NORTHERN ARIZONA UNIVERSITY
School of Informatics, Computing, and Cyber Systems

Using OpenMP. Part II

CS450/CS550: Parallel Programming

Week 10

NORTHERN ARIZONA UNIVERSITY
School of Informatics, Computing, and Cyber Systems

Loop parallelization issues

- Not all loops can be safely parallelized
- If there are inter-iteration dependencies, race conditions can appear
- Simple example – the Fibonacci sequence:

```
for (int i = 2; i < K; i++) {
  a[i] = a[i - 1] + a[i - 2];
}
```

- This situation is an example of the data dependency: the iteration `i` needs data produced by previous iterations `(i – 1)` and `(i – 2)`

Dependencies in loops

1. **Data dependency**:

- Current iteration needs data produced by the previous iterations
- Also knows as RAW – Read After Write
- Example:

```
a[i] = a[i - 2] * a[i - 1];
```

Dependencies in loops

2. **Anti-dependency:**

- Current iteration must use data before it is updated by the next iterations
- Also known as WAR – Write After Read
- Example:

```
a[i] = a[i + 1] + a[i + 2];
```

Dependencies in loops

3. **Output dependency:**

- Different iterations write to the same addresses
- Also known as WAW – Write After Write
- Example:

```
a[i] = 2 * a[i + 1];
a[i + 2] = 3 * a[i];
```

  - Iteration `0`: update `a[0]` and `a[2]`
  - Iteration `1`: update `a[1]` and `a[3]`
  - Iteration `2`: update `a[2]` and `a[4]`, etc.

Dealing with dependencies

- Loops should not be parallelized if the program logic contains dependencies
- Loops with output dependency or anti-dependency are inherently sequential
- Loops with data dependency have race conditions that could be fixed with locks:
  - But in this case, the loop likely becomes sequential
  - The execution time will become slower than sequential due to locking overhead

Justifying the sample size (reminder)

The necessary sample size for a random variable with a normal distribution:

$$n_p = \left\lceil \frac{u_\alpha^2 \cdot \sigma^2}{\varepsilon^2} \right\rceil$$

- α is a significance level ($\alpha = 0.05$)
- u_α is the value of a random variable with standard normal distribution $(N: 0, 1)$, such that $\mathrm{Prob}\{-u_\alpha < U < u_\alpha\} = 1 - \alpha$
- ε is a permissible maximum error of the estimated mean value: $\varepsilon = \mu \cdot \alpha$
- μ and σ are the parameters of the normally distributed variable (expected value and standard deviation)

NORTHERN ARIZONA UNIVERSITY
School of Informatics, Computing, and Cyber Systems

Inverting nested loops

- Consider the nested loop:

```
for (i = 1; i < n; i++)
  for (j = 1; j < m; j++)
    a[i][j] = a[i - 1][j];
```

- Slow parallelized version:

```
for (i = 1; i < n; i++)
  #pragma parallel for
  for (j = 1; j < m; j++)
    a[i][j] = a[i - 1][j];
```

- Fast parallelized version:

```
#pragma parallel for
for (j = 1; j < m; j++)
  for (i = 1; i < n; i++)
    a[i][j] = a[i - 1][j];
```

Conditionally parallelized loops

- The `if` clause syntax:

`#pragma omp parallel for` **`if`**`(`*`expression`*`)`

If the *`expression`* evaluates to true, the loop will be executed in parallel

- The `if` clause allows the compiler to insert code that determines at run-time whether the loop should be executed in parallel
- In some cases, the sequential execution could be more effective than the parallel version: e.g., if a loop does not have enough iterations, the time for forking and joining threads may exceed the time saved by dividing the loop iterations among threads

Justifying the parallelization

- To create an effective parallel code, the preliminary study of the program execution time may be needed
- The list of the input factors that may affect the execution time should be identified
- The plan of a full-factor experiment should be prepared:
  - the plan consists of the experiment series
  - the execution time for the unique combination of the input factors is studied in each series
  - in series, the studied function should be launched multiple times to guarantee the statistically significant results of measurements
- Based on the experiment results, the condition expression for the `if` clause is defined

Scheduling loops

- In some loops, the execution time for different iterations may vary considerably
- In such cases, the maximum possible speedup cannot be achieved by dividing the iterations among threads evenly: some threads will complete the job earlier and remain idle
- To avoid these situations, the schedule clause is used to specify how the iterations of a loop should be allocated to threads:

`#pragma omp parallel for` **`schedule`**`(`*`type`*`[, `*`chunk`*`])`

  - *`type`* is one of `static`, `dynamic`, `guided`, or `runtime`
  - *`chunk`* is the number of iterations assigned to a thread

Static scheduling

- The static schedule (default type) means the allocation of approximately the **same number** of contiguous iterations to each thread **before** the loop iterations start executing
- If the chunk size is provided for static scheduling, the *`chunk`* of iterations is allocated to threads in turns
- Increasing the chunk size can reduce overhead. Reducing the chunk size can allow finer balancing of workloads
- Static schedules have **low overhead** but may exhibit **high load imbalance**

Dynamic and guided scheduling

- In a dynamic schedule, only some of iterations are allocated to threads at the beginning of the loop execution
- Threads that completed the assigned iterations may get additional work
- Dynamic schedules have **higher overhead** but can **reduce load imbalance**
- In a guided schedule, a dynamic allocation of iterations to tasks is performed using the guided **self-scheduling heuristics**
- Guided self-scheduling begins by allocating **a large chunk** size to each thread and responds to further requests for chunks by allocating **chunks of decreasing size**
- The size of the chunks in guided schedules decreases **exponentially** to a minimum size of `chunk` (by default `chunk=1` for guided schedules)

The `runtime` type of scheduling

- If the `runtime` parameter is provided, the schedule is chosen at run-time based on the value of the environment variable `OMP_SCHEDULE`:
  - In csh-like shells:

    ```
    setenv OMP_SCHEDULE "static,1"
    ```

  - In bash-like shells:

    ```
    export OMP_SCHEDULE="static,1"
    ```

NORTHERN ARIZONA UNIVERSITY
School of Informatics, Computing, and Cyber Systems

Assignment #4

- Prepare the parallelized version of the provided sequential code
- Study the execution time of the parallelized code for different combinations of input parameters ($\mathrm{N} \in [50; 300],\ \mathrm{M} \in [50; 300]$)
- Maximize the speedup of your parallelized version (you may justify your decisions in commentaries to your code)
- The solution must satisfy the following conditions:
  - The program must not contain race conditions
  - The parallelized version must be as fast as possible (certainly faster than the sequential version)
  - The result returned by the parallelized version must be the same as for the sequential version for any combination of the input parameters N and M