Parallel programming with OpenMP

Parallel Programming Languages

Evolution of programming methods:

- MPI is still the dominant programming technique
- Hybrid OpenMP/MPI approach most effective on supercomputers
- GPU programming develops quickly
  - CUDA
  - OpenACC and OpenMP
- Message Passing directly within the GPU
- New specific parallel programming languages are developed:
  - Co-array Fortran, PGAS, X10, Chapel…
- New runtime systems to handle task-based parallelism:
  - Charm++, HPX, Kokkos

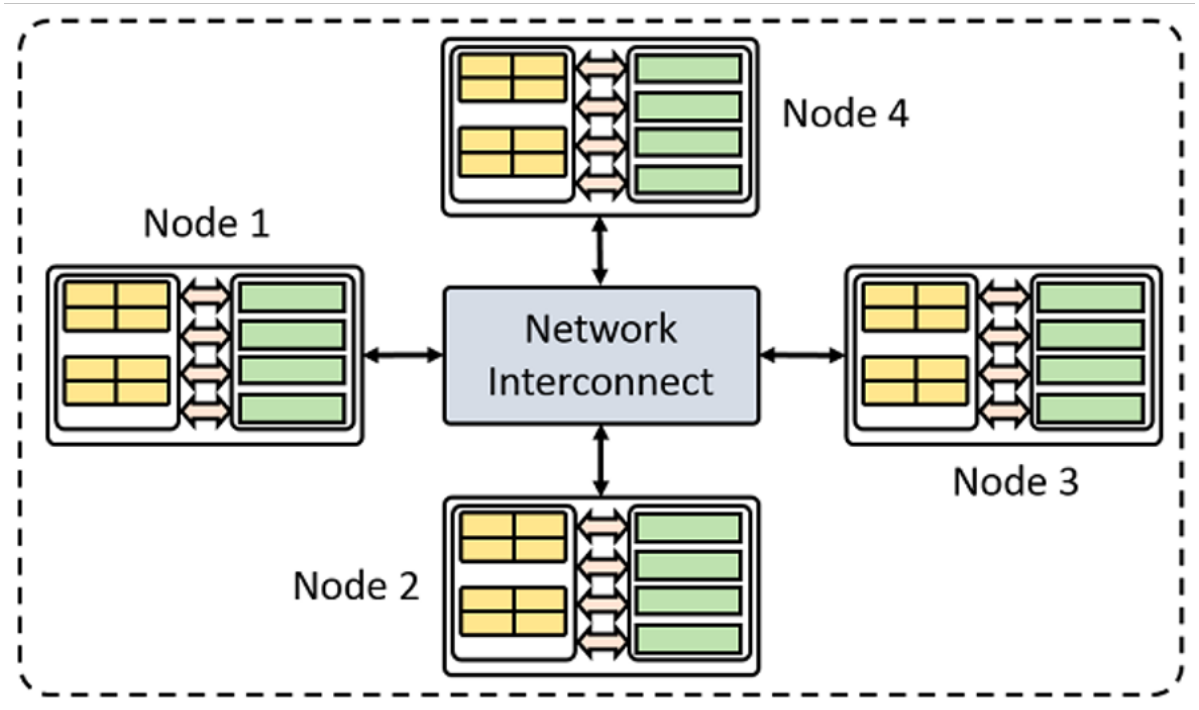
Distributed memory versus shared memory

MPI uses a distributed memory paradigm

- Data are transferred explicitly between nodes through the network

OpenMP uses a shared memory paradigm

- Data are shared implicitly within the node through the Random-Access Memory.



OpenMP: History

Multi-threading using directives:

Many vendors were using their own multi-threading directives (CRAY, NEC, IBM…) during the era of vector processing (mid-90s).

On October 28th 1997, they all met and adopted the *Open Multi Processing* standard.

OpenMP is specified by the Architecture Review Board (ARB).

OpenMP 2.0: Nov 2000: modern Fortran constructs

OpenMP 3.0: May 2008: task-based computing

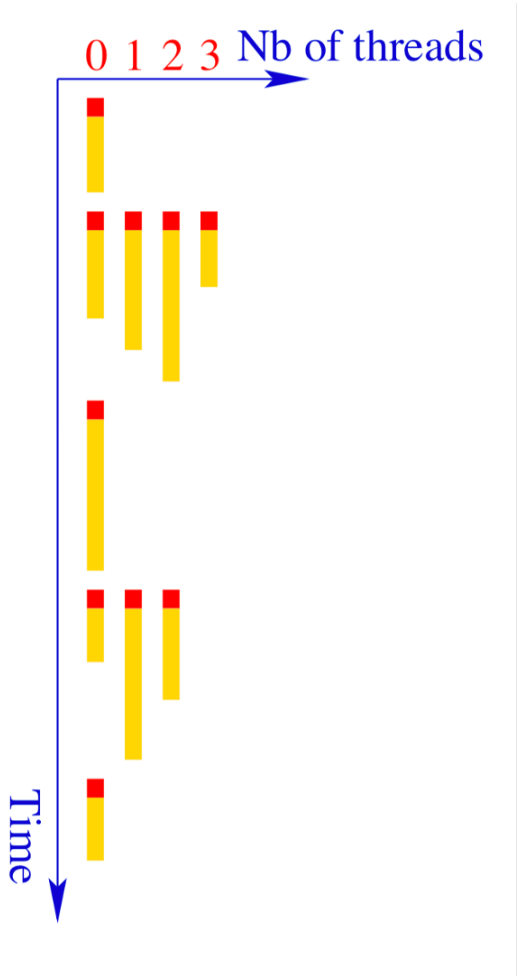
OpenMP 4.0: July 2013: external devices

OpenMP 5.0: under review

OpenMP: General Concepts

Multi-threading

- An OpenMP program is executed by only one process, called the master thread. The corresponding piece of code is called a *sequential region*.
- The master thread activates light-weight processes, called the workers or slave threads. This marks in the code the entry of a *parallel region*.
- Each thread executes a task corresponding to a block of instructions. During the execution of the task, variables can be read from or updated in memory.
- The variable can be defined in the local memory of the thread, called the stack memory. The variable is called a *private variable*.
- The variable can be defined in the main shared (RAM) memory also called the heap. The variable is called a *shared variable*.



OpenMP: Compilation

Compilation directives and clauses:

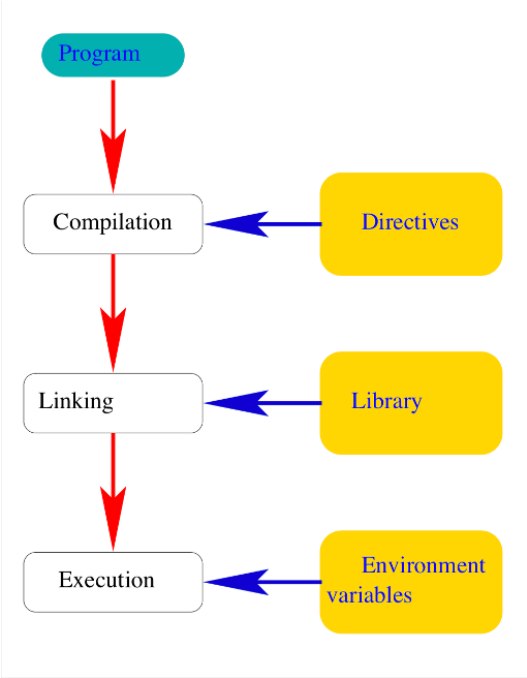
- They are put in the program to create the threads, define the work and data sharing strategy, and synchronize shared variables
- They are considered by the C, C++ or Fortran compilers as mere comment lines unless one specifies –openmp or –fopenmp on the compilation command line

Functions and routines:

- OpenMP contains several dedicated functions (like MPI). They are part of the OpenMP library than can be linked at link time.

Environment variables:

- OpenMP has several environment variables that can be set at execution time and changed the parallel computing behavior.



Example

Compilation:

```
ifort -openmp prog.f90
gfortran -fopenmp prog.f90
```

Environment variable:

```
export OMP_NUM_THREADS=4
```

OpenMP: Parallel Region

- In a parallel region, by default, the data sharing attribute of variables is shared.
- Within a single parallel region, all concurrent threads execute the same code in parallel.
- There is an implicit synchronization barrier at the end of the parallel region.

```
program parallel
 !$ use OMP_LIB
 implicit none
 real    :: a
 logical :: p

 a = 92290. ; p=.false.
 !$OMP PARALLEL
  !$ p = OMP_IN_PARALLEL()
  print *,"A = ",a, &
          "; p = ",p
 !$OMP END PARALLEL
end program parallel
```

```
dhcp-94-236:~ teyssier$ gfortran -fopenmp prog.f90
dhcp-94-236:~ teyssier$ export OMP_NUM_THREADS=4
dhcp-94-236:~ teyssier$ ./a.out
   92290.0000     T
   92290.0000     T
   92290.0000     T
   92290.0000     T
dhcp-94-236:~ teyssier$
```

- Using the DEFAULT clause, it is possible to change the default attribute to PRIVATE.
- If a variable is PRIVATE, it will be stored in the stack memory of each thread. Its value is undetermined when entering the parallel region.

```
program parallel
 implicit none
 real :: a

 a = 92000.
 !$OMP PARALLEL DEFAULT(PRIVATE)
   a = a + 290.
   print *,"A = ",a
 !$OMP END PARALLEL
end program parallel
```

```
dhcp-94-236:~ teyssier$ gfortran -fopenmp prog.f90
dhcp-94-236:~ teyssier$ export OMP_NUM_THREADS=4
dhcp-94-236:~ teyssier$ ./a.out
   290.000000
   290.000000
   290.000000
   290.000000
dhcp-94-236:~ teyssier$
```

OpenMP: Parallel Region

- Using the FIRSTPRIVATE clause, it is possible to force the initialization of a PRIVATE variable to the last value it has outside the parallel region.

```
program parallel
 implicit none
 real :: a

 a = 92000.
!$OMP PARALLEL DEFAULT(NONE) &
      !$OMP FIRSTPRIVATE(a)
   a = a + 290.
   print *,"A = ",a
!$OMP END PARALLEL
print*,"Out of region, A =",a
end program parallel
```

```
dhcp-94-236:~ teyssier$ gfortran -fopenmp prog.f90
dhcp-94-236:~ teyssier$ export OMP_NUM_THREADS=4
dhcp-94-236:~ teyssier$ ./a.out
   92290.0000
   92290.0000
   92290.0000
   92290.0000
 Out of parallel region, a=   92000.0000
dhcp-94-236:~ teyssier$
```

- Using the NUM_THREADS() clause allows to set the number of children in a parallel region.
- OMP_GET_NUM_THREADS() gives the size of the active team

```
program parallel
  implicit none

  !$OMP PARALLEL NUM_THREADS(2)
    print *,"Hello !"
  !$OMP END PARALLEL

  !$OMP PARALLEL NUM_THREADS(3)
    print *,"Hi !"
  !$OMP END PARALLEL
end program parallel
```

```
dhcp-94-236:~ teyssier$ gfortran -fopenmp prog.f90
dhcp-94-236:~ teyssier$ export OMP_NUM_THREADS=4
dhcp-94-236:~ teyssier$ export OMP_DYNAMIC=true
dhcp-94-236:~ teyssier$ ./a.out
 Hello !
 Hello !
 Hi !
 Hi !
 Hi !
dhcp-94-236:~ teyssier$
```

OpenMP: Parallel Loop

- A parallel loop is a do-loop where each iteration is independent from the others
- Work decomposition by distributing the loop iterations
- The parallel loop is the one that follows immediately after a **DO** directive
- Loop indices are always private integer variables.
- Loops without loop indices or while loop are not supported by OpenMP.

Parallel Loop: Work Scheduling

- The distribution strategy is set by a **SCHEDULE** directive.
- Proper scheduling can optimize the load-balancing of the work.
- By default, the runtime sets a global synchronization the end of the loop.
- The **NOWAIT** directive can remove the barrier.
- One can set many DO directives inside a **PARALLEL** region.

OpenMP: Parallel Loop

```
program parallel
  !$ use OMP_LIB
  implicit none
  integer, parameter :: n=4096
  real,dimension(n) :: a
  integer :: i, rank, nb_threads

  !$OMP PARALLEL PRIVATE(rank,nb_threads)
  rank=OMP_GET_THREAD_NUM()
  nb_threads=OMP_GET_NUM_THREADS()
  !$OMP DO
  do i=1,n
     print*,rank,i
     a(i) = 92290. + real(i)
  end do
  !$OMP END DO
  !$OMP END PARALLEL

end program parallel
```

```
Romain:~ teyssier$ gfortran -fopenmp prog.f90
Romain:~ teyssier$ export OMP_NUM_THREADS=4
Romain:~ teyssier$ ./a.out | more
           1        1025
           0           1
           3        3073
           2        2049
           1        1026
           0           2
           3        3074
           2        2050
           1        1027
           0           3
           3        3075
           2        2051
           1        1028
           0           4
           3        3076
           2        2052
           1        1029
           0           5
           3        3077
           2        2053
           1        1030
           0           6
           3        3078
           2        2054
           1        1031
           0           7
```

```
program parallel
  !$ use OMP_LIB
  implicit none
  integer, parameter :: n=4096
  real,dimension(n) :: a
  integer :: i, rank, nb_threads

  !$OMP PARALLEL PRIVATE(rank,nb_threads)
  rank=OMP_GET_THREAD_NUM()
  nb_threads=OMP_GET_NUM_THREADS()
  !$OMP DO SCHEDULE (STATIC,128)
  do i=1,n
     print*,rank,i
     a(i) = 92290. + real(i)
  end do
  !$OMP END DO
  !$OMP END PARALLEL

end program parallel
```

```
Romain:~ teyssier$ gfortran -fopenmp prog.f90
Romain:~ teyssier$ export OMP_NUM_THREADS=4
Romain:~ teyssier$ ./a.out | more
           2         257
           0           1
           1         129
           3         385
           2         258
           0           2
           1         130
           3         386
           2         259
           0           3
           1         131
           3         387
           2         260
           0           4
           1         132
           3         388
           2         261
           0           5
           1         133
           3         389
           2         262
           0           6
           1         134
           3         390
           2         263
           0           7
```

```
program parallel
  !$ use OMP_LIB
  implicit none
  integer, parameter :: n=4096
  real,dimension(n) :: a
  integer :: i, rank, nb_threads

  !$OMP PARALLEL PRIVATE(rank,nb_threads)
  rank=OMP_GET_THREAD_NUM()
  nb_threads=OMP_GET_NUM_THREADS()
  !$OMP DO SCHEDULE (RUNTIME)
  do i=1,n
     print*,rank,i
     a(i) = 92290. + real(i)
  end do
  !$OMP END DO
  !$OMP END PARALLEL

end program parallel
```

```
Romain:~ teyssier$ export OMP_SCHEDULE="GUIDED,16"
Romain:~ teyssier$ ./a.out | more
           0           1
           2        1025
           3        1793
           1        2369
           0           2
           2        1026
           3        1794
           1        2370
           0           3
           2        1027
           3        1795
           1        2371
           0           4
           2        1028
           3        1796
           1        2372
           0           5
           2        1029
           3        1797
           1        2373
           0           6
           2        1030
           3        1798
           1        2374
           0           7
           2        1031
           3        1799
```

OpenMP: Miscellaneous

- In a parallel loop, if one needs to perform a global operation, one uses the **REDUCTION** clause.

- Logical: .AND., .OR., .EQV., .NEQV., Intrinsic: MAX, MIN, IAND, IOR, IEOR, Arithmetic: +, x, -.

- Each thread computes partial results, which are combined at the end of the parallel loop.

```
program parallel
 implicit none
 integer, parameter :: n=5
 integer            :: i, s=0, p=1, r=1
 !$OMP PARALLEL
 !$OMP DO REDUCTION(+:s) REDUCTION(*:p,r)
    do i = 1, n
      s = s + 1
      p = p * 2
      r = r * 3
    end do
 !$OMP END PARALLEL
 print *,"s =",s, "; p =",p, "; r =",r
end program parallel
```

- The directive **!$OMP BARRIER** forces the synchronization of all threads within a parallel region.

- The directives **ATOMIC** and **CRITICAL** can be used to force a serial variable update and avoid race conditions.

OpenMP: Best Practices

- Minimize the number of parallel regions in the code.
- Adjust the number of threads to the size of the problem (threads come with overhead).
- Always parallelize the outermost loop on the outermost moving index of an array (non-consecutive in memory)
- Conflicts between threads can lead to poor cache memory management (the so-called cache misses). Level 1 and 2 cache memory management is key to OpenMP performance.
- Performance analysis with OpenMP can be done using several functions provided by OMP_LIB to measure time. The most useful is: `time = OMP_GET_WTIME()` gives the elapsed time in seconds.

OpenMP: Best Practices

- The performance of your code will depend on the architecture. Be aware of "false" shared memory architectures !
- On most present day machine, local cache memory is faster than main RAM memory.
- Mapping your data in memory is a key performance factor.
- On Linux, the memory is mapped to a given socket on a "first touch" basis. This is very important for "Non Uniform Memory Access" machines for which the main memory is not really shared but distributed across the node.