Using OpenMP

Rebecca Hartman-Baker
Oak Ridge National Laboratory
hartmanbakrj@ornl.gov

Outline

I. About OpenMP
II. OpenMP Directives
III. Data Scope
IV. Runtime Library Routines and Environment Variables
V. Using OpenMP
VI. Project: Computing Pi
I. ABOUT OPENMP

Source: http://xkcd.com/225/
About OpenMP

- Industry-standard shared memory programming model
- Developed in 1997
- OpenMP Architecture Review Board (ARB) determines additions and updates to standard
Advantages to OpenMP

- Parallelize small parts of application, one at a time (beginning with most time-critical parts)
- Can express simple or complex algorithms
- Code size grows only modestly
- Expression of parallelism flows clearly, so code is easy to read
- Single source code for OpenMP and non-OpenMP – non-OpenMP compilers simply ignore OMP directives
OpenMP Programming Model

• Application Programmer Interface (API) is combination of
  – Directives
  – Runtime library routines
  – Environment variables

• API falls into three categories
  – Expression of parallelism (flow control)
  – Data sharing among threads (communication)
  – Synchronization (coordination or interaction)
Parallelism

- Shared memory, thread-based parallelism
- Explicit parallelism (parallel regions)
- Fork/join model

Source: https://computing.llnl.gov/tutorials/openMP/
II. OPENMP DIRECTIVES

II. OpenMP Directives

- Syntax overview
- Parallel
- Loop
- Sections
- Synchronization
- Reduction
Syntax Overview: C/C++

- Basic format
  
  \#pragma omp directive-name [clause] newline

- All directives followed by newline

- Uses pragma construct (pragma = Greek for “thing”)

- Case sensitive

- Directives follow standard rules for C/C++ compiler directives

- Long directive lines can be continued by escaping newline character with \
Syntax Overview: Fortran

- Basic format:
  \[ \text{sentinel directive-name [clause]} \]

- Three accepted sentinels: \texttt{!$omp} \texttt{*$omp} \texttt{c$omp}

- Some directives paired with \texttt{end} clause

- Fixed-form code:
  - Any of three sentinels beginning at column 1
  - Initial directive line has space/zero in column 6
  - Continuation directive line has non-space/zero in column 6
  - Standard rules for fixed-form line length, spaces, etc. apply

- Free-form code:
  - \texttt{! $omp} only accepted sentinel
  - Sentinel can be in any column, but must be preceded by only white space and followed by a space
  - Line to be continued must end in \& and following line begins with sentinel
  - Standard rules for free-form line length, spaces, etc. apply
OpenMP Directives: Parallel

- A block of code executed by multiple threads
- Syntax:

```c
#pragma omp parallel private(list)\shared(list)
{
    /* parallel section */
}

!$omp parallel private(list) &
!$omp shared(list)
! Parallel section
!$omp end parallel
```
```c
#include <stdio.h>
#include <omp.h>
int main (int argc, char *argv[]) {
    int tid;
    printf("Hello world from threads:\n");
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        printf("<%d>\n", tid);
    }
    printf("I am sequential now\n");
    return 0;
}
```
program hello
integer tid, omp_get_thread_num
write(*,*) ‘Hello world from threads:’
!$OMP parallel private(tid)
tid = omp_get_thread_num()
write(*,*) ‘<‘, tid, ‘>’
!$omp end parallel
write(*,*) ‘I am sequential now’
end
Output (Simple Example)

Output 1
Hello world from threads:
<0>
<1>
<2>
<3>
<4>
I am sequential now

Output 2
Hello world from threads:
<1>
<2>
<0>
<4>
<3>
I am sequential now

Order of execution is scheduled by OS!!!!!!
OpenMP Directives: Loop

• Iterations of the loop following the directive are executed in parallel

• Syntax:

```
#pragma omp for schedule(type [,chunk]) \ 
private(list) shared(list) nowait
{
    /* for loop */
}
```

```
!$OMP do schedule(type [,chunk]) &
!$OMP private(list) shared(list)
C do loop goes here
!$OMP end do nowait
```

- `type` = {static, dynamic, guided, runtime}
- If `nowait` specified, threads do not synchronize at end of loop
Which Loops Are Parallelizable?

Parallelizable

• Number of iterations known upon entry, and does not change

• Each iteration independent of all others

• No data dependence

Not Parallelizable

• Conditional loops (many while loops)

• Iterator loops (e.g., iterating over a `std::list<...>` in C++)

• Iterations dependent upon each other

• Data dependence
/* Gaussian Elimination (no pivoting):
   \[ x = A\backslash b \] */

for (int i = 0; i < N-1; i++) {
    for (int j = i; j < N; j++) {
        double ratio = A[j][i]/A[i][i];
        for (int k = i; k < N; k++) {
            A[j][k] -= (ratio*A[i][k]);
            b[j] -= (ratio*b[i]);
        }
    }
}
Example: Parallelizable?
Example: Parallelizable?

• Outermost Loop (i):
  – $N-1$ iterations
  – Iterations depend upon each other (values computed at step $i-1$ used in step $i$)

• Inner loop (j):
  – $N-i$ iterations (constant for given $i$)
  – Iterations can be performed in any order

• Innermost loop (k):
  – $N-i$ iterations (constant for given $i$)
  – Iterations can be performed in any order
Example: Parallelizable?

/* Gaussian Elimination (no pivoting):
   \[ x = A\backslash b \] */

for (int i = 0; i < N-1; i++) {
    #pragma omp parallel for
    for (int j = i; j < N; j++) {
        double ratio = A[j][i]/A[i][i];
        for (int k = i; k < N; k++) {
            A[j][k] -= (ratio*A[i][k]);
            b[j] -= (ratio*b[i]);
        }
    }
}

Note: can combine \texttt{parallel} and \texttt{for} into single \texttt{pragma} line
OpenMP Directives: Loop Scheduling

• Default scheduling determined by implementation

• Static
  – ID of thread performing particular iteration is function of iteration number and number of threads
  – Statically assigned at beginning of loop
  – Load imbalance may be issue if iterations have different amounts of work

• Dynamic
  – Assignment of threads determined at runtime (round robin)
  – Each thread gets more work after completing current work
  – Load balance is possible
#include <omp.h>
#define CHUNKSIZE 100
#define N 1000
int main () {
    int i, chunk;
    float a[N], b[N], c[N];
    /* Some initializations */
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;
    #pragma omp parallel shared(a,b,c,chunk) private(i)
    {
        #pragma omp for schedule(dynamic,chunk) nowait
        for (i=0; i < N; i++)
            c[i] = a[i] + b[i];
    } /* end of parallel section */
    return 0;
}
OpenMP Directives: Sections

- Non-iterative work-sharing construct
- Divide enclosed sections of code among threads
- Section directives nested within sections directive

Syntax: C/C++

```c
#pragma omp sections
{
    #pragma omp section /* first section */
    #pragma omp section /* next section */
}
```

Fortran

```fortran
!$OMP sections
!$OMP section
C First section
!$OMP section
C Second section
!$OMP end sections
```
#include <omp.h>
#define N 1000
int main () {
    int i;
    double a[N], b[N], c[N], d[N];
    /* Some initializations */
    for (i=0; i < N; i++) {
        a[i] = i * 1.5;
        b[i] = i + 22.35;
    }
    #pragma omp parallel \
        shared(a,b,c,d) private(i)
    {
        #pragma omp sections nowait
        {
            #pragma omp section
            for (i=0; i < N; i++)
                c[i] = a[i] + b[i];
            #pragma omp section
            for (i=0; i < N; i++)
                d[i] = a[i] * b[i];
        } /* end of sections */
    } /* end of parallel section */
    return 0;
}
OpenMP Directives: Synchronization

• Sometimes, need to make sure threads execute regions of code in proper order
  – Maybe one part depends on another part being completed
  – Maybe only one thread need execute a section of code

• Synchronization directives
  – Critical
  – Barrier
  – Single
OpenMP Directives: Synchronization

• Critical
  – Specifies section of code that must be executed by only one thread at a time
  – Syntax: C/C++
    #pragma omp critical [name]
    !$OMP critical [name]
  – Names are global identifiers – critical regions with same name are treated as same region

• Single
  – Enclosed code is to be executed by only one thread
  – Useful for thread-unsafe sections of code (e.g., I/O)
  – Syntax: C/C++
    #pragma omp single
    !$OMP single

OpenMP Directives: Synchronization

• **Barrier**
  - Synchronizes all threads: thread reaches barrier and waits until all other threads have reached barrier, then resumes executing code following barrier
  - Syntax: C/C++
    ```
    #pragma omp barrier
    ```
  - Fortran
    ```
    !$OMP barrier
    ```
  - Sequence of work-sharing and barrier regions encountered must be the same for every thread
OpenMP Directives: Reduction

- Reduces list of variables into one, using operator (e.g., max, sum, product, etc.)

- Syntax

  ```
  #pragma omp reduction(op : list)
  !$OMP reduction(op : list)
  ```

  where `list` is list of variables and `op` is one of following:
  - C/C++: +, -, *, &, ^, |, &&, or ||
  - Fortran: +, -, *, .and., .or., .eqv., .neqv., or max, min, iand, ior, ieor
III. VARIABLE SCOPE

Variable Scope

- By default, all variables shared except
  - Certain loop index values – private by default
  - Local variables and value parameters within subroutines called within parallel region – private
  - Variables declared within lexical extent of parallel region – private
Default Scope Example

```c
void caller(int *a, int n) {
    int i,j,m=3;
    #pragma omp parallel for
    for (i=0; i<n; i++) {
        int k=m;
        for (j=1; j<=5; j++) {
            callee(&a[i], &k, j);
        }
    }
}
void callee(int *x, int *y, int z) {
    int ii;
    static int cnt;
    cnt++;
    for (ii=1; ii<z; ii++) {
        *x = *y + z;
    }
}
```

<table>
<thead>
<tr>
<th>Var</th>
<th>Scope</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>shared</td>
<td>Declared outside parallel construct</td>
</tr>
<tr>
<td>n</td>
<td>shared</td>
<td>same</td>
</tr>
<tr>
<td>i</td>
<td>private</td>
<td>Parallel loop index</td>
</tr>
<tr>
<td>j</td>
<td>shared</td>
<td>Sequential loop index</td>
</tr>
<tr>
<td>m</td>
<td>shared</td>
<td>Declared outside parallel construct</td>
</tr>
<tr>
<td>k</td>
<td>private</td>
<td>Automatic variable/parallel region</td>
</tr>
<tr>
<td>x</td>
<td>private</td>
<td>Passed by value</td>
</tr>
<tr>
<td>*x</td>
<td>shared</td>
<td>(actually a)</td>
</tr>
<tr>
<td>y</td>
<td>private</td>
<td>Passed by value</td>
</tr>
<tr>
<td>*y</td>
<td>private</td>
<td>(actually k)</td>
</tr>
<tr>
<td>z</td>
<td>private</td>
<td>(actually j)</td>
</tr>
<tr>
<td>ii</td>
<td>private</td>
<td>Local stack variable in called function</td>
</tr>
<tr>
<td>cnt</td>
<td>shared</td>
<td>Declared static (like global)</td>
</tr>
</tbody>
</table>
Variable Scope

• Good programming practice: explicitly declare scope of all variables

• This helps you as programmer understand how variables are used in program

• Reduces chances of data race conditions or unexplained behavior
Variable Scope: Shared

- Syntax
  - `shared(list)`

- One instance of shared variable, and each thread can read or modify it

- **WARNING:** watch out for multiple threads simultaneously updating same variable, or one reading while another writes

- Example
  ```c
  #pragma omp parallel for shared(a)
  for (i = 0; i < N; i++) {
    a[i] += i;
  }
  ```
### Variable Scope: Shared – Bad Example

```c
#pragma omp parallel for shared(n_eq)
for (i = 0; i < N; i++) {
    if (a[i] == b[i]) {
        n_eq++;
    }
}
```

- `n_eq` will not be correctly updated
- Instead, put `n_eq++;` in critical block (slow); introduce private variable `my_n_eq`, then update `n_eq` in critical block after loop (faster); or use `reduction` pragma (best)
Variable Scope: Private

- Syntax
  - private(list)

- Gives each thread its own copy of variable

- Example
  ```
  #pragma omp parallel private(i, my_n_eq)
  {
    #pragma omp for
    for (i = 0; i < N; i++) {
      if (a[i] == b[i]) my_n_eq++;
    }
    #pragma omp critical (update_sum)
    {
      n_eq+=my_n_eq;
    }
  }
  ```
Best Solution for Sum

#pragma parallel for reduction (+:n_eq)
for (i = 0; i < N; i++) {
    if (a[i] == b[i]) {
        n_eq = n_eq+1;
    }
}


IV. RUNTIME LIBRARY ROUTINES AND ENVIRONMENT VARIABLES

OpenMP Runtime Library Routines

- `void omp_set_num_threads(int num_threads)`
  subroutine `omp_set_num_threads (scalar_integer_expression)`
  - Sets number of threads used in next parallel region
  - Must be called from serial portion of code
OpenMP Runtime Library Routines

• `int omp_get_num_threads()`

integer function omp_get_num_threads()

  – Returns number of threads currently in team executing parallel region from which it is called

• `int omp_get_thread_num()`

integer function omp_get_thread_num()

  – Returns rank of thread

  – \( 0 \leq \text{omp_get_thread_num}() < \text{omp_get_num_threads}() \)
OpenMP Environment Variables

• Set environment variables to control execution of parallel code

• **OMP_SCHEDULE**
  - Determines how iterations of loops are scheduled
  - E.g., `setenv OMP_SCHEDULE “guided, 4”`

• **OMP_NUM_THREADS**
  - Sets maximum number of threads
  - E.g., `setenv OMP_NUM_THREADS 4`
V. USING OPENMP
Conditional Compilation

- Can write single source code for use with or without OpenMP

- Pragmas/sentinels are ignored

- What about OpenMP runtime library routines?
  - \_OPENMP macro is defined if OpenMP available: can use this to conditionally include omp.h header file, else redefine runtime library routines
Conditional Compilation

```c
#ifdef _OPENMP
    #include <omp.h>
#else
    #define omp_get_thread_num() 0
#endif

... int me = omp_get_thread_num();
...```
VI. PROJECT: COMPUTING PI

Project Description

• We want to compute $\pi$
• One method: method of darts*
• Ratio of area of square to area of inscribed circle proportional to $\pi$

*Disclaimer: this is a TERRIBLE way to compute $\pi$. Don’t even think about doing it this way except for the purposes of this project!
Method of Darts

- Imagine dartboard with circle of radius $R$ inscribed in square

- Area of circle: $\pi R^2$

- Area of square: $\left(2R\right)^2 = 4R^2$

- Area of circle divided by Area of square:
  $$\frac{\pi R^2}{4R^2} = \frac{\pi}{4}$$
Method of Darts

• So, ratio of areas proportional to $\pi$

• How to find areas?
  – Suppose we threw darts (completely randomly) at dartboard
  – Could count number of darts landing in circle and total number of darts landing in square
  – Ratio of these numbers gives approximation to ratio of areas
  – Quality of approximation increases with number of darts

• $\pi = 4 \times \frac{\# \text{ darts inside circle}}{\# \text{ darts thrown}}$
Method of Darts

• Okay, Rebecca, but how in the world do we simulate this experiment on computer?
  – Decide on length \( R \)
  – Generate pairs of random numbers \((x, y)\) s.t. \(-R \leq x, y \leq R\)
  – If \((x, y)\) within circle (i.e. if \((x^2 + y^2) \leq R^2\)), add one to tally for inside circle
  – Lastly, find ratio
#include <omp.h>
#include "random.h"
static long num_trials = 10000;

int main() {
    long i;
    long Ncirc = 0;
    double pi, x, y;
    double r = 1.0;  // radius of circle
    double r2 = r*r;
    for (i = 0; i < num_trials; i++) {
        x = random();
        y = random();
        if ((x*x + y*y) <= r2)
            Ncirc++;
    }
    pi = 4.0*((double)Ncirc)/(double)num_trials;
    printf("\n For %d trials, pi = %f\n", num_trials, pi);
}
The Code (random.h)*

```
#include <omp.h>
/* Random number generator -- and not a very good one, 
either */
static long MULTIPLIER = 1366;
static long ADDEND = 150889;
static long PMOD = 714025;
long random_last = 0;

/* This is not a thread-safe random number generator */
double random() {
    long random_next;
    random_next = (MULTIPLIER * random_last + ADDEND) % PMOD;
    random_last = random_next;
    return ((double)random_next/(double)PMOD);
}
```

* Source: SC08 OpenMP “Hands-On” Tutorial
Your Mission (should you choose to accept it)

- Take provided code (darts.c, darts.cc, or darts.f) and parallelize with OpenMP
- Run with different numbers of threads and track performance and accuracy of solution
- Oops! Random number generator is not thread-safe. How can we fix this? (Discussion)
Random Number Generator

• No such thing as random number generation – proper term is pseudorandom number generator (PRNG)

• Generate long sequence of numbers that seems “random”

• Properties of a good PRNG:
  – Very long period
  – Uniformly distributed
  – Reproducible
  – Quick and easy to compute
Pseudorandom Number Generator

- Generator from random.h is Linear Congruential Generator (LCG)
  - Short period (= PMOD, 714025)
  - Not uniformly distributed
    - known to have correlations
  - Reproducible
  - Quick and easy to compute
  - Poor quality (don’t do this at home)

Correlation of RANDU LCG (source: http://en.wikipedia.org/wiki/File:Randu.png)
Pseudorandom Number Generator

- Generator is not thread-safe – how to fix it?

- Problem: all threads have access to random_last
  - Second thread grabs random_last before first thread updates it, resulting in duplicate results
  - Makes reproducible sequence irreproducible – will not happen the same way every time
  - How can we make generator thread-safe?

- Bonus fun: Try different solutions and profile their performance
  - Use `omp_get_wtime()` for timings (elapsed time = end – start)
Bibliography/Resources: OpenMP


- LLNL OpenMP Tutorial, [https://computing.llnl.gov/tutorials/openMP/](https://computing.llnl.gov/tutorials/openMP/)

- OpenMP.org, [http://openmp.org/](http://openmp.org/)

- OpenMP 3.0 API Summary Cards:
Appendix: Better Ways to Compute $\pi$

- Look it up on the internet, e.g. http://oldweb.cecm.sfu.ca/projects/ISC/data/pi.html

- Compute using the BBP (Bailey-Borwein-Plouffe) formula

\[
\pi = \sum_{n=0}^{\infty} \left( \frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6} \right) \left( \frac{1}{16} \right)^n
\]

- For less accurate computations, try your programming language’s constant, or quadrature or power series expansions
Appendix: Better Ways to Generate Pseudorandom Numbers

For serial codes

- Mersenne twister
- GSL (Gnu Scientific Library), many generators available (including Mersenne twister)

For parallel codes

- SPRNG, regarded as leading parallel pseudorandom number generator
- PPRNG, Bill Cochran’s new parallel pseudorandom number generator, supposedly superior to SPRNG
  [http://runge.cse.uiuc.edu/~wkcochra/pprng/](http://runge.cse.uiuc.edu/~wkcochra/pprng/)