Overview

• Quick introduction.
• Parallel loops.
• Parallel loop directives.
• Parallel sections.
• Some more advanced directives.
• Summary.
Shared memory

- All processors have access to local memory
- Simpler programming
- Concurrent memory access
- More specialized hardware
- CHPC: PC Linux 2-way (Arches)
OpenMP basics

- Compiler directives to parallelize
  - Fortran – source code comments
    !$omp parallel/!$omp end parallel
  - C/C++ - #pragmas
    #pragma omp parallel
- Small set of subroutines, environment variables
  !$ iam = omp_get_num_threads()
Example 1 – numerical integration

\[ \int_{a}^{b} f(x) \approx \sum_{i=1}^{n} \frac{1}{2} h[f(x_{i-1}) + f(x_i)] = \]

\[ \frac{1}{2} h[f(x_0) + f(x_n)] + \sum_{i=1}^{n-1} h[f(x_i)] \]
program trapezoid
  integer n, i
  double precision a, b, h, x, integ, f

1. print*,'Input integ. interval, no. of trap:'
   read(*,*)a, b, n
   h = (b-a)/n
   integ = 0.

2. !$omp parallel do reduction(+:integ) private(x)
   do i=1,n-1
      x = a+i*h
      integ = integ + f(x)
   enddo

3. integ = integ + (f(a)+f(b))/2.
   integ = integ*h
   print*,'Total integral = ',integ
end
Program output

```
%pgf77 -mp trap.f -o trap
%setenv OMP_NUM_THREADS 2
%trap
Input integ. interval, no. of trap:
0 10 100
Total integral = 333.3500000000001
```
Parallel do directive

• **Fortran**

```fortran
$omp parallel do [clause [, clause]]
[$omp end parallel do]
```

• **C/C++**

```c
#pragma omp parallel for [clause [clause]]
```

• **Loops must have precisely determined trip count**
  - no do-while loops
  - no change to loop indices, bounds inside loop (C)
  - no jumps out of the loop (Fortran – exit, goto; C – break, goto)
  - cycle (Fortran), continue (C) are allowed
  - stop (Fortran), exit (C) are allowed
Clauses

- **Control execution of parallel loop**
  - **scope**
    - sharing of variables among the threads
  - **if**
    - whether to run in parallel or in serial
  - **schedule**
    - distribution of work across the threads
  - **ordered**
    - perform loop in certain order
  - **copyin**
    - initialize private variables in the loop
Data sharing

- **Private** – each thread creates a private instance
  - not initialized upon entry to parallel region
  - undefined upon exit from parallel region
  - default for loop indices, variables declared inside parallel loop

- **Shared** – all threads share one copy
  - update modifies data for all other threads
  - default everything else

- **Changing default behavior**
  - default (shared | private | none)
Variable initialization, reduction

- **firstprivate/lastprivate clause**
  - initialization of a private variable
  \[
  \text{!$omp parallel do } \text{firstprivate} (x)\]

- **finalization of a private variable**
  \[
  \text{!$omp parallel do } \text{lastprivate} (x)\]

- **reduction clause**
  - performs global operation on a variable
  \[
  \text{!$omp parallel do } \text{reduction} \ (+ : \text{sum})\]
Removing data dependencies

- **Serial trapezoidal rule**
  
  ```
  integ = 0.
  do i=1,n-1
    x = a+i*h
    integ = integ + f(x)
  enddo
  ```

- **Parallel solution**
  
  ```
  integ = 0.
  !$omp parallel do
  do i=1,n-1
    x = a+i*h
    integ = integ + f(x) private(x) reduction (+:integ)
  enddo
  ```
Data dependence classification

- **Anti-dependence**
  - race between statement $S_1$ writing and $S_2$ reading
  - removal: *privatization*, multiple do loops

- **Output dependence**
  - values from the last iteration used outside the loop
  - removal: *lastprivate* clause

- **Flow dependence**
  - data at one iteration depend on data from another iteration
  - removal: reduction, rearrangement, often impossible
Parallel overhead

- Parallelization costs CPU time
- Nested loops
  parallelize the outermost loop
- if clause
  parallelize only when it is worth it – above certain number of iterations:

```c
!$omp parallel do if (n .ge. 800)
do i = 1, n
  ...
enddo
```
Load balancing - scheduling

- **user-defined work distribution schedule** (type[, chunk])
- **chunk** – number of iterations contiguously assigned to threads
- **type**
  - static – each thread gets a constant chunk
  - dynamic – work distribution to threads varies
  - guided – chunk size exponentially decreases
  - runtime – schedule decided at the runtime
Static schedule timings

Time (sec) vs Chunk

- 2 Threads
- 4 Threads
- 8 Threads

on SGI Origin 2000

Default Niter/ Nproc
Schedule comparison

Time (sec)

Chunk

Dynamic schedule
Guided schedule
Static schedule

NUM_OMP_THREADS = 8

on SGI Origin 2000

Default
Example 2  MPI-like parallelization

#include <stdio.h>
#include "omp.h"
#define min(a,b) ((a) < (b) ? (a) : (b))

1. int istart,iend;
#pragma omp threadprivate(istart,iend)

int main (int argc, char* argv[]){
int n,nthreads,iam,chunk; float a, b;
double h, integ, p_integ;
double f(double x);
double get_integ(double a, double h);

2. printf("Input integ. interval, no. of trap:\n");
scanf("%f %f %d",&a,&b,&n);
h = (b-a)/n;
integ = 0.;
Example 2, cont.

3. 
   
   #pragma omp parallel shared(integ)
   private(p_integ,nthreads,iam,chunk){
   nthreads = omp_get_num_threads();
   iam = omp_get_thread_num();
   chunk = (n + nthreads -1)/nthreads;
   istart = iam * chunk + 1;
   iend = min((iam+1)*chunk+1,n);

   4. p_integ = get_integ(a,h);

   5. 
      #pragma omp critical
      integ += p_integ;
   }

   6. 
      integ += (f(a)+f(b))/2.;
      integ *= h;
      printf("Total integral = %f\n",integ);
      return 0;}

Example 2, cont.

double get_integ(double a, double h)
{
  int i;
  double sum, x;

  sum = 0;
  for (i=istart; i<iend; i++)
  {
    x = a+i*h;
    sum += f(x);
  }
  return sum;
}
Parallel regions

- **Fortran**
  
  !$omp parallel ... !$omp end parallel

- **C/C++**
  
  #pragma omp parallel

- **SPMD parallelism** – replicated execution
- **must be a self-contained block of code** – 1 entry, 1 exit
- **implicit barrier at the end of parallel region**
- **can use the same clauses as in** parallel do/for
threadprivate variables

• global/common block variables are private only in lexical scope of the parallel region

• possible solutions
  • pass private variables as function arguments
  • use threadprivate – identifies common block/global variable as private
  • !$omp threadprivate (/cb/ [,cb/] …)
    #pragma omp threadprivate (list)
  • use copyin clause to initialize the threadprivate variable

  e.g. !$omp parallel copyin(istart,iend)
Mutual exclusion

- **critical section**
  - limit access to the part of the code to one thread at the time
  
  ```
  !$omp critical [name]
  ...
  !$omp end critical [name]
  ```

- **atomic section**
  - atomically updating single memory location
  
  ```
  sum += x
  ```

- **runtime library functions**
Library functions, environmental variables

• **thread set/inquiry**

  `omp_set_num_threads(integer)`
  
  `OMP_NUM_THREADS`

  `integer omp_get_num_threads()`

  `integer omp_get_max_threads()`

  `integer omp_get_thread_num()`

• **set/query dynamic thread adjustment**

  `omp_set_dynamic(logical)`

  `OMP_DYNAMIC`

  `logical omp_get_dynamic()`
• **lock/unlock functions**

omp_init_lock()
omp_set_lock()
omp_unset_lock()
logical omp_test_lock()
omp_destroy_lock()

• **other**

integer omp_get_num_procs()
logical omp_in_parallel()

OMP_SCHEDULE
Work-sharing

• Distribution of work among the processes
• Work sharing constructs
  - do directive – same as parallel do
  - section directive – divide code into sections which run on separate threads
  - single directive – assign work to a single thread
• Restrictions:
  - continuous block; no nesting
  - all threads must reach the same construct
  - constructs can be outside lexical scope of the parallel construct (e.g. subroutine)
Event synchronization

- **barrier** - !$omp barrier
  - synchronizes all threads at that point
- **ordered** - !$omp ordered
  - imposes order across iterations of a parallel loop
- **master** - !$omp master
  - sets block of code to be executed only on the master thread
- **flush** - !$omp flush
  - synchronizes memory and cache on all threads
Summary

• parallel do/for loops
  ▪ variable scope, reduction
  ▪ parallel overhead, loop scheduling
• parallel regions
  ▪ mutual exclusion
  ▪ work sharing
  ▪ synchronization

http://www.chpc.utah.edu/short_courses/intro_openmp
References

• **Web**

  http://www.openmp.org/
  http://www.openmp.org/specs/

• **Book**

  Chandra, Dagum, Kohr,... - Parallel Programming in OpenMP