Parallel Programming with OpenMP

CS240A, T. Yang, 2013
Modified from Demmel/Yelick’s and Mary Hall’s Slides
Introduction to OpenMP

• What is OpenMP?
  • Open specification for Multi-Processing
  • “Standard” API for defining multi-threaded shared-memory programs
  • openmp.org – Talks, examples, forums, etc.

• High-level API
  • Preprocessor (compiler) directives ( ~ 80% )
  • Library Calls ( ~ 19% )
  • Environment Variables ( ~ 1% )
A Programmer’s View of OpenMP

• OpenMP is a portable, threaded, shared-memory programming \textit{specification} with “light” syntax
  • Exact behavior depends on OpenMP \textit{implementation}!
  • Requires compiler support (C or Fortran)

• OpenMP will:
  • Allow a programmer to separate a program into \textit{serial regions} and \textit{parallel regions}, rather than \(T\) concurrently-executing threads.
  • Hide stack management
  • Provide synchronization constructs

• OpenMP will not:
  • Parallelize automatically
  • Guarantee speedup
  • Provide freedom from data races
Motivation – OpenMP

```c
int main() {

    // Do this part in parallel

    printf( "Hello, World!\n" );

    return 0;
}
```
```c
int main() {

    omp_set_num_threads(4);

    // Do this part in parallel
    #pragma omp parallel
    {
        printf( "Hello, World!\n" );
    }

    return 0;
}
```
OpenMP parallel region construct

- Block of code to be executed by multiple threads in parallel
- Each thread executes the **same code redundantly** (SPMD)
  - Work within work-sharing constructs is distributed among the threads in a team
- Example with C/C++ syntax
  
  ```c
  #pragma omp parallel [ clause [ clause ] ... ] new-line structured-block
  
  ```

- clause can include the following:
  - private (list)
  - shared (list)
OpenMP Data Parallel Construct: Parallel Loop

- All pragmas begin: #pragma
- Compiler calculates loop bounds for each thread directly from *serial* source (computation decomposition)
- Compiler also manages data partitioning
- Synchronization also automatic (barrier)
Programming Model – Parallel Loops

• Requirement for parallel loops
  • No data dependencies (reads/write or write/write pairs) between iterations!

• Preprocessor calculates loop bounds and divide iterations among parallel threads

```c
#pragma omp parallel for
for( i=0; i < 25; i++ )
{
    printf("Foo");
}
```
OpenMP: Parallel Loops with Reductions

- OpenMP supports reduce operation
  
  ```c
  sum = 0;
  #pragma omp parallel for reduction(+:sum)
  for (i=0; i < 100; i++) {
    sum += array[i];
  }
  ```

- Reduce ops and init() values (C and C++):
  
  + 0   bitwise & ~0   logical & 1
  - 0   bitwise | 0   logical | 0
  * 1   bitwise ^ 0
Example: Trapezoid Rule for Integration

- Straight-line approximation

\[
\int_a^b f(x) \, dx \approx \sum_{i=0}^{l} c_i f(x_i) = c_0 f(x_0) + c_1 f(x_1)
\]

\[
= \frac{h}{2} [f(x_0) + f(x_1)]
\]
\[ \int_{a}^{b} f(x) \, dx = \int_{x_{0}}^{x_{1}} f(x) \, dx + \int_{x_{1}}^{x_{2}} f(x) \, dx + \cdots + \int_{x_{n-1}}^{x_{n}} f(x) \, dx \]

\[ = \frac{h}{2} \left[ f(x_{0}) + f(x_{1}) \right] + \frac{h}{2} \left[ f(x_{1}) + f(x_{2}) \right] + \cdots + \frac{h}{2} \left[ f(x_{n-1}) + f(x_{n}) \right] \]

\[ = \frac{h}{2} \left[ f(x_{0}) + 2f(x_{1}) + \cdots + 2f(x_{i}) + \cdots + 2f(x_{n-1}) + f(x_{n}) \right] \]
Serial algorithm for composite trapezoid rule

/* Input:  a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++) {
    x_i = a + i*h;
approx += f(x_i);
}
approx = h*approx;
h = (b−a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n−1; i++)
    approx += f(a + i*h);
approx = h*approx;

# pragma omp parallel for num_threads(thread_count) \
    reduction(+: approx)
for (i = 1; i <= n−1; i++)
    approx += f(a + i*h);
approx = h*approx;
Programming Model – Loop Scheduling

• schedule clause determines how loop iterations are divided among the thread team
  • static([chunk]) divides iterations statically between threads
    • Each thread receives [chunk] iterations, rounding as necessary to account for all iterations
    • Default [chunk] is \( \text{ceil}( \# \text{iterations} / \# \text{threads} ) \)
  • dynamic([chunk]) allocates [chunk] iterations per thread, allocating an additional [chunk] iterations when a thread finishes
    • Forms a logical work queue, consisting of all loop iterations
    • Default [chunk] is 1
  • guided([chunk]) allocates dynamically, but [chunk] is exponentially reduced with each allocation
## Loop scheduling options

<table>
<thead>
<tr>
<th></th>
<th>static</th>
<th>dynamic(3)</th>
<th>guided(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(2)
Impact of Scheduling Decision

• Load balance
  • Same work in each iteration?
  • Processors working at same speed?

• Scheduling overhead
  • Static decisions are cheap because they require no run-time coordination
  • Dynamic decisions have overhead that is impacted by complexity and frequency of decisions

• Data locality
  • Particularly within cache lines for small chunk sizes
  • Also impacts data reuse on same processor
More loop scheduling attributes

• **RUNTIME** The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause.

• **AUTO** The scheduling decision is delegated to the compiler and/or runtime system.

• **NO WAIT / nowait**: If specified, then threads do not synchronize at the end of the parallel loop.

• **ORDERED**: Specifies that the iterations of the loop must be executed as they would be in a serial program.

• **COLLAPSE**: Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause (collapsed order corresponds to original sequential order).
OpenMP environment variables

**OMP_NUM_THREADS**
- sets the number of threads to use during execution
- when dynamic adjustment of the number of threads is enabled, the value of this environment variable is the maximum number of threads to use
- For example,
  ```csh
groupsetenv OMP_NUM_THREADS 16
```
  ```tcsh
groupsetenv OMP_NUM_THREADS=16
```
  ```sh
export OMP_NUM_THREADS=16
```
  ```ksh
export OMP_NUM_THREADS=16
```
  ```bash
export OMP_NUM_THREADS=16
```

**OMP_SCHEDULE**
- applies only to `do/for` and `parallel do/for` directives that have the schedule type `RUNTIME`
- sets schedule type and chunk size for all such loops
- For example,
  ```csh
groupsetenv OMP_SCHEDULE GUIDED,4
```
  ```tcsh
groupsetenv OMP_SCHEDULE= GUIDED,4
```
  ```sh
export OMP_SCHEDULE= GUIDED,4
```
  ```ksh
export OMP_SCHEDULE= GUIDED,4
```
  ```bash
export OMP_SCHEDULE= GUIDED,4
```
Programming Model – Data Sharing

- Parallel programs often employ two types of data
  - Shared data, visible to all threads, similarly named
  - Private data, visible to a single thread (often stack-allocated)

- PThreads:
  - Global-scoped variables are shared
  - Stack-allocated variables are private

- OpenMP:
  - `shared` variables are shared
  - `private` variables are private

```c
// shared, globals
int bigdata[1024];

void* foo(void* bar) {
    int tid;
    #pragma omp parallel
    /* Calculation goes */
    shared (bigdata)
    private (tid)
    {
        /* Calc. here */
    }
}
```
Programming Model - Synchronization

- OpenMP Synchronization
  - OpenMP Critical Sections
    - Named or unnamed
    - No *explicit* locks / mutexes

- Barrier directives

- Explicit Lock functions
  - When all else fails – may require flush directive

- Single-thread regions within parallel regions
  - *master, single* directives

```c
#pragma omp critical
{
   /* Critical code here */
}
```

```c
#pragma omp barrier
omp_set_lock( lock l );
/* Code goes here */
omp_unset_lock( lock l );
```

```c
#pragma omp single
{
   /* Only executed once */
}
```
Microbenchmark: Grid Relaxation (Stencil)

```c
for( t=0; t < t_steps; t++) {
    #pragma omp parallel for \
    shared(grid,x_dim,y_dim) private(x,y)

    for( x=0; x < x_dim; x++) {
        for( y=0; y < y_dim; y++) {
            grid[x][y] = /* avg of neighbors */
        }
    }
    // Implicit Barrier Synchronization

    temp_grid = grid;
    grid = other_grid;
} other_grid = temp_grid;
```
Microbenchmark: Ocean

Normalized Speedup, Ocean 2050x2050

Threading Strategy

Normalized Speedup

Dynamic   Static   Squares   Pthreads

1p   2p   4p   8p   16p

CS267 Lecture 6
Microbenchmark: Ocean

Normalized Speedup, Ocean 258x258

Threading Strategy

Dynamic | Static | Squares | Pthreads

Normalized Speedup

1p | 2p | 4p | 8p | 16p

CS267 Lecture 6
OpenMP Summary

- OpenMP is a compiler-based technique to create concurrent code from (mostly) serial code
- OpenMP can enable (easy) parallelization of loop-based code
  - Lightweight syntactic language extensions
- OpenMP performs comparably to manually-coded threading
  - Scalable
  - Portable
- Not a silver bullet for all applications
More Information

- openmp.org
  - OpenMP official site

- www.llnl.gov/computing/tutorials/openMP/
  - A handy OpenMP tutorial

- www.nersc.gov/nusers/help/tutorials/openmp/
  - Another OpenMP tutorial and reference