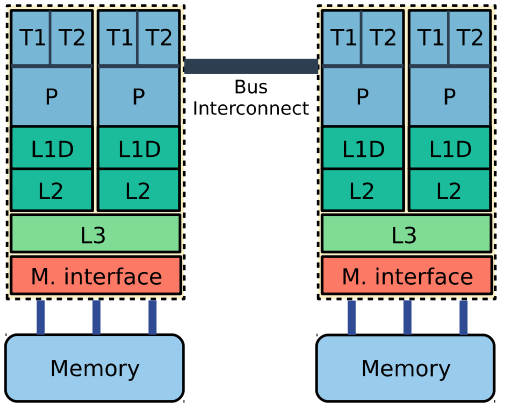
OpenMP is designed for multi-processor/core, shared memory machine in **UMA/NUMA** architecture.



UMA

OpenMP www.openmp.org

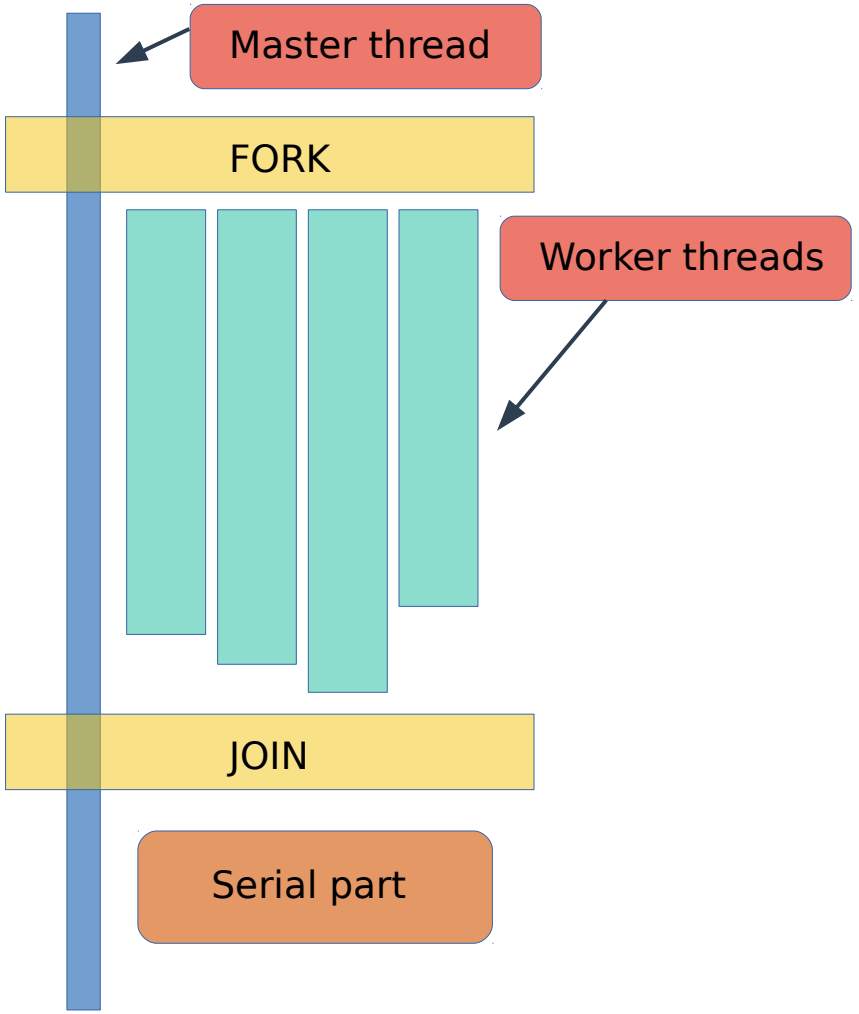
- A set of compiler directives and API for multithreading applications
- An explicit (not automatic) programming model, offering the programmer full control over parallelization



NUMA

Fork-join model

- OpenMP uses the **fork-join** model for parallel execution
- The program start as a single process: the **master thread**
- During the execution the master thread creates a team of parallel threads → **FORK**
- The subsequent part is executed in parallel
- When all the threads terminate the execution, they synchronize and terminate → **JOIN**



OpenMP directives

OpenMP directives for C/C++ are specified with the pragma preprocessing directive. The syntax is formally as follow:

```
C/C++:     #pragma omp directive-name [clause[,], … ]
Fortran:   $!omp directive-name [clause[,], … ]
```

The parallelization has to be expressed explicitly:

C/C++:

```
#pragma omp parallel
{
 …
}
```

Fortran:

```
!$omp parallel
      …
!$omp end parallel
```

OpenMP: Hello world

```
#include <iostream>
#include <omp.h>

int main(int argc, char** argv)
{
#pragma omp parallel
 {
   int my_id = omp_get_thread_num();
   int n_threads = omp_get_num_threads();
   std::cout << "Hello world from:" << my_id
             << " out of:" << n_threads std::endl;
 }
 return 0;
}
```

OpenMP include file

Start the parallel region (one entry)

Runtime library functions

End the parallel region (one exit)

Sample Output:

```
Hello world from:1 out of:4
Hello world from:3 out of:4
Hello world from:4 out of:4
Hello world from:2 out of:4
```

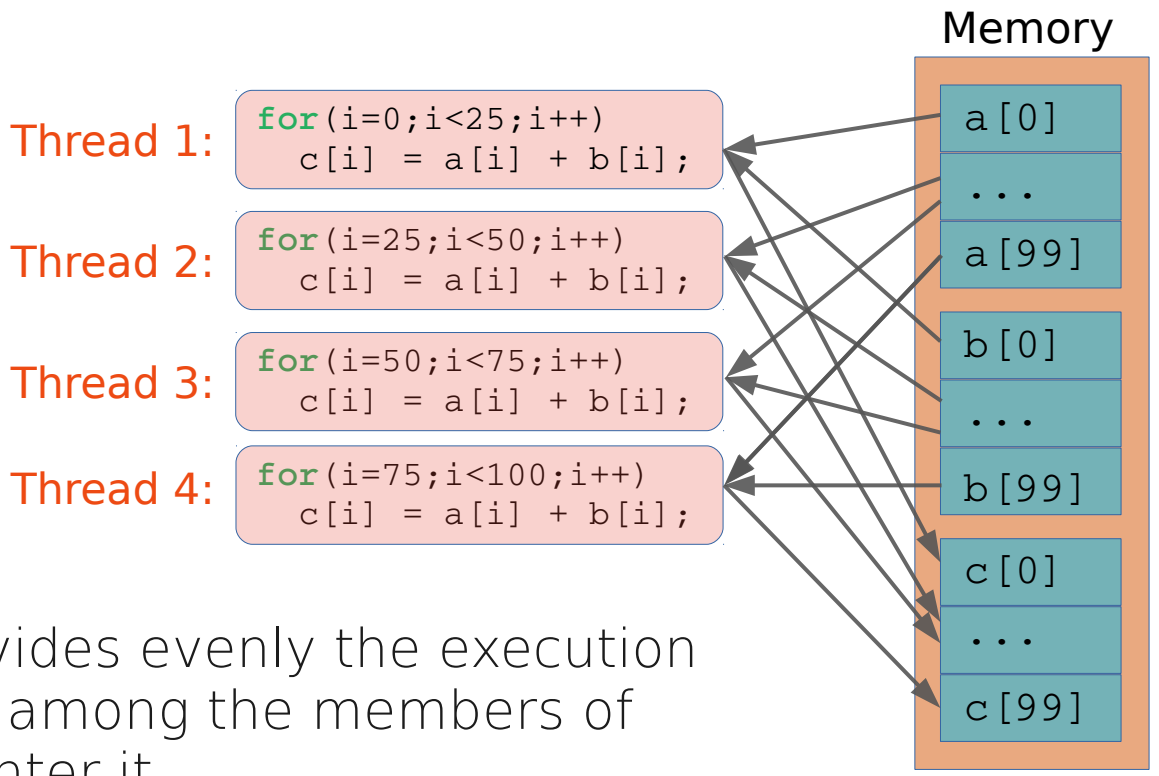
The output is not deterministic!

The number of threads can be specified via the environment variable: `OMP_NUM_THREADS=...`

For construct

Ofter in a program **loops** take the most computing time!
The **for** construct specifies that the iterations of one or more associated loops will be executed in parallel by threads in a team.

```
#include <iostream>
#include <omp.h>
#define N 100
int main(int argc, char** argv)
{
 int i;
 float a[N], b[N], c[N];
#pragma omp parallel for
 for(i=0;i<N;i++)
   c[i] = a[i] + b[i];

 return 0;
}
```



A work-sharing construct divides evenly the execution of the enclosed code region among the members of the thread-team that encounter it.

OpenMP loop scheduling

The OpenMP for loop can be scheduled in different ways according to the **workload** and the **performance** of the loop. This specifies how iterations of the associated loops are divided into contiguous chunks and distributed among the threads of the team. The most used scheduling are:

- **Guided**: the iterations are assigned to threads in the team in chunks as the executing threads request them. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be assigned. The chunk decrease with time.

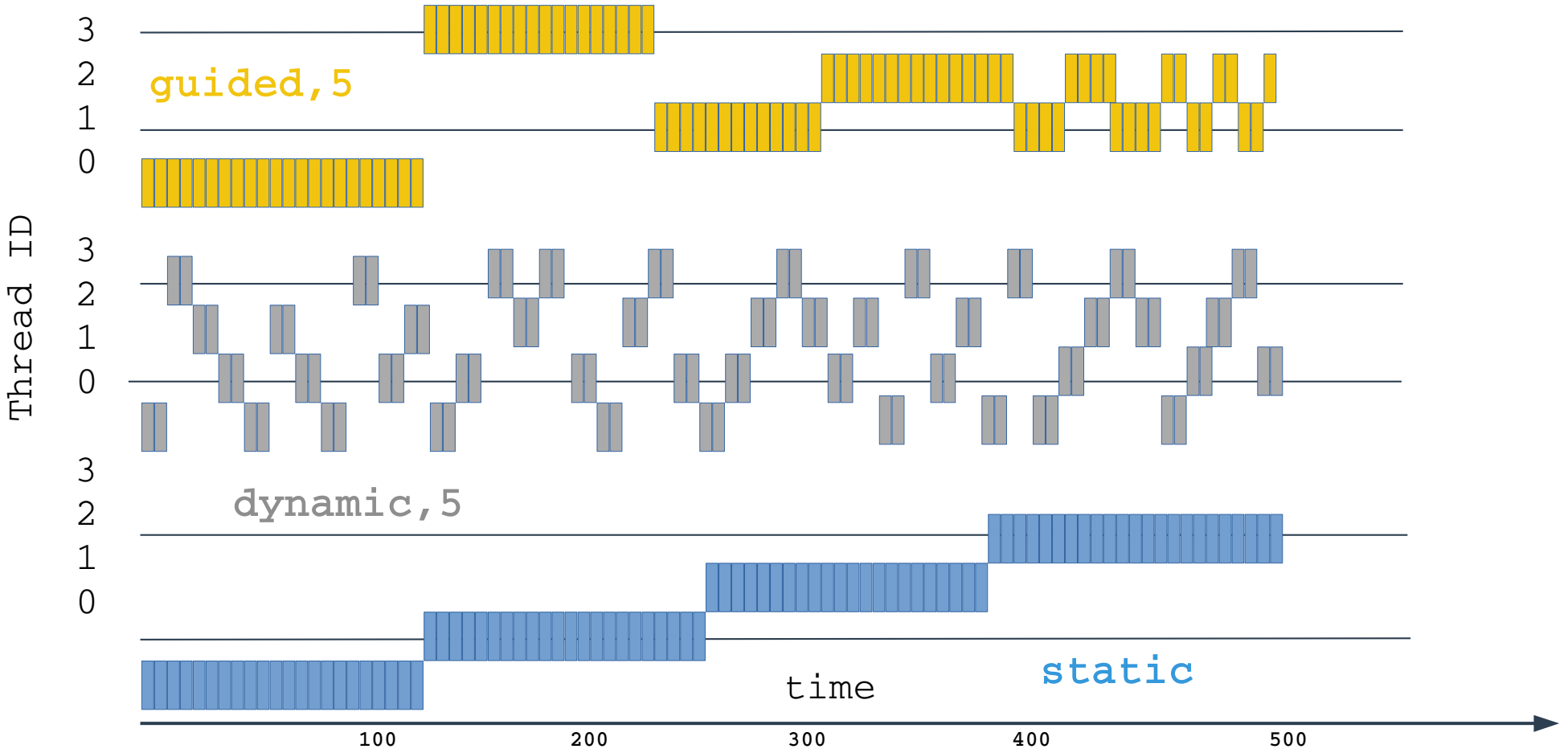
- **Dynamic**: iterations are distributed to threads in the team in chunks as the threads request them. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed.

- **Static**: iterations are divided into chunks of size chunk_size, and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number.

OpenMP loop scheduling

```
#pragma omp parallel for schedule(...)
```

500 iterations
on 4 threads



Synchronization: barrier

When a thread reaches the **parallel** directive, it creates the team of threads and all the threads execute the same code.
The is always a implicit **barrier** at the and of the parallel region.

One can explicitly use the barrier directive to synchronize the threads.

```
#include <iostream>
#include <omp.h>
#define N 100
int main(int argc, char** argv)
{
 int i;
 float a[N], b[N], c[N];
#pragma omp parallel for
 for(i=0;i<N;i++)
   c[i] = a[i] + b[i];

 return 0;
}
```

Implicit barrier

Explicit barrier

```
void work1(int k) { // large amount of
work }
void work2(int k) { // large work that must
come after work1 is finished }
int main() {
  int n=1000000;
  #pragma omp parallel private(i) shared(n)
  {
    #pragma omp for
    for (i=0; i<n; i++) work1(i);
    #pragma omp barrier
    #pragma omp for
    for (i=0; i<n; i++) work2(i);
  }
...
```

Critical and atomic

The critical construct restricts execution of the associated structured block to a single thread at a time. An optional name may be used to identify the critical construct.

```
for (i = 0; i < SIZE; i++) a[i] = rand();
max = a[0];
#pragma omp parallel for num_threads(4)
for (i = 1; i < SIZE; i++)
{
  if (a[i] > max)
  {
   #pragma omp critical
   {
    // compare a[i] and max again because max
    // could have been changed by another
    // thread after the comparison outside
    // the critical section
    if (a[i] > max)
       max = a[i];
    }
  }
}
```

The atomic construct ensures that a specific ***storage location*** is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads.

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

```
double pi,x;
int i,N;
pi=0.0;
N=1000;
#pragma omp parallel for
for(i = 0;i < N; i++)
{
   x = (double)i/N;
   #pragma omp atomic
   // An atomic operation has much lower
   // overhead. It takes advantage on the
   // hardware providing an atomic increment
   // operation; in that case there's no
   // lock/unlock needed on entering/exiting
   // the line of code.
   pi += 4 / (1+x*x);
}
pi=pi/N;
```

Data-sharing attribute

Reduction: The reduction clause specifies an operator and one or more list of items. For each list item, a private copy is created in each implicit task, and is initialized appropriately for the operator. After the end of the region, the original list item is updated with the values of the private copies using the specified operator.

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

```
double calculate_PI(int N)
{
 double fX =0.0;
 double fSum =0.0;
 double fH = 1.0/(double) N;

#pragma omp parallel for reduction(+:fSum)
 for(i=0;i<N;i++)
 {
   fX = fH * ( (double)i + 0.5);
   fsum += f(fX);
 }
 return (fH * fSum);
}
```

reduction (operator: list)

+, *, -, /, &, ^, |
Not overloaded

```
double f(double value)
{
  return( 4.0 / 1.0 + value*value);
}
```

Hands-on session: pi.c

- Go to the folder **code/examples/openmp**
- The **pi.c** is the code which compute PI, compile and run
- The compute intensive part resides only in the `CalcPi()` function: Try to use Openmp

#Threads	Runtime [sec]	Speedup	Efficiency
1			
2			
3			
4			
6			
12			

Storage attributes

In OpenMP is also possible to change the storage attribute for the variables inside a parallel region using the following clauses:

- **Shared**: declares a list of one or more items to be shared by the threads.
- **Private**: declares one or more items to be private to a thread. The other threads cannot access this data. Changes can only visible to the thread owning the data.
- **Firstprivate**: declares one or more list items to be private to a thread, and initializes each of them with the value that the corresponding original item has when the construct is encountered.
- **Lastprivate**: the final value of a private inside a parallel loop can be transmitted to the shared variable outside the.

```
int main(void) {
int i;
int x = 42;

#pragma omp parallel for private(x)
for(i = 0; i<= 10; i++){
  x=i;
  printf("Thread number: %d x: %d\n",
          omp_get_thread_num(),x);
}
 printf("x is %d\n", x);
}
```

***Output*: private**

```
Thread number: 0 x: 0
Thread number: 0 x: 1
...
x is 42
```

***Output*: shared**

```
Thread number: 0 x: 0
...
Thread number: 1 x: 5
x is 5
```

***Output*: firstprivate**

```
Thread number: 2 x: 6
Thread number: 2 x: 7
...
x is 42
```

***Output*: lastprivate**

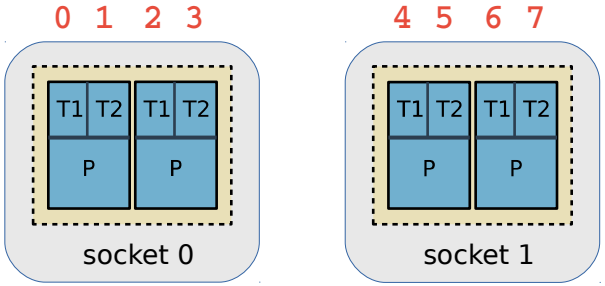
```
Thread number: 2 x: 6
...
Thread number: 3 x: 10
...
Thread number: 1 x: 5
x is 10
```

Thread affinity

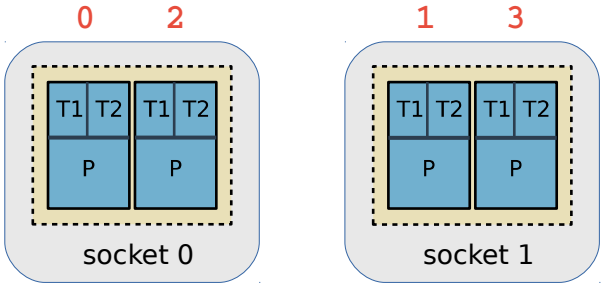
The Intel® runtime library has the ability to bind OpenMP threads to physical processing units. Such binding is known as: ***thread affinity.***

Depending on the topology of the machine, thread affinity can have a great impact to the performance.

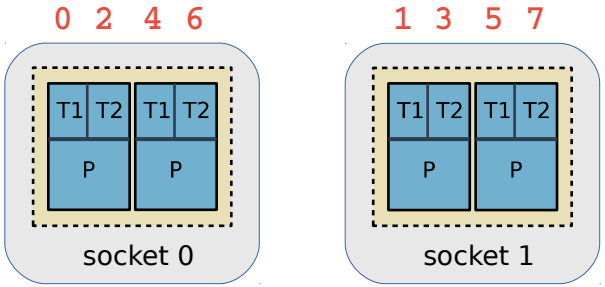
`KMP_AFFINITY=compact, granularity=fine`



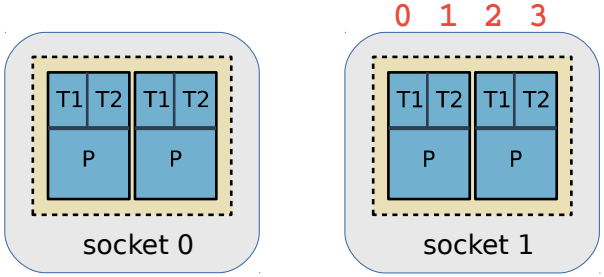
`KMP_AFFINITY=scatter, granularity=core`



`KMP_AFFINITY=scatter, granularity=core`



`KMP_AFFINITY=compact, granularity=fine,0,4`



Hands-on session: Nbody example

code/nbody/ver4/GSimulation.cpp:

```
...
for (int s=1; s<=get_nsteps(); ++s)
{
   ts0 += time.start();
//---------------------------------
// TODO: Parallelize this region
//---------------------------------
// start of parallel region
//---------------------------------
  for (i = 0; i < n; i++)          // update acceleration
  {
  __assume_aligned(particles->pos_x, alignment);
  ...
     for (j = 0; j < n; j++)
       if (i < j || i > j)
       {
         real_type distance, dx, dy, dz;
         real_type distanceSqr = 0.0f;
         real_type distanceInv = 0.0f;

         dx = particles[j].pos[0]  - particles[i].pos[0];
        ...
       }
         // update position and velocity
       ...
//-----------------------------------
// TODO: end of region to parallelize
//-----------------------------------
```

To insert OpenMP pragma

To insert OpenMP pragma
Is there some data-sharing?

Hands-on session

- Go to the folder **code/nbody/ver4**
- In the file: Gsimulation.cpp, try to use Openmp pragmas
- Is there some data-sharing? Do we need a reduction operation?
- Play also changing the thread affinity

#Threads	Runtime [sec]	GFlops	Speedup	Efficiency
1				
2				
3				
4				
6				
12				

Hands-on session: solution

- Go to the folder **code/nbody/ver4**
- In the file: Gsimulation.cpp, try to use Openmp pragmas
- Is there some data-sharing? Do we need a reduction operation?
- Play also changing the thread affinity

Hands-on session: solution

- Go to the folder **code/nbody/ver4**
- In the file: Gsimulation.cpp, try to use Openmp pragmas
- Is there some data-sharing? Do we need a reduction operation?
- Play also changing the thread affinity
- Solution:
  i. cp GSimulation.cpp GSimulation.cpp_bkp
  ii.patch GSimulation.cpp < openmp.patch

Final results of the Nbody example

Version	Optimization / Comments	Performance	% of the Peak E5-2650 v2
base	-O2 / 1 thread	2.28 GFs	5.4% (0.6 %)
ver1	-O3 -xAVX / scalar optimization / 1 thread	6.07 GFs	14.4% (1.8 %)
ver2	#pragma simd / 1 thread	10.7 GFs	25.6% (3.2 %)
ver3	AoS → SoA (no auto-vector) / 1 thread	2.08 GFs	5.0% (0.6%)
ver3_1	removed vector dependence / 1 thread	17.5 GFs	42.1% (5.2%)
ver3_2	alignment allocation / 1 thread	17.8 GFs	42.8% (5.3%)
ver3_2p	__assume_align(…) / 1 thread	20.4 GFs	61.4% (7.6%)
ver4	#pragma omp … / 8 threads / 1 socket	85.8 GFs	25%
ver4	affinity compact, granularity=fine / 16 threads / 1 socket	112.8 GFs	34%