OpenMP Tutorial

Arnamoy Bhattacharyya

Scalable Parallel Computing Laboratory

ETH Zurich

Oct 2, 2014

OpenMP is an API for Parallel Programming

First developed by the OpenMP Architecture Review Board (1997), now a standard

Designed for shared memory multiprocessors

Consists:

1.Set of compiler directives
2.Library functions
3.Environment variables

NOT a language

OpenMP vs MPI

Pros:

- considered by some to be easier to program and debug (compared to MPI)
- data layout and decomposition is handled automatically by directives.
- unified code for both serial and parallel applications: OpenMP constructs are treated as comments when sequential compilers are used.
- original (serial) code statements need not, in general, be modified when parallelized with OpenMP. This reduces the chance of inadvertently introducing bugs and helps maintenance as well.
- both coarse-grained and fine-grained parallelism are possible

Cons:

- currently only runs efficiently in shared-memory multiprocessor platforms
- requires a compiler that supports OpenMP.
- lacks fine-grained mechanisms to control thread-processor mapping.
- synchronization between subsets of threads is not allowed.
- mostly used for loop parallelization

Pros and Cons of MPI

Pros

- does not require shared memory architectures which are more expensive than distributed memory architectures
- can run on both shared memory and distributed memory architectures
- highly portable with specific optimization for the implementation on most hardware

Cons

- requires more programming changes to go from serial to parallel version
- can be harder to debug

OpenMP is based on *Fork/Join* model

When program starts, one *Master* thread

Master thread executed sequential portion of the program

At the beginning of parallel region, master thread *forks* new threads

All the threads together now forms a "*team*"

At the end of the parallel region, the forked threads die

Picture!!

Special Features of OpenMP

Parallelism can be added incrementally

Sequential program is a special case of multi-threaded program

Demo of Hello World OpenMP

```
#pragma omp parallel
  printf(“Hello world\n”);
```

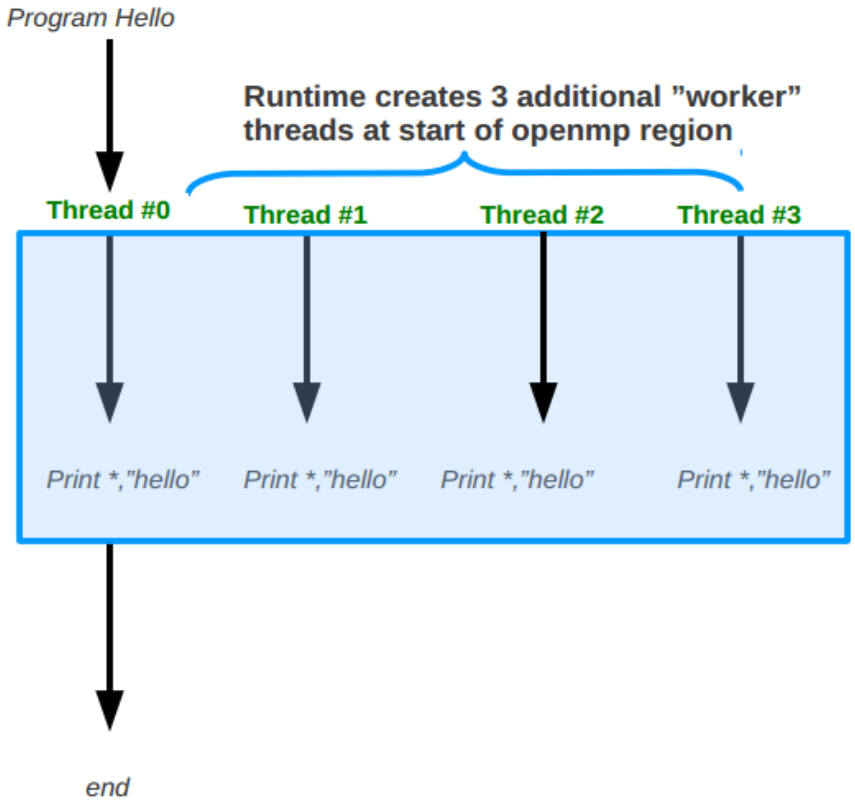
```
MPI_Init (&argc, &argv);/* starts MPI */

printf( "Hello world \n” );

MPI_Finalize();
```

gcc -fopenmp *args*
icc -openmp *args*

At run time



OMP region, every thread executes all the instructions in the OMP block

http://people.math.umass.edu/~johnston/PHI_WG_2014/OpenMPSlides_tamu_sc.pdf

For Loop:

```
#pragma omp parallel
#pragma omp for

for(.....)
```

For Loop:

```
#pragma omp parallel for

for(.....)
```

Tells the compiler that the for loop “immediately” following has to be executed in parallel

The number of loop iterations MUST be computable at runtime

Loop must not contain break, return or exit

Loop must not contain goto label outside of the loop

For Loop:

```
int first, *marked, prime, size;

#pragma omp parallel for

for(i = first; i <size; i+=prime)
    marked[i] = 1;
```

Threads are assigned an independent set of iterations

Thread must wait at the end of the construct

Loops are not enough. Sometimes we may need a BLOCK of code to Be executed in parallel

```
 #pragma omp parallel
{
      doSomeWork(res,M);
      doSomeMoreWork(res,M);
}
```

The for pragma can be used inside a block of code that is marked with parallel pragma

```
#pragma omp parallel
{
	doSomeWork(res,M);
	#pragma omp for
	{
		for(i<-1....M)
			res[i] = huge();
	}
	doSomeMoreWork(res,M);
}
```

There is implicit barrier at the end of the for loop

Most of the time OpenMP is used in case of loops

You have to make sure that the consistency of the program remains.

Floyd's algorithm to solve all-pair shortest path problem

Which Loop to execute in parallel?

```
main()  {
        int i, j, k;
        float **a, **b;

        for(k=0; k<N;k++)
                for(i=0;i<N;i++)
                        for(j=0; j<N;j++)
                                a[i][j] = MIN(a[i][j], a[i][k]+a[k][j]);

}
```

Loop carried dependence

Can be executed in parallel

Can be executed in parallel

There is ***overhead*** at every instance of fork-join

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

We want to maximize the amount of work that is done for each fork join

We parallelize the middle loop for maximum gain.

```
main()  {
        int i, j, k;
        float **a, **b;

        for(k=0; k<N;k++)
                #pragma omp parallel for
                for(i=0;i<N;i++)
                        for(j=0; j<N;j++)
                                a[i][j] = MIN(a[i][j], a[i][k]+a[k][j]);

}
```

But 'j' is shared, which might cause problem

```
main()  {
        int i, j, k;
        float **a, **b;

        for(k=0; k<N;k++)
                #pragma omp parallel for private(j)
                for(i=0;i<N;i++)
                        for(j=0; j<N;j++)
                                a[i][j] = MIN(a[i][j], a[i][k]+a[k][j]);

}
```

Tells the compiler to make the listed variables private
To each thread

Dot Product

```
float dot_product(float *a, float * b, int N)
{
	float sum = 0.0;

	#pragma omp parallel for private(sum)

	for(int i=0; i<N; i++)
		sum += a[i]*b[i];
}
```

Why the private clause will not work in this example?

We will use reduction

reduction(op:list)

A PRIVATE copy of each list variable is created and initialized depending On the “op”

The copies are updated by threads

At the end of construct, local copies are combined through 'op' into A single value and then combined with the original SHARED variable Value.

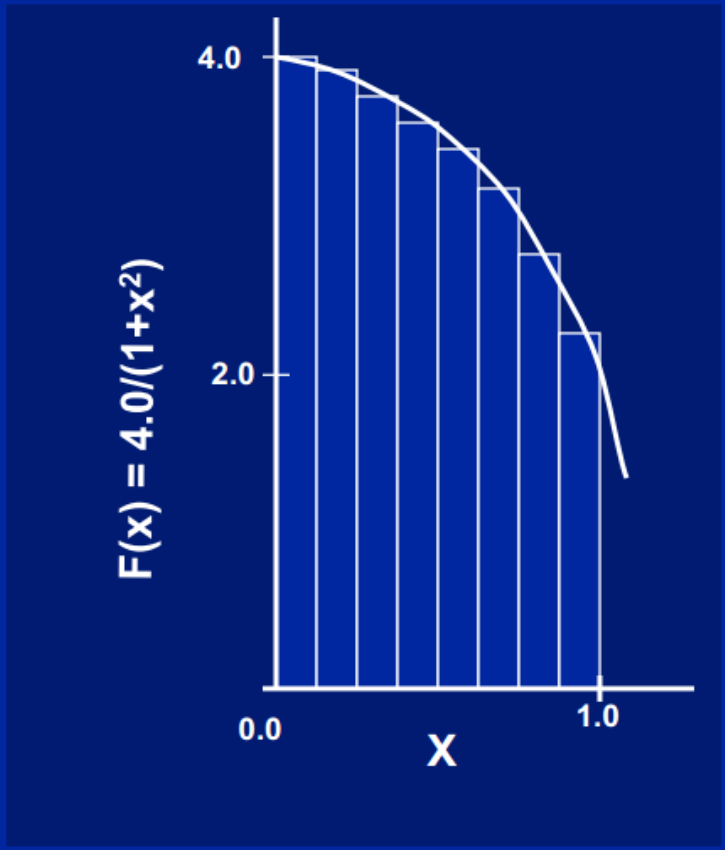
```
float dot_product(float *a, float * b, int N)
{
	float sum = 0.0;

	#pragma omp parallel for reduction(+:sum)
	for(int i=0; i<N; i++)
		sum += a[i]*b[j];
}
```

Local copy of sum in each thread

All local copies are combined at the end and stored in shared copy

Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

http://openmp.org/mp-documents/omp-hands-on-SC08.pdf

```
static long num_rects = 100000;
double step;
int main ()
{
        int i; double x, pi, sum = 0.0;

        step = 1.0/(double) num_rects;

        for (i=0;i< num_rects; i++){
              x = (i+0.5)*step;
              sum = sum + 4.0/(1.0+x*x);
        }
        pi = step * sum;
}
```

What variables can be shared?

num_rects, step

```
static long num_rects = 100000;
double step;
int main ()
{
        int i; double x, pi, sum = 0.0;

        step = 1.0/(double) num_rects;

        for (i=0;i< num_rects; i++){
              x = (i+0.5)*step;
              sum = sum + 4.0/(1.0+x*x);
        }
        pi = step * sum;
}
```

What variables needs to be private?

x, i

```
static long num_rects = 100000;
double step;
int main ()
{
        int i; double x, pi, sum = 0.0;

        step = 1.0/(double) num_rects;

        for (i=0;i< num_rects; i++){
              x = (i+0.5)*step;
              sum = sum + 4.0/(1.0+x*x);
        }
        pi = step * sum;
}
```

What variables are reduced?

sum

```
static long num_rects = 100000;
double step;
int main ()
{
	int i; double x, pi, sum = 0.0;

	 step = 1.0/(double) num_rects;

	#pragma omp parallel for private(x) reduction(+:sum)
	 for (i=0;i< num_rects; i++){
		x = (i+0.5)*step;
		sum = sum + 4.0/(1.0+x*x);
	 }
	 pi = step * sum;
}
```

Synchronization

Barrier

Lock

Ordered

Synchronization: Barrier

Barrier: Each thread waits until all threads arrive.

```
#pragma omp parallel shared (A, B, C) private(id)
{
id=omp_get_thread_num();
A[id] = big_calc1(id);

#pragma omp barrier
#pragma omp for

for(i=0;i<N;i++){C[i]=big_calc3(i,A);}

#pragma omp for nowait

for(i=0;i<N;i++){ B[i]=big_calc2(C, i); }

A[id] = big_calc4(id);
}
```

Synchronization:Locks

```
omp_lock_t writelock;
omp_init_lock(&writelock);

#pragma omp parallel for
for ( i = 0; i < x; i++ )
{
    // some stuff

   omp_set_lock(&writelock);
   // one thread at a time stuff
   omp_unset_lock(&writelock);

   // some stuff
}
omp_destroy_lock(&writelock);
```

Synchronization:Ordered

used when part of the loop must execute in serial order

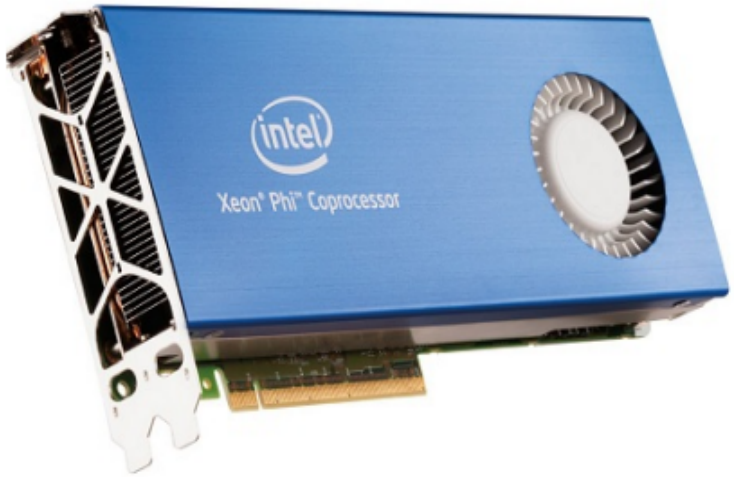
```
#pragma omp parallel for private(myval) ordered
{
   for(i=1; i<=n; i++){
      myval = do_lots_of_work(i);
      #pragma omp ordered
      {
         printf("%d %d\n", i, myval);
      }
   }
}
```

What is Xeon Phi?

You might have heard it referred to as MIC (many integrated cores) or KNC (Knights Corner).

- It is a recently released accelerator based on x86 architecture (Pentium III)
- Designed as a 1-Tflops co-processor

"Now you can think reuse rather than recode with x86 compatibility" (Intel website)



https://hpcforge.org/plugins/mediawiki/wiki/pracewp8/images/6/68/XeonPhi.pdf

What Xeon Phi is

Technical specifications:

- 60 x86-based cores at 1.052Ghz , 4 hardware threads per core,
- Coherent cache,
- 512-bit SIMD, FMAs (1011 Gflops Peak, ~800 measured)
- 8GB (less than 6 in reality) of GDDR5 (320 GB/s peak, ~170 measured)
- PCIe Gen 2
- TDP of 220W
- Lightweight Linux OS, IP addressable (ssh).

This is the key feature of the MIC:

- - Node vector performance is 5 times that of SB node.
- - Core scalar performance is 1/10th of SB core.

https://hpcforge.org/plugins/mediawiki/wiki/pracewp8/images/6/68/XeonPhi.pdf

Xeon Phi vs. GPUs"

There are some similarities between Xeon Phi and GPUs"

- Both require a host CPU"
- Communicate with the host CPU via PCI"
- Very fast at data-parallel computational tasks (like many of those we tackle in HPC)"
- Very slow at other tasks!"
- Allow the host CPU to “offload” work to the device"

But with some interesting differences"

- Based on x86 architecture"
- OpenMP code for x86 can be compiled to run with little or no modification."
- Offer some useful alternatives to offload operation (ssh, native, MPI)"

Some interesting new Pragmas for OpenMP

OpenMP 4.0 for Devices

#pragma omp simd [clauses]

Single threaded loop with vector instructions

Example

```
void sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp simd reduction(+:sum)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```

https://software.intel.com/en-us/articles/openmp-40-for-simd-and-affinity-features-with-intel-xeon-processors-and-intel-xeon-phi

SIMD loop clauses:

safelen(length)

Number of loop iterations that can be executed in SIMD mode
Without dependence violation

```
for(i = 0; i < 2000; i++)
{
        if(i <200)
                arr[i] = val;
        else
                arr[i] = arr[i-1] + 1;
}
```

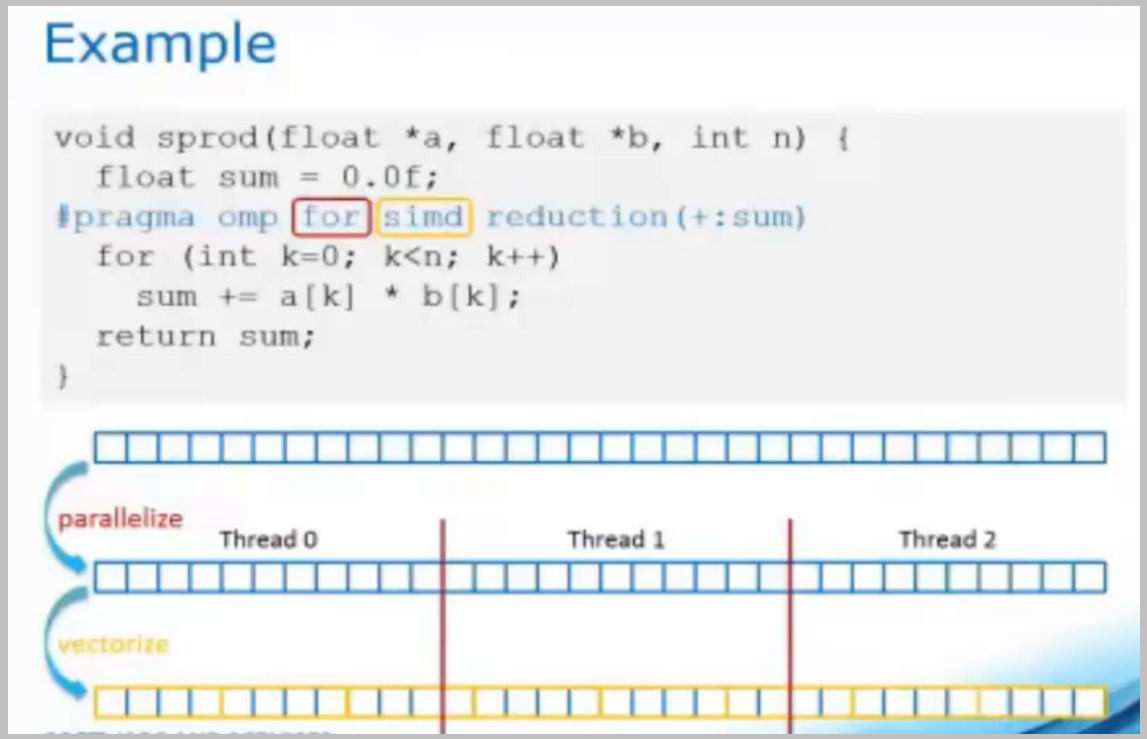
safelen(200)

https://software.intel.com/en-us/articles/openmp-40-for-simd-and-affinity-features-with-intel-xeon-processors-and-intel-xeon-phi

Parallelize and vectorize at the same time:

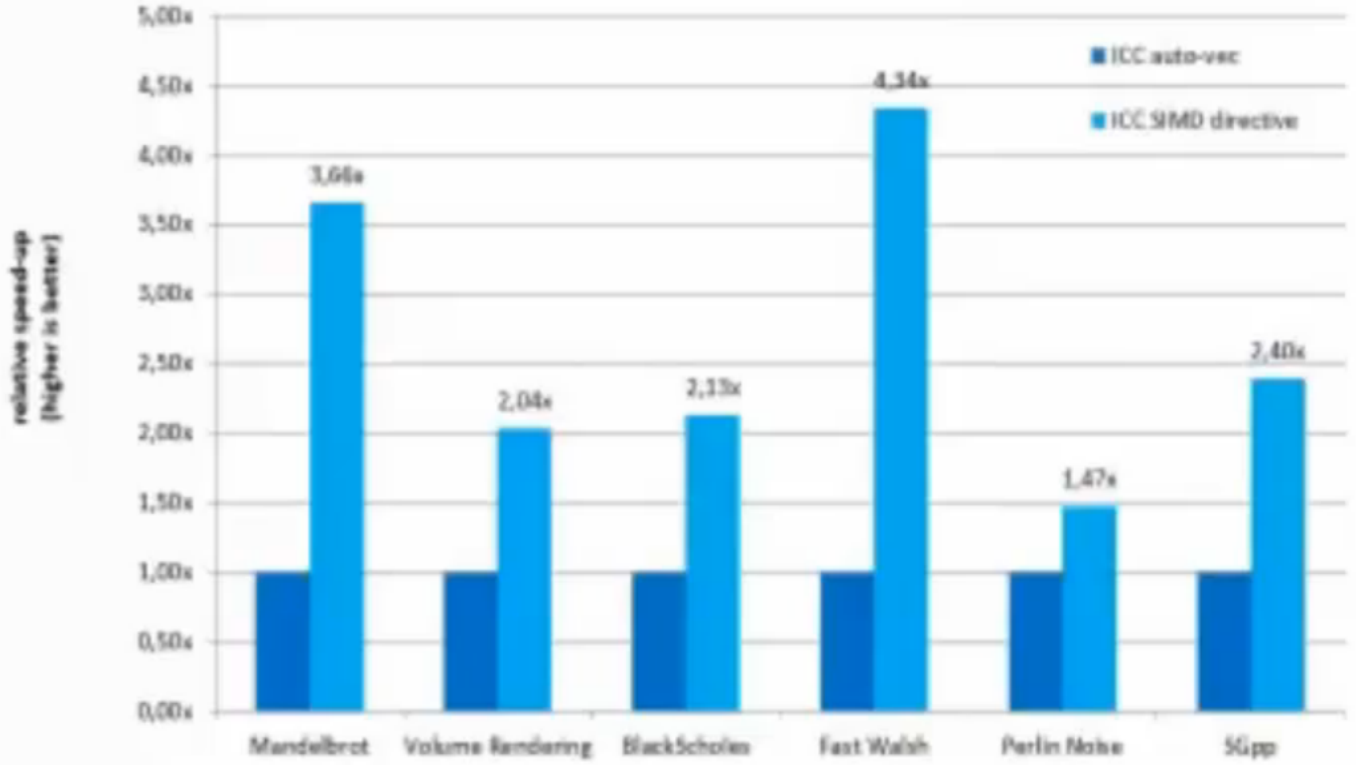
#pragma omp for simd

1. Distribute the loop iterations across thread teams
2. Subdivide loop chunks to fit a vector register



https://software.intel.com/en-us/articles/openmp-40-for-simd-and-affinity-features-with-intel-xeon-processors-and-intel-xeon-phi

SIMD Constructs & Performance



M. Klemm, A. Duran, X. Tian, H. Saito, D. Caballero, and X. Martorell. Extending OpenMP with Vector Constructs for Modern Multicore SIMD Architectures. In Proc. of the Intl. Workshop on OpenMP, pages 59-72, Rome, Italy, June 2012. LNCS 7312.

https://software.intel.com/en-us/articles/openmp-40-for-simd-and-affinity-features-with-intel-xeon-processors-and-intel-xeon-phi

Coprocessor execution

Transfer Control/ data from the host to the device (coprocessor)

#pragma omp target [data] [clause]

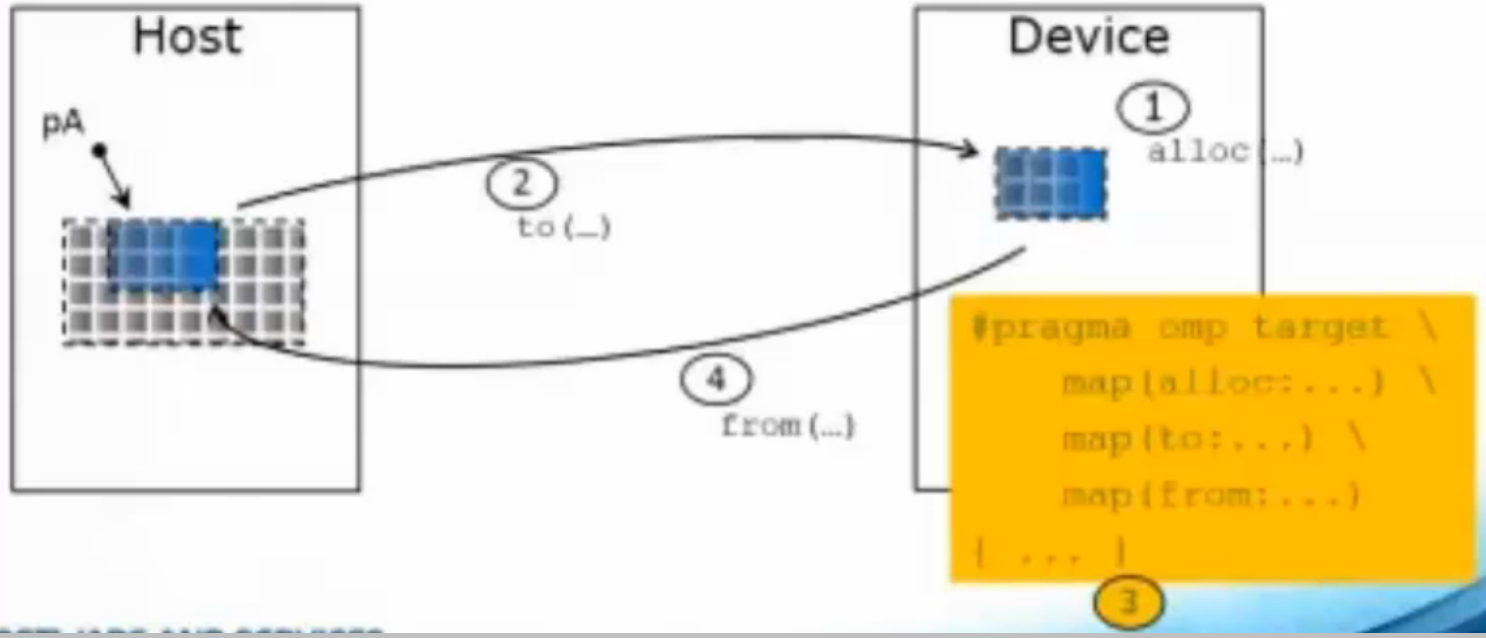
Clauses can be:

device(scalar integer expression)
map(alloc | to | from | tofrom : list)

https://software.intel.com/en-us/articles/openmp-40-for-simd-and-affinity-features-with-intel-xeon-processors-and-intel-xeon-phi

Execution Model

- Data environment is lexically scoped
  - Data environment is destroyed at closing curly brace
  - Allocated buffers/data are automatically released



https://software.intel.com/en-us/articles/openmp-40-for-simd-and-affinity-features-with-intel-xeon-processors-and-intel-xeon-phi

Example

```
#pragma omp target data device(0) map(alloc:tmp[:N]) map(to:input[:N]) map(from:res)
  {
#pragma omp target device(0)
#pragma omp parallel for
    for (i=0; i<N; i++)
      tmp[i] = some_computation(input[i], i);

    update_input_array_on_the_host(input);

#pragma omp target update device(0) to(input[:N])

#pragma omp target device(0)
#pragma omp parallel for reduction(+:res)
    for (i=0; i<N; i++)
      res += final_computation(input[i], tmp[i], i)
  }
```

https://software.intel.com/en-us/articles/openmp-40-for-simd-and-affinity-features-with-intel-xeon-processors-and-intel-xeon-phi

Affinity matters:

Thread affinity in OpenMP 4.0

Concept of places:

1. Set of threads running on one or more processors
2. Can be defined by user
3. Pre-defined places available

threads: one place per hyperthread
cores: One place exists per physical core
sockets: One place per processor package

And affinity policies:

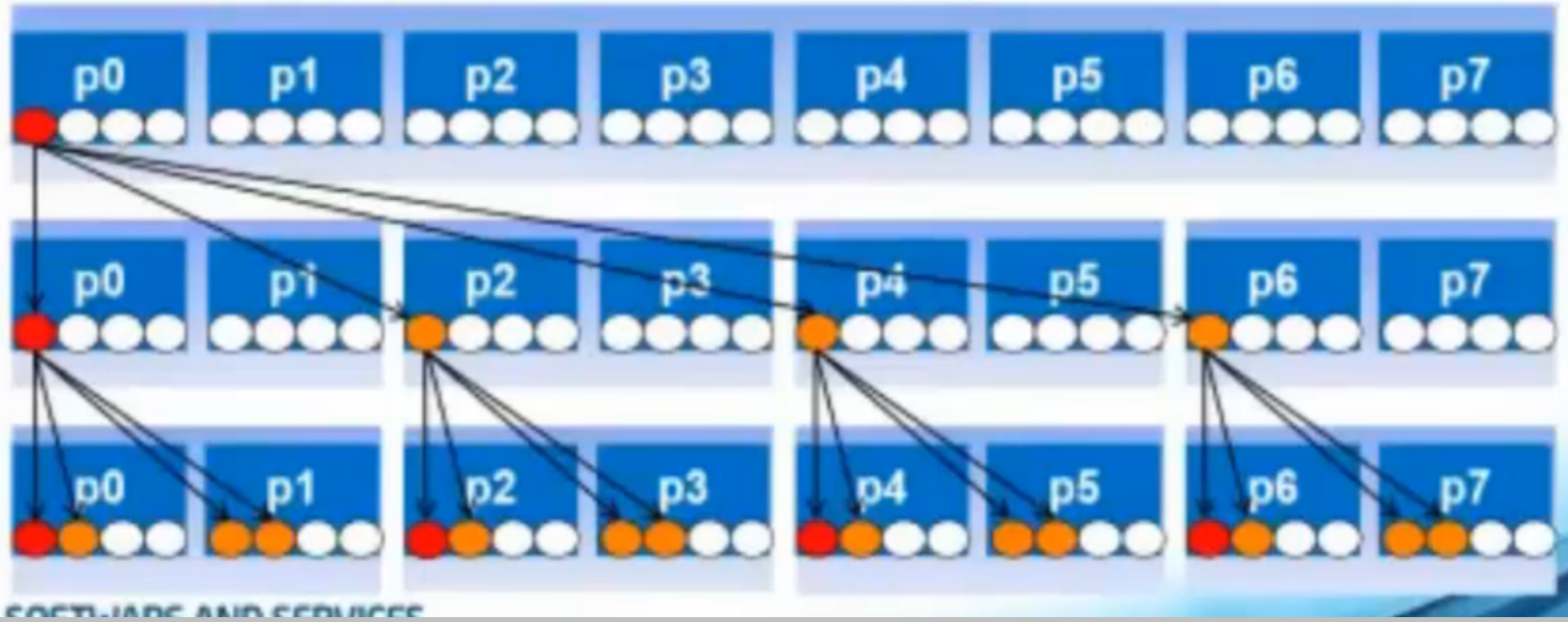
spread: spread openmp threads evenly among the places
close: pack threads close to the Master thread in a place
master: Collocate threads with the Master thread

https://software.intel.com/en-us/articles/openmp-40-for-simd-and-affinity-features-with-intel-xeon-processors-and-intel-xeon-phi

- Example (Intel® Xeon Phi™ Coprocessor):
  Distribute outer region, keep inner regions close

```
OMP_PLACES=cores(8)

#pragma omp parallel proc_bind(spread)
   #pragma omp parallel proc_bind(close)
```



https://software.intel.com/en-us/articles/openmp-40-for-simd-and-affinity-features-with-intel-xeon-processors-and-intel-xeon-phi