Introduction to OpenMP

Martin Čuma
Center for High Performance Computing
University of Utah
m.cuma@utah.edu
Overview

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

https://www.surveymonkey.com/r/8HLJ3WK
Shared memory

- All processors have access to local memory
- Simpler programming
- Concurrent memory access
- More specialized hardware
- CHPC: Linux clusters 12 - 32 core nodes

Dual quad-core node

Many-core node (e.g. SGI)
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()
    
    OMP_NUM_THREADS=4
• Shared memory, thread based parallelism
• Explicit parallelism
• Nested parallelism support
• Fork-join model
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 code

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
em001:%module load pgi
em001:%pgf77 -mp= numa trap.f -o trap
em001:%setenv OMP_NUM_THREADS 12
em001:%trap
Input integ. interval, no. of trap:
0 10 100
Total integral = 333.3500000000001
Parallel do directive

- **Fortran**
  
  !$omp parallel do [clause [, clause]]
  !$omp end parallel do

- **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** *(shared, private)*
    sharing of variables among the threads
  ▪ **if**
    whether to run in parallel or in serial
  ▪ **schedule**
    distribution of work across the threads
  ▪ **collapse** *(n)*
    combine nested loops into a single loop for more parallelism
  ▪ **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)
Data reduction

• Threads distribute work
• Need to collect work at the end
  ‒ sum up total
  ‒ find minimum or maximum
• Reduction clause – global operation on a variable
  
  !$omp parallel do reduction(+:var)
  #pragma omp parallel for reduction(+:var)

• Allowed operations - commutative
  ‒ +, *, max, min, logical
Data dependence

- Data in one loop iteration often depend on data written in another loop iteration.
- Anti-dependence: race between statement $S_1$ writing and $S_2$ reading.
  - Removal: privatization
- 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

Examples:

\[
x = a(i) \\
b(i) = c + x
\]

\[
a(i) = a(i+1) + x
\]
• Serial trapezoidal rule

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

• Parallel solution

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

**x** – anti-dependence – privatization
**integ** – flow dependence - reduction
Variable initialization and finalization

- **firstprivate/lastprivate clause**
  - initialization of a private variable
    \$omp parallel do firstprivate(x)
  - finalization of a private variable
    \$omp parallel do lastprivate(x)
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
  ...
endo
```
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 run time
Static schedule timings

on SGI Origin 2000

Default Niter/ Nproc

Time (sec) vs Chunk

2 Threads
4 Threads
8 Threads

Niter/Nproc on SGI Origin 2000

http://www.chpc.utah.edu
Different schedule timings

on SGI Origin 2000

NUM_OMP_THREADS = 8
Example 2
numerical integration

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

\[ \frac{1}{2} h [f(x_0) + f(x_n)] + \sum_{i=1}^{n-1} h [f(x_i)] \]

New concepts:
- Threadprivate
- Parallel regions
Example 2
MPI-like parallelization

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

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);

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

1. istart, iend – global variables

f, get_integ – local functions

2.
Example 2, cont.

3. 
```c
#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);
}
```

parallel section, explicit computation distribution

4. ```c
   p_integ = get_integ(a,h);
```

5. ```c
   #pragma omp atomic
       integ += p_integ;
   }
   
   integ += (f(a)+f(b))/2.;
   integ *= h;
   printf("Total integral = %f\n",integ);
   return 0;
```
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
  \texttt{!omp parallel ... !omp end parallel}
- C/C++
  \texttt{#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 \texttt{parallel do/for}
Work-sharing constructs

- DO/for loop – distributes loop - `do` directive
- Sections – breaks work into separate, discrete sections - `section` directive
- Workshare – parallel execution of separate units of work - `workshare` directive
- Single/master – serialized section of code - `single` directive
• 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)
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

• also available via runtime library functions
task construct

- Used to parallelize irregular, recursive algorithms
- All tasks run independent of each other in parallel, on up to OMP_NUM_THREADS
- Use `taskwait` to wait for all tasks to finish
- Each task has its own data space – use `mergeable` for shared variables to reduce storage needs
- Use `depend` to specify data dependencies
- Often started from serial section
Calculate Fibonacci number using recursion

```c
int fib(int n) {
    int i, j;
    if (n<2) return n;
    else {
        #pragma omp task shared(i)
        i=fib(n-1);
        #pragma omp task shared(j)
        j=fib(n-2);
        #pragma omp taskwait
        return i+j;
    }
    #pragma omp parallel {
        #pragma omp single {
            fibn = fib(n); }
    }
}
```

recursive function

independent task #1

independent task #2

wait till completion of both tasks

main program – need to start parallel section in which the tasks will run
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
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()`
Library functions, environmental variables

• 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
• nested parallel loops
• accelerator support (4.0)
• user defined reduction (4.0)
• thread affinity (4.0)
• SIMD (=vectorization) (4.0)
Summary

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

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

• Web
  http://www.openmp.org/
  https://computing.llnl.gov/tutorials/openMP
  http://openmp.org/mp-
       documents/OpenMP_Examples_4.5.0.pdf

• Books
  Chapman, Jost, van der Pas – Using OpenMP
  Pacheco – Introduction to Parallel Computing

Survey  https://www.surveymonkey.com/r/8HLJ3WK