Parallel Programming: Introduction to OpenMP

A Shared-Memory Programming Model

12/30/2015
Two families of parallel machines

- **Shared-memory architecture:**
  - A set of processors which have access to a common memory and connected by a bus
  - Bus contention prevents bus architectures from scaling
  - Generally do not comprise more than 32 processors
  - Symmetric Multi Processing (SMP) machines

- **Distributed-memory architecture:**
  - Each processor has its own private memory and connected by a network
  - Highly scalable
  - Clusters

→ Each of these two parallel machines has its advantages and disadvantages.
Two parallel programming model

Shared-memory: thread (e.g., OpenMP)

"A thread is a runtime entity that is able to independently execute a stream of instructions"
Chapman et al. 2008

Distributed-memory: massage passing (e.g., MPI)

TOP500 statistics of interconnect
• Shared-memory machines exist for a long time.
• In the past, each vendor had its own standard of compiler directives and libraries to make use of their specific parallel computers.
• **OpenMP** is the result of large agreement between hardware vendors and compiler developers.
• **Open**: the standard is defined through a specification accessible to anyone
• **MP**: Multi Processing

• **OpenMP** is now an industry standard for the development of parallel program on shared-memory machines.

• **OpenMP** is owned, written and maintained by the OpenMP Architecture Review Board.
  
  Members: AMD, IBM, Intel, Cray, HP, Fujitsu, Nvidia, NEC, Microsoft, Texas Instruments, Oracle Corporation, ...

• OpenMP represents a collection of **compiler directives, library routines** and **environment variables**.
OpenMP Directives
Instructions which tell compiler how to execute in a parallel way

```fortran
program t1
implicit none

!$omp parallel
write(*,*) 'This is a parallel program.'
!$omp end parallel

end program
```

```bash
>gfortran -fopenmp openmp_t1.f90
>a.exe
    This is a parallel program.
    This is a parallel program.
    This is a parallel program.
    This is a parallel program.
```

Advantage of directives hidden behind the comment symbols:
    a single code for both sequential and parallel environments
The fork-join programming model

master thread

!$omp parallel

!$omp end parallel
OpenMP Directives

- Create threads for parallel processing
  - `parallel`
- Share works to the threads
  - `do`
  - `sections`
  - `single`
- Synchronize the threads
  - `barrier`
  - `ordered`
parallel

Creating threads for parallel processing

```fortran
program t2
implicit none
integer :: omp_get_thread_num

!$omp parallel
write(*,*) 'This is processor', omp_get_thread_num()
!$omp end parallel
end program
```

```
>a.exe
This is processor 3
This is processor 1
This is processor 0
This is processor 2
```

**omp_get_thread_num():** OpenMP intrinsic function; returns an identity of a thread

The sequence changes in each run
How to use parallelism to gain performance?

The works you want to do:

```fortran
do i = 1, 8
    tmp = sqrt(i*2.5)
    a(i) = a(i) + tmp
end do
```

Assign the whole works to each threads:

```
!$omp parallel
  do i = 1, 8
    tmp = sqrt(i*2.5)
    a(i) = a(i) + tmp
  end do
!$omp end parallel
```

No performance gain!
Therefore, works should be shared among the threads.
program t3
implicit none
real :: tmp, a(8)=2.0
integer :: i

!$omp parallel default(none) shared(a) private(tmp,i)
$omp sections
$omp section
do i = 1, 4
  tmp = sqrt(i*2.5)
a(i) = a(i) + tmp
end do
$omp section
do i = 5, 8
  tmp = sqrt(i*2.5)
a(i) = a(i) + tmp
end do
$omp end sections
$omp end parallel
end program
Clause: default, shared, private

- **default**: Set the default scope of all variables (none, shared, private)
- **shared**: All threads share the same variable
- **private**: Each thread has its own copy of the variable

```
!$omp parallel default(none) shared(a) private(tmp,i)
 !$omp sections
 !$omp section
 do i = 1, 4
   tmp = sqrt(i*2.5)
   a(i) = a(i) + tmp
 end do
 !$omp section
 do i = 5, 8
   tmp = sqrt(i*2.5)
   a(i) = a(i) + tmp
 end do
 !$omp end sections
 !$omp end parallel
```

If `tmp` is assign `shared`, the **race condition** occurs.

Thd0: Assign `tmp` for `i = 1`
Thd1: Assign `tmp` for `i = 5`
Thd0: Add `tmp` to `a(1)`
...
Distributing do loop to threads automatically

program t4
implicit none
real :: tmp, a(8)=2.0
integer :: i
!$omp parallel default(none) shared(a) private(tmp,i)
!$omp do
do i = 1, 8
   tmp = sqrt(i*2.5)
   a(i) = a(i) + tmp
end do
!$omp end do
!$omp end parallel
end program

The schedule clause can control the way of sharing works
program t5
implicit none
real :: tmp, a(8)=2.0
integer :: i

!$omp parallel default(none) shared(a) private(tmp,i)
!$omp single
do i = 1, 8
   tmp = sqrt(i*2.5)
   a(i) = a(i) + tmp
end do
!$omp end single
!$omp end parallel

end program
Combined syntax

parallel do
parallel sections
parallel single

```c
!$omp parallel do default(none) shared(a) private(tmp,i)
do i = 1, 8
  tmp = sqrt(i*2.5)
  a(i) = a(i) + tmp
end do
!$omp end parallel do

!$omp parallel sections default(none) shared(a) private(tmp,i)
!$omp section
do i = 1, 4
  tmp = sqrt(i*2.5)
  a(i) = a(i) + tmp
end do
!$omp section
do i = 5, 8
  tmp = sqrt(i*2.5)
  a(i) = a(i) + tmp
end do
!$omp end parallel sections
```
program t6
implicit none
real :: tmp, a(8)=2.0
integer :: i, tid, omp_get_thread_num

!$omp parallel default(none) shared(a) private(tmp,i)
tid = omp_get_thread_num()
!$omp do nowait
  do i = 1, 8
    tmp = sqrt(i*2.5)
    a(i) = a(i) + tmp
  end do
!$omp end do
if (tid == 0) a = 2.0*a
!$omp barrier
!$omp do
  do i = 1, 8
    a(i) = sqrt(a(i))
  end do
!$omp end do
!$omp end parallel
end program
ordered

Executing a block of code in a parallel loop in sequential order

```plaintext
program t7
implicit none
integer :: i, a, omp_get_thread_num
!
$omp parallel do ordered
do i = 1, 10
   a = i * 5
$omp ordered
   write(*,*) 'This is processor', omp_get_thread_num(), &
      'writes i=', i
$omp end ordered
end do
$omp end parallel do
end program
```

```
> a.exe
This is processor 0 writes i= 1
This is processor 0 writes i= 2
This is processor 0 writes i= 3
This is processor 1 writes i= 4
This is processor 1 writes i= 5
This is processor 1 writes i= 6
This is processor 2 writes i= 7
This is processor 2 writes i= 8
This is processor 2 writes i= 9
This is processor 3 writes i= 10
```
**reduction**

Updating a shared variable with a supported operator

```fortran
program t8
implicit none
integer :: i, isum = 0

!$omp parallel do default(none) private(i) reduction(+:isum)
do i = 1, 10
  isum = isum + i
end do
!$omp end parallel do
write(*,*) isum

end program
```

> a.exe
   55

**Supported operators:**
+ , * , - , max, min,
 .and., .or., .eqv., .neqv., ...
Example:

```fortran
program openmp_ft
implicit none
integer,parameter :: nna = 1024, nrep = 500
real :: a(0:nna-1), fa_i(0:nna-1,nrep) = 0.0, &
          fa_r(0:nna-1,nrep) = 0.0
real :: cf = 2.0*acos(-1.0)/nna
integer :: n, j, irep
real(kind=8) :: t0, t1, t2, t3, t4, omp_get_wtime

!-- Give values
call random_number(a)

t0 = omp_get_wtime()
do irep = 1, nrep
   do n = 0, nna-1
      do j = 0, nna-1
         fa_r(n,irep) = fa_r(n,irep) + a(j)*cos(cf*n*j)
         fa_i(n,irep) = fa_i(n,irep) - a(j)*sin(cf*n*j)
      end do
   end do
end do
```

**omp_get_wtime()**: OpenMP intrinsic function; double precision real; returns wall clock time (cpu_time does not work in a multi-thread program)

---

Sequential processing
t1 = omp_get_wtime()
do irep = 1, nrep
   do n = 0, nna-1
!$omp parallel do default(none) shared(cf,a,fa_r,fa_i,irep,n) &
!$omp   private(j)
   do j = 0, nna-1
      fa_r(n,irep) = fa_r(n,irep) + a(j)*cos(cf*n*j)
      fa_i(n,irep) = fa_i(n,irep) - a(j)*sin(cf*n*j)
   end do
!$omp end parallel do
   end do
end do

t2 = omp_get_wtime()
do irep = 1, nrep
!$omp parallel do default(none) shared(cf,a,fa_r,fa_i,irep) &
!$omp   private(n,j)
   do n = 0, nna-1
   do j = 0, nna-1
      fa_r(n,irep) = fa_r(n,irep) + a(j)*cos(cf*n*j)
      fa_i(n,irep) = fa_i(n,irep) - a(j)*sin(cf*n*j)
   end do
   end do
!$omp end parallel do
end do
t3 = omp_get_wtime()

!$omp parallel do default(none) shared(cf,a,fa_r,fa_i) &
!$omp private(n,j,irep)
do irep = 1, nrep
  do n = 0, nna-1
    do j = 0, nna-1
      fa_r(n,irep) = fa_r(n,irep) + a(j)*cos(cf*n*j)
      fa_i(n,irep) = fa_i(n,irep) - a(j)*sin(cf*n*j)
    end do
  end do
end do
!$omp end parallel do

t4 = omp_get_wtime()

write(*,*) 'Elapsed time (sequential processing): ', t1-t0
write(*,*) 'Elapsed time (fine-grain parallelism): ', t2-t1
write(*,*) 'Elapsed time (coarse-grain 1 parallelism): ', t3-t2
write(*,*) 'Elapsed time (coarse-grain 2 parallelism): ', t4-t3

end program
About 4 times speed up are achieved for the most coarse-grain parallel. It proves the ease-of-use and high performance of OpenMP.
**Homework**

Consider the multiplication of a matrix $b_{ij}$ and a vector $c_i$:

$$
\begin{bmatrix}
  b_{11} & b_{12} & b_{13} & \cdots & b_{1n} \\
  b_{21} & \phantom{b_{12}} & \phantom{b_{13}} & \cdots & \phantom{b_{1n}} \\
  b_{31} & \phantom{b_{12}} & \phantom{b_{13}} & \cdots & \phantom{b_{1n}} \\
  \vdots & \phantom{b_{12}} & \phantom{b_{13}} & \cdots & \phantom{b_{1n}} \\
  b_{m1} & b_{m2} & b_{m3} & \cdots & b_{mn}
\end{bmatrix}
\begin{bmatrix}
  c_1 \\
  c_2 \\
  c_3 \\
  \vdots \\
  c_n
\end{bmatrix} =
\begin{bmatrix}
  a_1 \\
  a_2 \\
  a_3 \\
  \vdots \\
  a_m
\end{bmatrix}
$$

$$
a_1 = b_{11}c_1 + b_{12}c_2 + \cdots + b_{1n}c_n \\
a_2 = b_{21}c_1 + b_{22}c_2 + \cdots + b_{2n}c_n \\
\vdots \\
a_m = b_{m1}c_1 + b_{m2}c_2 + \cdots + b_{mn}c_n
$$

1. Compile and execute the non-parallelized program to get the cpu time.
2. Use OpenMP to parallelize the two different accesses of arrays (red and blue), and find out how much cpu time it will gain.
3. Use `default(none)` clause and specify $a$, $b$, $c$, $i$, $j$ as shared or private
4. Use `omp_get_wtime()` to measure the performance.