Advanced OpenMP

Dr. Nitya Hariharan (Intel)

Preliminaries: Part 1

- Disclosure
  - The material in the lecture slides are adapted from lectures/tutorials by different sources
    - Tim Mattson (Intel)
    - Michael Klemm (Intel)
    - Edward Smyth (NAG)
    - Blaise Barney, LLNL
    - Hernandez et al., OpenMP 5.0/5.1 tutorial at ECP 2020 annual meeting
    - OpenMP 5.0.1 specification and examples https://www.openmp.org/resources/
    - Examples from “Using OpenMP” by Chapman et al.

Preliminaries: Part 2

- The lectures are split into different parts going from basics to advanced topics.
- Large number of OpenMP constructs/clauses, refer to the standard for completeness
- Each section will have some exercises for you to try out.
- As far as possible, attempt the exercises on your own and try out different solutions. And please ask queries during the discussion hour.
- Grey boxes indicate some questions you can try and answer, solutions are given for some of them.
- You will find links to relevant talks or discussion boards in the notes, do have a look

Recap

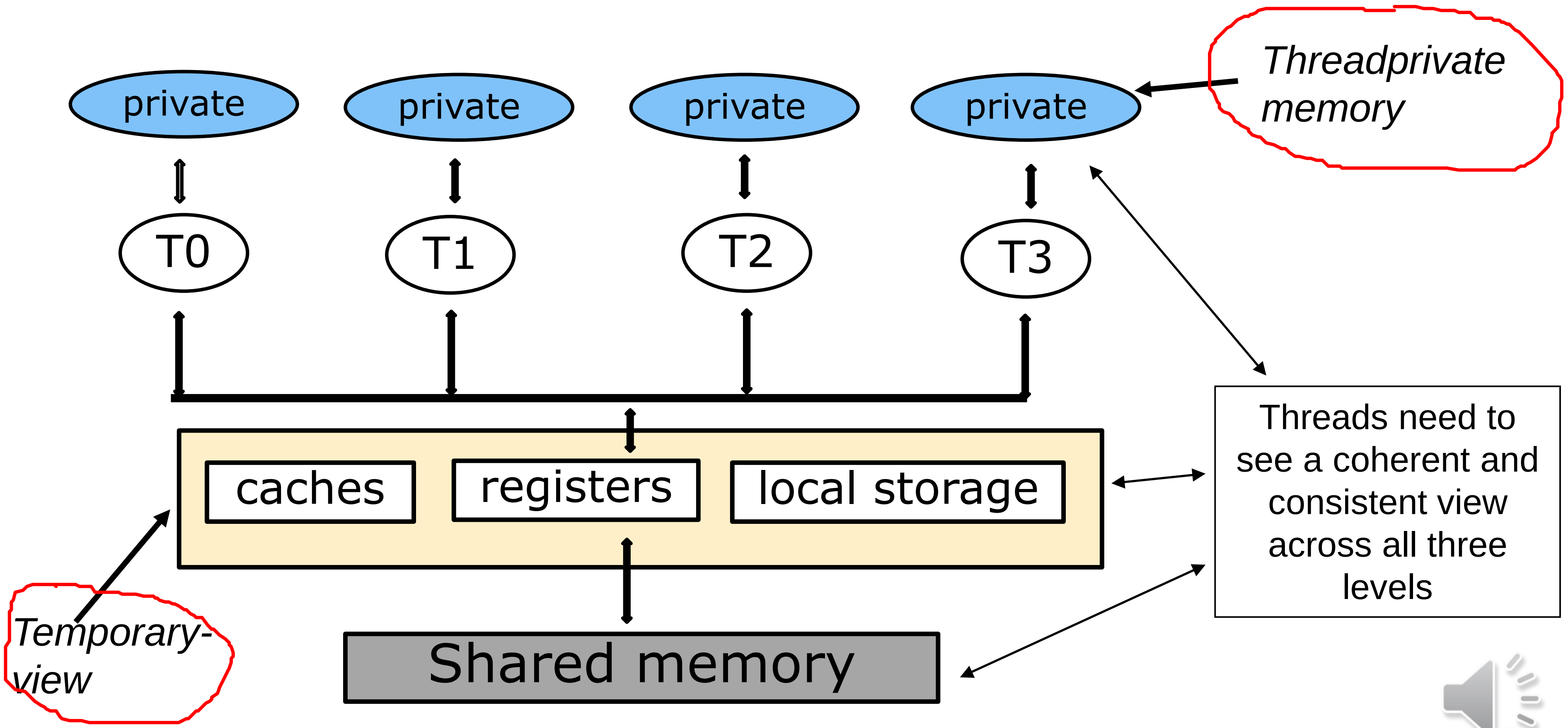
- OpenMP evolution and adoption
- Shared memory programming
- Creating a parallel region and distributing work among threads
- Basic OpenMP clauses

Outline

- OpenMP memory model
- Synchronization – advanced
- TASKS and SECTIONS
- Vectorization constructs
- Thread affinity
  - First touch policy
- OpenMP run time routines
- OpenMP environment variables

OpenMP Memory model

- “Relaxed-consistency” and shared-memory model
- Threads share memory and have their own *threadprivate* data
- *Temporary view* of memory - multiple cache levels and registers



OpenMP relaxed-consistency model

- The *temporary-view* of memory allows threads to cache variables and not refer to memory all the time
- Faster access but makes things complicated – recall false sharing
- The *temporary-view* can be made consistent with memory or across threads at specific points through flush operations
  - Strong flush
  - Release flush
  - Acquire flush

flush operations

- Strong flush –
  - enforce consistency of a **thread's** *temporary-view* with memory
  - flush applied to set of variables called *flush-set*
  - restricts reordering of memory operations (compiler flags sometimes do this)

- Release or acquire flushes – enforce consistency of the memory views of **two synchronizing threads**
  - Both work in tandem
  - A release flush will propagate values of shared variables to memory that other threads can then use an acquire flush to read from
  - An acquire flush will discard values of shared variables in *temporary-view,* use value propagated by the release flush
  - Implicit acquire and release flush on entry and exit, respectively, from a CRITICAL construct

flush operations

- Implied by synchronization during
  - entry/exit of parallel regions or critical regions
  - implicit and explicit barriers

- Explicitly flush through the OpenMP FLUSH construct

  #pragma omp flush [memory-order-clause] [(list)]

  where *memory-order-clause* can be
  - *acq_rel* – both acquire and release flush
  - *release* – release flush
  - *acquire* - acquire flush

- FLUSH construct without a *memory-order-clause* results in –
  - A strong flush - *temporary-view* of only the thread executing FLUSH is made consistent
  - FLUSH without a list flushes all shared variables within the structured block
  - FLUSH with a list acts on only the listed variables

- Revisit this while discussing synchronization

Outline

- OpenMP memory model
- Synchronization
- TASKS and SECTIONS
- Vectorization constructs
- Thread affinity
  - First touch policy
- OpenMP run time routines
- OpenMP environment variables

Synchronization

- Used to impose order constraints and to protect access to shared data
- Control what data each thread can access and when

```
for(i=0; i<n;i++)
{
new_elem = ..;
ptr = head->next;
prev = head;
//insert new_elem
}
```

Q

- What happens when two threads try and update the pointers at the same time?
- How do you parallelize such loops?

- Use synchronization clauses
  - CRITICAL
  - BARRIER
  - More advanced ATOMIC or FLUSH constructs

OpenMP CRITICAL clause

- Mutual exclusion: Only one thread at a time can enter a **critical** region.

Threads wait their turn, and update the list one at a time $\longrightarrow$

```
#pragma omp parallel
for(i=0; i<n;i++)
{
new_elem = ..;
#pragma omp critical
 {
    ptr = head->next;
    prev = head;
    //insert new_elem
  }
}
```

- This is almost like a serial code with the whole parallel region inside a critical region. This can be improved with locks (later slides).

OpenMP BARRIER clause

- Threads wait till other threads have reached the barrier.
- Conditional clauses not allowed

Threads wait for others $\longrightarrow$

```
#pragma omp parallel
for(i=0; i<n;i++)
{
    //do calculation
#pragma omp barrier
    //compare results
  }
}
```

Synchronization – ATOMIC construct

- ATOMIC construct
  - ensures that a memory locations is accessed “*atomically*”
  - Only one thread carries out the operation

#pragma omp atomic atomic-clause memory-order-clause expression-statement

- where atomic-clause can be *read, write, update, capture*
- memory-order-clause can be *seq_cst, acq_rel, release, acquire, relaxed*
- default atomic-clause is *update*

```
#pragma omp atomic read
 new_val = x;
```

x cannot be updated until operation finishes

```
#pragma omp atomic write
   x = new_val;
```

x cannot be read until operation finishes

ATOMIC construct

```
#pragma omp atomic update
 x++;
```

read and write to *x* is atomic

```
#pragma omp atomic capture
  v = x++;
```

Operation on *x* is atomic, updated value written to *v*

- Strong flush implied at the entry and exit from the atomic operation

- Can use ATOMIC for memory consistency between threads
  - Recall the release and acquire flush (see slide)
  - release flush - *write/update/capture* clause with *release, acq_rel, seq_cst*
  - acquire flush – *read/capture* clause with *acquire, acq_rel, seq_cst*

Synchronization – FLUSH construct

- We looked at strong flushes, release and acquire flush
- OpenMP has an implicit flush in shared constructs, an explicit flush is not really needed
- Using the FLUSH construct can be error prone if not used correctly, see code below

```
a = b = 0

Thread 1

atomic(b=1)
flush(b)
flush(a)
atomic(tmp=a)
if(tmp==0) then
   protected section
endif
```

```
a = b = 0

Thread 2

atomic(a=1)
flush(a)
flush(b)
atomic(tmp=b)
if(tmp==0) then
   protected section
endif
```

Note – the compiler can move *flush(b)* on thread 1 and *flush(a)* on thread 2 to after the protected section, if neither a or b is used within it.

Q

What happens if the compiler reorders the flush statements to after the protected section?

Synchronization – FLUSH construct

- There is nothing preventing both threads from running the code in the protected section.
- The corrected code below ensures only one thread runs the code in the protected section at any given time

```
a = b = 0

Thread 1

atomic(b=1)
flush(a,b)
atomic(tmp=a)
if(tmp==0) then
   protected section
endif
```

```
a = b = 0

Thread 2

atomic(a=1)
flush(a,b)
atomic(tmp=b)
if(tmp==0) then
   protected section
endif
```

- Compiler cannot reorder the flush statements for *a* or *b*
- Data assignment is complete and is flushed before *if* statement.

- OpenMP does a good job with implicit flushes, use an explicit FLUSH **only** if you need it

Synchronization – ORDERED construct

- Specifies that structured block will be executed in order of loop iterations
- Can be a stand-alone directive with specified cross-iteration dependencies
- Makes the loop sequential with code outside the loop running in parallel

#pragma omp ordered [clause [[,] clause]]
structured block

clause can be *threads, simd*

#pragma omp ordered clause [[[,] clause] ...]

clause can be *depend(source)*
depend(sink: vec)

```
#pragma omp parallel for ordered shared(pos)
for(int i=1; i< n; i++)
{
#pragma omp ordered
print_neighbor(pos[i]);
}
```

iteration order maintained across threads and within a thread

ORDERED construct

- Can use the *depend* clause to specify cross-iteration dependencies

```
#pragma omp parallel for ordered shared(pos)
for(int i=1; i< n; i++)
{
#pragma omp ordered depend(sink: i-1)
neighbor[i] = get_neighbor(atom[i], atom[i-1]);
#pragma omp ordered depend(source)
}
```

depend(sink) requires iteration *i-1* to complete before *i*

depend(source) indicates completion of iteration *i* to satisfy cross-iteration dependencies

```
#pragma omp parallel for ordered shared(pos)
for(int i=1; i< n; i++)
{
#pragma omp ordered depend(sink: i-1, i+1)
neighbor[i] = get_neighbor(atom[i], atom[i-1], atom[i+1]);
pragma omp ordered depend(source)
}
```

This is incorrect as dependency is on lexicographically later iteration *i+1*, this might not happen before *i*

Synchronization - LOCK routines

- OpenMP provides a set of lock routines for synchronization

- Operate on OpenMP locks represented by lock variables

- Lock can be different states
  - *uninitialized, unlocked* and *locked*
  - if lock is in *unlocked* state, a task can **set** the lock, and change it to *locked*

- *simple* and *nestable* locks supported
  - *simple* locks can be only set once by the task that owns it
  - *nestable* locks can be set multiple times by owning task, nesting count maintained
  - *simple* and *nestable* lock routines available, respectively, for both

- OpenMP takes care of ensuring routines read and update the most current value of lock variable, no FLUSH operations needed

Lock routines

- *omp_init_[nest]_lock* - initializes a *simple* lock, no task owns it at this stage
- *omp_destroy_[nest]_lock* - uninitializes a *simple* lock
- *omp_set_[nest]_lock* - waits until a *simple* lock is available and then sets it
  - The task is suspended until the lock is available
- *omp_unset_[nest]_lock* - unsets a *simple* lock
- *omp_test_[nest]_lock* - tests a *simple* lock and sets it if it is available
  - The task is not suspended in this case
- The routines with “*nest*” are for *nestable* locks

Ownership of locks

- Till OpenMP 2.5 locks were owned by threads
  - Lock unset by *omp_unset_lock* must be executed by thread that owns the lock
- OpenMP 3.0 onwards, locks are owned by task regions
  - Lock unset by *omp_unset_lock* in a task region must be owned by same task region

```
omp_init_lock(&lock);
omp_set_lock(&lock);

#pragma omp parallel for
for(int i=0;i<n;i++)
{
#pragma omp master
{
omp_unset_lock(&lock);
}
omp_destroy_lock(&lock);
```

Q

- The code is conforming as per OpenMP 2.5 and not per OpenMP 3.0. Why?

Simple locks - example

```
int find_total_mass()
{
omp_lock_t lock;
omp_init_lock(&lock);
#pragma omp parallel for
for(int i=0;i<n;i++)
{
omp_set_lock(&lock);
  total_mass += mass[i];
omp_unset_lock(&lock);

while (!omp_test_lock(&lock)) {
  find_active_neighbors(list);
}

total_mass = sum_neighbour_mass(list);
omp_unset_lock(&lock);
}

omp_destroy_lock(&lock);
return total_mass;
}
```

initialize lock, not owned by any task here

only one thread can add to *total_mass*

do something else till we have the lock

we have the lock, so do some work

release the lock

locks vs CRITICAL

```
#pragma omp parallel
for(int i=0;i<n;i++)
{
  //get random number between 1 and 10
ind = get_random_number(1,10);
arr[ind] = do_something(arr[ind]);
if(check_condition(arr[ind]))
{
  check = false;
} }
```

The user needs to ensure two threads don't access the same index *ind*

You could put a critical region around it, but it will be an overkill, threads won't generate same *ind* value all the time

```
omp_init_lock(&lock[i]); //n locks
#pragma omp parallel for
for(int i=0;i<n;i++)
{
  //get random number between 1 and 10
ind = get_random_number(1,10);
omp_set_lock(&lock[ind]);
   arr[ind] = do_something(arr[ind]);
   if(check_condition(arr[ind]))
  {
     check = false;
  }
omp_unset_lock(&lock[ind]);
}
```

Put the lock around an individual array item, based on index *ind*, the rest of the threads need not wait if *ind* is not the same

In general

- Too many locks or atomic regions or critical sections are bad for scalability
- Too many locks or atomic regions also make the code error prone and difficult to debug
- Let OpenMP handle things for you as far as possible, without using such explicit constructs
- If you are using too many locks or atomic operations or critical regions, it doesn’t mean you are an expert in OpenMP. It means you need to redesign your code ☺

Outline

- OpenMP memory model
- Synchronization
- TASKS and SECTIONS
- Vectorization constructs
- Thread affinity
  - First touch policy
- OpenMP run time routines
- OpenMP environment variables

OpenMP TASK construct

- Tasking constructs provide units of work to threads
- This is done by work-sharing constructs too, but they look at “data parallel”, while tasks look at “task parallel”
  - Typically in “data-parallelism” we want to have spatial and temporal locality
  - In “task-parallelism” each thread has an individual unit of work which can involve non-locality and irregular memory access
- Can use this to split up work, for example, traversing nodes in a list or walking through a graph
  - Generate nested tasks if the graph has child nodes and process each node within a separate task

OpenMP TASK

```
void walk(node *p)
{
  if(p->left)   //does node have left child
#pragma omp task
     walk(p->left);

  if(p->right)      //does node have right child
#pragma omp task
    walk(p->right);

  sum_weight(p);  //sum weight of current node
}

void main()
{
..
node *p = head;
#pragma omp parallel
walk(p);
}
```

- *p* is FIRSTPRIVATE by default
- The tasks are not executed in a particular order, there's no guarantee *sum_weight* will not be called before the left and right nodes have been traversed fully and their weights added.
- Note that the calls to the *walk* routine are recursive.

OpenMP TASKWAIT construct

```
void walk(node *p)
{
  if(p->left)   //does node have left child
#pragma omp task
     walk(p->left);

  if(p->right)      //does node have right child
#pragma omp task
    walk(p->right);
  #pragma omp taskwait
  sum_weight(p);   //sum weight of current node
}

void main()
{
..
node *p = head;
#pragma omp parallel
walk(p);
}
```

- The TASKWAIT construct specifies a wait on the completion of the child tasks of the current task
- Adding a TASKWAIT to the code shown gives the right post ordering traversal of the graph
- The tasks that are created for left and right traversal of the graph will have completed before *sum_weight* on current node is called

Generating large number of tasks

```
void generate ()
{
 const int num_elem=9999999;
 int arr[num_elem];
#pragma omp parallel
#pragma omp single
{
   for(int i=0;i<num_elem;i++)
    #pragma omp task
     check(arr[i]);
}
}
```

- If the number of tasks reaches a limit then the thread generating the tasks, say the *"parent"* thread, can be stopped from creating further tasks and starts executing unassigned tasks.
- Once the number of unassigned tasks is low, the *"parent"* thread can start generating tasks again.
- While executing unassigned tasks, if the *"parent"* thread takes a long time to finish its work, the other threads idle till the *"parent"* thread is done –the tasks are *"tied"* to the *"parent"*.

```
#pragma omp parallel
#pragma omp single
{
   for(int i=0;i<num_elem;i++)
    #pragma omp task untied
     check(arr[i]);
}
}
```

The *untied* clause allows any thread to resume the task generating loop

TASKWAIT vs TASKGROUP

```
void generate ()
{
#pragma omp parallel
#pragma omp single
{
#pragma omp task
{
printf("task 1\n");
#pragma omp task
{
printf("task 2\n");
}
}
#pragma omp taskwait
#pragma omp task
{
printf("task 3\n");
}
}
```

- TASKWAIT construct suspends a thread till all the child tasks generated before the TASKWAIT region are completed.
- With TASKGROUP, the thread waits till all the child tasks and their descendant tasks complete execution.
- Here the TASKWAIT will wait for task 1 to be completed before task 3 is scheduled. The TASKWAIT is bound to the parallel region, of which tasks 1 and 3 are child tasks.
- Task 2 is a child of task 1, the TASKWAIT construct doesn't wait for it to finish before task 3 is executed.

TASKWAIT vs TASKGROUP

```
#pragma omp parallel
#pragma omp single
{
#pragma omp taskgroup
{
#pragma omp task
{
printf("task 1\n");
#pragma omp task
{
printf("task 2\n");
}
}
} */end taskgroup
#pragma omp task
{
printf("task 3\n");
}
}
```

The TASKGROUP construct ensures task 2, which is a child of task 1, is also completed before task 3 is scheduled.

Some points about TASKS

- Think of a producer-consumer relation, a thread within a parallel region generates the tasks and adds them to a queue.
- All the threads in that parallel region, including the one generating the tasks, consume it.
- The TASK construct makes is easier to parallelize something like a while loop, where the number of iterations might not be known beforehand. Or even something recursive like calculating Fibonacci or traversing a graph.
- The runtime can decide if execution of the task is immediate or delayed, but the user can synchronize completion of tasks – BARRIER/TASKWAIT/TASKGROUP.
- The usual process is to have a single thread generate tasks, and all threads execute them – notice the use of *#pragma omp single*

Q

Can the MASTER clause be used here, what extra care do you need to take in such a case?

OpenMP SECTIONS

- We looked at some work-sharing constructs earlier – recall *#pragma omp parallel for*

- SECTIONS is another construct to allows a set of structured blocks to be shared between threads. Each structured block is executed by one thread

```
#pragma omp sections [clause [[,] clause] …]
    {
    [#pragma omp section]
      structured block
    [#pragma omp section]
      structured block
    }
```

clause can be PRIVATE, FIRSTPRIVATE, LASTPRIVATE, REDUCTION, NOWAIT

```
#pragma omp sections
{
    #pragma omp section
      neighbors_xaxis();

    #pragma omp section
      neighbors_yaxis();
}
```

- Implicit barrier at the end of each SECTIONS, unless NOWAIT clause specified. Only one NOWAIT clause allowed within a SECTIONS construct

OpenMP SECTION – FIRSTPRIVATE clause

```
int neighbor_count=0;
#pragma omp parallel
#pragma omp sections firstprivate(neighbor_count)
{
    #pragma omp section {
     neighbor_count += neighbors_xaxis();
    printf("neighbors incl x-axis %d\n", neighbor_count);
    }
    #pragma omp section {
    neighbor_count +=  neighbors_yaxis();
    printf("neighbors incl y-axis %d\n", neighbor_count);
    }
}
```

Q

- Say number of neighbors in both x and y-axis is 5 each, what value will be printed?
- Will the code give repeatable results?

OpenMP TASKS vs SECTIONS

- We saw both TASKS and SECTIONS, and both of them seem to be used for task based parallelism
- In both cases, a thread is executing an independent piece of work, so how are they different?
- **Exercise** – find out how TASKS and SECTIONS are different. Can you use that information to say which would be better for performance?

Outline

- OpenMP memory model
- Synchronization
- TASKS and SECTIONS
- Vectorization constructs
- Thread affinity
  - First touch policy
- OpenMP run time routines
- OpenMP environment variables

OpenMP SIMD support

- SIMD execution – Single Instruction Multiple Data
  - makes use of vector processing units or SIMD units
  - processors nowadays have 512 bit or 256 bit vector units
- OpenMP SIMD construct allows execution in SIMD units for loops with no loop-carried backward dependency
  - Assure the compiler that loop can be vectorized
  - Supported since 4.0
- *declare SIMD* construct can be used for function calls
  - replace scalar function call with vector version
  - vectorize loops with function calls
  - supported since OpenMP 4.5
- Work-sharing SIMD constructs allows to vectorize parallel loops
  - *omp for simd*
  - simultaneous thread execution in multiple SIMD units

OpenMP SIMD construct

```
#pragma omp simd [clause[[,]clause]..]
    for loops
```

where *clause* can be

- *safelen(N)* – cross-iteration dependencies only for vectors above size *N*
- *simdlen(N)* – *N* number of iterations that can be processed concurrently
- *aligned(x:N)* – variable *x* in the list is aligned to *N* number of bytes
- *collapse* – number of loops to be collapsed into one SIMD construct
- *private, lastprivate, reduction* and more

```
#pragma omp for simd [clause[[,]clause]..]
    for loops
```

Work-sharing SIMD construct

where *clause* can be any of the clauses accepted by the *for* or *simd* construct

OpenMP SIMD construct

```
#pragma omp simd
for (i=0;i<n;i++)
  a[i] = b[i] + c[i] * d
```

```
#pragma omp simd private(temp) reduction(+:sum)
  for (i=0;i<n;i++)
   temp = a[i] * b[i];
   sum = sum  + temp;
```

```
#pragma omp simd safelen(8)
    for (i=m;i<n;i++)
    a[i] = a[i-m] * b[i];
```

If you know that *a[i]* will only depend on values 8 elements, or less, away in the array *a*, use *safelen*. This helps the compiler to vectorize the loop accordingly.

```
#pragma omp simd simdlen(16)
    for (i=0;i<n;i++)
    c[i] = a[i] * b[i];
```

Vectorize the loop and process 16 single-precision elements concurrently (512 bit vector processing)

```
#pragma omp simd collapse(2)
    for (i=0;i<n;i++)
     for (j=0;j<m;j++)
      c[i,j] = a[i,j] + b[i,j];
```

Collapse *i* and *j* loops and vectorize over *n*m* iterations

OpenMP work-sharing SIMD construct

```
#pragma omp for simd collapse(2)
  for (i=0;i<n;i++)
    for (j=0;j<m;j++)
     c[i,j] = a[i,j] + b[i,j];
```

Collapse *i* and *j* loops and distribute across threads, vectorize over *n*m/num_threads* iterations per thread

```
#pragma omp for simd simdlen(16)
  for (i=0;i<n;i++)
   c[i] = a[i] * b[i];
```

Distribute loop across threads and process 16 single-precision elements concurrently, per thread

```
#pragma omp for simd align(a,b,c:64)
  for (i=0;i<n;i++)
   c[i] = a[i] * b[i];
```

Distribute loop across threads and vectorize iterations per thread. Align *a,b,c* at 64 byte boundary, this makes the vectorization more efficient. Refer to peel, main and remainder loops during vectorization.

OpenMP declare SIMD construct

#pragma omp declare simd [clause[[,]clause]..]

where *clause* can be

- *simdlen(N)* – *N* number of iterations that can be processed concurrently
- *aligned(x:N)* – variable *x* in the list is aligned to *N* number of bytes
- *linear* – value of variable in list changes linearly with iteration
- *uniform* – variable has invariant/constant value across all invocations of the function/subroutine

- apply to a function (C/C++/Fortran) or subroutine (Fortran)
- process multiple arguments from one invocation in a SIMD loop
- Compiler can create faster, vector code for loop by guaranteeing SIMD properties of the called function/subroutine

OpenMP declare SIMD construct

```
#pragma omp declare simd
int add_values(x,y)
{
return x+y;
}
```

Compiler can inline function *add_values* and vectorize it across the loop over *n*

```
#pragma omp simd private(temp) reduction(+:sum)
    for (i=0;i<n;i++)
    sum = sum  + add_values(a[i], b[i]);
```

```
#pragma omp simd uniform(increment)
int add_increment(x,y)
     return x+y+increment;
```

Value of *increment* remains same across SIMD lanes

```
#pragma omp declare simd linear(i:1)
int add_arrays(int *x, int *y)
     return x[i]+y[i]+increment;
```

Guarantee to the compiler that *x* and *y* are accessed in unit-strides, compiler generates loads accordingly

Outline

- OpenMP memory model
- Synchronization
- TASKS and SECTIONS
- Vectorization constructs
- Thread affinity
  - First touch policy
- OpenMP run time routines
- OpenMP environment variables

Thread Affinity and Data Locality

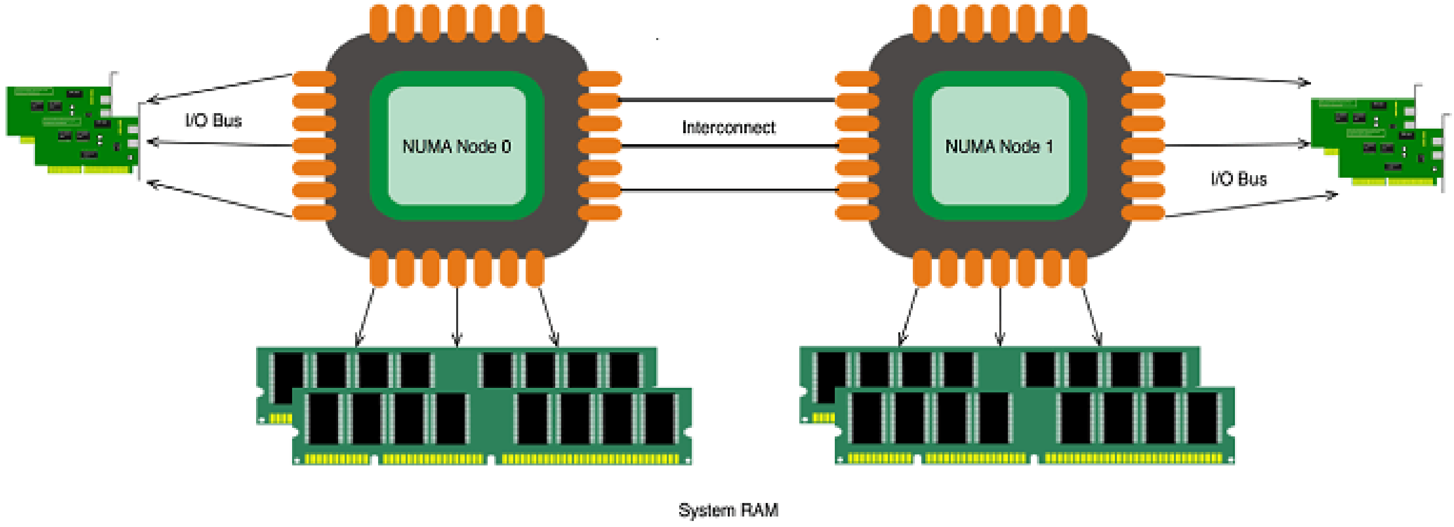
- Affinity
  - Process Affinity: bind processes to CPUs
  - Thread Affinity: bind threads to CPUs that are allocated to their parent process

- Data Locality
  - Memory Locality: allocate memory as close as possible to the core on which the task requesting the memory is running.
  - Cache Locality: use data in cache as much as possible

- Correct process, thread and memory affinity is the basis for getting optimal performance.

Memory Locality

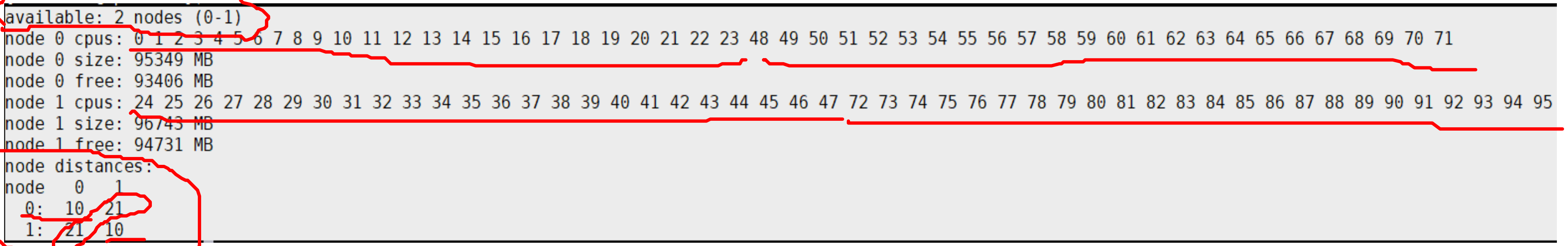
- Most systems today are Non-Uniform Memory Access (NUMA)
  - The physical memory attached to the processor is a NUMA “domain”
- When you access memory attached to the other processor, the data must cross the interconnect
  - “remote access” with longer access times
- The access is “non-uniform“ – this has an impact on how you program



*https://software.intel.com/content/www/us/en/develop/articles/use-intel-quickassist-technology-efficiently-with-numa-awareness.html

How to check "NUMA-ness" of a node

- **numactl**: controls NUMA policy for processes or shared memory
  - **numactl -H**: provides NUMA info of the CPUs



```
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
node 0 size: 95349 MB
node 0 free: 93406 MB
node 1 cpus: 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
node 1 size: 96743 MB
node 1 free: 94731 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
```

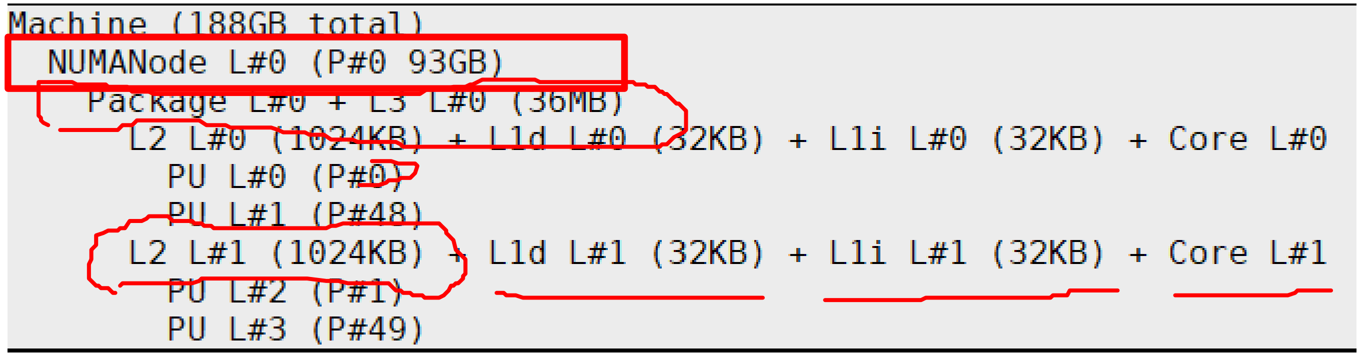
- An Intel Cascade Lake (CLX*) node has 24 physical cores (48 logical cores) per socket.
- Two NUMA domains per node, 24 cores per NUMA domain, two hardware threads (CPUs) per core.
- Memory bandwidth is non-homogeneous among NUMA domains.
  - CPUs 0-23, 48-71 are closer to memory in NUMA domain 0
  - CPUs 24-47, 72-95 are closer to memory in NUMA domain 1

*CLX: 24-core **Intel® Xeon® Platinum 8268 @ 2.9 GHz**

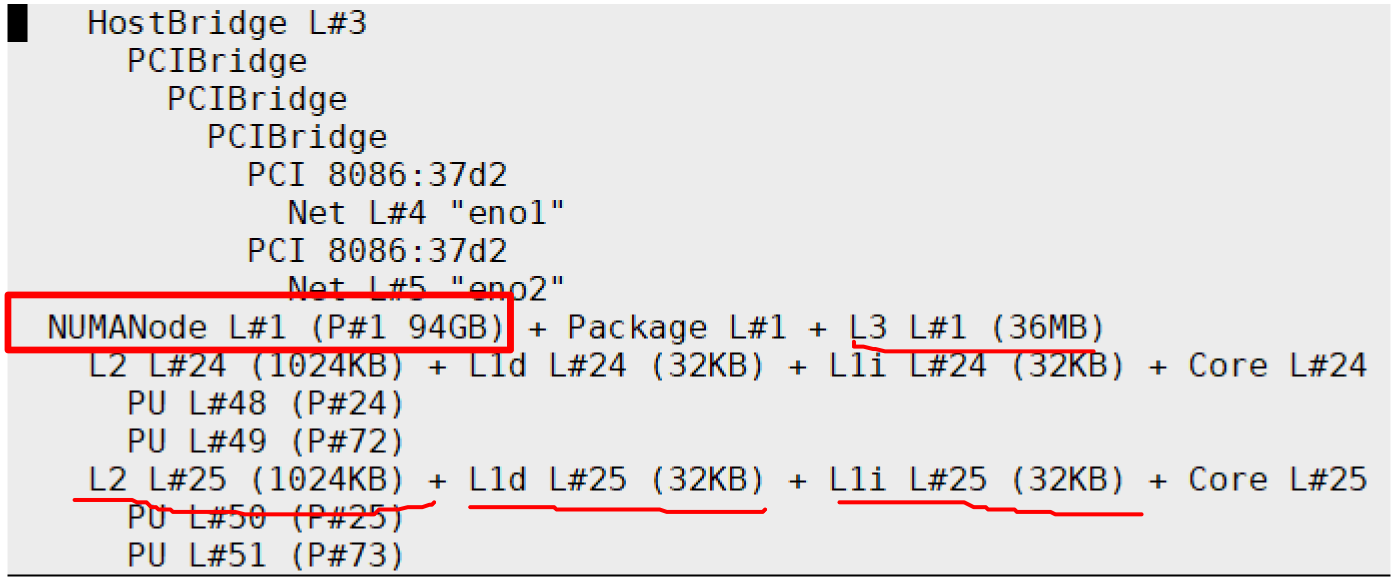
How to check “NUMA-ness” of a node

- **Portable Hardware Locality (hwloc)**
  - **hwloc-ls:** you can also use this tool to get information about the system topology, NUMA nodes, cache info, and the mapping of procs.

% hwloc-ls



```
Machine (188GB total)
  NUMANode L#0 (P#0 93GB)
    Package L#0 + L3 L#0 (36MB)
      L2 L#0 (1024KB) + L1d L#0 (32KB) + L1i L#0 (32KB) + Core L#0
        PU L#0 (P#0)
        PU L#1 (P#48)
      L2 L#1 (1024KB) + L1d L#1 (32KB) + L1i L#1 (32KB) + Core L#1
        PU L#2 (P#1)
        PU L#3 (P#49)
```



```
    HostBridge L#3
      PCIBridge
        PCIBridge
          PCIBridge
            PCI 8086:37d2
              Net L#4 "eno1"
            PCI 8086:37d2
              Net L#5 "eno2"
  NUMANode L#1 (P#1 94GB) + Package L#1 + L3 L#1 (36MB)
    L2 L#24 (1024KB) + L1d L#24 (32KB) + L1i L#24 (32KB) + Core L#24
      PU L#48 (P#24)
      PU L#49 (P#72)
    L2 L#25 (1024KB) + L1d L#25 (32KB) + L1i L#25 (32KB) + Core L#25
      PU L#50 (P#25)
      PU L#51 (P#73)
```

How does all this NUMA-ness affect performance?

Without first-touch: serial data initialization

```
 for (j=0; j<N; j++) {
a[j] = 1.0; b[j] = 2.0; c[j] = 0.0;}
```

With first-touch: || data initialization

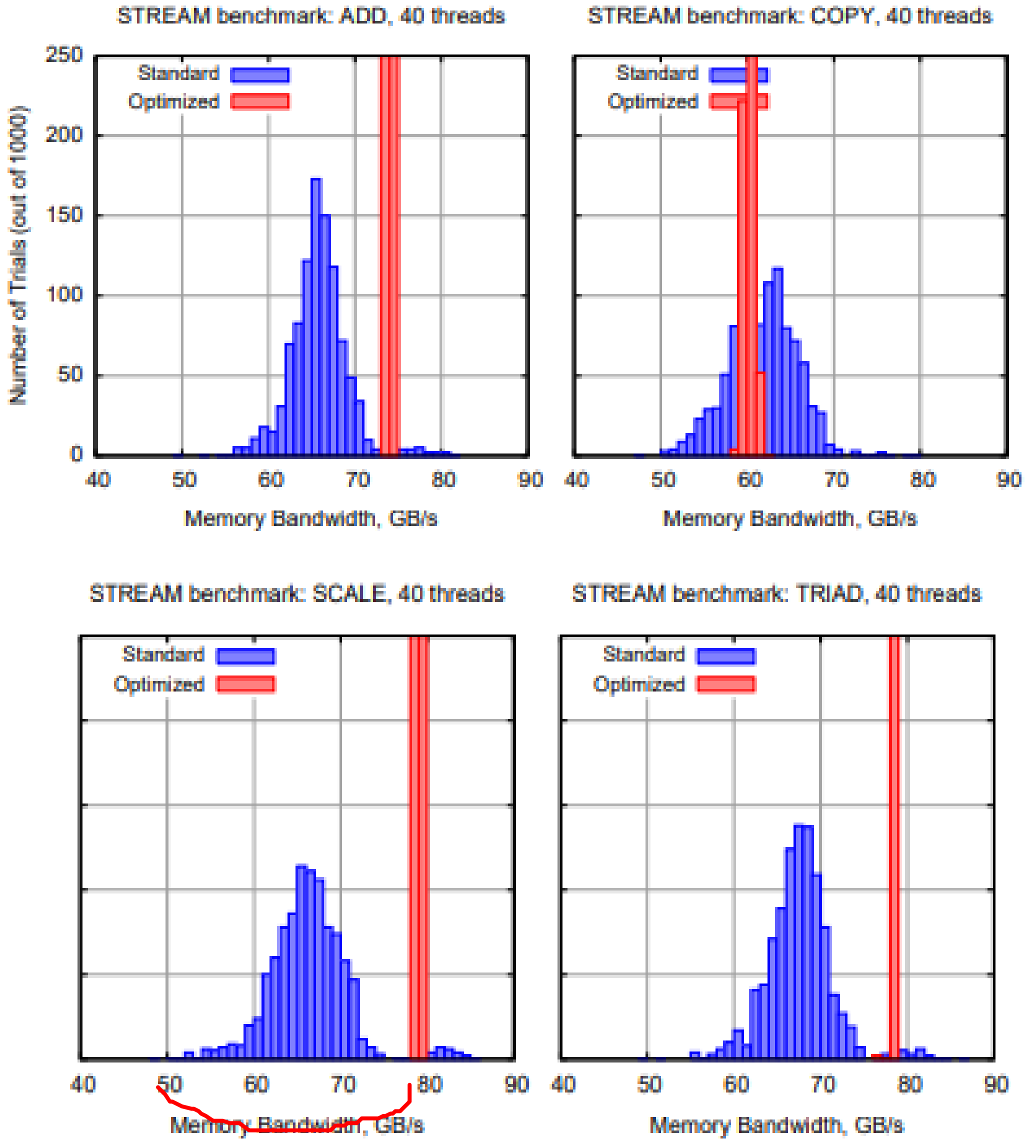
```
#pragma omp parallel for
 for (j=0; j<VectorSize; j++) {
 a[j] = 1.0; b[j] = 2.0; c[j] = 0.0;}
```

Step 2 Compute

```
#pragma omp parallel for
for (j=0; j<VectorSize; j++) {
a[j]=b[j]+d*c[j];}
```

- When you allocate memory, the NUMA domain it is affinitized to is not decided.
- It is decided when you initialize the memory.
- Memory will be local to the thread which initializes it. This is called the "**first touch policy**".
- Try to initialize memory in the same pattern as you intend to computation across threads

How does all this NUMA-ness affect performance?



STREAM benchmark

- Run on Intel Xeon E7-4870 CPU4
- 30 MB of L3 cache and 10 cores with hyper-threading technology @ 2.40 GHz

- Red bars are histograms for bandwidth measurements with KMP_AFFINITY=scatter
- Notice how the bandwidth usage has improved with thread affinity set
- We will discuss KMP_AFFINITY in coming slides

*https://colfaxresearch.com/terabyte-ram-servers-memory-bandwidth-benchmark-and-how-to-boost-ram-bandwidth-by-20-with-a-single-command/

Perfecting “first-touch” policy is hard

- Hard to do “perfect touch” for real applications.
- What if you have some dynamic load balancing in the code, data gets reshuffled among threads
- Keep threads less than the number of CPUs in NUMA domain
- As far as possible keep the data initialization and compute on the same thread. And experiment with the different thread placement policies (we will see next).
- There is no one size fits all approach here, applications have different data access patterns. Example – structured grid codes have unit-strided access. Codes based on Monte-Carlo method thrive on randomness.

Runtime Environment Variable: OMP_PLACES

- OpenMP 4.0 added OMP_PLACES environment variable
  - To control thread allocation
  - defines a series of places to which the threads are assigned

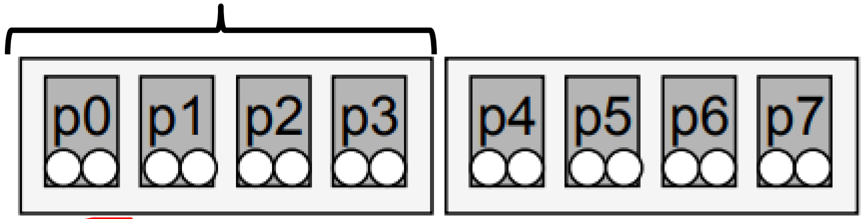
- Allowed values
  - threads: each place corresponds to a single hardware thread on the target machine.
  - cores: each place corresponds to a single core (having one or more hardware threads) on the target machine.
  - sockets: each place corresponds to a single socket (consisting of one or more cores) on the target machine.
  - A list with explicit CPU ids along with intervals for placement

- Examples:
  - export OMP_PLACES=threads
  - export OMP_PLACES=cores

Runtime Environment Variable: OMP_PLACES

- OMP_PLACES can also be
  - A list with explicit place values of CPU ids, such as:
    - *"{0,1,2,3},{4,5,6,7},{8,9,10,11},{12,13,14,15}"*
    - *"{0:4},{4:4},{8:4},{12:4}"* (default stride is 1)
    - Format: *{lower-bound:length:stride}*. Thus, specifying *{0:3:2}* is the same as specifying *{0,2,4}*

Socket with 4 cores



- Examples: 2 socket machine with 2 processors, 4 cores each
  - export OMP_PLACES=*"{0:4:2},{1:4:2}"* (which is equivalent to *"{0,2,4,6},{1,3,5,7}"*)
  - export OMP_PLACES=*"{0:8:1}"* (which is equivalent to *"{0,1,2,3,4,5,6,7}"*

*Image - https://www.openmp.org/wp-content/uploads/openmp-examples-5-0-1.pdf

Runtime Environment Variable: OMP_PROC_BIND

- Controls thread affinity within and between OpenMP places
- OpenMP 3.1 only has OMP_PROC_BIND, either TRUE or FALSE.
  - If true, the runtime will not move threads around
- OpenMP 4.0 still allows the above. Added options:
  - close: bind threads close to the master thread
  - spread: bind threads as evenly distributed (spread) as possible
  - master: bind threads to the same place as the master thread
- Examples:
  - OMP_PROC_BIND=spread
  - OMP_PROC_BIND=spread,close (for nested levels)

Runtime Environment Variable:OMP_PROC_BIND

- Use 4 cores total, 2 hyper-threads per core, and OMP_NUM_THREADS=4 as an example
- close: Bind threads as close to master thread as possible

Node	Core 0		Core 1		Core 2		Core 3	
	HT1	HT2	HT1	HT2	HT1	HT2	HT1	HT2
Thread	0 (M)	1	2	3				

- spread: Bind threads as far apart as possible.

Node	Core 0		Core 1		Core 2		Core 3	
	HT1	HT2	HT1	HT2	HT1	HT2	HT1	HT2
Thread	0 (M)		1		2		3	

- master: bind threads to the same place as the master thread

Node	Core 0		Core 1		Core 2		Core 3	
	HT1	HT2	HT1	HT2	HT1	HT2	HT1	HT2
Thread	0 (M), 1, 2, 3							

Affinity Clauses for OpenMP Parallel Construct

- The *"num_threads"* and *"proc_bind"* clauses can be used
  - The values set with these clauses take precedence over values set by runtime environment variables

- ***OMP_PLACES***=*"{0,1},{2,3},{4,5},{6,7},{8,9},{10,11},{12,13},{14,15}"*   *OR*
  OMP_PLACES=*"0:8:2"*

  - C/C++:

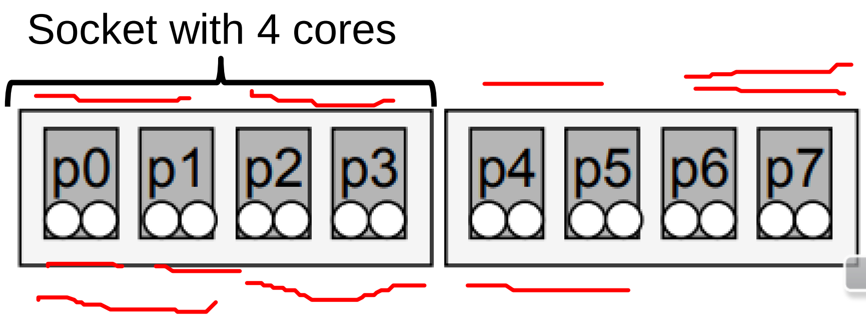
    #pragma omp parallel num_threads(4) proc_bind(spread)

  - Fortran:

    !$omp parallel num_threads (4) proc_bind (spread)

    …

    !$omp end parallel



Runtime APIs for Thread Affinity Support

- OpenMP 4.5 added runtime functions to determine the effect of thread affinity clauses

- Query functions for OpenMP thread affinity were added
  - ***omp_get_num_places:*** returns the number of places
  - ***omp_get_place_num_procs***: returns number of processors in the given place
  - ***omp_get_place_proc_ids:*** returns the ids of the processors in the given place
  - ***omp_get_place_num:*** returns the place number of the place to which the current thread is bound

Runtime APIs - examples

Assume the following

- *OMP_PROC_BIND="TRUE"*
- *OMP_NUM_THREADS=4*
- *OMP_PLACES="{0,2,4,6}"*

- ***omp_get_num_places()*** – returns 1 place
- ***omp_get_place_num_procs(0)*** - returns 4
- ***omp_get_place_proc_ids(0, ids)*** **-** returns the ids of the processors in the place 0, in our example it‘s 0, 2, 4 and 6
- ***omp_get_place_num(2)*** **-** returns 0, since thread 2 is bound to place 0

KMP_AFFINITY

- Specific to the Intel® runtime library
- Binds OpenMP threads to physical processes
- Supported on Windows* and Linux* that have support for thread affinity
- Intel OpenMP thread affinity interface
  - Three types of interface available to control thread affinity
  - High-level – with environment variable KMP_AFFINITY
  - Mid-level - environment variable to specify which processors are bound to OpenMP threads. Compatible to GNU GOMP_AFFINITY
  - **Low-level** – APIs to enable OpenMP threads to make call to runtime for setting processor set. For advanced users

KMP_AFFINITY

- *KMP_AFFINITY=[<modifier,…>] <type> [,<permute>][,<offset>]*

- *modifier* can be
  - *noverbose or verbose* – to get thread placement information
  - *proclist={proc-list}*
  - *granularity=<specifier> where specifier* can be *fine, thread or core*

- *type* can be
  - *none, disabled, explicit, balanced, scatter, compact*
  - This specifies the placement of threads on the core

KMP_AFFINITY

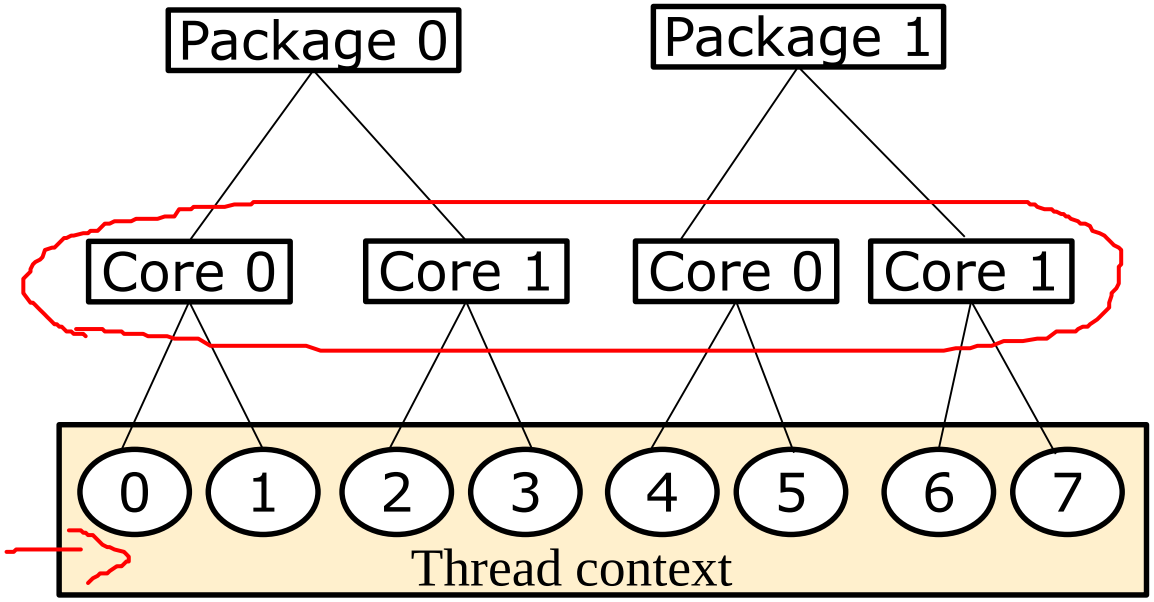
- *type = balanced*
  - threads are placed on separate cores till all cores atleast have one thread each
  - if multiple threads are to be placed on same core, balanced ensures the openmp threads numbers are close to each other
  - Different to *scatter* that does not do so
  - *KMP_AFFINITY=balanced* is same as *OMP_PROC_BIND=spread*

- *type = scatter*
  - Distributes threads as evenly as possible in the entire system
  - Opposite of compact

- *type = compact*
  - Places thread T as close to T-1 as possible
  - permute and offset allowed for both compact and scatter

KMP_AFFINITY

- Consider a machine with two processors, with 2 cores each and hyper-threading enabled



								KMP_AFFINITY
0	1	2	3	4	5	6	7	*granularity=fine,compact*
0	4	2	6	1	5	3	7	*granularity=fine,scatter*
3	4	5	6	7	0	1	2	*granularity=fine,compact,0,3*
0	4	1	5	2	6	3	7	*granularity=fine,compact,1,0*

Summary for Thread Affinity and Data Locality

- Best data locality, and optimal process and thread affinity crucial for good performance
- Exploit first touch data policy, optimize code for cache locality.
  - Pay special attention to avoid false sharing.
- Threads far apart (spread) may improve aggregated memory bandwidth and available cache size for your application but may also increase synchronization overhead. And putting threads “close” have the reverse impact as “spread”.
- We have not discussed nested OpenMP here, but this is also an important feature in OpenMP. Once you are comfortable with process affinity and process placement, we urge you to look at the concept.

Outline

- OpenMP memory model
- Synchronization
- TASKS and SECTIONS
- Vectorization constructs
- Thread affinity
  - First touch policy
- OpenMP run time routines
- OpenMP environment variables

OpenMP runtime routines

- Recall that the OpenMP ICVs can also be changed with runtime routines
  - *omp_set_num_threads*

- OpenMP provides runtime routines that a user can call from within an application to change or query the state of threads, processes or parallel environment

- Each of these runtime routines set the value of an ICV and return the value when probed via a function call

- We look at few of them that are commonly used, in addition to the lock routines

OpenMP runtime routines

- *omp_set_num_threads*
  - Set the number of threads to be used in a parallel region
- *omp_get_num_threads*
  - Return the number of threads in the currently executing parallel region
  - This returns a value of 1 in a sequential region
- *omp_get_max_threads*
  - Upper bound on number of threads that can be used to form a new team of threads
  - Used if the parallel construct doesn't have the *num_threads* clause
- *omp_get_thread_num*
  - Get thread number of the calling thread
  - In a team of N threads, value ranges from 0 to $N - 1$
- *omp_set/get_schedule*
  - set/get the runtime schedule – STATIC, DYNAMIC, GUIDED or AUTO

Outline

- OpenMP memory model
- Synchronization – advanced
- TASKS and SECTIONS
- Vectorization constructs
- Thread affinity
- OpenMP run time routines
- OpenMP environment variables

OpenMP environment variables

- OpenMP provides environment variables that can also be used to set ICVs
- Modifications to environment variables during runtime are ignored by OpenMP
- Can use runtime routines to change the ICV values, as seen earlier
- Set through the *setenv*, *export* or *set* command
- Variable names are in uppercase, values are case insensitive and may have leading or trailing white space

OMP_SCHEDULE	Controls the schedule and chunk size of loops specified with the *schedule(runtime)* directive. export **OMP_SCHEDULE**="guided,4"
OMP_NUM_THREADS	Sets number of threads to be used in a parallel region. This has some rules of precedence, and can be overridden through function call or *num_threads* clause export **OMP_NUM_THREADS**=8
OMP_STACKSIZE	Controls the stack size for each thread, size set as size/size**B**/size**K**/size**M**/size**G** export **OMP_STACKSIZE**=100M (100 Mega bytes)
OMP_WAIT_POLICY	Whether the waiting threads should be ACTIVE and consume processor cycles while waiting. Or be PASSIVE and go to sleep or yield to other threads while waiting export **OMP_WAIT_POLICY**=PASSIVE
OMP_TARGET_OFFLOAD	Controls the offloading behaviour **MANDATORY** – program is terminated if device not found **DEFAULT** – run on default device, probably the host
OMP_PLACES	Specifies the places a thread is bound to, **threads, cores, sockets** export **OMP_PLACES**=threads export **OMP_PLACES**={0,1,2,3} (4 hardware threads)
OMP_PROC_BIND	Controls if threads can be moved between OpenMP places, scattered or compact or bound to master thread export **OMP_PROC_BIND**=**false/true/master/close/spread**