Shared Memory Programming with OpenMP (3)

2014 Spring

Jinkyu Jeong
(jinkyu@skku.edu)
SCHEDULING LOOPS
Scheduling Loops (2)

- parallel for directive
  - Basic partitioning policy $\rightarrow$ block partitioning

<table>
<thead>
<tr>
<th>Iteration space</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Thread 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

- Is this optimal?
  - **Yes**, when each iteration takes equal time
  - **No**, when each iteration takes different time
    - ex) larger index takes longer time
Scheduling Loops (2)

- **Example of f(i)**
  - f(i) calls sin function i times
  - Time(f(i)) is linear to i

- **Block partitioning**

- **Cyclic partitioning**

```c
double f(int i) {
    int j, start = i*(i+1)/2, finish = start + i;
    double return_val = 0.0;
    for (j = start; j <= finish; j++) {
        return_val += sin(j);
    }
    return return_val;
} /* f */
```
### Scheduling Loops (3)

- **Example of f(i)**
  - f(i) calls sin function i times
  - Time(f(i)) is linear to i
  - n = 10,000

```java
double f(int i) {
    int j, start = i*(i+1)/2, finish = start + i;
    double return_val = 0.0;
    for (j = start; j <= finish; j++) {
        return_val += sin(j);
    }
    return return_val;
} /* f */
```

<table>
<thead>
<tr>
<th></th>
<th>One thread</th>
<th>Two threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Block</td>
<td>Cyclic</td>
</tr>
<tr>
<td></td>
<td>partitioning</td>
<td>partitioning</td>
</tr>
<tr>
<td>Run-time</td>
<td>3.67s</td>
<td><strong>2.77s</strong></td>
</tr>
<tr>
<td>Speed-up</td>
<td>1x</td>
<td><strong>1.33x</strong></td>
</tr>
</tbody>
</table>

- Scheduling loop (load balancing) is important
Schedule Clause

- **Format**
  - `#pragma omp parallel for schedule(type [, chunk])`
    - `type` := static, dynamic, guided, runtime
    - `chunk` := positive integer
  - **static**
    - Divide iterations by `chunk` (near equal in size by default)
    - Statically assign threads in a round-robin fashion
  - **dynamic**
    - Divide iterations by `chunk` (1 by default)
    - Dynamically assign a chunk to an idle thread (master/worker)
  - **guided**
    - Chunk size is reduced in an exponentially decreasing manner
    - Dynamically assign a chunk to an idle thread (master/worker)
    - Minimum chunk size is specified by `chunk` (1 by default)
  - **runtime**
    - Determined at runtime with OMP_SCHEDULE environment variable
Schedule Clause (Illustration)

- **Dividing iteration space**
  - **Static** schedule on iteration space
    
    | 0 | ¼N | ½N | ¾N | N-1 |
    |---|----|----|----|-----|
    | ![Static Schedule](image) |
  
  - **Dynamic** schedule on iteration space (master/worker)
    
    ![Dynamic Schedule](image) |
  
  - **Guided** schedule on iteration space (master/worker)
    
    ![Guided Schedule](image) |
The Static Schedule Type

- **Static** schedule on iteration space

\[
\begin{array}{cccc}
0 & \frac{1}{4}N & \frac{1}{2}N & \frac{3}{4}N \\
\hline
& \text{Orange} & \text{Red} & \text{Gray} & \text{Gray}
\end{array}
\]

- Example

12 iterations\((0, 1, \ldots, 11)\) and 3 threads

\[
\begin{array}{llll}
\text{schedule (static, 1)} & \text{schedule (static, 2)} & \text{schedule (static, 4)} \\
\text{Thread 0: 0, 3, 6, 9} & \text{Thread 0: 0, 1, 6, 7} & \text{Thread 0: 0, 1, 2, 3} \\
\text{Thread 1: 1, 4, 7, 10} & \text{Thread 1: 2, 3, 8, 9} & \text{Thread 1: 4, 5, 6, 7} \\
\text{Thread 2: 2, 5, 8, 11} & \text{Thread 2: 4, 5, 10, 11} & \text{Thread 2: 8, 9, 10, 11}
\end{array}
\]
The Dynamic Schedule Type

- **Dynamic** schedule on iteration space (master/worker)

  - The iterations are broken up into chunks of chunksize consecutive iterations
  - Each thread executes a chunk
  - When a thread finishes a chunk, it requests another one from the run-time system
  - This continues until all the iterations are completed
  - The chunksize is 1 by default
The Guided Schedule Type

- **Guided** schedule on iteration space (master/worker)

  - The initial value of chunksize
    $\Rightarrow$ # of iterations / # of threads
  - Each thread executes a chunk, and when a thread finishes a chunk, it requests another one from runtime system
  - However, in a guided schedule, as chunks are completed the size of the new chunks decreases
  - If no **chunksize** is specified, the size of the chunks decreases down to 1
  - If **chunksize** is specified, it decreases down to **chunksize**
    - The very last chunk can be smaller than **chunksize**
Example

Two threads, 10000 iterations, chunksize = 1

<table>
<thead>
<tr>
<th>Thread</th>
<th>Chunk</th>
<th>Size of Chunk</th>
<th>Remaining Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 – 5000</td>
<td>5000</td>
<td>4999</td>
</tr>
<tr>
<td>1</td>
<td>5001 – 7500</td>
<td>2500</td>
<td>2499</td>
</tr>
<tr>
<td>1</td>
<td>7501 – 8750</td>
<td>1250</td>
<td>1249</td>
</tr>
<tr>
<td>1</td>
<td>8751 – 9375</td>
<td>625</td>
<td>624</td>
</tr>
<tr>
<td>0</td>
<td>9376 – 9687</td>
<td>312</td>
<td>312</td>
</tr>
<tr>
<td>1</td>
<td>9688 – 9843</td>
<td>156</td>
<td>156</td>
</tr>
<tr>
<td>0</td>
<td>9844 – 9921</td>
<td>78</td>
<td>78</td>
</tr>
<tr>
<td>1</td>
<td>9922 – 9960</td>
<td>39</td>
<td>39</td>
</tr>
<tr>
<td>1</td>
<td>9961 – 9980</td>
<td>20</td>
<td>19</td>
</tr>
<tr>
<td>1</td>
<td>9981 – 9990</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>9991 – 9995</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>9996 – 9997</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>9998 – 9998</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>9999 – 9999</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
The Runtime Schedule Type

- Environment variable `OMP_SCHEDULE` determines the schedule of loop at run-time

- **OMP_SCHEDULE**
  - static, dynamic, or guided
  - Example
    - `$ export OMP_SCHEDULE="static, 1"$
    - `$ ./omp_program 4`
    - static scheduling with chunksize of 1
CRITICAL SECTION AND LOCKS
Critical Directive

- **Format**
  - `# pragma omp critical`
  - Provides mutual exclusion of the following structured block to all threads in a team

| # pragma omp critical Enqueue(&queue, &message) | # pragma omp critical Dequeue(&queue, &message) |

- **Problem**
  - Distinct critical sections are treated as one composite critical section
  - Serialization of all threads

- Critical sections for queue

| # pragma omp critical Enqueue(&queue, &message) | # pragma omp critical Dequeue(&queue, &message) |

- Critical sections for stack

| # pragma omp critical Push(&stack, &message) | # pragma omp critical Pop(&stack, &message) |
Named Critical Directive

- **Format**
  - `# pragma omp critical(name)`
  - Specifies the name of a critical section
  - OpenMP provides mutual exclusion to the critical sections having the same name

  ```
  # pragma omp critical(queue)
  Enqueue(&queue, &message)
  # pragma omp critical(queue)
  Dequeue(&queue, &message)
  # pragma omp critical(stack)
  Push(&stack, &message)
  # pragma omp critical(stack)
  Pop(&stack, &message)
  ```

- **Problem**
  - Distinction of critical sections is made at compilation time
  - No critical section distinction between different data structures
Lock APIs in OpenMP

- When to distinct critical sections based on the data structure

- Usage
  - `omp_lock_t lock;`
  - `omp_init_lock(&lock); omp_destroy_lock(&lock);`
  - `omp_set_lock(&lock); omp_unset_lock(&lock);`

- Example

```c
/* q_p = msg_queues[dest] */
omp_set_lock(&q_p->lock);
Enqueue(q_p, my_rank, msg);
omp_unset_lock(&q_p->lock);

/* q_p = msg_queues[my_rank] */
omp_set_lock(&q_p->lock);
Dequeue(q_p, &src, &msg);
omp_unset_lock(&q_p->lock);
```
Atomic Directive

- **Format**
  - #pragma omp atomic

- **It only protects critical sections that consist of a single C assignment statement**

- **Valid statement format:**
  
  ```
  x <op>= <expression>;
  x++;
  ++x;
  x--;  
  --x;
  ```

- **Supported operations:**
  
  ```
  +, *, -, /, &, ^, |, <<, or >>
  ```
Barrier Directive

- OpenMP
  - Barrier synchronization
    - Wait until all the threads in a team reach to a point
  - `#pragma omp barrier`

```c
main() {
    #pragma omp parallel
    sub();
}
sub() {
    work1();
    #pragma omp barrier
    work2();
}
```
OpenMP Programming Practice

- **OpenMP**
  - Start with a **parallelizable algorithm**
  - Implement serially, mostly ignoring
    - Data races
    - Synchronization
    - Threading syntax
  - Test & Debug
  - Annotation with **directives** for parallelization & synchronization
  - Test & Debug

- **Ideal way**
  - Start with some algorithm
  - Implement serially, ignoring
    - Data races
    - Synchronization
    - Threading syntax
  - Test & Debug
  - **Auto-magically** parallelize

SSE3054: Multicore Systems | Spring 2014 | Jinkyu Jeong (jinkyu@skku.edu)
OpenMP Summary

- **OpenMP is:**
  - An API that may be used to explicitly direct multi-threaded, shared memory parallelism
  - Portable
    - C/C++ and Fortran support
    - Implemented on most Unix variants and Windows
  - Standardized
    - Major computer HW and SW vendors jointly defines (OpenMP.org)

- **OpenMP does NOT:**
  - Support distributed memory systems
    - but Cluster OpenMP does
  - Automatically parallelize
  - Have data distribution controls
  - Guarantee efficiency, freedom from data races, ...