CSCI-UA.0480-003
Parallel Computing

Lecture 14: OpenMP III

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
Question

We have seen that in reduction, OpenMP uses local variables and initializes them to identity value (e.g. 1 for multiplication and 0 for addition). What is the identity value for:

- `&&`
- `||`
- `&`
- `|`
- `^`
MORE ABOUT LOOPS IN OPENMP: SORTING
Bubble Sort

```c
for (list_length = n; list_length >= 2; list_length--)
    for (i = 0; i < list_length-1; i++)
        if (a[i] > a[i+1]) {
            tmp = a[i];
            a[i] = a[i+1];
            a[i+1] = tmp;
        }
```

Loop-carried dependency in inner loop
Loop-carried dependency in outer loop

What can we do?
Serial Odd-Even Transposition Sort

```c
for (phase = 0; phase < n; phase++)
    if (phase % 2 == 0)
        for (i = 1; i < n; i += 2)
            if (a[i-1] > a[i]) Swap(&a[i-1], &a[i]);
    else
        for (i = 1; i < n-1; i += 2)
            if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
```
# Serial Odd-Even Transposition Sort

<table>
<thead>
<tr>
<th>Phase</th>
<th>Subscript in Array</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>9 &lt;-&gt; 7</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>7 &lt;-&gt; 6</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>
Serial Odd-Even Transposition Sort

for (phase = 0; phase < n; phase++)
if (phase % 2 == 0)
    for (i = 1; i < n; i += 2)
        if (a[i-1] > a[i]) Swap(&a[i-1], &a[i]);
else
    for (i = 1; i < n-1; i += 2)
        if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);

No dependence in inner loops

Outer-loop carried dependence
What if a thread proceeds from phase \( p \) to phase \( p+1 \) before other threads?

Performance issue:
For each outer iteration, OpenMP may fork-join threads.
Repeated overhead per iteration.
Can we do better?
for directive does not fork any threads. But uses whatever threads that have been forked before in the enclosing parallel block.
Odd-even sort with two parallel for directives and two for directives.
(Times are in seconds.)
Array of 20,000 elements

<table>
<thead>
<tr>
<th>thread_count</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two parallel for directives</td>
<td>0.770</td>
<td>0.453</td>
<td>0.358</td>
<td>0.305</td>
</tr>
<tr>
<td>Two for directives</td>
<td>0.732</td>
<td>0.376</td>
<td>0.294</td>
<td>0.239</td>
</tr>
</tbody>
</table>
SCHEDULING LOOPS
Take a look at this:

```c
sum = 0.0;
for (i = 0; i <= n; i++)
    sum += f(i);
```

- Usually, the default for many OpenMP implementations is to parallelize the above iterations as block of consecutive \( \frac{n}{\text{thread\_count}} \) iterations to each thread.
- What if \( f(i) \) has latency that increases with \( i \)? What is the best schedule then?
Example of function $f$.

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;
} /* $f$ */
\[
\begin{align*}
\text{sum} &= 0.0; \\
\text{for} \ (i = 0; \ i \ <= \ n; \ i++) \\
\quad \text{sum} &= \text{f}(i);
\end{align*}
\]

Wouldn't this be better? (why?)

<table>
<thead>
<tr>
<th>Thread</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0, (n/t), 2n/t, ...</td>
</tr>
<tr>
<td>1</td>
<td>1, (n/t + 1), 2n/t + 1, ...</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>(t-1)</td>
<td>(t-1), (n/t + t - 1), 2n/t + t - 1, ...</td>
</tr>
</tbody>
</table>

Assignment of work using cyclic partitioning.
Results

• $f(i)$ calls the sin function $i$ times.
• Assume the time to execute $f(2i)$ requires approximately twice as much time as the time to execute $f(i)$.

• $n = 10,000$
  – one thread
  – run-time = 3.67 seconds.
Results

- \( n = 10,000 \)
  - two threads
  - default assignment
  - run-time = 2.76 seconds
  - speedup = 1.33

- \( n = 10,000 \)
  - two threads
  - cyclic assignment
  - run-time = 1.84 seconds
  - speedup = 1.99
The Schedule Clause

• Default schedule:

    sum = 0.0;
    
    # pragma omp parallel for num_threads(thread_count) \ 
       reduction(+:sum)
    for (i = 0; i <= n; i++)
       sum += f(i);

• Cyclic schedule:

    sum = 0.0;
    
    # pragma omp parallel for num_threads(thread_count) \ 
       reduction(+:sum) schedule(static,1)
    for (i = 0; i <= n; i++)
       sum += f(i);
schedule ( type [, chunksize] )

• Type can be:
  – *static*: the iterations can be assigned to the threads before the loop is executed.
  – *dynamic* or *guided*: the iterations are assigned to the threads while the loop is executing.
  – *auto*: the compiler and/or the run-time system determine the schedule.
  – *runtime*: the schedule is determined at run-time.

• The chunksize is a positive integer.
The Static Schedule Type

Example: twelve iterations, 0, 1, \ldots, 11, and three threads

\begin{align*}
schedule(\text{static,1}) & \quad schedule(\text{static,2}) \\
Thread 0 & : \quad 0, 3, 6, 9 \quad & Thread 0 & : \quad 0, 1, 6, 7 \\
Thread 1 & : \quad 1, 4, 7, 10 \quad & Thread 1 & : \quad 2, 3, 8, 9 \\
Thread 2 & : \quad 2, 5, 8, 11 \quad & Thread 2 & : \quad 4, 5, 10, 11
\end{align*}

\begin{align*}
schedule(\text{static,4}) \\
Thread 0 & : \quad 0, 1, 2, 3 \\
Thread 1 & : \quad 4, 5, 6, 7 \\
Thread 2 & : \quad 8, 9, 10, 11
\end{align*}

How to implement the default block scheduling using the static schedule?
The Dynamic Schedule Type

- The iterations are broken up into chunks of \texttt{chunksize} consecutive iterations.
- Each thread executes a chunk, and when a thread finishes a chunk, it requests another one from the runtime system.
- This continues until all the iterations are completed.
- The \texttt{chunksize} can be omitted. When it is omitted, a default \texttt{chunksize} of 1 is used.
The Guided Schedule Type

• Each thread also executes a chunk, and when a thread finishes a chunk, it requests another one.
• 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, with the exception that the very last chunk can be smaller than chunksize.
Example:
Assignment of trapezoidal rule iterations 1-9999 using a guided schedule with two threads.

<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

- The system uses the environment variable `OMP_SCHEDULE` to determine at run-time how to schedule the loop.
- The `OMP_SCHEDULE` environment variable can take on any of the values that can be used for a static, dynamic, or guided schedule.
- Example:
  
  ```
  export OMP_SCHEDULE ="static,1"
  ```
Another Way for Controlling the Schedule

- Using the `omp_set_schedule` function. Syntax:
  ```c
  void omp_set_schedule(omp_sched_t kind, int chunk_size);
  ```

- Where `kind` is one of:
  - `omp_sched_static`
  - `omp_sched_dynamic`
  - `omp_sched_guided`
  - `omp_sched_auto`
Keep in mind:

• There is an overhead in using the schedule directive
• The overhead is higher in dynamic than static schedules
• The overhead of guided is the greatest of all three.
• So: if we get satisfactory performance without schedule then don’t use schedule.
Rules of thumb

- If each iteration requires roughly the same amount of computation → default is best
- If the cost of each iteration increases/decreases linearly as the loop executes → static with small chunksize
- If the cost cannot be determined → you need to try several schedules: schedule(runtime) and try different options with OMP_SCHEDULE
Can we parallelize the following loop? If yes, do it. If not, why not?

\[
a[0] = 0;
for( \text{i = 1; i < n; i++} )
\quad a[\text{i}] = a[\text{i-1}] + \text{i};
\]
Conclusions

• OpenMP depends on compiler directives, runtime library, and environment variable.

• The main concept to parallelize a program with OpenMP is how to have independent for-loops.

• May aspects of OpenMP are still implementation dependent, so you need to be careful!