Introduction to OpenMP

Xiaoxu Guan

High Performance Computing, LSU

April 6, 2016
Overview

- Overview of Parallel Computing
Overview

- Overview of Parallel Computing
- Parallel Programming on *Shared*-Memory and *Distributed*-Memory Machines
Overview

• Overview of Parallel Computing
• Parallel Programming on Shared-Memory and Distributed-Memory Machines

• Introduction to OpenMP
  ◦ Prerequisite for Parallel Computing;
  ◦ Constructs for Parallel Execution;
  ◦ Data Communications;
  ◦ Synchronization;
Overview

• Overview of Parallel Computing

• Parallel Programming on Shared-Memory and Distributed-Memory Machines

• Introduction to OpenMP
  ◦ Prerequisite for Parallel Computing;
  ◦ Constructs for Parallel Execution;
  ◦ Data Communications;
  ◦ Synchronization;

• OpenMP Programming: Directives/Pragams, Environment Variables, and Run-time Libraries
  ◦ Variables Peculiar to OpenMP Programming;
  ◦ Loop Level Parallelism;
  ◦ Non-Loop Level Parallelism;
Overview

• Overview of Parallel Computing
• Parallel Programming on **Shared**-Memory and **Distributed**-Memory Machines
• Introduction to OpenMP
  ◦ Prerequisite for Parallel Computing;
  ◦ Constructs for **Parallel Execution**;
  ◦ **Data Communications**;
  ◦ **Synchronization**;
• OpenMP Programming: Directives/Pragams, Environment Variables, and Run-time Libraries
  ◦ Variables Peculiar to OpenMP Programming;
  ◦ Loop Level Parallelism;
  ◦ Non-Loop Level Parallelism;
• **Summary and Further Reading**
Overview of parallel computing

• Why parallel or concurrency computing?
• Goes beyond the single-core capability (memory and flops per unit time), and therefore increase performance;
• Reduces wall-clock time, and saves energy;
• Finish those impossible tasks in my lifetime;
• Handles larger and larger-scale problems;
• There is no free lunch, however!
• Different techniques other than serial coding are needed;
• Effective parallel algorithms in terms of performance;
• Increasing flops per unit time is one of our endless goals in the HPC community;
• Think in parallel;
• Start parallel programming as soon as possible;
Parallel programming

- Parallel programming environment;
  - Essential language extensions to the existing language (Fortran 95);
  - New constructs for directives/pragmas to existing serial programs (**OpenMP** and HPF);
  - Run-time libraries that support data communication and synchronization (**MPI** and Pthreads);
- **OpenMP** stands for **Open Multi-Processing** (API);
- **OpenMP** is one of the directives/pragmas approaches that support parallelism on **shared** memory systems;
- **OpenMP** is supported by Fortran, and C/C++;
- **OpenMP** allows us to start from a serial code and provides an incremental approach to express parallelism;
The “Three Knights” in OpenMP

(1) **Directives/pragmas** need to express parallelism;
(2) **Run-time libraries** can dynamically control or change code execution at run-time;
(3) **Environment variables** specify the run-time options;

- How does OpenMP achieve parallel computing?
  - Specify parallel execution – parallel constructs allowing parallel execution;
  - Data communication – data constructs for communication among threads;
  - Synchronization – synchronization constructs;
- OpenMP directives/pragmas:
  - Fortran: !$omp, c$omp, or !$omp [clauses]
  - C/C++: #pragma [clauses]
Parallel execution

- Constructs for parallel execution: OpenMP starts with a single thread, but it supports the directives/pragmas to spawn multiple threads in a fork-join model;

- OpenMP `do` and `parallel` directives;
- OpenMP also allows you to change the number of threads at run-time;
Data communication

- When multiple threads were spawned, each thread was assigned to a unique thread ID from 0 to \( N - 1 \). Here \( N \) is the total number of threads;
- The key point is that there are three types of variables: private, shared, and reduction variables;
- At run-time, there is always a common region in global memory that allows all threads to access, and this memory region is used to store all shared variables;
- Each thread was also assigned a private memory region to store all private variables; Thread \( a \) cannot access the private variables stored in the private memory space for thread \( b \);
- Data Communications are achieved through read and write operations among the threads;
Synchronization

- In OpenMP, synchronization is used to (1) control the access to *shared* variables and (2) coordinate the workflow;
- Event and mutual exclusion synchronization;
- **Event synchronization** includes *barrier* directives, which are either explicit or implicit; a thread has to wait until all threads reach the same point;
- **Mutual exclusion** is supported through *critical*, *atomic*, *single*, and *master* directives. All these are used to control how many threads, which thread, or when a thread can execute a specified code block or modify shared variables;
- Be careful with synchronization!
Compile OpenMP code

- Compiler options that enable OpenMP directive/pragmas:

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Fortran</th>
<th>C</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>ifort -openmp</td>
<td>icc -openmp</td>
<td>icpc -openmp</td>
</tr>
<tr>
<td>PGI</td>
<td>pgf90 -mp</td>
<td>pgcc -mp</td>
<td>pgCC -mp</td>
</tr>
<tr>
<td>GCC</td>
<td>gfortran -fopenmp</td>
<td>gcc -fopenmp</td>
<td>g++ -fopenmp</td>
</tr>
</tbody>
</table>

- If the above flags are left out, OpenMP code is compiled as serial code (except Intel compilers but -openmp-stubs needed);

- Load modules on the HPC or LONI machines:
  $ module load [package name]
  $ soft add [+package name] (resoft) # intel, pgi, or gcc.

- Set up an environment variable:
  $ export OMP_NUM_THREADS=[number of threads]
Loop level parallelism
First OpenMP “Hello World!” in Fortran and C

Fortran (hello.f90)

```fortran
program hello_world
    implicit none

    integer :: id, omp_get_thread_num

    !$omp parallel
    id = omp_get_thread_num()
    write(*,'(1x,a,i3)') "Hello World! from", id
    !$omp end parallel

end program hello_world
```

$ export OMP_NUM_THREADS=20
    # for instance, on SuperMIC in bash shell
$ ifort -o hello hello.f90 -openmp
First OpenMP “Hello World!” in Fortran and C

C (hello.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main( ) {
    int id;

    //pragma omp parallel {
    id = omp_get_thread_num();
    printf("Hello World! from %3d\n", id);
    //}
}
```

$ export OMP_NUM_THREADS=20
   # for instance, on SuperMIC in bash shell
$ icc -o hello hello.c -openmp
Loop-level parallelism

- Loop-level parallelism is one of the fine-grained approaches supported by OpenMP;
- `parallel do` directives in Fortran and `parallel for` pragmas in C/C++;

```fortran
!$omp parallel do [clauses]
do i = imin, imax, istep
   loop body ...
end do
!$omp end parallel do
```

```c/c++
#pragma omp parallel for [clauses]
for (i = imin; i < imax; increment_expr)
{
   loop body ...;
}
```
Loop-level parallelism

### Fortran

```fortran
!$omp parallel [clauses]
!$omp do [clauses]
do i = imin, imax, istep
  loop body ...
end do
!$omp end do
!$omp end parallel
```

### C/C++

```c
#pragma omp parallel [clauses]
{
  #pragma omp for [clauses]
  for (i = imin; i < imax; increment_expr)
    {
      loop body ...;
    }
}
```
Loop-level parallelism

- How about nested multiple loops? Where do we add `parallel for`, right above outer loop or inner loop?

```
1 for (i = imin; i < imax; increment_i)   C/C++
2 {                                         (inner loop)
3   #pragma omp parallel for
4     for (j = jmin; j < jmax; increment_j)
5       { loop body ...; }
6 }
```

```
1 #pragma omp parallel for                     C/C++ (outer loop)
2 for (i = imin; i < imax; increment_i)
3 {                                           
4     for (j = jmin; j < jmax; increment_j)
5       { loop body ...; }
6 }
```
More words on parallel loops

- OpenMP only supports Fortran `do` loops and C/C++ `for` loops that the number of loop iterations is known for at run-time;
- However, it doesn’t support other loops, including `do-while` and `repeat-until` loops in Fortran and `while` loops and `do-while` loops in C/C++. In these cases, the trip count of loop is unknown before entering the loop;
- Loop body has to follow `parallel do` or `parallel for` immediately, and nothing in between them!
- There is an implicit `barrier` at the end of `parallel do` or `for` loops;
- All loops must have a single entry point and single exit point. We are not allowed to jump into a loop or branch out of a loop;
How to control loops?

- Once we entered the parallel region, for some variables multiple threads need to use the same named variables, but they store different values at different memory locations; these variables are called **private** variables;
- This leads to the fact that all private variables are **undefined** or **uninitialized** before entry and after exit from parallel regions;
- The **shared** variables are also necessary to allow data communication between threads;
- **Default** scopes for variables: By default all the variables are considered to be **shared** in parallel regions, unless they are explicitly declared as **private**, **reduction**, or **other** types;
- Remember Fortran and C/C++ may have different settings regarding default rules;
How to control loops?

- Let’s see how we can do it, for instance, in parallel loops;
- OpenMP provides a means to change the default rules;
- Clauses `default(none)`, `default(private)`, and `default(shared)` in Fortran;
- But only `default(none)` and `default(shared)` in C/C++;

```fortran
ALLOCATE( da(1:nsize), db(1:nsize) )
!$omp parallel do default(none),
!$omp private(i,temp),
!$omp shared(imin,imax,istep,scale,da,db)
do i = imin, imax, istep
temp = scale * da(i)
da(i) = temp + db(i)
end do
!$omp end parallel do
```
How to control loops?

- OpenMP reduction operations;
- The reduction variable is very special that it has both characters of private and shared variables;
- Compiler needs to know what type of operation is associated with the reduction variable; operation = +, *, max, min, etc;
- reduction(operation : variables_list)

```fortran
1  ALLOCATE( da(1:nsize) )
2  prod = 1.0d0
3  !$omp parallel do default(none), private(i), &
4  !$omp reduction(* : prod)
5    do i = imin, imax, istep
6      prod = prod * da(i)
7    end do
8  !$omp end parallel do
```

What happens if we compile it?
How to control loops?

- Two special “private” variables: `firstprivate` and `lastprivate`; they are used to initialize and finalize some private variables;
- `firstprivate`: upon entering a parallel `do/for`, the private variable for each `slave` thread has a copy of the `master` thread’s value;
- `lastprivate`: upon exiting a parallel `do/for`, no matter which thread executed the last iteration (sequential), the private variable was copied back to the `master` thread;
- Why do we need them? (1) all private variables are undefined outside of a parallel region, (2) they provide a simply way to exchange data to some extent through these special private variables;
How to control loops?

- In a parallel region, a given variable can only be one of private, shared, or reduction, but it can be both of firstprivate and lastprivate;

```c
1 double ashift = shift;
2 #pragma omp parallel for default(none), 
3    firstprivate(ashift), shared(a), 
4    private(i)
5 {
6   for (i = imin; i <= imax; ++i)
7     {
8       ashift = ashift + (double) i;
9       a[i] = a[i] + ashift;
10     }
11 }
```
How to control loops?

- Exception of the default rules: Fortran and C/C++ behave differently;
- The index in a parallel loop is always private. The index in a sequential loop is also private in Fortran, but is shared in C by default!
- Is the following code correct?
- Has the loop $j$ been parallelized?

```c
#pragma omp parallel for
for (i = imin; i <= imax; ++i)
{
  for (j = jmin; j <= jmax; ++j)
    a[i][j] = (double) (i + j);
}
```

- Do we have the same issues in the Fortran version?
How to control loops?

- Exception of the default rules. Fortran and C/C++ behave differently;
- The index in a parallel loop is always private. The index in a sequential loop is also private in Fortran, but is shared in C by default!
- Is the following code correct?
- Has the loop j been parallelized?

```
1  #pragma omp parallel for private(i,j)   C/C++
2  for (i = imin; i <= imax; ++i)
3  {
4      for (j = jmin; j <= jmax; ++j)
5          a[i][j] = (double) (i + j) ;
6  }
```

- Do we have the same issues in the Fortran version?
How to control loops?

- Parallelize multiple nested loops;
- The `collapse(n)` ($n \geq 1$) for nested parallel loops;
- Each thread takes a chunk of the $i$ loop and a chunk of the $j$ loop at the same time;
- No statements in between;

```c
#pragma omp parallel for private(i,j), "collapse(2)"
for (i = imin; i <= imax; ++i)
{
    for (j = jmin; j <= jmax; ++j)
        a[i][j] = (double) (i + j);
}
```
Restrictions on parallel loops

- Not all loops are parallelizable. What can we do?
- Think parallely and change your algorithms;
- We have to maintain the correctness of the results;
- One of the common mistakes is data race;

```c
#pragma omp parallel for
{
  for (i = imin; i <= imax; ++i)
    r[i] = r[i] + r[i-1] ;
}
```

- **Data race** means that in a parallel region, the same memory location is referred by **two or more** statements, and at least one of them is a **write** operation;
- Data race requires more attention and might lead to incorrect results!
Restrictions on parallel loops

- A closer look at the data race: let's run it on 2 threads and assume that $r[0]=a; r[1]=b; r[2]=c$; and $imin=1; imax=2$.
- Note $r[1]$ is referred twice, and thus we have two scenarios:

<table>
<thead>
<tr>
<th>if thread 0 finished first</th>
<th>if thread 1 finished first</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread 0</td>
<td>thread 0</td>
</tr>
<tr>
<td>thread 1</td>
<td>thread 1</td>
</tr>
<tr>
<td>$i = 1$</td>
<td>$i = 2$</td>
</tr>
<tr>
<td>$i = 2$</td>
<td>$i = 1$</td>
</tr>
<tr>
<td>$r[0]=a$</td>
<td>$r[1]=b$</td>
</tr>
<tr>
<td>$r[1]=a+b$</td>
<td>$r[0]=a$</td>
</tr>
<tr>
<td>$r[2]=a+b+c$</td>
<td>$r[1]=b+a$</td>
</tr>
</tbody>
</table>

- OpenMP standard does not guarantee which thread finishes first or later;
How to control loops again?

- OpenMP supports three loop schedulings as clauses:
  static, dynamic, and guided in the code, plus run-time
  scheduling;
- schedule( type[, chunk_size] )

For static, if chunk_size is given, loop iterations are divided
into multiple blocks and each block contains chunk_size
iterations. The iterations will be assigned to threads in a
round-robin fashion. If chunk_size is not present, the loop
iterations will be (nearly) evenly divided and assigned to each
thread.

<table>
<thead>
<tr>
<th>thread 0</th>
<th>thread 1</th>
<th>thread 2</th>
<th>thread 3</th>
</tr>
</thead>
</table>
| 1  
5  
9  
13 | 2  
6  
10  
14 | 3  
7  
11 | 4  
8  
12 |

14 iterations
on 4 threads
in round-robin fashion
How to control loops again?

- For `dynamic`, if `chunk_size` is given, the partition is almost the same as those of `static`. The difference is that with `static`, the mapping between loop iterations and threads are done during *compilation*, while for `dynamic`, it will be done at *run-time* (therefore, more potentially overhead); if `chunk_size` is not present, then it was set to 1.

- The *guided* scheduling means the `chunk_size` assigned to threads decreases exponentially;

- Run-time scheduling: set the environment variable `OMP_SCHEDULE`;

  - `$ export OMP_SCHEDULE=10`, for instance;

- Each scheduling has its own pros and cons, so be careful with `chunk_size` and potential overhead;
Non-loop-level parallelism
Parallel regions

- In addition to `parallel do` or `for`, most importantly OpenMP supports the parallelism beyond loop levels;

```fortran
1  !$omp parallel [clauses]
2       code block
3  !$omp end parallel
```

```c++
1  #pragma omp parallel [clauses]
2       { code block ; }
```

- Each thread in the parallel team executes the same block of code, but with different data;
- In parallel directives, **clauses** include:
  private(list), shared(list), reduction(operation : list), default(none | private | shared), if(logical operation), copyin(list);
Any differences?

```fortran
!$omp parallel
id = omp_get_thread_num()
write(*,*) "Hello World! from ", id
!$omp end parallel
```

```fortran
!$omp parallel
do k = 1, 5
  id = omp_get_thread_num()
  write(*,*) "Hello World! from ", id, k
end do
!$omp end parallel
```

```fortran
!$omp parallel do
do k = 1, 5
  id = omp_get_thread_num()
  write(*,*) "Hello World! from ", id, k
end do
!$omp end parallel do
```
Global variables in OpenMP

- In addition to `automatic` or `static` variables in Fortran and C/C++, we also need `global` variables;
- `Command blocks` or `modules` in Fortran, while `globals` in C/C++, and we might have issues with `private` variables;
- `Global/local` variables between different code units for a given thread;
- `Private/shared` variables between multiple threads in a given code unit;
- The default data scoping rule is only apply to its `lexical` region, and all rest are `shared`; How can we make `private` variables “propagate” to `other` code units?
- OpenMP introduced the `threadprivate` directive to solve data scoping issues;
Global variables in OpenMP

- !$omp threadprivate (list_common_variables) in Fortran;
- #pragma omp threadprivate (list_variables) in C/C++;
- We have **global** but **private** variables;
- The **threadprivate** variables are special private variables; thus thread \(a\) cannot access the **threadprivate** variables stored on thread \(b\);
- The **threadprivate** variables persist from one parallel region to another, because they are globals;
- Furthermore, OpenMP supports the **copyin** (list) clause to initialize global variables on slave threads to be the values on the master thread;
- #pragma omp parallel copyin (a,b,c) { code block; }
- Sounds familiar with the Intel Xeon Phi programming?
Work-sharing directives

```
1 program mapping
2 implicit none
3 integer :: i,id,nothread, &
4   omp_get_thread_num, omp_get_num_threads
5
6 !$omp parallel private (k,id), shared(nothread)
7   id = omp_get_thread_num()
8   nothread = omp_get_num_threads()
9 !$omp do
10   do k = 1, 40
11   write(*,'(1x,2(a,i4))') "id = ",id, " k = ",k
12   end do
13 !$omp end do [nowait]
14 !$omp end parallel
15 end program mapping
```
The point is that !$omp do directive does not spawn threads. Instead, only !$omp parallel spawns multiple threads!

$id = omp_get_thread_num()$

!$omp do

!$omp do needs to be embedded in an existing parallel region.
Work-sharing directives

- Work-sharing constructs do not spawn multiple threads; they need to be embedded in a parallel region; if not, only one thread will run work-sharing constructs;
- There is an implicit barrier at the end of a work-sharing construct, but no implicit barrier upon the entry to it;
- Three work-sharing constructs:
  - !$omp do #pragma for
  - !$omp section #pragma section
  - !$omp single #pragma single
- A given thread may work on zero, one or more omp sections; but only one thread runs omp single at a given time;
- Be sure there are no data dependencies between sections;
- All threads must encounter the same workflow (though it may or may not execute the same code block at run-time);
Work-sharing directives

!$omp do  
#pragma for

!$omp section  
#pragma section

!$omp single  
#pragma single
Work-sharing directives

```c
#include <omp.h>
#define nsize 500
main() { int i, j, k ;
double a[nsize], b[nsize], c[nsize] ;
for (k = 0; k <= nsize, ++k) {
a[k] = (double) k; b[k] = a[k]; c[k] = 0.5*a[k];}

#pragma omp parallel {
    #pragma omp sections {
        #pragma omp section { code block_1; }
        #pragma omp section { code block_2; }
        #pragma omp section { code block_3; }
    }
}
```
Synchronization
Synchronization

- OpenMP provides the constructs for **mutual exclusion**:
  
critical, atomic, master, barrier, and run-time routines;

  ![omp critical [name] code block]
  ![omp end critical [name] in Fortran;]
  
#pragma omp critical [name] {code block;} in C/C++;

- [name] is an optional; But in Fortran, name here should be unique (cannot be the same as those of **do** loops or **if/endif** blocks, etc);

- At a given time, critical only allows **one** thread to run it, and all other threads also need to go through critical section, but have to wait to enter critical section;

- Don’t jump into or branch out of a critical section;

- It is useful in a parallel region;

- It might have a tremendous impact on code performance;
Synchronizaton

• The other way to think of reduction variable (say addition):

```fortran
1      tsum = 0.0d0 ; nsize = 10000
2      !$omp parallel private(temp), shared(tsum,nsize)
3            temp = 0.0d0
4      !$omp do
5          do i = 1, nsize
6                temp = temp + array(i)
7          end do
8      !$omp end do
9
10     !$omp critical
11          tsum = tsum + temp
12     !$omp end critical
13
14     !$omp end parallel
```
Synchronization

- Using `atomic` to protect a shared variable:

```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define nsize 1000
int main () {
  int i; double x = 0.0, answer;
  #pragma omp parallel for private(i) shared(x) {
    for (i = 0; i < nsize; ++i) {
      #pragma omp atomic
      x += (double) i;
    }
    answer = (double) 0.5*(nsize-1)*nsize;
  }
  printf("%f\n", x);
  printf("correct answer is %f\n", answer);
}
```
OpenMP run-time libraries

- `integer omp_get_num_threads()`
  `int omp_get_num_threads(void)`
  # No. of threads in the current collaborating parallel region;

- `integer omp_get_thread_num()`
  `int omp_get_thread_num(void)`
  # Return the thread IDs in a parallel team;

- `integer omp_get_num_procs()`
  `int omp_get_num_procs(void)`
  # Get the number of “processors” available to the code;

- `call omp_set_num_threads(num_threads)`
  `omp_set_num_threads(num_threads)`
  # Set number of threads to be `num_threads` for the following parallel regions;

- `omp_get_wtime()`
  # Measure elapsed wall-clock time (in seconds) relative to an arbitrary reference time;
Summary and Further Reading

- OpenMP loop-level, non-loop level parallelism, synchronization, and run-time libraries;
- How to protect shared variables; pay attention to them and synchronization; data races;
- Global and local variables in OpenMP programming (global private variables);
- Develop a defensive programming style;

Parallel Programming in OpenMP, R. Chandra et al. (Morgan Kaufmann Publishers, 2001).

Questions?

sys-help@loni.org