OpenMP

https://rc.colorado.edu/training/downloads

August 13, 2017

Timothy Brown

Research Computing

UNIVERSITY OF COLORADO **BOULDER**

Overview

- Background
- OpenMP
- Compiler Directives
- Parallel Control
- Data Scope
- Work Sharing
- Scheduling
- Synchronization
- Vectorization
- Accelerators

Overview

Background

OpenMP

Compiler Directives

Parallel Control

Data Scope

Work Sharing

Scheduling

Synchronization

Vectorization

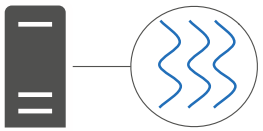
Accelerators

Parallelism

Parallelism can be achieved across many levels

- Nodes – MPI

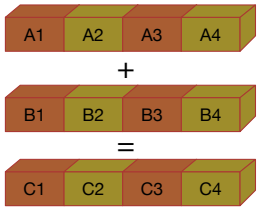
- Threads – OpenMP



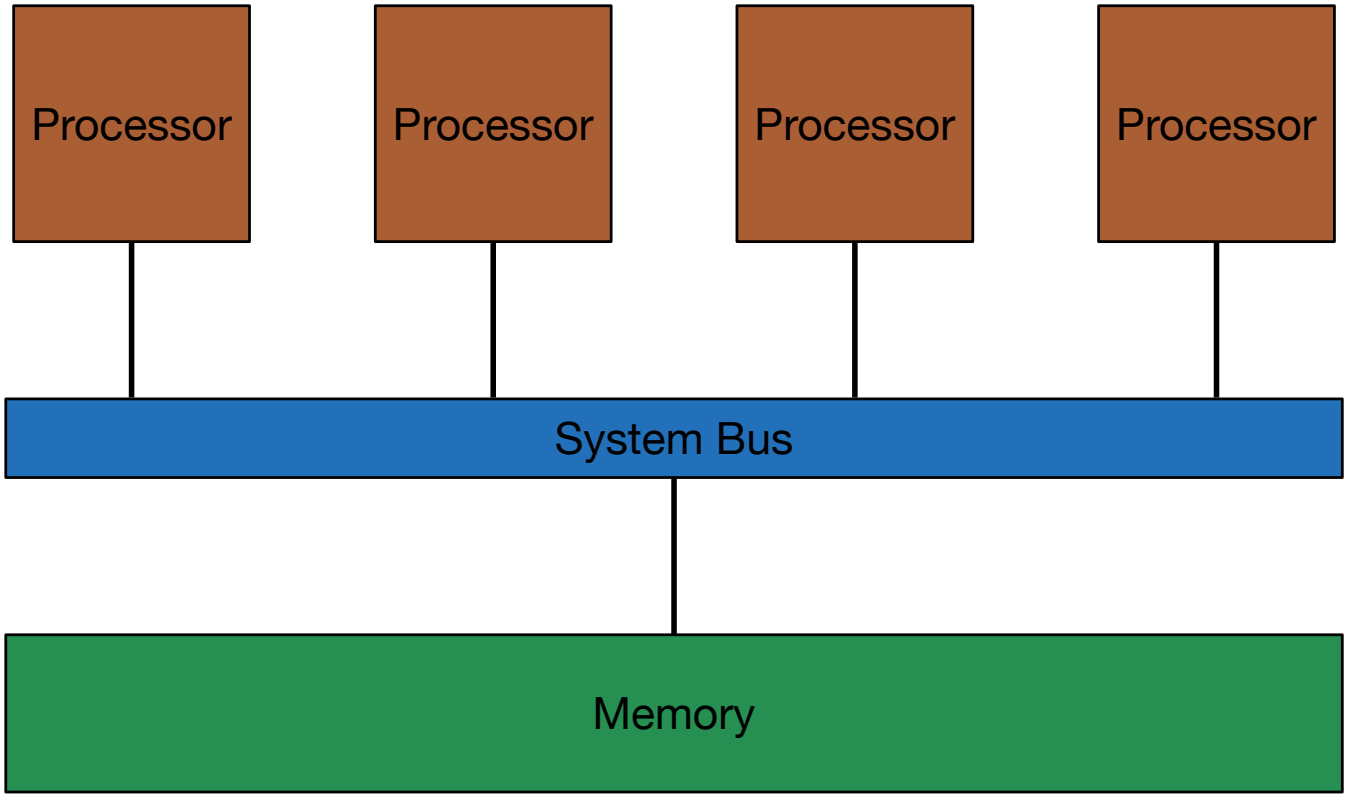
- Instructions – ILP

```
I1: add R1,   R2, R3
I2: sub R4,   R1, R5
I3: xor  R10, R2, R11
```

- Data – SIMD



Shared Memory Model



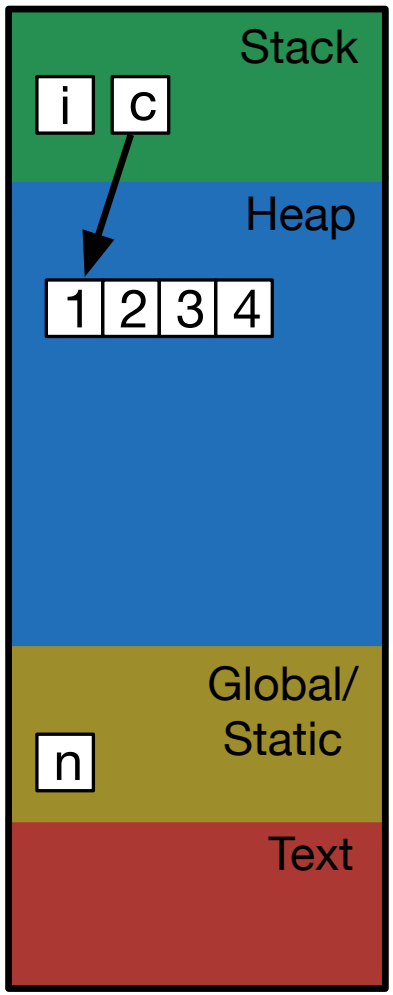
- All processors see a single view of data.
- Processors interact and synchronize through shared variables.

Memory Model

- Contents of memory segments:
  - static variables
  - variables on the run-time stack
  - functions on the run-time stack
  - dynamically allocated data on the heap

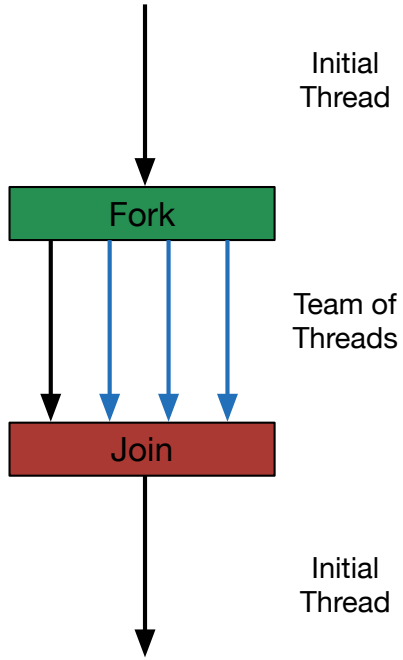
```c
#include <stdlib.h>
static const int n = 4;
int
main(int argc, char **argv)
{
        int i = 0;
        int *c = NULL;

        c = malloc(n * sizeof(int));
        for (i = 0; i < n; ++i) {
                c[i] = i +1;
        }
        free(c);
        return(0);
}
```



Fork/Join Parallelism

- Program starts as a single thread of execution, initial thread.
- A team of threads is forked at the beginning of a parallel region.
- At the end of a parallel region the threads join (either die or are suspended).



Overview

- Background
- OpenMP
- Compiler Directives
- Parallel Control
- Data Scope
- Work Sharing
- Scheduling
- Synchronization
- Vectorization
- Accelerators

OpenMP

- The OpenMP application programming interface (API) supports multi-platform shared-memory parallel programming in
  - C/C++
  - Fortran
- The API defines a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer.

`http://openmp.org`

Philosophy

- Goal: Add parallelism to a functioning serial code.
- Requires: Shared memory machine.
- How: Add *compiler directives* to parallelize parts of code.
- Pro: Often very easy to add to existing code.
- Con: Large shared memory machines are expensive.

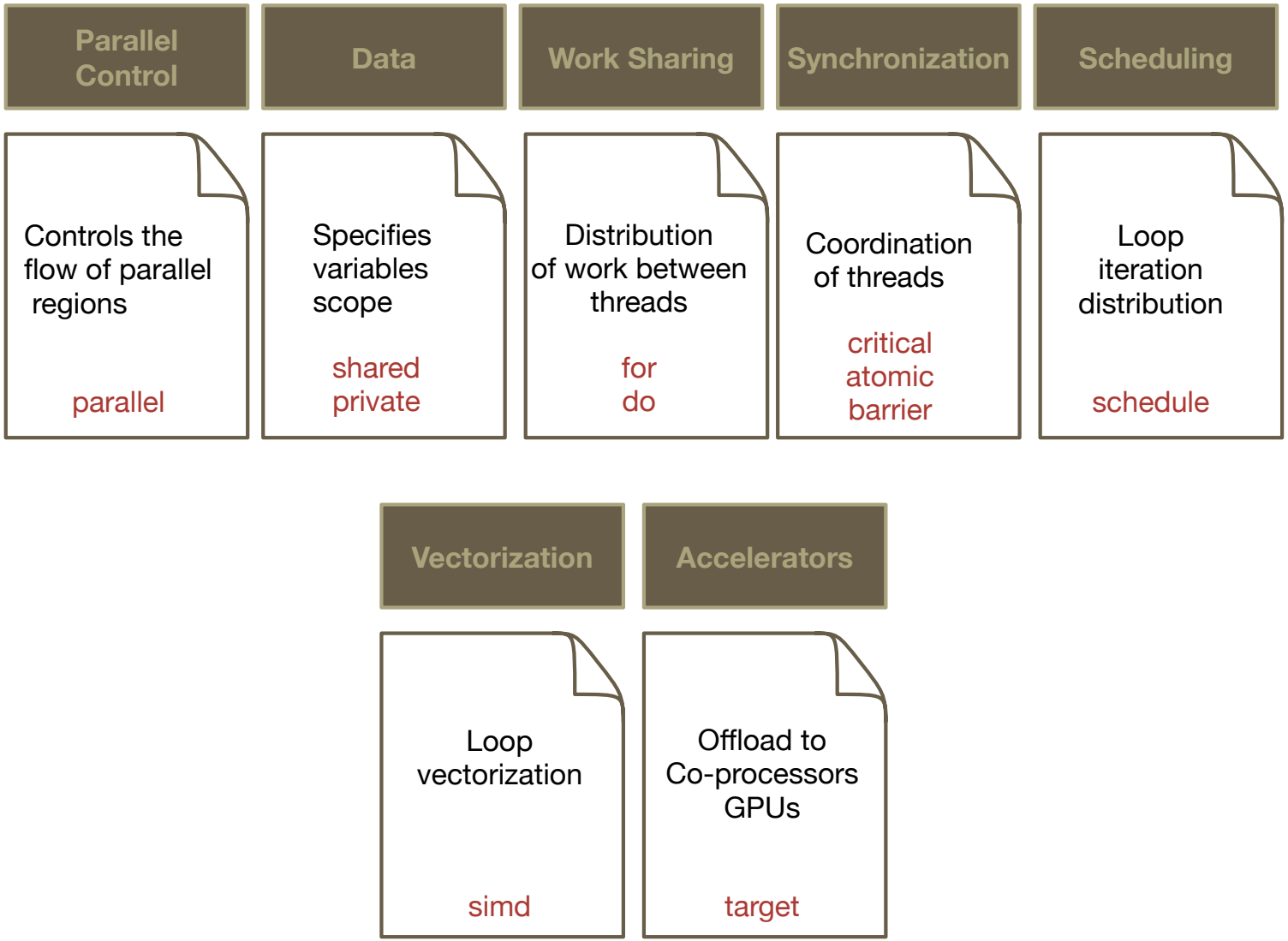
Resources

- *Using OpenMP* the book
  `https://mitpress.mit.edu/index.php?q=books/using-openmp`
- API Qucik Reference
  - `http://openmp.org/mp-documents/OpenMP-4.0-C.pdf`
  - `http://openmp.org/mp-documents/OpenMP-4.0-Fortran.pdf`
- OpenMP v4 full API
  `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`
- OpenMP Examples
  `http://openmp.org/mp-documents/openmp-examples-4.0.2.pdf`

Overview

- Background
- OpenMP
- Compiler Directives
- Parallel Control
- Data Scope
- Work Sharing
- Scheduling
- Synchronization
- Vectorization
- Accelerators

Compiler Directives



Overview

- Background
- OpenMP
- Compiler Directives
- Parallel Control
- Data Scope
- Work Sharing
- Scheduling
- Synchronization
- Vectorization
- Accelerators

Parallel Regions

- We tell OpenMP compiler to parallelize code.
- Mark parallel blocks.
- The compiler will spawn threads and split the work up.
- We can tell the compiler the number of threads too.

C

```c
#pragma omp parallel
{
        ...
}
```

Fortran

```fortran
!$OMP parallel
        ...
!$OMP end parallel
```

- OpenMP also provides library calls.
  - C function prototypes are in `omp.h`.
  - Fortran module interface is in `omp_lib`.
- For compatibility, you should `#ifdef` guard these calls.
- Remember to use the pre-processor for Fortran too (`.F90`).

C

```c
#ifdef _OPENMP
#include <omp.h>
#endif
```

Fortran

```fortran
#ifdef _OPENMP
  use omp_lib
#endif
```

Login to Janus

1. Log in to Janus.

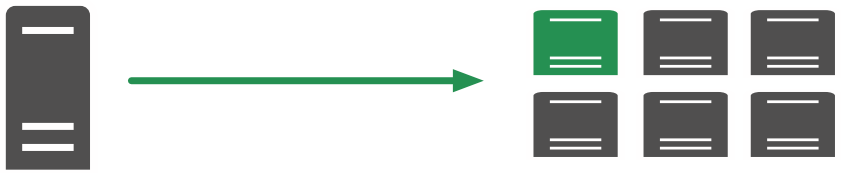
```
laptop ~$ ssh user0000@tutorial-login.rc.colorado.edu
```

2. Load the slurm and intel module.

```
[user0000@tutorial-login ~]$ ml slurm gcc
```

3. Start a compute job.

```
[user0000@tutorial-login ~]$ sinteractive  \
                             -t 02:00:00 -N 1    \
                             --reservation=rmacc-tutorials
```



Exercise

- **Figure out how many threads we have.**
- OpenMP has functions to find out our thread number and the total number of threads.
  - `omp_get_thread_num()`
  - `omp_get_num_threads()`
- We can set the number of threads:
  - Function `omp_set_thread_num()`.
  - Specifying `num_threads()` clause after the `parallel` directive.
  - With the environment variable **`OMP_NUM_THREADS`**.

All of the examples taught are online, in a github repository:
`https://github.com/ResearchComputing/RMACC_2015`

nthreads/nthreads_c.c

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

int
main(int argc, char **argv)
{

#pragma omp parallel
{
        printf("Hello world! From thread %d\n",
                omp_get_thread_num());
} /* End omp parallel */

        return(EXIT_SUCCESS);
}
```

- There is also a Fortran version `nthreads/nthreads_f.f90`.

- Compiling the program.

  GCC `gcc -fopenmp -o thread_num_c thread_num_c.c`
  Intel `icc -qopenmp -o thread_num_c thread_num_c.c`
  IBM `xlc -qsmp=omp -o thread_num_c thread_num_c.c`

- Execute the program, specifying different numbers of threads.
  1. `./thread_num_c`
  2. `env OMP_NUM_THREADS=1 ./thread_num_c`
  3. `env OMP_NUM_THREADS=64 ./thread_num_c`

- What is the output?
  - Threads printed out their identification number.
  - Random order of numbers. Threads execute independently and in general order will be random.

Overview

- Background
- OpenMP
- Compiler Directives
- Parallel Control
- Data Scope
- Work Sharing
- Scheduling
- Synchronization
- Vectorization
- Accelerators

Variables

- We must tell the compiler how to use variables.
  - A `shared` variable has the same address in memory in every thread.
  - A `private` variable has a different address in memory in every thread.
  - A `firstprivate` is private, however it is pre-initialize.

Clauses specifies the scope of variables.

C

```
#pragma omp parallel          \
            default(none)     \
            shared(x)         \
            private(i,j)      \
```

Fortran

```
!$OMP parallel          &
!$OMP default(none)     &
!$OMP shared(x)         &
!$OMP private(i,j)
                ...
!$OMP end parallel
```

- The default is shared.
- Try and always specify `default(none)`, so as not to confuse a variables behavior. Then explicitly define every variable.
- In C, you are able to declare variables within structured blocks to reduce it's scope. This will make it a private variable. For example, `i` is shared while `j` is private.

C

```c
int i = 0;
#pragma omp parallel      \
            default(none) \
            shared(i)
{
        int j = 0;
}
```

Overview

- Background
- OpenMP
- Compiler Directives
- Parallel Control
- Data Scope
- Work Sharing
- Scheduling
- Synchronization
- Vectorization
- Accelerators

Work Sharing

There are four work sharing constructs

- Loop Distribute iterations over the threads.
- Sections Distribute independent work units.
- Single Only one thread executes the block of code.
- Workshare Parallelize array syntax (Fortran only)

Loops

Distribute iterations over the threads.

- Makes it easy to indicate when the iterations of a loop may be executed in parallel.
- Each loop iteration must be independent of other iterations.
- Implied barrier at the end of the loop.

C

```c
#pragma omp for
for (i=0; i<10; ++i) {
    a[i] = b[i] + c[i];
}
```

Fortran

```fortran
!$OMP do
do i=1,10
    a(i) = b(i) + c(i)
end do
```

Loop Variable Clauses

- `firstprivate` pre-initialized, private variable.
- `lastprivate` last value is accessible.
- `reduction` operator is applied to the shared variable.

First Private

- Used to create private variables having initial values identical to the variable controlled by the master thread as the loop is entered.
- Variables are initialized once per thread, not once per loop iteration.
- If a thread modifies a variable's value in an iteration, subsequent iterations will get the modified value.

C

```
i = 10;
#pragma omp parallel      \
            default(none) \
            firstprivate(i)
{
    int id = omp_get_thread_num();
    i += id;
}
```

Last Private

- Sequentially last iteration: iteration that occurs last when the loop is executed sequentially.
- Used to copy back to the master thread's copy of a variable the private copy of the variable from the thread that executed the sequentially last iteration.

C

```c
#pragma omp parallel for  \
            default(none) \
            shared(n)     \
            private(i)    \
            lastprivate(a)
    for(i=0; i < n; ++i) {
        a = i +1;
    }
    printf("Final value of a: %d\n", a);
```

Reduction

- Reductions are so common that OpenMP provides support for them.
- Specify reduction operation and reduction variable.
- OpenMP takes care of storing partial results in private variables and combining partial results after the loop.

C

```c
#pragma omp parallel for  \
            default(none) \
            shared(x, n)  \
            private(i)    \
            reduction(+:t)
    for(i=0; i < n; ++i) {
        t += x[i];
    }
```

Reduction Operands

Operator	Initial Value
+	0
−	0
×	1

C	
Operator	**Initial Value**
&	~0
|	0
^	0
&&	1
||	0

Fortran	
Operator	**Initial Value**
min	huge positive number
max	huge negaive number
.eqv.	.true.
.iand.	all bits on
.ior.	0
.ieor.	0
.neqv.	.false.
.and.	.true.
.or.	.false.

Exercise

- **Perform an array addition.**
- Use a sufficiently large array (eg. 10^7).
- Define the default data scope to be `none`.
- Time the serial vs OpenMP versions.

Sections

- Non-iterative work sharing construct.
- The ability to assign different threads to different portions of code (function parallelism).
- Each section is executed by a thread.
- A thread can execute more than one section.

C

```c
#pragma omp sections
{
    #pragma omp section
    A()
    #pragma omp section
    B()
}
```

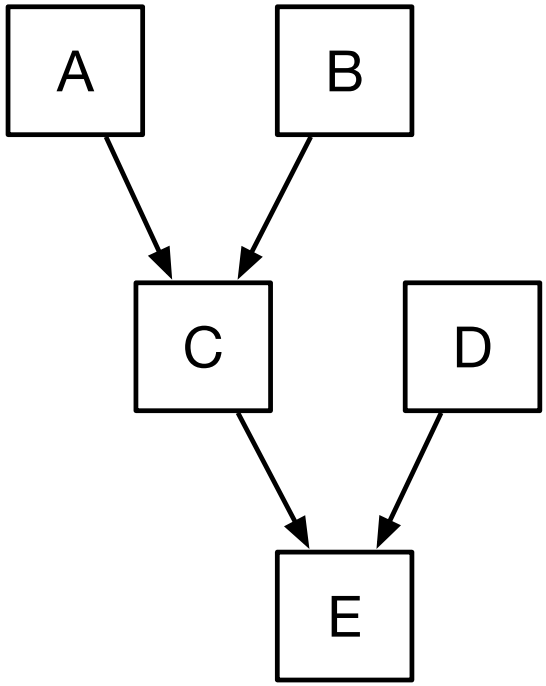
Fortran

```fortran
!$OMP sections
!$OMP section
    call A()
!$OMP section
    call B()
!$OMP end sections
```

```
program sections
    use omp_lib
    implicit none
    integer :: x, y, i, j

!$OMP parallel      &
      default(none) &
      shared(x, y, i, j)

!$OMP sections
!$OMP section
    call A(x)
!$OMP section
    call B(y)
!$OMP end sections
```




```
!$OMP sections
!$OMP section
    call C(x, y, i)
!$OMP section
    call D(j)
!$OMP end sections
!$OMP end parallel
    call E(i,j)
```

Single

- Only one thread will execute the block.
- Not necessarily the master thread.
- Useful for sections of code that are not thread safe (for example
- All other threads wait at the end of the block. I/O).

C

```c
#pragma omp single
{
  write(iunit, data,
        N * sizeof(double));
}
```

Fortran

```fortran
!$OMP single
  write(iunit,iostat=ierr) &
        data(1:N)
!$OMP end single
```

Workshare

- Fortran only
- Work is divided into separate units.
- Each unit is executed only once.
- Parallelization of array assignments.

Fortran

```
!$OMP workshare
    A = B + C
!$OMP end workshare
```

Overview

- Background
- OpenMP
- Compiler Directives
- Parallel Control
- Data Scope
- Work Sharing
- Scheduling
- Synchronization
- Vectorization
- Accelerators

Scheduling

- Supported on the loop construct only.
- Controls the distribution of loop iterations over threads.
- Clauses are:
  - **schedule(static)** Each CPU receives one set of contiguous iterations.
  - **schedule(static, *x*)** Iterations are divided round-robin fashion in chunks of size *x*.
  - schedule(dynamic, *x*) Iterations handed out in chunks of size *x* as CPUs become available.
  - **schedule(guided, *x*)** Each of the iterations are handed out in pieces of exponentially decreasing size, with *x* minimum number of iterations to dispatch each time.
  - **schedule(runtime)** Schedule and chunk size taken from the `OMP_SCHEDULE` environment variable.

Overview

- Background
- OpenMP
- Compiler Directives
- Parallel Control
- Data Scope
- Work Sharing
- Scheduling
- Synchronization
- Vectorization
- Accelerators

Synchronization

Synchronization helps to organize access to shared data by multiple threads.

- High level:
  - atomic
  - critical
  - barrier
  - ordered
- Low level:
  - locks
  - flush

Atomic

- Enables multiple threads to update shared data without interference.
- Applied only to a single statement.
- Try and use a `reduction` clause instead.

C

```c
int i = 0;
int n = 10;
int x = 0;
#pragma omp parallel for  \
            default(none) \
            shared(n, x)  \
            private(i)
    for (i=0; i < n; ++i) {
        #pragma omp atomic
        x += foo();
    }
```

Critical

- Enables multiple threads to update shared data without interference.
- Applied to a code block.

C

```c
#pragma omp parallel
{
    #pragma omp critical
    {
        funA(x)
        funB(y)
    }
}
```

Fortran

```fortran
!$OMP parallel
!$OMP critical
    call subA(x)
    call subB(y)
!$OMP end critical
!$OMP end parallel
```

Barrier

- Synchronizes all threads in team.
- When a `barrier` directive is reached, a thread will wait at that point until all other threads have reached that barrier.
- All threads resume executing in parallel the code that follows the barrier.

C

```c
#pragma omp parallel      \
            default(none) \
            shared(x, t)  \
            private(i)    \
            firstprivate(n)
{
    int id = omp_get_thread_num();
    x[id] = id;
    sleep(id);
    #pragma omp barrier
    #pragma omp for reduction(+:t)
    for (i=0; i < n; ++i) {
        t += x[i]
    }
    #pragma omp master
    {
        printf("Slept [s]: %d\n", t);
    }
}
```

Ordered

- The ordered region executes in sequential order.

C

```c
#pragma omp parallel      \
            default(none) \
            private(i,j)  \
            shared(t, n)
{
        #pragma omp for ordered reduction(+:t)
        for (i=0; i < n; ++i) {
                j = i;
                #pragma omp ordered
                {
                        t += j;
                }
        }
} /* end of omp parallel */
printf("t: %d\n", t);
```

Overview

- Background
- OpenMP
- Compiler Directives
- Parallel Control
- Data Scope
- Work Sharing
- Scheduling
- Synchronization
- Vectorization
- Accelerators

Vectorization

- Parallelism that exploits the hardware feature – instruction level parallelism (ILP).
- Enables the execution of multiple iterations of the loops concurrently by means of SIMD instructions.
- Data alignment is important.

Data Alignment

- Tells the compiler to create data objects in memory on specific byte boundaries.
- Increases the efficiency of data loads and stores to and from the processor.
- Aligning consists of:
  - Aligning the base-pointer where the space is allocated for the array (or pointer).
  - Making sure the starting indices have good-alignment properties for each vectorized loop (for each thread)

Data Alignment C

- For static arrays, use compiler attributes.

```
#define ALIGNMENT 64
#define ATT_ALIGN __attribute__((aligned(ALIGNMENT)))
    int a[3] ATT_ALIGN;
```

- For dynamic arrays, use `posix_memalign` instead of `malloc`.

```
    int n = 1024;
    double *b = NULL;

    ierr = posix_memalign((void **)&b, ALIGNMENT,
                          n * sizeof(double));
    /* do something */
    if (b) {
            free(b);
            b = NULL;
    }
```

Data Alignment Fortran

- Use the directive `attributes align`.

```
integer, parameter              :: n = 1024
double precision, allocatable :: a(:)
!dir$ attributes align: 64:: a

allocate(a(n), stat=ierr)
! do something
if (allocated(a)) then
    deallocate(a)
end if
```

SIMD Construct

- To create a vectorized loop using **only** SIMD instructions.

C

```c
#pragma omp simd
for (i = 0; i < n; ++i) {
    a[i] = b[i] + c[i];
}
```

Fortran

```fortran
!$OMP simd
do i=1,n
    a(i) = b(i) + c(i)
end do
!$OMP end simd
```

SIMD Clauses

- Accepts the following clauses
  - `safelen(`*`x`*`)`
  - `linear(`*`list`*`[:`*`linear-step`*`])`
  - `aligned(`*`list`*`[:`*`alignment`*`])`
  - `private(`*`list`*`)`
  - `lastprivate(`*`list`*`)`
  - `reduction(`*`op`*`:`*`list`*`)`
  - `collapse(`*`x`*`)`

safelen

- No two iterations executed concurrently with SIMD instructions can have a greater distance in the logical iteration space than its value.

C

```c
#pragma omp simd safelen(4)
for (i = 0; i < n; ++i) {
    a[i] = a[i+4] + 1;
}
```

linear

- Declares the items to be private to a SIMD lane.
- Has a linear relationship with respect to the iteration space of a loop.

C

```c
int step = 4;
float a[N] = {0};
float sum = 0.0f;
float *p = a;

#pragma simd reduction(+:sum) linear(p:step)
for (int i = 0; i < N; ++i) {
   sum += *p;
   p += step;
}
```

aligned

- Declares one or more list items to be aligned to the specified number of bytes.

C

```c
#pragma simd aligned(a:64,b:64)
for (int i = 0; i < N; ++i) {
   a[i] = b[i] + 1;
}
```

Loop SIMD

- Combines worksharing loop construct and the SIMD contruct.
  - Parallel over threads.
  - Vectorized over SIMD.
- Construct accepts the same clases as the `for` construct.

C

```c
#pragma omp parallel \
     for simd        \
     aligned(a,b,c:64)
for (i = 0; i < n; ++i) {
    a[i] = b[i] + c[i];
}
```

Fortran

```fortran
!$OMP parallel &
!$OMP for simd &
!$OMP aligned(a,b,c:64)
do i=1,n
    a(i) = b(i) + c(i)
end do
!$OMP end parallel for simd
```

Function SIMD

- Declare function and subroutines.
- Accepts the following clauses
  - `simdlen(`*x*`)`
  - `uniform(`*list*`)`
  - `linear(`*list*`[:`*linear-step*`])`

C

```c
#pragma declare simd uniform(fact)
double add(double a, double b, double fact)
{
    double c = 0.0;
    c = a + b + fact;
    return c;
}
```

Fortran

```fortran
function add(a, b, fact) result(c)
!$omp declare simd(add) uniform(fact)
    implicit none
    double precision :: a, b, fact, c
    c = a + b + fact
```

Overview

- Background
- OpenMP
- Compiler Directives
- Parallel Control
- Data Scope
- Work Sharing
- Scheduling
- Synchronization
- Vectorization
- Accelerators

Accelerators

- Host-centric: the execution of an OpenMP program starts on the host device and it may offload target regions to target devices.
- If a target device is not present, or not supported, or not available, the target region is executed by the host device.
- If a construct creates a data environment, the data environment is created at the time the construct is encountered.

Target Construct

C

```c
#pragma omp target          \
            device(0)       \
            map(to:x,y)     \
            map(from:z)
#pragma omp parallel for    \
        default(none)       \
        private(i)          \
        shared(x, y, z, n)
for (i=0; i < n; ++i) {
    z[i] = x[i] + y[i];
}
```

Fortran

```fortran
!$omp target device(0)    &
!$omp        map(to:x,y)  &
!$omp        map(from:z)
!$omp parallel do         &
!$omp       default(none) &
!$omp       private(i)    &
!$omp       shared(x, y, z, n)
    do i=1,n
        z(i) = x(i) + y(i)
    end do
```

Questions?

Online Survey

<Timothy.Brown-1@colorado.edu>

License