Parallel Programming in OpenMP

Xiaoxu Guan

High Performance Computing, LSU

May 29, 2017
Overview

- Parallel programming
  - Prerequisite for parallel computing:
  - Constructs for parallel execution
  - Data communications
  - Synchronization

- OpenMP programming: directives/pragmas, environment variables, and run-time libraries
  - Variables peculiar to OpenMP programming;
  - Loop level parallelism;
  - Nested thread parallelism;
  - Non-loop level parallelism;
  - Data race and false sharing;

- Summary
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;
Shared-memory parallel programming

- System level and user’s application level;

  - **Pthreads** specification is from the IEEE POSIX standard;
    - Many control knobs at low level;
    - Difficult to use and relatively heavyweight threads;

- **Intel Cilk Plus**
  - C/C++ language extensions;
  - Supported by GCC and Intel C/C++ compilers;
  - Fork-join mechanism and efficient load-balancing via work-stealing;

- **Intel TBB (Threading Building Blocks)**
  - C++ libraries instead of language extension;
  - Supports task and loop-level parallelism;

- **OpenCL**
  - Offload work to devices, C/C++, Python, Java APIs;
  - Low-level model;
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

- 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 it, 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 on shared variables 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 directives/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>

- Compilers support **conditional compilation** in disabling OpenMP. Intel compiler also provides the flag `-openmp-stubs` at the compiler level;

- 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_f.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=16
   # on Mike-II in bash shell, or inline setting
$ ifort -o hello hello.f90 -openmp
First OpenMP “Hello World!” in Fortran and 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=16
   # on Mike-II in bash shell, or inline setting
$ icc -o hello hello.c -openmp
Loop-level parallelism

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

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

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

- Other form of parallel loops:

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

```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?

```c++
for (i = imin; i < imax; increment_i) {
    #pragma omp parallel for
    for (j = jmin; j <= jmax; increment_j) {
        loop body ...;
    }
}
```

```c++
#pragma omp parallel for
for (i = imin; i <= imax; increment_i) {
    for (j = jmin; j <= jmax; increment_j) {
        loop body ...;
    }
}
```
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 (but ...);
How to control variables in loops?

- Once we have 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;
- 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** scope 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 variables in 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 variables in 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 variables in 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 variables in 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 variables in 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 variables in 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 private(i,j)
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?

- Parallelize multiple nested loops;
- The `collapse(n)` for nested parallel loops \( n \geq 1 \);
- 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/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 1</td>
</tr>
<tr>
<td>$i = 1$</td>
<td>$i = 2$</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></td>
<td>$r[1]=b+a$</td>
</tr>
</tbody>
</table>

- OpenMP standard does not guarantee which thread finishes first or later;
Data dependence in loops

- How to identify data dependence and possibly remove it;
- A rule of thumb: (1) focus on those variables that are accessed twice or more. If the same memory location is only accessed (r/w) once, we don’t have data dependence for that variable. (2) analyze all variables (mostly array elements) in a loop body. (3) pay attention to the global variables.

C/C++

```c
for (i = imin; i <= imax; ++i)
{ b[i] = b[i] + b[i-1] ; }
```

```c
for (i = imin; i <= imax; ++i)
{ b[i] = b[i] + a[i-1] ; }
```

```c
for (i = 2; i <= imax; i+=2)
{ a[i] = a[i] + a[i-1] ; }
```
Data dependence in loops

- How to identify data dependence and possibly remove it;
- A rule of thumb: (1) focus on those variables that are accessed twice or more. If the same memory location is only accessed (r/w) once, we don’t have data dependence for that variable. (2) analyze all variables (mostly array elements) in a loop body. (3) pay attention to the global variables.

```c
// C/C++
for (i = imin; i <= imax; ++i)
    { b[i] = b[i] + b[i-1] ; }
```

```c
for (i = imin; i <= imax; ++i)
    { b[i] = b[i] + a[i-1] ; }
```

```c
for (i = 1; i <= imax/2; ++i)
    { b[i] = b[i] + b[i+imax/2] ; }
```
Data dependence in loops

- **Dataflow** analysis for potential data dependence;
- Access sequence in two or more statements is critical;
- **Three** types of dataflows:
  1. (1) flow dep.  2. (2) anti-flow dep.  3. (3) output dep.

```c
1 for (i=0;i<10;++i)
2 { tmpi=sin(i);
3 b[i]=a[i]+tmpi; }
```

```c
1 for (i=0;i<10;++i)
2 { tmpl=fact*c[i];
3 c[i]=b[i]+tmpi; }
```
Data dependence in loops

- **Dataflow** analysis for potential data dependence;
- Access sequence in two or more statements is critical;
- **Three** types of dataflows:
  1. Flow dependence
  2. Anti-flow dep.
  3. Output dep.

\[
\begin{align*}
S_1 & \text{ writes the var.} \\
S_2 & \text{ reads the same var.}
\end{align*}
\]

\[
\begin{align*}
S_1 & \text{ reads the var.} \\
S_2 & \text{ writes the same var.}
\end{align*}
\]
Data dependence in loops

- **Dataflow** analysis for potential data dependence;
- Access sequence in two or more statements is critical;
- **Three** types of dataflows:
  1. Flow dependence
  2. Anti-flow dependence
  3. Output dependence

\[ S_1 \text{ writes the var.} \]
\[ S_2 \text{ reads the same var.} \]

\[ S_1 \text{ reads the var.} \]
\[ S_2 \text{ writes the same var.} \]

(1) Flow dependence

(2) Anti-flow dep.

```
for (i = imin; i <= imax; ++i)
    { tmp = a[i] + b[i];
      c[3] = sin(tmp); }
```
Data dependence in loops

- **Dataflow** analysis for potential data dependence;
- Access sequence in two or more statements is critical;
- **Three** types of dataflows:
  - (1) flow dep.
  - (2) anti-flow dep.
  - (3) output dep.

1. $S_1$ writes the var.
   $S_2$ reads the same var.
2. $S_1$ reads the var.
   $S_2$ writes the same var.

(1) Flow dependence

```
for (i = imin; i <= imax; ++i)
{
    tmp = a[i] + b[i];
    c[3] = sin(tmp);
}
```

(2) Anti-flow dep.

Multiple statements write to the same memory location

(3) Output dependence
Data dependence in loops

- Is it possible to remove anti-flow and output dependences?
- The answer is yes in most cases: change the data structure;
- Can we parallelize the following serial code?

```
1 // array a[] and b[] are ready to use. v0 C/C++
2 for (i=0; i<nsize, i++)
3    a[i] = a[i+1] + b[i];
```

- This is a typical example of anti-flow dependence;

  (1) For the given i-th iteration, read a[i+1] (RHS, S1);
  (2) In the next i+1-th iteration, write a[i+1] (LFS, S2);
Data dependence in loops

- Is it possible to remove **anti-flow** and **output** dependences?
- The answer is **yes** in most cases: change the **data structure**;
- Data dependence follows a certain **pattern**;
- Define a new array with **shifted** indices;

```c
1 // make a new array by shifting index. v1 C/C++
2 #pragma omp parallel for
3 for (i=0; i<nsize, i++) a_new[i] = a[i+1];
4
5 // array a[] and b[] are ready to use.
6 #pragma omp parallel for
7 for (i=0; i<nsize, i++)
8 a[i] = a_new[i] + b[i];
```

- Now the data dependences were removed;
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>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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;
False sharing

• Let’s consider the following question: For a given integer matrix, how do we count the total number of even matrix elements?

• This is how the serial code looks like:

```c
// count the number of even integers.
counter_even = 0;
for (i=0; i<nosize; i++)
for (j=0; j<nosize; j++)
{
    itmp = matrix[i][j]%2;
    if (itmp == 0) ++counter_even;
}
printf("Number of even integers is %d\n", counter_even);
```

• How do we parallelize these two loops?
False sharing

- Generally, loop-level parallelism is considered **fine-grained** parallelism;
- Let’s try something different: manually decompose the data;
- This is called **coarse-grained** parallelism:
  - We decompose the matrix in a **row-wise** and each thread takes care of one block (for the **outer** loop);
  - Then we define a counter array: each thread has its own counter;
  - Each thread counts the number of even matrix elements in its own block;
  - Finally, the **master** thread makes a summation;
- Reduce the overhead in the loop scheduling and maintain good **scalability** and **performance**;
False sharing

1 // define the arrays.
2 counter = (int*)malloc(nothread*sizeof(int));
3 start_idx = (int*)malloc(nothread*sizeof(int));
4 end_idx = (int*)malloc(nothread*sizeof(int));
5 chunk_size = nosize / nothread + 1;
6 for (id=0; id<nothread; id++)
7 { start_idx[id] = id*chunk_size;
8   end_idx[id] = MIN(start_idx[id]+chunk_size-1, nosize-1); i }
9 for(id=0; id<nothread; id++) counter[id] = 0;
10 #pragma omp parallel private(id,i,j,itmp)
11 { id = omp_get_thread_num();
12   for (i=start_idx[id]; i<=end_idx[id]; i++)
13     for (j=0; j<nosize; j++)
14       { itmp = matrix[i][j]%2;
15         if (itmp == 0) ++counter[id]; } }
False sharing

- **False sharing** affects performance: the array `counter[]`;
- Thread only accesses its array element, but **adjacent**;
- What is **false sharing**? If **two** or **more** threads that access the **same** cache line, at least one of them is to modify (write) that cache line.

**MESI** protocol:
(1) Modified (M);
(2) Exclusive (E);  
(3) Shared (S), and  
(4) Invalid (I).

- Overhead to synchronize the cache and main memory;
False sharing

- Understand memory and cache **hierarchy**: 
- Lower-level cache (smaller, faster), but higher-level cache (larger, slower);

<table>
<thead>
<tr>
<th>Cache Level</th>
<th>Size</th>
<th>Access Time</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 (D)</td>
<td>32 KB</td>
<td>4 cycles</td>
<td>40 - 100 GB/s</td>
</tr>
<tr>
<td>L2</td>
<td>256 KB</td>
<td>12 cycles</td>
<td>30 - 60 GB/s</td>
</tr>
<tr>
<td>L3 shared</td>
<td>30 MB</td>
<td>25 - 30 cycles</td>
<td>20 - 40 GB/s</td>
</tr>
<tr>
<td>Main Memory</td>
<td></td>
<td>180 - 350 cycles</td>
<td>4 - 10 GB/s</td>
</tr>
</tbody>
</table>

Memory Hierarchy of Xeon Sandy-Bridge Processor
False sharing

- Can we **remove** the false sharing?
- The shared array `counter[]` is the root of the problems;

```c
#pragma omp parallel
private(id,i,j,itmp,counter_own)
{
  counter_own = 0;
  id = omp_get_thread_num();
  for (i=start_idx[id]; i<=end_idx[id]; i++)
    for (j=0; j<nosize; j++)
      { itmp = matrix[i][j]%2;
        if (itmp == 0) ++counter_own; }
  counter[id] = counter_own; }

counter_even = 0;
for(id=0; id<nothread; id++)
  counter_even += counter[id];
```
False sharing

- Any improvement?
- Has the false sharing been removed completely?
- What happens if we simple do parallel for?

```c
#include <omp.h>

v3 C/C++

// count the number of even integers.

counter_even = 0;

#pragma omp parallel for private(i,j,itmp)
for (i=0; i<nosize; i++)
    for (j=0; j<nosize; j++)
    {
        itmp = matrix[i][j]%2;
        #pragma omp critical
        if (itmp == 0) ++counter_even;
    }

Using critical to protect the access to the shared variable;

Any room to improve it?
```
False sharing

- Using **reduction** instead of **critical**:

```c
// start a timer.
double time_start = omp_get_wtime();

// count the number of even integers.
int counter_even = 0;

#pragma omp parallel for private(i,j,itm) reduction(+:counter_even)
for (i=0; i<nosize; i++)
  for (j=0; j<nosize; j++)
  {
    int tmp = matrix[i][j]%2;
    if (tmp == 0) ++counter_even;
  }

time_end = omp_get_wtime();

printf("Elapsed time (sec) is \n\n%5.5f\n", time_end-time_start);
```
False sharing

- Compare the performance of the different versions;

- Which version is the best?
- In the ideal case, the walltime × No. of threads is **flat**;
- The **coarse-grained** data decomposition (**v2**) is the best in terms of both speedup and walltime;
False sharing

- Compare the performance of the different versions;

\[ \begin{array}{c|cccc}
\text{Number of OpenMP threads} & 2 & 4 & 6 & 8 \\
\hline
\text{v1} & & & & \\
\text{v2} & & & & \\
\text{v4} & & & & \\
\end{array} \]

- Which version is the best?
- Performance also depends on the problem size;
- The coarse-grained data decomposition (v2) is the best in terms of the speedup (v4) and walltime (v2);
False sharing

v1: manually decomposition with false sharing;
v2: manually decomposition with false sharing removed (almost);
v3: using `parallel for` with `critical`;
v4: using `parallel for` with `reduction`;

- Don’t confuse false sharing with data race!
- Data race affects the correctness of your results;
- False sharing affects the code performance, but has nothing to do with the correctness of data;
- Both are caused by shared variables;

“Sharing is the root of all contention!” (H. Sutter 2009)
Nested thread parallelism
Nested thread parallelism

- The Intel MKL contains BLAS, LAPACK, Sparse BLAS and solvers, FFT routines, ...
- All BLAS/LAPACK routines were written in Fortran, and Intel compiler provides the flag `-mkl` to link to it. It also supports a CBLAS wrapper for C/C++ code;
- In some cases we need to call the MKL routines in a parallel region. Note the problems of oversubscribing resources;
- `-mkl=parallel` Uses the threaded MKL routines. It is the same as `-mkl` (default);
- `-mkl=sequential` Uses the non-threaded MKL routine.
- `-mkl=cluster` Uses the cluster part and the sequential part of the MKL routines on multiple nodes. It is possible to link to the cluster part and parallel routines at the same time;
Nested thread parallelism

- Consider **matrix-matrix** products that need to be repeated multiple times;
- Link it to the MKL routine `dgemm` in Fortran or `cblas_dgemm` routine in C;

```c
// repeat "iteration" times: C = A×B.  C/C++
#pragma omp parallel for
for (k=0; k<iteration; k++)
{
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, nsize, nsize, nsize,
                alpha, matrix_a, nsize, matrix_b, nsize, beta, matrix_c, nsize);
}
```

- The MKL routine was embedded in the **parallel** region;
- **Explicit** OpenMP threading and **implicit** MKL threading;
Nested thread parallelism

- Compile it with `-openmp -mkl` flags;
- On a **compute node**, run the following commands and monitor the **load** average:

  (1) Without setting anything but run `$ ./threaded_dgemm`

  How many threads from **OpenMP parallel for** and how many threads from the implicit **MKL** routine?

  (2) How do we **explicitly** control the no. of the **OpenMP** threads and the no. of the **MKL** threads?

    ```
    $ OMP_NESTED=true MKL_DYNAMIC=false
    OMP_NUM_THREADS=2 MKL_NUM_THREADS=4 ./threaded_dgemm
    ```

    In this case, we ask 2 OpenMP threads and each of them spawns 4 MKL threads;
Nested thread parallelism

- **OMP_NUM_THREADS** controls the number of OpenMP threads;
- **MKL_NUM_THREADS** controls the number of MKL threads;
- Setting **MKL_NUM_THREADS=1** at run-time is kind of equivalent to `-mkl=sequential` during compilation;
- What if we **don’t** explicitly specify both of the thread counts?
  
  ```
  $ OMP_NESTED=true MKL_DYNAMIC=false ./threaded_dgemm
  ```
  
  How many threads spawned in total?
  
  - **OMP_NESTED** enable/disable **nested parallel regions**. If not defined, nested parallel regions are **disabled** by default;
  - **MKL_DYNAMIC=true** allows the run-time system to detect and decide how many threads need to spawn. This is good to avoid **oversubscription**. If it is **false**, each **MKL** thread has a chance to spawn the max of thread counts, which is bad;
Nested thread parallelism

- The **default** value of MKL_DYNAMIC is **true**.
- By **default** the MKL routines will use only 1 thread if they are called in an OpenMP **parallel** region;
- “**I’m not a programmer, why should I be concerned about it?**”
- Some third-party applications or packages that were built on the top of the Intel MKL may **overwrite** the default behavior at run-time (call `omp_set_nested()` and `mkl_set_dynamic()`);

1. Modify the `threaded_dgemm.c` code by adding
   ```
   omp_set_nested(1); mkl_set_dynamic(0);
   ```
2. Compile it and run it with `./thread_dgemm`
3. Are there any problems and how can we fix it?
4. Run it with `$ OMP_NUM_THREADS=4 ./threaded_dgemm`
   - How many threads in total?
- Use the env `OMP_NUM_THREADS` to control it;
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
1  !$omp parallel
2    id = omp_get_thread_num()
3    write(*,*) "Hello World! from ", id
4  !$omp end parallel
```

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

```fortran
1  !$omp parallel do
2    do k = 1, 5
3      id = omp_get_thread_num()
4      write(*,*) "Hello World! from ", id, k
5    end do
6  !$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;
- **common** 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 similar to the Intel Xeon Phi programming?
Work-sharing directives

program mapping

implicit none

integer :: i, id, nothread, &
omp_get_thread_num, omp_get_num_threads

!$omp parallel private (k, id), shared(nothread)
id = omp_get_thread_num()
nothread = omp_get_num_threads()

!$omp do
  do k = 1, 40
    write(*,'(1x,2(a,i4))') "id = ", id, " k = ", k
  end do
!$omp end do [nowait]
!$omp end parallel
end program mapping
Work-sharing directives

The point is that `!$omp do` directive does not spawn threads. Instead, only `!$omp parallel` spawns multiple threads!

```
id = omp_get_thread_num()

!$omp do
    id = omp_get_thread_num()
!$omp end 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 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

```c
!$omp do  #pragma for
d/for 1 2 3 4 5 6 7 8

!$omp section  #pragma section
S1 S3 S2

!$omp single  #pragma single
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; }
    }
}
```

C/C++
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 the critical section, but have to wait to enter the 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;
Synchronization

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

```fortran
  tsum = 0.0d0 ; nsize = 10000
  !$omp parallel private(temp), shared(tsum,nsize)
  temp = 0.0d0
  !$omp do
  do i = 1, nsize
    temp = temp + array(i)
  end do
  !$omp end do
  !$omp critical
  tsum = tsum + temp
  !$omp end critical
  !$omp end parallel
```
Synchronization

- Using **atomic** to protect a shared variable:

```c/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 = 1; 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()`
  
  ```c
  int omp_get_num_threads(void)
  # No. of threads in the current collaborating parallel region;
  ```

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

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

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

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

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

Questions?

sys-help@loni.org