Introduction to OpenMP

2018 HPC Workshop: Parallel Programming

Alexander B. Pacheco
Research Computing
July 17 - 18, 2018
Distributed Memory Model

- Each process has its own address space
  - Data is local to each process
- Data sharing is achieved via explicit message passing
- Example
  - MPI
Shared Memory Model

- All threads can access the global memory space.
- Data sharing achieved via writing to/reading from the same memory location
- Example
  - OpenMP
  - Pthreads
Clusters of SMP nodes

- The shared memory model is most commonly represented by Symmetric Multi-Processing (SMP) systems
  - Identical processors
  - Equal access time to memory
- Large shared memory systems are rare, clusters of SMP nodes are popular.
Shared vs Distributed

**Shared Memory**

- **Pros**
  - Global address space is user friendly
  - Data sharing is fast

- **Cons**
  - Lack of scalability
  - Data conflict issues

**Distributed Memory**

- **Pros**
  - Memory scalable with number of processors
  - Easier and cheaper to build

- **Cons**
  - Difficult load balancing
  - Data sharing is slow
Parallelizing Serial Code

Compiler Flags for Automatic Parallelization

- GCC  
  -floop-parallelize-all

- Intel  
  -parallel

- XL  
  -qsmp=auto

- PGI  
  -Mconcur=<flags>

When to consider using OpenMP?

- The compiler may not be able to do the parallelization
  1. A loop is not parallelized
     - The data dependency analysis is not able to determine whether it is safe to parallelize or not
  2. The granularity is not high enough
     - The compiler lacks information to parallelize at the highest possible level
OpenMP is an Application Program Interface (API) for thread based parallelism; Supports Fortran, C and C++

- Uses a fork-join execution model

- OpenMP structures are built with program directives, runtime libraries and environment variables

- OpenMP has been the industry standard for shared memory programming over the last decade
  - Permanent members of the OpenMP Architecture Review Board: AMD, Cray, Fujutsu, HP, IBM, Intel, Microsoft, NEC, PGI, SGI, Sun

- OpenMP 4.0 was released in June 2014
Advantages of OpenMP

- **Portability**
  - Standard among many shared memory platforms
  - Implemented in major compiler suites

- **Ease to use**
  - Serial programs can be parallelized by adding compiler directives
  - Allows for incremental parallelization - a serial program evolves into a parallel program by parallelizing different sections incrementally
Fork-Join Execution Model

- Parallelism is achieved by generating multiple threads that run in parallel
  - A fork (F) is when a single thread is made into multiple, concurrently executing threads
  - A join (J) is when the concurrently executing threads synchronize back into a single thread
- OpenMP programs essentially consist of a series of forks and joins.
Building Block of OpenMP

- Program directives
  - Syntax
    - C/C++: #pragma omp <directive> [clause]
    - Fortran: !$omp <directive> [clause]
  - Parallel regions
  - Parallel loops
  - Synchronization
  - Data Structure
  - ...

- Runtime library routines

- Environment variables
OpenMP Basic Syntax

- **Fortran: case insensitive**
  - Add: `use omp_lib` or include "omp_lib.h"
  - Usage: Sentinel directive [clauses]
  - Fortran 77
    - Sentinel could be: !$omp, *$omp, c$omp and must begin in first column
  - Fortran 90/95/2003
    - Sentinel: !$omp
  - End of parallel region is signified by the end sentinel statement: !$omp end directive [clauses]

- **C/C++: case sensitive**
  - Add `#include <omp.h>`
  - Usage: #pragma omp directive [clauses] newline
Compiler Directives

- **Parallel Directive**
  - `parallel`

- **Worksharing Constructs**
  - Fortran: `do, workshare`
  - C/C++: `for`
  - Fortran/C/C++: `sections`

- **Synchronization**
  - `master, single, ordered, flush, atomic`
Clauses

- private(list), shared(list)
- firstprivate(list), lastprivate(list)
- reduction(operator:list)
- schedule(method[,chunk_size])
- nowait
- if(scalar_expression)
- num_thread(num)
- threadprivate(list), copyin(list)
- ordered
- more · · ·
Runtime Libraries

- Number of Threads: `omp_{set,get}_num_threads`
- Thread ID: `omp_get_thread_num`
- Scheduling: `omp_{set,get}_dynamic`
- Nested Parallelism: `omp_in_parallel`
- Locking: `omp_{init,set,unset}_lock`
- Wallclock Timer: `omp_get_wtime`
- more · · ·
Environment Variables

- OMP_NUM_THREADS
- OMP_SCHEDULE
- OMP_STACKSIZE
- OMP_DYNAMIC
- OMP_NESTED
- OMP_WAIT_POLICY
- more · · ·
The **parallel** directive forms a team of threads for parallel execution.

Each thread executes the block of code within the OpenMP Parallel region.

**C**

```c
#include <stdio.h>

int main() {
    #pragma omp parallel
    {
        printf("Hello world\n");
    }
}
```

**Fortran**

```fortran
program hello

    implicit none

    !$omp parallel
    print *, 'Hello World'
    !$omp end parallel

end program hello
```
Compilation and Execution

- Use any compiler of your choices
  - PGI Compiler
    - module load pgi
    - pgcc -mp -o hellocmp hello.c
    - pgfortran -mp -o hellofmp hello.f
  - GNU Compiler
    - module load gcc
    - gcc -fopenmp -o hellocmp hello.c
    - gfortran -fopenmp -o hellofmp hello.f
  - Intel Compiler
    - module load intel
    - icc -qopenmp -o hellocmp hello.c
    - ifort -qopenmp -o hellofmp hello.f

```bash
[alp514.sol](752): module load gcc
[alp514.sol](753): gcc -fopenmp -o hellocmp hello.c
[alp514.sol](754): gfortran -fopenmp -o hellofmp hello.f90
[alp514.sol](755): export OMP_NUM_THREADS=4
[alp514.sol](756): srun -p lts -n 1 -c 4 ./hellocmp
Hello world
Hello world
Hello World
Hello World
```
Hello World: C

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

int main () {
    #pragma omp parallel
    {
        printf("Hello from thread %d out of %d threads\n", omp_get_thread_num(), omp_get_num_threads());
    }
    return 0;
}
```

OpenMP include file
Parallel region starts here
Runtime library functions
Parallel region ends here

Hello from thread 0 out of 4 threads
Hello from thread 3 out of 4 threads
Hello from thread 1 out of 4 threads
Hello from thread 2 out of 4 threads
program hello

  implicit none
  integer :: omp_get_thread_num, omp_get_num_threads

  !$omp parallel

  print '(a,i3,a,i3,a)', 'Hello from thread',omp_get_thread_num(), &, 
     ' out of ' omp_get_num_threads(),' threads'

  !$omp end parallel

end program hello
Exercise 1: Hello World

- Write a “hello world” program with OpenMP where
  1. If the thread id is odd, then print a message "Hello world from thread x, I’m odd!"
  2. If the thread id is even, then print a message "Hello world from thread x, I’m even!"

C

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

int main() {
    int id;
    /* Add Opmp pragma */
    {
        id = /* Get Thread ID */
        if (id%2==1)
            printf("Hello world from thread %d, I am odd\n", id);
        else
            printf("Hello world from thread %d, I am even\n", id);
    }
}
```

Fortran

```fortran
program hello
    implicit none
    integer i
    ! Add OMP Directive
    i = ! Get Thread ID
    if (mod(i,2).eq.1) then
        print *,’Hello from thread’,i,’ , I am odd!’
    else
        print *,’Hello from thread’,i,’ , I am even!’
    endif
    ! End OMP Directive
end program hello
```
#Solution

##C/C++

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

int main() {
    int id;
    #pragma omp parallel private(id)
    {
        id = omp_get_thread_num();
        if (id%2==1)
            printf("Hello world from thread %d, I am odd\n", id);
        else
            printf("Hello world from thread %d, I am even \n", id);
    }
}
```

##Fortran

```fortran
program hello
    use omp_lib
    implicit none
    integer i
    !$omp parallel private(i)
    i = omp_get_thread_num()
    if (mod(i,2).eq.1) then
        print *,’Hello from thread’,i,’,’ , I am odd!’
    else
        print *,’Hello from thread’,i,’,’ , I am even!’
    endif
    !$omp end parallel
end program
```

```bash
[alp514.sol](1898): make helloc
pgcc -mp -o helloc hello.c
[alp514.sol](1899): export OMP_NUM_THREADS=4
[alp514.sol](1900): srun -p eng -n 1 -c 4 ./helloc
Hello world from thread 0, I am even
Hello world from thread 3, I am odd
Hello world from thread 1, I am odd
Hello world from thread 2, I am even
```

```bash
[alp514.sol](1906): pgfortran -mp -o hellof hello.f90
[alp514.sol](1907): interact -p eng -n 1 -c 4
[alp514.sol-b110](893): OMP_NUM_THREADS=4 ./hellof
Hello from thread 0, I am even!
Hello from thread 2, I am even!
Hello from thread 3, I am odd!
Hello from thread 1, I am odd!
```
Work Sharing: Parallel Loops

- We need to share work among threads to achieve parallelism
- Syntax:
  - Fortran: !$omp parallel
  - C/C++: #pragma omp parallel
- Loops are the most likely targets when parallelizing a serial program
- Syntax:
  - Fortran: !$omp do
  - C/C++: #pragma omp for
- Other work sharing directives available
  - Sections: !$omp sections or #pragma sections
  - Tasks: !$omp task or #pragma omp task
- The parallel and work sharing directive can be combined as
  - !$omp parallel do
  - #pragma omp parallel sections
Example: Parallel Loops

**C/C++**

```c
#include <omp.h>

int main() {
    int i = 0, n = 100, a[100];
    #pragma omp parallel for
    for (i = 0; i < n ; i++) {
        a[i] = (i+1) * (i+2) ;
    }
}
```

**Fortran**

```fortran
program paralleldo

    implicit none
    integer :: i, n, a(100)

    i = 0
    n = 100
    !$omp parallel
    !$omp do
    do i = 1, n
        a(i) = i * (i+1)
    end do
    !$omp end do
    !$omp end parallel
end program paralleldo
```
OpenMP provides different methods to divide iterations among threads, indicated by the `schedule` clause.

- **Syntax:** `schedule (<method>, [chunk size])`

**Methods include**

- **Static:** the default schedule; divide iterations into chunks according to `size`, then distribute chunks to each thread in a round-robin manner.
- **Dynamic:** each thread grabs a chunk of iterations, then requests another chunk upon completion of the current one, until all iterations are executed.
- **Guided:** similar to Dynamic; the only difference is that the chunk size starts large and shrinks to `size` eventually.
# Load Balancing II

## 4 threads, 100 iterations

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Iterations mapped onto thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>0: 1-25, 1: 26-50, 2: 51-75, 3: 76-100</td>
</tr>
<tr>
<td>Static, 20</td>
<td>0: 1-20, 81-100, 1: 21-40, 2: 41-60, 3: 61-80</td>
</tr>
<tr>
<td>Dynamic</td>
<td>0: 1, ..., 1: 2, ..., 2: 3, ..., 3: 4, ...</td>
</tr>
<tr>
<td>Dynamic, 10</td>
<td>0: 1-10, ..., 1: 11-20, ..., 2: 21-30, ..., 3: 31-40, ...</td>
</tr>
<tr>
<td>Schedule</td>
<td>When to Use</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Static</td>
<td>Even and predictable workload per iteration; scheduling may be done at compilation time, least work at runtime.</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Highly variable and unpredictable workload per iteration; most work at runtime</td>
</tr>
<tr>
<td>Guided</td>
<td>Special case of dynamic scheduling; compromise between load balancing and scheduling overhead at runtime</td>
</tr>
</tbody>
</table>
Work Sharing: Sections

- Gives a different block to each thread

### C/C++

```c
#pragma omp parallel
{
#pragma omp sections
{
#pragma omp section
some_calculation();
#pragma omp section
some_more_calculation();
#pragma omp section
yet_some_more_calculation();
}
}
```

### Fortran

```fortran
!$omp parallel
!$omp sections
!$omp section
call some_calculation
!$omp section
call some_more_calculation
!$omp section
call yet_some_more_calculation
!$omp end sections
!$omp end parallel
```
Scope of variables

- **Shared(list)**
  - Specifies the variables that are shared among all threads

- **Private(list)**
  - Creates a local copy of the specified variables for each thread
  - the value is uninitialized!

- **Default(shared|private|none)**
  - Defines the default scope of variables
  - C/C++ API does not have default (private)

- Most variables are shared by default
  - A few exceptions: iteration variables; stack variables in subroutines; automatic variables within a statement block.
Exercise: SAXPY

- SAXPY is a common operation in computations with vector processors included as part of the BLAS routines
  \[ y \leftarrow \alpha x + y \]
- SAXPY is a combination of scalar multiplication and vector addition
- Parallelize the following SAXPY code

```c
#include <stdio.h>
#include <time.h>

int main() {
    int i;
    long long int n=100000000;
    float a=2.0;
    float x[n];
    float y[n];
    clock_t start_time, end_time;
    /* Parallelize this block of code (optional) */
    for (i = 0; i < n; i++){
        x[i] = 1.0;
        y[i] = 2.0;
    }
    start_time = clock();
    /* Parallelize this block of code */
    for (i = 0; i < n; i++){
        y[i] = a*x[i] + y[i];
    }
    end_time = clock();
    printf("SAXPY Time: %f\n", (double)(end_time - start_time)/CLOCKS_PER_SEC);
}
```

```fortran
program saxpy

implicit none
integer :: i,n
real,dimension(:),allocatable :: x, y
real :: a,start_time, end_time
n=100000000
allocate(x(n),y(n))
! Parallelize this block of code (optional)
! x = 1.0d0
! y = 2.0d0
! a = 2.0d0

call cpu_time(start_time)
! Parallelize this block of code
do i = 1, n
    y(i) = y(i) + a * x(i)
end do
! Parallelize this block of code
call cpu_time/end_time
deallocate(x,y)
print '(a,f8.6)', 'SAXPY Time: ', end_time - start_time
end program saxpy
```
Solution: SAXPY

C

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

int main() {
    long long int i, n=500000000;
    float a=2.0;
    float x[n];
    float y[n];
    double start_time, end_time;
    for (i = 0; i < n; i++) {
        x[i] = 1.0;
        y[i] = 2.0;
    }
    start_time = omp_get_wtime();
    #pragma omp parallel for private(i)
    for (i = 0; i < n; i++) {
        y[i] = a*x[i] + y[i];
    }
    end_time = omp_get_wtime();
    printf ("SAXPY Time: %f\n", end_time - start_time);
}
```

Fortran

```fortran
program saxpy
implicit none
integer, parameter :: dp = selected_real_kind(15)
integer, parameter :: ip = selected_int_kind(15)
integer(ip) :: i, n
real(dp),dimension(:), allocatable :: x, y
real(dp) :: a, start_time, end_time
n=500000000
allocate(x(n), y(n))
!$omp parallel sections
!$omp section
x = 1.0
!$omp section
y = 1.0
!$omp end parallel sections
a = 2.0
!$omp parallel do default(shared) private(i)
do i = 1, n
    y(i) = y(i) + a * x(i)
end do
!$omp end parallel do
!$omp end parallel sections
!$omp end parallel do
!$omp parallel sections
!$omp section
a = 2.0
!$omp section
y = 1.0
!$omp end parallel sections
!$omp end parallel do
!$omp end parallel sections
!$omp end parallel do
!$omp parallel do default(shared) private(i)
do i = 1, n
    y(i) = y(i) + a * x(i)
end do
!$omp end parallel do
!$omp end parallel do
!$omp end parallel sections
!$omp end parallel do
allocate(x, y)
print '(a,f8.6)', 'SAXPY Time: ', end_time - start_time
end program saxpy
```

<table>
<thead>
<tr>
<th>Language</th>
<th>Serial Time</th>
<th>OpenMP (10 Threads)</th>
<th>SpeedUp</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0.050000</td>
<td>0.011806</td>
<td>4.235</td>
</tr>
<tr>
<td>Fortran</td>
<td>0.050255</td>
<td>0.011834</td>
<td>4.247</td>
</tr>
</tbody>
</table>
Exercise: Matrix Multiplication I

- Most Computational code involve matrix operations such as matrix multiplication.
- Consider a matrix $C$ of two matrices $A$ and $B$:
  
  Element $i,j$ of $C$ is the dot product of the $i^{th}$ row of $A$ and $j^{th}$ column of $B$.
Exercise: Matrix Multiplication II

- Parallelize the following MATMUL code

**C**

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

#define dt(start, end) ((end).tv_sec - start.tv_sec) + \
    1/1000000.0*(end).tv_usec - start.tv_usec)

int main() {
    int i, j, k;
    int nra=1500, nca=2000, ncb=1000;
    double a[nra][nca], b[nca][ncb], c[nra][ncb];
    struct timeval icalc, scalc, ecalc;
    double flops, sum, timing;
    flops = 2.0 * nra * nca * ncb;
    gettimeofday(&icalc, NULL);
    for (i = 0; i < nra; i++) {
        for (j = 0; j < nca; j++) {
            a[i][j] = (double)(i+j);
        }
    }
    for (j = 0; j < nca; j++) {
        for (k = 0; k < ncb; k++) {
            b[j][k] = (double)(i*j);
        }
    }
    for (i = 0; i < nra; i++) {
        for (k = 0; k < ncb; k++) {
            c[i][k] = 0.0;
        }
    }
    gettimeofday(&scalc, NULL);
    /* Parallelize the following block of code */
    for (i = 0; i < nra; i++) {
        for (k = 0; k < ncb; k++) {
            sum = 0.0;
            for (j = 0; j < nca; j++) {
                sum = sum + a[i][j] * b[j][k];
            }
            c[i][k] = sum;
        }
    }
    gettimeofday(&ecalc, NULL);
    timing = dt(scalc, ecalc);
    printf("Init Time: %6.3f Calc Time: %6.3f GFlops: %7.3f\n", dt(icalc, scalc), timing, le-9 * flops/timing );
}
```

**Fortran**

```fortran
program matrix_mul

implicit none

integer, parameter :: dp = selected_real_kind(14)
integer :: i,j,k
integer, parameter :: nra=1500, nca=2000, ncb=1000
real(dp) :: a(nra,nca), b(nca,ncb), c(nra,ncb)
real(dp) :: flops, sum
real(dp) :: init_time, start_time, end_time
flops = 2d0 * float(nra) * float(nca) * float(ncb)
call cpu_time(init_time)
c = 0d0
   do i = 1,nra
       do j = 1,nca
           a(i,j) = i + j
       end do
   end do
do i = 1,nca
   do j = 1,ncb
       b(i,j) = i * j
   end do
do i = 1,nra
   do j = 1,nca
       do k = 1,ncb
           sum = 0d0
           do i = 1, nra
               sum = sum + a(i,j) * b(j,k)
           end do
           c(i,k) = sum
       end do
   end do
call cpu_time(start_time)
! Parallelize the following block of code
   do j = 1, nca
       do k = 1, ncb
           sum = 0d0
           do i = 1, nra
               sum = sum + a(i,j) * b(j,k)
           end do
           c(i,k) = sum
       end do
   end do
   call cpu_time(end_time)
   ! Parallelize the following block of code
   do j = 1, nca
       do k = 1, ncb
           sum = 0d0
           do i = 1, nra
               sum = sum + a(i,j) * b(j,k)
           end do
           c(i,k) = sum
       end do
   end do
   call cpu_time(end_time)
   print '(a,f6.3,f6.3,f6.3,f7.3)', 'Init Time: ', start_time - init_time, &
   ' Calc Time: ', end_time - start_time, &
   ' GFlops: ', 1d-9 * flops/(end_time - start_time)
end program matrix_mul
```
Solution: MATMUL

C

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define dt(start, end) ((end.tv_sec - start.tv_sec) + 1/1000000.0*(end.tv_usec - start.tv_usec))

int main() {
  int i, j, k;
  int nra=1500, nca=2000, ncb=1000;
  double a[nra][nca], b[nca][ncb], c[nra][ncb];
  struct timeval icalc, scalc, ecalc;
  double flops, sum, timing;
  flops = 2.0 * nra * nca * ncb;

  gettimeofday(&icalc, NULL);
  for (i = 0; i < nra; i++) {
    for (j = 0; j < nca; j++){
      a[i][j] = (double)(i+j);
    }
  }
  for (j = 0; j < nca; j++){
    for (k = 0; k < ncb; k++){
      b[j][k] = (double)(i+j);
    }
  }
  for (i = 0; i < nra; i++) {
    for (k = 0; k < ncb; k++){
      c[i][k] = 0.0;
    }
  }

  gettimeofday(&scalc, NULL);
  #pragma omp parallel for private(sum) shared(a,b,c)
  do {for (j = 0; j < nca; j++){
    sum = 0.0;
    for (i = 0; i < nra; i++){
      sum = sum + a[i][j] * b[j][k];
    }
    c[i][k] = sum;
  } }
  gettimeofday(&ecalc, NULL);

  timing = dt(ecalc, ecalc);
  printf("Init Time: %6.3f Calc Time: %6.3f GFlops: %7.3f\n", dt(icalc, scalc), timing, le-9*flops/timing );
}
```

Fortran

```fortran
program matrix_mul
  implicit none
  integer, parameter :: dp = selected_real_kind(14)
  integer :: i,j,k
  integer, parameter :: nra=1500, nca=2000, ncb=1000
  real(dp) :: a(nra,nca), b(nca,ncb), c(nra,ncb)
  real(dp) :: flops, sum
  integer, dimension(8) :: value
  flops = 2d0 * float(nra) * float(nca) * float(ncb)
  call date_and_time(VALUES=value)
  init_time = float(value(6)*60) + float(value(7)) + float(value(8))/1000d0
  c = 0d0
  do i = 1,nra
    do j = 1,nca
      a(i,j) = i + j
    end do
  end do
  do i = 1,nca
    do j = 1,ncb
      b(i,j) = i * j
    end do
  end do
  call date_and_time(VALUES=value)
  start_time = float(value(6)*60) + float(value(7)) + float(value(8))/1000d0
  !$omp parallel do private(sum) shared(a,b,c)
  do {for (j = 0; j < nca; j++){
    sum = 0d0
    for (i = 0; i < nra; i++){
      sum = sum + a(i,j) * b(j,k);
    }
    c(i,k) = sum;
  } }
  !$omp end parallel do
  call date_and_time(VALUES=value)
  end_time = float(value(6)*60) + float(value(7)) + float(value(8))/1000d0
  print '(a,f6.3,a,f6.3,a,f7.3)', 'Init Time: ', start_time - init_time, 
    ' Calc Time: ', end_time - start_time, 
    ' GFlops: ', le-9 * flops/(end_time - start_time)
end program matrix_mul
```
Pitfalls: False Sharing

- Array elements that are in the same cache line can lead to false sharing.
  - The system handles cache coherence on a cache line basis, not on a byte or word basis.
  - Each update of a single element could invalidate the entire cache line.

```c
!$omp parallel
myid = omp_get_thread_num()
nthreads = omp_get_numthreads()
do i = myid+1, n , nthreads
   a(i) = some_function(i)
end do
!$omp end parallel
```
Pitfalls: Race Condition

- Multiple threads try to write to the same memory location at the same time.
  - Indeterministic results

- Inappropriate scope of variable can cause indeterministic results too.

- When having indeterministic results, set the number of threads to 1 to check
  - If problem persists: scope problem
  - If problem is solved: race condition

```c
!$omp parallel do
do  i = 1, n
  if (a(i) > max) then
    max = a(i)
  end if
end do
!$omp end parallel do
```
Synchronization: Barrier

- “Stop sign” where every thread waits until all threads arrive.
- Purpose: protect access to shared data.
- Syntax:
  - Fortran: !$omp barrier
  - C/C++: #pragma omp barrier
- A barrier is implied at the end of every parallel region
  - Use the nowait clause to turn it off
- Synchronizations are costly so their usage should be minimized.
Synchronization: Critical and Atomic

- Critical: Only one thread at a time can enter a critical region

```ompl
!$omp parallel do
do i = 1, n
  b = some_function(i)
!$omp critical
  call some_routine(b,x)
end do
!$omp end parallel do
```

- Atomic: Only one thread at a time can update a memory location

```ompl
!$omp parallel do
do i = 1, n
  b = some_function(i)
!$omp atomic
  x = x + b
end do
!$omp end parallel do
```
Private Variables

- Not initialized at the beginning of parallel region.
- After parallel region
  - Not defined in OpenMP 2.x
  - 0 in OpenMP 3.x

```c
void wrong()
{
    int tmp = 0;
    #pragma omp for private(tmp)
    for (int j = 0; j < 100; ++j)
        tmp += j
    printf("%d\n", tmp)
}
```

- OpenMP 2.5: tmp undefined
- OpenMP 3.0: tmp is 0
Special Cases of Private

- **Firstprivate**
  - Initialize each private copy with the corresponding value from the master thread

- **Lastprivate**
  - Allows the value of a private variable to be passed to the shared variable outside the parallel region

```c
void wrong()
{
    int tmp = 0;
    #pragma omp for firstprivate(tmp) lastprivate(tmp)
    for (int j = 0; j < 100; ++j)
        tmp += j
    printf("%d\n", tmp)
}
```

The value of tmp is the value when j=99

tmp initialized as 0
Exercise: Calculate pi by Numerical Integration

- We know that
  \[ \int_{0}^{1} \frac{4.0}{(1 + x^2)} \, dx = \pi \]

- So numerically, we can approximate pi as the sum of a number of rectangles
  \[ \sum_{i=0}^{N} F(x_i) \Delta x \approx \pi \]

Meadows et al, A “hands-on” introduction to OpenMP, SC09
Exercise: Rewrite for OpenMP parallelization

C/C++

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

int main() {
    int i;
    long long int n=100000000;
    clock_t start_time, end_time;
    double x, pi;
    double sum = 0.0;
    double step = 1.0/(double) n;

    start_time = clock();
    /* Parallelize the following block of code */
    for (i = 0; i < n; i++) {
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
    end_time = clock();

    printf("pi = %17.15f\n",pi);
    printf("time to compute = %g seconds\n", (double) (end_time - start_time)/CLOCKS_PER_SEC);
    return 0;
}
```

Fortran

```fortran
program pi_serial

  implicit none
  integer, parameter :: dp=selected_real_kind(14)
  integer :: i
  integer, parameter :: n=100000000
  real(dp) :: x,pi,sum,step,start_time,end_time

  sum = 0d0
  step = 1.d0/float(n)

  call cpu_time(start_time)
  ! Parallelize the following block of code
  do i = 0, n
    x = (i + 0.5d0) * step
    sum = sum + 4.d0 / (1.0d0+x**2)
  end do
  pi = step * sum

  call cpu_time(end_time)

  print '(a,f17.15)', "pi = ", pi
  print '(a,f9.6,a)', "time to compute =",end_time - start_time, " seconds"

end program pi_serial
```
**C/C++**

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

int main() {
    long long int i, n = 10000000000;
    double start_time, end_time;
    double x, pi;
    double sum = 0.0;
    double step = 1.0 / (double) n;

    start_time = omp_get_wtime();
    #pragma omp parallel for default(shared) private(i, x)
    for (i = 0; i < n; i++) {
        x = (i + 0.5) * step;
        #pragma omp atomic
        sum += 4.0 / (1.0 + x * x);
    }
    pi = step * sum;
    end_time = omp_get_wtime();

    printf("pi = %17.15f\n", pi);
    printf("time to compute = %g seconds\n", (double) (end_time - start_time));
    return 0;
}
```

**Fortran**

```fortran
program pi_omp

implicit none
integer, parameter :: dp=selected_real_kind(14)
integer, parameter :: ip=selected_int_kind(15)
integer(ip) :: i
integer(ip), parameter :: n=10000000000
real(dp) :: x, pi, sum, step, start_time, end_time
integer, dimension(8) :: value

sum = 0d0
step = 1.d0/float(n)

call date_and_time(VALUES=value)
start_time = float(value(6)*60) + float(value(7)) + float(value(8))/1000d0
!$omp parallel do default(shared) private(i,x)
do i = 0, n
    x = (i + 0.5d0) * step
    !$omp atomic
    sum = sum + 4.d0 / (1.d0 + x ** 2)
end do
!$omp end parallel do
pi = step * sum

call date_and_time(VALUES=value)
end_time = float(value(6)*60) + float(value(7)) + float(value(8))/1000d0
if ( start_time > end_time ) end_time = end_time + 3600d0

print '(a,f17.15)', "pi = ", pi
print '(a,f9.3,a)', "time to compute = ", end_time - start_time, " seconds"
end program pi_omp
```

42 / 51
What is the value of pi if you did not have the *atomic* directive?
The reduction clause allows accumulative operations on the value of variables.

Syntax: reduction (operator:variable list)

A private copy of each variable which appears in reduction is created as if the private clause is specified.

Operators

1. Arithmetic
2. Bitwise
3. Logical
Example: Reduction

C/C++

```c
#include <omp.h>
int main() {
    int i, n = 100, sum, a[100], b[100];
    for (i = 0; i < n; i++) {
        a[i] = i;
        b[i] = 1;
    }
    sum = 0;
    #pragma omp parallel for reduction(+:sum)
    for (i = 0; i < n; i++) {
        sum += a[i] * b[i];
    }
}
```

Fortran

```fortran
program reduction
  implicit none
  integer :: i, n, sum, a(100), b(100)
  n = 100 ; b = 1; sum = 0
  do i = 1 , n
    a(i) = i
  end do
  !$omp parallel do reduction(+:sum)
  do i = 1, n
    sum = sum + a(i) * b(i)
  end do
  !$omp end parallel do
end program reduction
```
Exercise 3: pi calculation with reduction

- Redo exercise 2 with reduction
Solution: pi calculation with reduction I

### C

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

int main() {
    long long int i, n = 10000000000;
    double start_time, end_time;
    double x, pi;
    double sum = 0.0;
    double step = 1.0/(double) n;

    start_time = omp_get_wtime();
    #pragma omp parallel default(shared) private(i, x) reduction(+:sum)
    {
        #pragma omp for
        for (i = 0; i < n; i++) {
            x = (i+0.5)*step;
            sum += 4.0/(1.0+x*x);
        }
    }
    pi = step * sum;
    end_time = omp_get_wtime();

    printf("pi = %17.15f\n",pi);
    printf("time to compute = %g seconds\n", (double)(end_time - start_time));
    return 0;
}
```

### Fortran

```fortran
program pi_omp

implicit none
integer, parameter :: dp=selected_real_kind(14)
integer, parameter :: ip=selected_int_kind(15)
integer(ip) :: i
integer(ip), parameter :: n=10000000000
real(dp) :: x,pi,sum,step,start_time,end_time
integer, dimension(8) :: value

sum = 0.d0
step = 1.d0/float(n)
call date_and_time(VALUES=value)
start_time = float(value(6)*60) + float(value(7)) + float(value(8))/1000d0
!$omp parallel do default(shared) private(i,x) reduction(+:sum)
do i = 0, n
    x = (i + 0.5d0) * step
    sum += 4.d0 / (1.d0+x*x);
end do
!$omp end parallel do
pi = step * sum

call date_and_time(VALUES=value)
end_time = float(value(6)*60) + float(value(7)) + float(value(8))/1000d0
if ( start_time > end_time ) end_time = end_time + 3600d0
print '(a,f17.15)', "pi = ", pi
print '(a,f9.3,a)', "time to compute =", end_time - start_time, " seconds"
end program pi_omp
```

47 / 51
Solution: pi calculation with reduction II

```bash
altair:openmp apacheco$ gcc -fopenmp pi_omp.c -o pic_ompr
altair:openmp apacheco$ gfortran -fopenmp pi_omp.f90 -o pif_ompr
altair:solution apacheco$ echo "Serial C Code"; ./pic
Serial C Code
pi = 3.141592653590426
time to compute = 1.72441 seconds
altair:solution apacheco$ echo "OMP C Code with Atomic"; ./pic_omp
OMP C Code with Atomic
pi = 3.141592653590195
time to compute = 6.10142 seconds
altair:solution apacheco$ echo "OMP C Code with Reduction"; ./pic_ompr
OMP C Code with Reduction
pi = 3.141592653589683
time to compute = 0.48712 seconds
altair:solution apacheco$ echo "Serial F90 Code"; ./pif
Serial F90 Code
pi = 3.141592673590427
time to compute = 0.988196 seconds
altair:solution apacheco$ echo "OMP F90 Code with Atomic"; ./pif_omp
OMP F90 Code with Atomic
pi = 3.141592673590174
time to compute = 7.368610 seconds
altair:solution apacheco$ echo "OMP F90 Code with Reduction"; ./pif_ompr
OMP F90 Code with Reduction
pi = 3.141592673589683
time to compute = 0.400939 seconds
```
Modify/query the number of threads

- `omp_set_num_threads()`, `omp_get_num_threads()`, `omp_get_thread_num()`, `omp_get_max_threads()`

Query the number of processors

- `omp_num_procs()`

Query whether or not you are in an active parallel region

- `omp_in_parallel()`

Control the behavior of dynamic threads

- `omp_set_dynamic()`, `omp_get_dynamic()`
Environment Variables

- OMP_NUM_THREADS: set default number of threads to use.
- OMP_SCHEDULER: control how iterations are scheduled for parallel loops.
References

▶ https://docs.loni.org/wiki/Using_OpenMP
▶ http://en.wikipedia.org/wiki/OpenMP
▶ http://www.nersc.gov/nusers/help/tutorials/openmp
▶ http://www.llnl.gov/computing/tutorials/openMP
▶ http://www.citutor.org