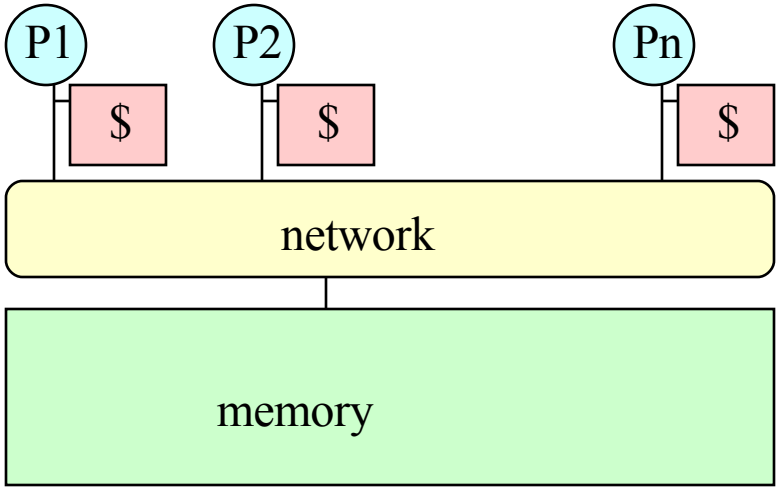
IPM School of Physics
Workshop on High Performance Computing - HPC08

Shared Memory programming paradigm: openMP

Luca Heltai <luca.heltai@sissa.it>
Stefano Cozzini <cozzini@democritos.it>
SISSA - Democritos/INFM

Recap Basic Shared Memory Architecture

- Processors all connected to a large shared memory
- Local caches for each processor
- Cost: much cheaper to cache than main memory



How to program with shared memory ?

- Automatic (implicit) parallelization:
  - compilers do (?) the job for you
- Manual parallelization:
  - Insert parallel directives by yourself to help compilers
  - OpenMP THE standard
- Multi threading programming:
  - more complex but more efficient
  - use a threads library to create task by yourself

Parallelization Strategy

How to parallelize a code with automatic parallelization :

1. Perform profiling to find the time consuming parts
2. Use APC with default settings to auto-parallelize these parts
3. Profile the parallelized code
4. Study the individual behavior of the parallelized parts
5. If the speed-up is not sufficient, check why
6. Try to find options&directives to achieve your goal
7. If found, go to step 3
8. If not found, implement changes in the code and go to step 2

Help the Compiler: Directives and Pragmas

- Special Fortran comments or C pragmas can be inserted in application sources directing the compiler to generate the appropriate parallel code
- Features
  - powerful and easy to use
  - enabled by compiler options
  - code can be still be maintained for portability and serial execution
  - can be mixed with message passing to create hybrid programs
- What happens at compile time ?
  - Parallel region is converted to a subroutine/function
  - shared variables are passed as arguments
  - local/private variables are made local to subroutines/functions

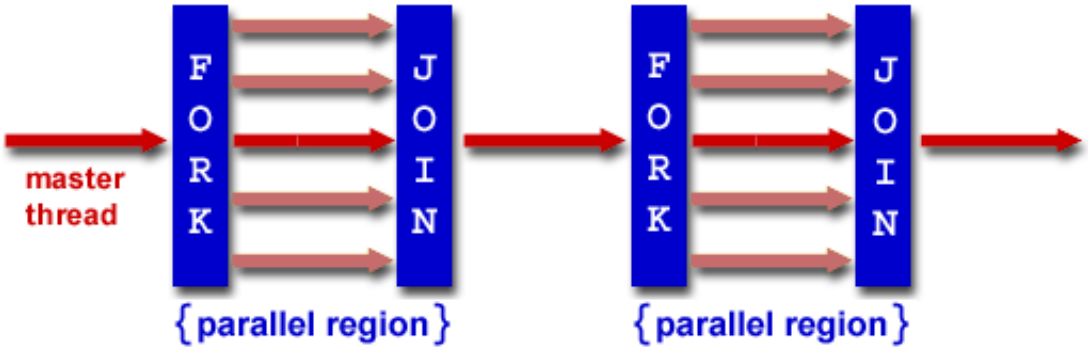
Which set of directives ?

The OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows NT platforms. Jointly defined by a group of major computer hardware and software vendors, OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer.

from WWW.OPENMP.ORG

OpenMP programming model

- Explicit Parallelism:
  - OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.
- Fork - Join Model:



Terminology

- OpenMP Team := Master + Workers
- A Parallel Region is a block of code executed by all threads simultaneously
  - The master thread always has thread ID 0
  - Thread adjustment (if enabled) is only done before entering a parallel region
  - An "if" clause can be used to guard the parallel region; in case the condition evaluates to "false", the code is executed serially
- A work-sharing construct divides the execution of the enclosed code region among the members of the team; in other words: they split the work

Examples: Assertions

```
do i=1,n
    A(I+K)=A(I)+B(I)
end do
```

- If K>-1 or less then N no dependencies !
- Compiler cannot know this...

```
C$ASSERT NO_DEPENDENCIES
do i=1,n
    A(I+K)=A(I)+B(I)
end do
```

Examples: Pragmas

```
do i=1,n
    A(I+K)=A(I)+B(I)
end do
```

- Make the execution parallel !

```
C$OMP PARALLEL DO
do i=1,n
    A(I+K)=A(I)+B(I)
end do
```

Fine-Grained vs. Coarse-grained

- Fine-grained parallelism (loop decomposition)
  - can be implemented incrementally (one loop a time)
  - does not require a deep knowledge of the code
  - a lot of loop have to be parallelized to achieve a decent speed-up
  - potentially many synchronization points
- Coarse-grained parallelism (domain decomposition)
  - parallelizes larger loops at higher levels, enclosing many smaller loops
  - more code is parallelized
  - fewer loop is parallelized, reducing overhead
  - requires deeper knowledge of the code

Local and shared data/1

- Shared Memory does not mean that all data is actually shared
- There is often a need for “local” data as well
- Consider the following example:

```
for (i=0; i<10; i++)
 a[i]= b[i] + c[i];
```

- Let’s assume we run this on 2 processors:

processor 1	for I=0,2,4,6,8
processor 2	for I=1,3,5,7,9

Local and shared data/2

Processor 1

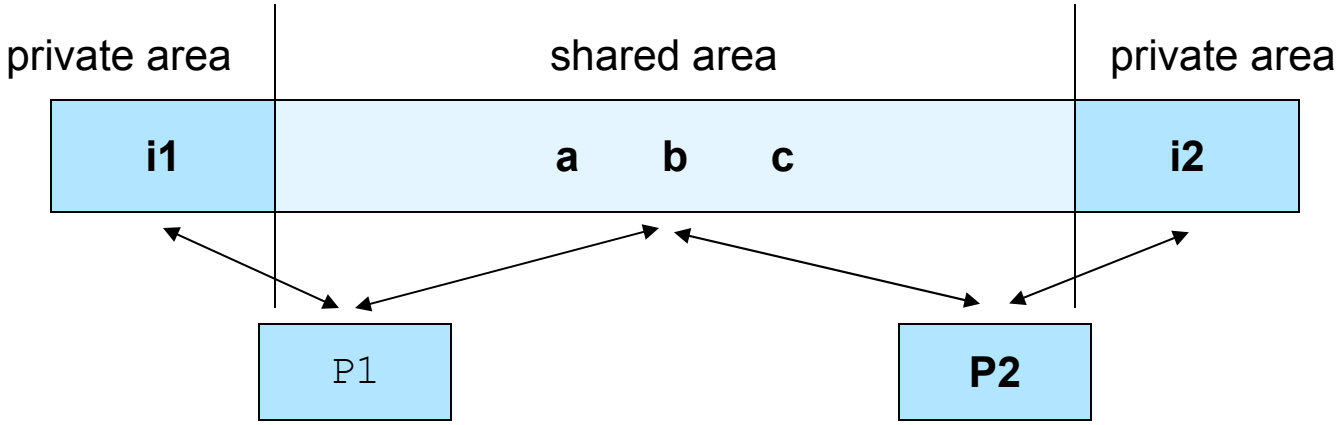
```
For i1=0,2,4,6,8,do:
a[i1]=b[i1]+c[i1];

Read:  b,c   Write:  a
```

Processor 2

```
For i2=1,3,5,7,9 do:
a[i2]=b[i2]+c[i2];

Read:  b,c   Write:  a
```



hello world in openMP

```
#include <omp.h>
int main()  {
  int iam =0, np = 1;
#pragma omp parallel private(iam, np)
    {
#if defined (_OPENMP)
      np = omp_get_num_threads();
      iam = omp_get_thread_num();
#endif
      printf("Hello from thread %d out of %d \n", iam, np);
    }
}
```

compile&run the code

```
cerbero~ 37>icc -openmp openMP.c
openMP.c(4) : (col. 1) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
cerbero~ 38>./a.out
Hello from thread 0 out of 2
Hello from thread 1 out of 2
```

Gnu compiler (only from version 4.3 on):

```
gcc -fopenmp openMP.c -o hello
```

Example: Manual Parallelization

```
program tests
      implicit none
      integer i,n
      parameter(n=10000000)
      real*8 a(n),s

!$OMP PARALLEL
!$OMP PARALLEL DO SHARED(a,n) PRIVATE(i)
      do i=1,n
      a(i)=dcos(dfloat(i))**2+1.d0
      enddo
!$OMP END PARALLEL DO
!$OMP PARALLEL DO SHARED(a,n) PRIVATE(i) REDUCTION(+:s)
      do i=1,n
        s= s +a(i)
      end do
!$OMP END PARALLEL DO
!$OMP END PARALLEL
      print*, a(1),a(n),s
      stop
      end
```

Directives.

- *C: directives are case sensitive*
  - *Syntax:* #pragma omp directive [clause [clause] …]
- *Continuation: use \ in pragma*
- *Conditional compilation:* _OPENMP *macro is set*

- *Fortran: directives are case insensitive*
  - *Syntax:* sentinel directive [clause [[,] clause]…]
  - *The sentinel is one of the following:*
    - ✔ *!$OMP or C$OMP or *$OMP* *(fixed format)*
    - ✔ *!$OMP* *(free format)*
- *Continuation: follows the language syntax*
- *Conditional compilation:* !$ *or* C$ -> 2 spaces

A more "complex" example

```
int main (int argc, char *argv[]) {

    unsigned int rank = 0, threads = 1;
    #pragma omp parallel private(rank, threads)
    {
      rank    = omp_get_thread_num ();
      threads = omp_get_num_threads();
      #pragma omp master
      {
         cout << "I'm the master: "
              << rank << endl;
      }
      #pragma omp for
      for(int i=0; i<4; ++i)
        sleep(1);
    }
    return 0;
}
```

Run time library routines

Name	Functionality
omp_set_num_threads	Set number of threads
omp_get_num_threads	Return number of threads in team
omp_get_max_threads	Return maximum number of threads
omp_get_thread_num	Get thread ID
omp_get_num_procs	Return maximum number of processors
omp_in_parallel	Check whether in parallel region
omp_set_dynamic	Activate dynamic thread adjustment (but implementation is free to ignore this)
omp_get_dynamic	Check for dynamic thread adjustment
omp_set_nested	Activate nested parallelism (but implementation is free ignore this)
omp_get_nested	Check for nested parallelism
omp_get_wtime	Returns wall clock time
omp_get_wtick	Number of seconds between clock ticks

openMP env variables

- OpenMP provides four environment variables for controlling the execution of parallel code.
- All environment variable names are uppercase.
- Most important: OMP_NUM_THREADS

  Sets the maximum number of threads to use during execution.
- Others:
  - OMP_SCHEDULE: Applies only to DO, PARALLEL DO (Fortran) and for, parallel for (C/C++) directives which have their schedule clause set to RUNTIME. The value of this variable determines how iterations of the loop are scheduled on processors. For example:

    setenv OMP_SCHEDULE "guided, 4"

    setenv OMP_SCHEDULE "dynamic"
  - OMP_DYNAMIC: Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. Valid values are TRUE or FALSE.
  - OMP_NESTED: Enables or disables nested parallelism. Valid values are TRUE or FALSE

OpenMP libraries

- Today many vendor provided libraries are openMP enabled
- OpenMP Parallelism comes for free
- OpenMP libraries can be used jointly with MPI internodes in a mixed MPI/openMP programming model.
- Examples for AMD ACML and Intel MKL are provided in afternoon sessions

Conclusions

- Auto-parallelizing compilers can do a lot but they are limited in their capabilities to analyze data dependencies.
- Manual parallelization with directives is easy to implement, however care must be given to treatment of shared data
- OpenMP is the standard for this approach
- It is important to understand the exact semantic of the parallelization directives
- To obtain scalable performance it is often necessary to parallelize at coarse-grain level