Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National Supercomputing Centre

DAY 1: Introduction to OpenMP

Multi-threaded Programming, Tuning and Optimization on Multi-core MPP Platforms

15-17 February 2011

CSCS, Manno

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National Supercomputing Centre

Introductory Course on OpenMP Programming

CSCS, National Supercomputing Service

Agenda

- Basic information
  - Intro to programming model.
  - Directives for work parallelization and synchronization.
- Hands-on Lab
  - Writing compiling and executing simple OpenMP programs.
  - Identifying and resolving common issues.

Agenda

- Advanced topics
  - Data-scoping constructs
  - Constructs introduced in OpenMP 3.0
- Hands-on Lab
  - Experiments using data-scoping constructs
  - Examples with OpenMP 3.0 directives.

What is OpenMP?

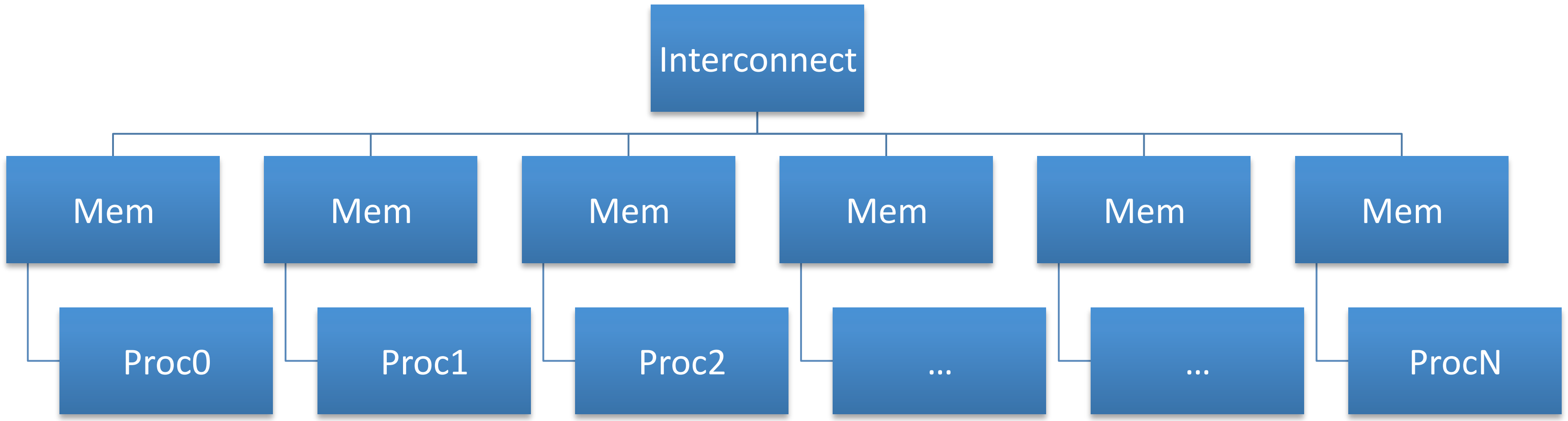
- OpenMP = Open Multi-Parallelism
- It is an API to explicitly direct *multi-threaded* *shared-memory parallelism*.
- Comprised of three primary API components
  - Compiler directives
  - Run-time library routines
  - Environment variables

How is OpenMP not MPI?

MPI is an API for controlling *distributed-memory* parallelism on multi-processor architectures.

Each processor has it's own unique memory

Information is passed between memory locations through the interconnect via the MPI API.

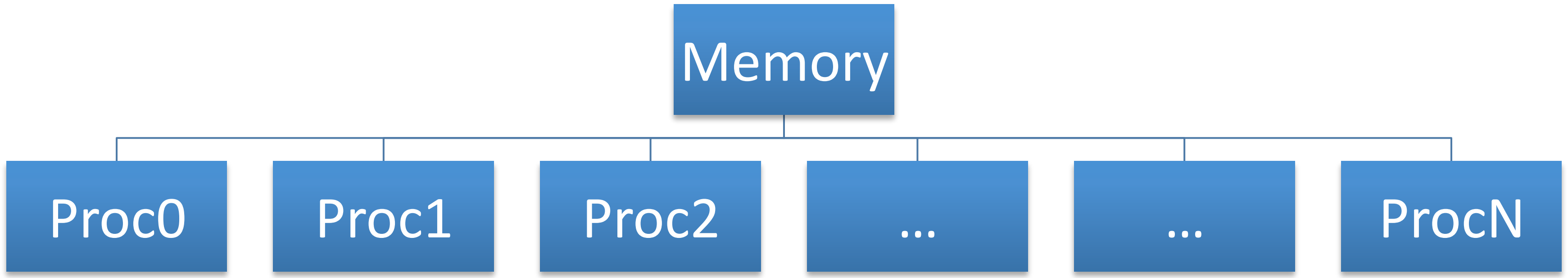


OpenMP

A process, such as an MPI task, owns a lot of state information about the process, including the memory, file handles, etc. Threads, launched by the process, share the state information, including memory, of the launching process and so are considered *light weight*.

Since memory references amongst a team of threads are shared: *OpenMP requires that the programmer ensures that memory references are handled correctly*.

It is possible, for both paradigms to be used in one application to improve either speed, or scaling, or both. This is the so called *hybrid* parallel programming model.



Why use OpenMP?

- Exploit *more* parallelism to increase scaling and performance
  - SMPD parallelism
    - E.g. MPI tasks on one spatial dimension, OpenMP threads on another.
  - Functional parallelism
    - Threads executing different tasks, perhaps on the same data, perhaps not.
  - Improve load balance via new OpenMP constructs that enable *work stealing*.

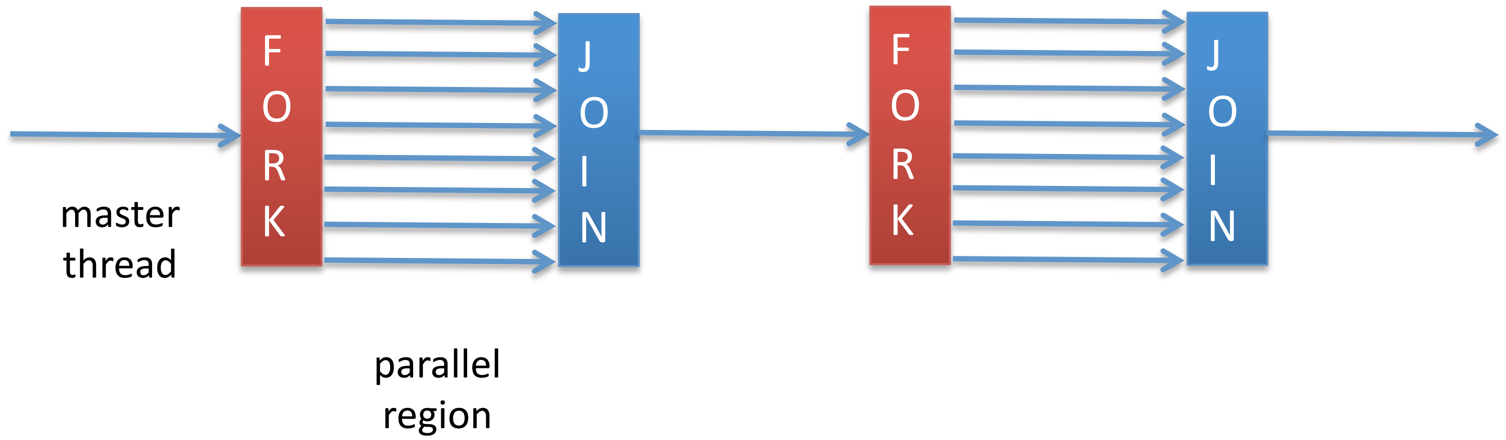
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National Supercomputing Centre

Creating parallelism

Fork-and-join model

- OpenMP programs begin as a single process, the master thread, until they reach a parallel region, which then spawns a team of threads.



Parallel regions

- Threads are created with the `parallel` directive.
- NB: Directives are comments (in Fortran) or pragmas (in C/C++). Thus, you can create portable code that works with or without OpenMP depending on the architecture or your available compilers.

Fortran example

```
double precision :: x(1000)
integer id,n
integer omp_get_thread_id
integer omp_get_num_threads

!$omp parallel private(id)

      id = omp_get_thread_id()
      n  = omp_get_num_threads()
      call foo( id, x )

!$omp end parallel
```

- Outside of parallel region, there is only 1 thread (master).
- Inside of parallel region there are N threads (will see how to set this later)
- All threads share X, id is private to each thread.
- There is an implicit barrier at the end of the parallel region

Fortran example

- In the previous example, we also saw two functions from the run time library
  - `omp_get_thread_num()`
    - Returns unique thread id number for each thread in the team.
  - `omp_get_num_threads()`
    - Returns the number of threads in the team.
- There are more (over 20) but these are the two most common, if they are used at all.

C example

```
double x[1000];

#pragma omp parallel
{
        int id = omp_get_thread_id()
        int n = omp_get_num_threads()
        foo( id, x );
}
```

Synchronization 1

- Synchronization is used to impose order constraints and to protect shared data.
  - Master
  - Single
  - Critical
  - Barrier
- Will see a others later

`master` directive

```
!$omp parallel private(id)

      id = omp_get_thread_id()

!$omp master
      print *, ‘myid = ‘, id
!$omp end master

!$omp end parallel
```

- In this example, all threads are assigned a thread ID number (0-12, say).
- Because of the `master` directive, only the master thread (id=0) prints out a message.

`single` directive

```
!$omp parallel private(id)

       id = omp_get_thread_id()

!$omp single
       print *, ‘myid = ‘, id
!$omp end single [nowait]

!$omp end parallel
```

- Again, all threads are assigned a thread ID number.
- Because of the `single` directive, only one thread prints out a message.
- Which thread executes the `single` section may change from one execution to the next.
- The optional `nowait` directive overrides the implicit barrier in a directive.

critical directive

```
!$omp parallel private(id)

       id = omp_get_thread_id()

!$omp critical
       print *, 'myid = ', id
!$omp end critical

!$omp end parallel
```

- All threads will print their id number.
- Within the `critical` section, only one thread out of the team will be executing at any time.
- Thus, for six threads, there will be six print statements but they will not necessarily be ordered by id number.

`barrier` directive

```
!$omp parallel
       call foo1()
!$omp barrier
       call foo2()
!$omp end parallel
```

- The `barrier` directive requires that all threads in the team arrive at the barrier before execution continues.
- In this example, the function foo1 may perform some action, e.g. on shared data, that may affect other threads in the function foo2. Thus, all threads execute foo1, stop at the barrier and then continue on to foo2.

`atomic` directive

- The `atomic` protects memory locations from being updated by more than one thread.

```
n = 0
!$omp parallel

!$omp atomic
   n = n + 1

!$omp end parallel
```

Warning

- In general, try to avoid the use of synchronization directives, especially barriers, as they may cause significant performance degradation.
- If possible, try to re-factor your algorithm to avoid using them. Consider using temporary variables in OpenMP sections to accomplish this.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National Supercomputing Centre

Data scoping

Private/Shared Data

- In parallel regions, four types of data attributes can exist
  - `shared` (default)
    - Accessible by all threads
  - `private`
    - Accessible only by the current thread
    - NB: Loop counters are automatically private
- Also
  - `none`
  - `firstprivate`
- The default can be changed using the `default` directive

```
!$omp parallel default(private)
!$omp parallel default(shared)
```

Private/Shared data

- Individual variables in parallel regions can be declared `private` or `shared`

```
!$omp parallel private(x0,y0)
      x0 = xarray(…)
      y0 = yarray(…)
      f(…) = foo1(x0,y0)
!$omp end parallel
```

- Here, `x0`, and `y0` are private variables, taken from the shared arrays `x()`, and `y()` that are used to compute some variable that is stored in the shared array `f()`.
- It is also possible to directly specify that variables be `shared`.

```
!$omp parallel private(x0,y0) shared(xarray,yarray,f)
      x0 = xarray(…)
      y0 = yarray(…)
      f(…) = foo1(x0,y0)
!$omp end parallel
```

firstprivate

- The `firstprivate` directive allows you to set `private` variables to the value of their original prior to entry into the parallel or worksharing construct.

```
A = 1
B = 2
!$omp parallel private(A) firstprivate(B)
      ….
!$omp end parallel
```

- In this example, A has an undefined value on entry into the parallel region while B has the value specified in the previous parallel region.
- This can be costly for large data structures.

lastprivate

- Upon exiting worksharing constructs (do loops or sections), it may be useful to store the last value of a private variable do it can be used in the serial section.

```
A = 1
B = 2
!$omp parallel firstprivate(B)
!$omp do lastprivate(A)
do i = 1, 1000
      A = i
end do
!$omp end do
!$omp end parallel
```

- In this example, upon exiting the do loop, A=1000.

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National Supercomputing Centre

Loop Worksharing (do/for)

• Motivating example

```
DO I = 1, N
       a(i) = b(i) + c(i)
END DO
```

- The OpenMP worksharing construct `do` (in Fortran) or `for` (in C/C++) enables the programmer to distribute the work of loops across threads.

```
!$omp parallel
!$omp do
DO I = 1, N
       a(i) = b(i) + c(i)
END DO
!$omp end do [nowait]
!$omp end parallel
```

- In this example, OpenMP determines, by default, the amount of work to give to each thread by dividing N by the number of threads. We will see later how to change this behavior.

Loop worksharing

- For convenience, the two statements can be combined

```
!$omp parallel do
DO I = 1, N
       a(i) = b(i) + c(i)
END DO
!$omp end parallel do
```

Reductions

- Very often, a programmer needs to compute a variable that is the sum of other data, e.g.

```
Real :: x(M), avg
Avg = 0.0
DO I = 1, N
      avg = avg + x(i)
END DO
Avg = avg / FLOAT(M)
```

- This operation is called a reduction and there is support in OpenMP for parallelizing this sort of thing rather trivially.

`reduction` directive

```
Real :: x(M), avg
!$omp parallel do reduction(+:avg)
DO I = 1, N
       avg = avg + x(i)
END DO
!$omp end parallel do
```

- In this example, the `avg` variable is automatically declared `private` and initialized to zero.
- The general form of the reduction directive is

```
reduction(operator:variable)
```

Reductions

- Some of the most common reduction operators and initial values are as follows

Operator	Initial value
+	0
*	1
-	0

C/C++ Only

Operator	Initial value
&	~0
|	0
^	0
&&	1
||	0

Fortran Only

Operator	Initial value
MIN	Largest pos. number
MAX	Most negative number
.AND.	.TRUE.
.OR.	.FALSE.
.NEQV.	.FALSE.
.IEOR.	0
.IOR.	0
.IAND.	All bits on
.EQV.	.TRUE.

`order` directive

- Some expressions in do/for loops need to be executed sequentially because the results are order dependent, e.g.

```
DO I = 1, N
      a(i) = 2 * a(i-1)
END DO
```

- In order to parallelize this loop, it is mandatory to use the `ordered` directive

```
!$omp do ordered
DO I = 1, N
!$omp ordered
      a(i) = 2 * a(i-1)
!$omp end ordered
END DO
!$omp end do
```

← Let OpenMP know an ordered statement is coming later

Scheduling

- When a do-loop is parallelized and its iterations distributed over the different threads, the most simple way of doing this is by giving to each thread the same number of iterations.
  - not always the best choice, since the computational cost of the iterations may not be equal for all of them.
  - different ways of distributing the iterations exist, this is called scheduling.

`schedule` directive

- The `schedule` directive allows you to specify the chunking method for parallelization of `do` or `parallel do` loops. Work is assigned to threads in a different manner depending on the scheduling type or chunk size used.
  - `static (default)`
  - `dynamic`
  - `guided`
  - `runtime`

`schedule` directive

```
!$omp parallel do schedule(type[, chunk])
DO I = 1, N
       a(i) = b(i) + c(i)
END DO
!$omp end parallel do
```

- The `schedule` clause accepts two parameters.
  - The first one, `type`, specifies the way in which the work is distributed over the threads.
  - The second one,`chunk`, is an optional parameter specifying the size of the work given to each thread: its precise meaning depends on the type of scheduling used.

`schedule` directive

- `static (default)`
  - work is distributed in equal sized blocks. If the chunk size is specified, that is the unit of work and blocks are assigned to threads in a round-robin fashion.
- `dynamic`
  - work is assigned to threads one at a time. If the chunk size is not specified, the chunk size is one.
  - Faster threads get more work, slower threads less.
- `guided`
- `runtime`

`schedule` directive

- `guided`
  - Similar to dynamic but each block of work is a fixed fraction of the preceding amount, decreasing to `chunk_size` (1, if not set)
  - Fewer chunks = less synchronization = faster?
- `runtime`
  - Allows scheduling to be determined at run time.
  - Method and chunk size specified by the environment variable `OMP_SCHEDULE`, e.g.
    - `setenv OMP_SCHEDULE “guided, 25”`

Sections

- Sections are a means of distributing independent blocks of work to different threads.
- For example, you may have three functions that do not update any common data

```
…
call foo1(…)
call foo2(…)
call foo3(…)
…
```

Section directive

- Using sections, each of these functions can be excuted by different threads

```
!$omp parallel
!$omp sections [options]
!$omp section
call foo1(…)          !thread 1
!$omp section
call foo2(…)          !thread 2
!$omp section
call foo3(…)          !thread 3
!$omp end sections[nowait]
!$omp end parallel
```

Sections

- May be the only way to parallelize a region.
- If you don’t have enough sections, some threads my be idle.
  - Still may be useful and provide a performance boost if you can’t thread your blocks or functions.
- Can also use `!$omp parallel sections` shortcut.

Workshare (Fortran only)

- In Fortran, the following can be parallelized using the `workshare` directive
  - `forall`
  - `where`
  - Array notation expression
    - e.g. `A = B + C`, where `A`, `B`, and `C` are arrays.
  - Transformational array functions
    - e.g. `matmul`, `dot_product`, `sum`, `maxval`, `minval`, etc.

workshare example

```
real(8) :: a(1000), b(1000)
!$omp parallel
!$omp workshare

forall(i=1:1000)
   b(i) = 10*I
end forall

a = a + b

!$omp end workshare[nowait]
!$omp end parallel
```

Useful links

- OpenMP Consortium
  - Summary of Fortran Syntax (PDF)
  - Summary of C/C++ Syntax (PDF)

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National Supercomputing Centre

LAB 1

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National Supercomputing Centre

OpenMP 3.0

OpenMP 3.0 features

- OMP_STACKSIZE
- Loop collapsing
- Nested parallelism
- Tasks

OMP_STACKSIZE

- `omp_stacksize` *`size`*
- New environment variable that controls the stack size for threads.
  - Valid sizes are *size*, *sizeB*, *sizeK, sizeM*, *sizeG* bytes.
  - If B, K, M, G not specified, size is in kilobytes(K).

Collapse(n)

- New clause for do/for constructs
- Specifies how many loops in a nested loop should be collapsed into one large iteration space.

```
!$omp parallel do collapse(2)
DO J = 1, M
DO I = 1, N
      a(i,j) = b(i,j) + c(i,j)
END DO
END DO
!$omp end parallel do
```

Nested Parallelism

- It is possible to nest parallel sections within other parallel sections

```
!$omp parallel
   print *, ‘hello’
!$omp parallel
       print *, ‘hi’
!$omp end parallel
!$omp end parallel
```

- Can be useful, say, if individual loops have small counts which would make them inefficient to process in parallel.

Nested parallelism

- Nested parallelism needs to be enabled by either
  - Setting an environment variable
    - `setenv OMP_NESTED TRUE`
    - `export OMP_NESTED=TRU`
  - Using the OpenMP run-time library function
    - `call omp_set_nested(.true.)`
- Can query to see if nesting is enabled
  - `omp_get_nested()`

Nested parallelism

- Unfortunately, nested parallelism is not currently implemented by all vendors.
  - Fortran
    - PGI NO
    - Cray YES, need to set omp_set_max_active_levels()
    - Intel YES
  - The situation should be true for their C/C++ products.

Tasks

- Why task parallelism? Gives us an elegant way of dealing with
  - Unbounded loops
  - Recursive algorithms
  - Producer/consumer algorithms
  - etc

Example: list traversal

```
Void traverseList( List list )
{
      ListElement elem;
#pragma omp parallel private(elem)
      for( elem=list->first; elem; elem=elem->next)
            #pragma omp single nowait
                  foo(elem);
}
```

- Awkward
- Poor performance (note single directive)

Example: tree traversal

```
Void traverseTree( Tree *tree )
{
#pragma omp parallel sections
{
#pragma omp section
      if( tree->right )
            traverseTree( tree->left );
#pragma omp section
      if( tree->left )
            traverseTree( tree->right );
}

foo(tree);
}
```

- Too many parallel sections and synchronizations.

Tasks

- Tasks are work units which may execute immediately or be deferred.
- Tasks are composed of
  - Code blocks to execute
  - A data environment
    - Initialized when the work unit is created
  - Internal control variables

Task directive

- `!$omp task / !$omp end task`
- `#pragma omp task`
- Can be nested inside
  - Parallel regions
  - Other tasks
  - Inside worksharing constructs

Example with tasks

```
void traverseList( List list )
{
        ListElement elem;
        for( elem=list->first; elem; elem=elem->next)
                #pragma omp task
                        foo(elem);
}
```

- What is the scope of `elem`?

Scoping for tasks

- `shared(`*`list`*`)`
- `private(list)`
- `firstprivate(list)`
  - Data is copied at creation
- `default(shared|none)`
- Global variables are shared
- Otherwise
  - `firstprivate`
  - `shared` if specified

Example with tasks

```
void traverseList( List list )
{
        ListElement elem;
        for( elem=list->first; elem; elem=elem->next)
                #pragma omp task
                        foo(elem);
}
```

- Elem is `firstprivate`
- How can you guarantee that traversal is finished?

Synchronizing tasks

- Barriers (explicit or implicit)
  - All tasks created by any thread of the current team are guaranteed to be completed at barrier exit.
  - Taskwait
    - !$omp taskwait
      - Encountering task waits until child (only direct children!) tasks complete.

Example

```
void traverseList( List list )
{
        ListElement elem;
        for( elem=list->first; elem; elem=elem->next)
                #pragma omp task
                        foo(elem);
#pragma omp taskwait
}
```

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National Supercomputing Centre

LAB 2