Part B - Languages

OpenMP - Dependencies

Describe the effect of dependencies on parallelization
Show how to resolve dependencies to enable parallelization
Introduce accompanying profiling and debugging tools

Dependencies | Loop-Level Parallelism | Nested Iterations | Exercises

Not all serial programs can be converted into parallel programs by simply adding directives. If the results produced by a parallelized program are to be consistent with those produced by its serial version, statements that execute concurrently should not depend on one another. If they do, the results that the parallelized program produces will vary with the order in which the threads execute. Wherever dependencies exist, the sets of instructions that contain them should execute within the same thread.

This chapter describes the dependencies that may obstruct loop-level parallelism and shows how to resolve them through code restructuring and the OpenMP `parallel for` construct. This chapter also introduces Iteration Space Dependency Graphs, which help identify the iterations that are independent of one another.

DEPENDENCIES

Dependencies between statements are classified under two distinct categories:

- data dependencies
- control flow and resource dependencies

Data Dependencies

Data dependencies are of four types:

- Flow Dependency: read after write (symbol: δ^f)

```
a = 5;
b = a + 5;
```

- Anti-Dependency: write after read (symbol: δ^α)

```
b = a + 5;
a = 5;
```

- Output Dependency: write after write (symbol: δ^o)

```
a = 5;
a = b + 5;
```

- Input Dependency: read after read (symbol: δ^i)

```
b = a + 5;
c = a;
```

Control Flow and Resource Dependencies

Dependencies other than data dependencies include:

- Control Flow Dependency: (symbol: δ^c)

```
a = f(x);
if (a < 0)
    x = y + 3;
else
    x = y - 3;
```

- Resource Dependency (hardware dependent): (symbol: δ^r)

```
b = a + 5;
a = 5;
```

LOOP-LEVEL PARALLELISM

OpenMP implements loop-level parallelism using the **`for`** construct. This construct parallelizes the iteration in the structured block that follows the hosting directive (**`for`**, **`parallel for`**, **`parallel for simd`**,).

Canonical Form

For OpenMP to parallelize he iteration correctly, the iteration must be in *canonical form*. Canonical form satisfies the following requirements:

- the control variable
  - is of **`int`**, **`pointer`**, or **`iterator`** type
  - remains unmodified throughout the body of the iteration
  - changes only within the **`for`** clause and the change is iteration invariant
- the limit on the control variable is an iteration invariant
- the body of the iteration does not transfer control outside itself
- the body of the iteration may include a call to **`exit()`**

If the iteration is not in canonical form, the **`for`** construct may produce unexpected results.

The Body of the Iteration

Implementing a **`for`** construct raises the possibility of out-of-order execution of the iterations within the loop that immediate follows the construct. If a statement in one iteration depends on a statement in another iteration, we say that the body of the iteration has a *loop-carried dependency*.

Removing Loop-Carried Data Dependencies

Sources: Barlas, G. (2015), Jordan, H.F., Alaghband, G. (2003).

Some loop-carried dependencies can be eliminated, some can be localized and some cannot be removed. The examples below show some improvements. The source code with the loop-carried dependency is listed on the left. The restructured code with the loop-carried dependency removed is listed on the right.

- A loop-carried flow dependency caused by an induction variable. An *induction* variable is a variable that depends on the control variable. In the following example, **`i`** is the control variable and **`x`** is an induction variable. Each iteration evaluates **`f(x)`**. **`x`** changes with each iteration. We can remove such dependencies by embedding the evaluation of the induction variable within the body of the iteration itself:

```
float x = b;
for (int i = 0; i < n; i++) {
    y[i] = f(x);
    x = x + a;
}
```

```
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    float x = a * i + b;
    y[i] = f(x);
}
```

- A loop-carried flow dependency caused by consumption of a value produced in a previous iteration. In the following example, `a[i]` is evaluated based on `x[i-1]`. We can remove such a dependency by skewing the loop. Note the adjusted limit on the control variable and the two statements that address the edge cases.

```
for (int i = 1; i < n; i++) {
    a[i] = f(x[i - 1]);
    x[i] = x[i] + y[i];
}
```

```
a[1] = f(x[0]);
#pragma omp parallel for
for (int i = 1; i < n - 1; i++) {
    x[i] = x[i] + y[i];
    a[i + 1] = f(x[i]);
}
x[n - 1] = x[n - 1] + y[n - 1];
```

- Two loop-carried flow dependencies within a nested loop. The inner loop can be parallelized since there is no loop-carried dependency on j. Parallelization of the inner loop removes one of the dependencies. The outer loop cannot be parallelized.

```
for (int i = 1; i < n; i++)
    for (int j = 0; j < m; j++)
        a[i][j] = a[i - 1][j] +
                  a[i + 1][j];
```

```
for (int i = 1; i < n; i++) {
    #pragma omp parallel for
    for (int j = 0; j < m; j++)
        a[i][j] = a[i - 1][j] +
                  a[i + 1][j];
```

- A loop-carried anti-dependency that accesses an unmodified element of an array to modify another element. We can remove this dependency by storing the unmodified elements in a separate array.

```
for (int i = 0; i < n - 1; i++) {
    x[i] = x[i + 1] + z;
}
```

```
for (int i = 0; i < n - 1; i++)
    xx[i] = x[i + 1];
#pragma omp parallel for
for (int i = 0; i < n - 1; i++)
    x[i] = xx[i] + z;
```

- A loop-carried output dependency where the value at the final iteration is needed after the iterations complete. We resolve this dependency using the `lastprivate` clause.

```
for (int i = 0; i < n; i++) {
    y[i] = m * x[i] + c;
    w = fabs(y[i]);
}
```

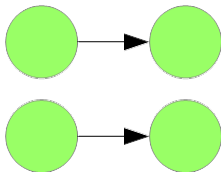
```
#pragma omp parallel for lastprivate(w)
for (int i = 0; i < n; i++) {
    y[i] = m * x[i] + c;
    w = fabs(y[i]);
}
```

NESTED ITERATIONS

Dependencies within nested iterations can be identified graphically using an Iteration Space Dependency Graph (ISDG).

Iteration Space Dependency Graph

An ISDG illustrates the dependencies in an iteration and identifies the groups of iterations that are independent of one another. Nodes on the graph represent iterations whiled edges on the graph represent data flows. The edges are directed with the source at the node that produces the data value and the sink at the node that consumes the data value.



Iteration Space Dependency Graph - Data Nodes and Dependencies

Examples

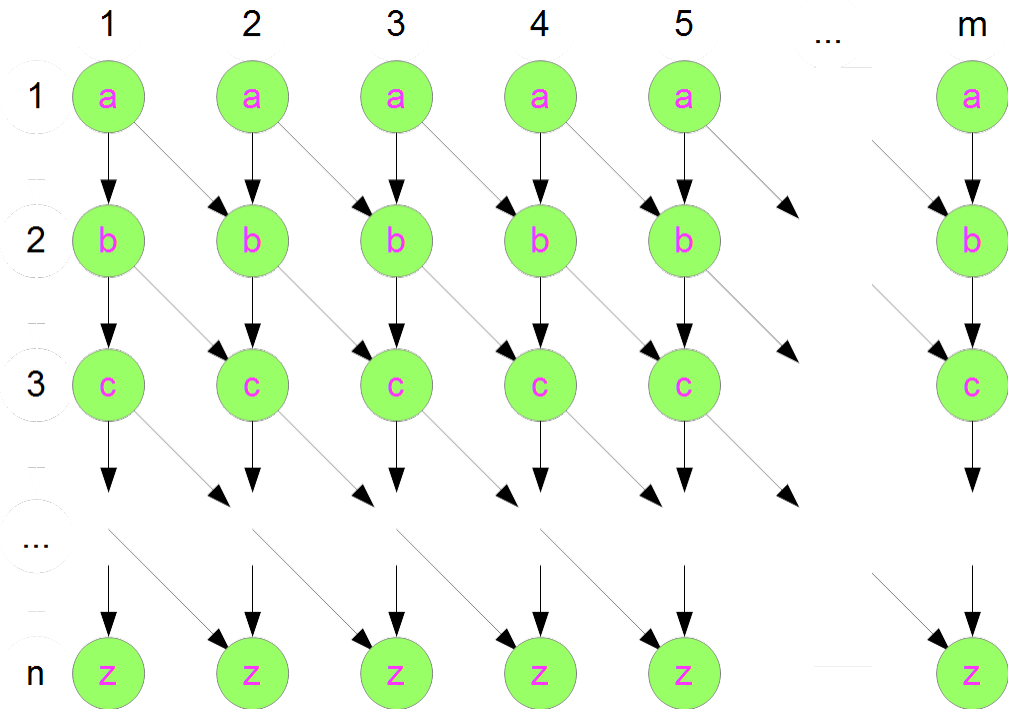
Independent Elements along Rows

Consider the statements listed below. The value for each element in row i is the sum of two adjacent values in row i-1.

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

The figure below shows the ISDG for this nested iteration. The columns are iterations of the inner loop. The rows are iterations of the outer loop.

Since there are no edges between the nodes of the same row, the inner loop is parallelizable. The nodes identified by the same letter are independent of one another.



Iteration Space Dependency Graph - Example 1

There is no loop-carried dependency on the inner iteration. We can write

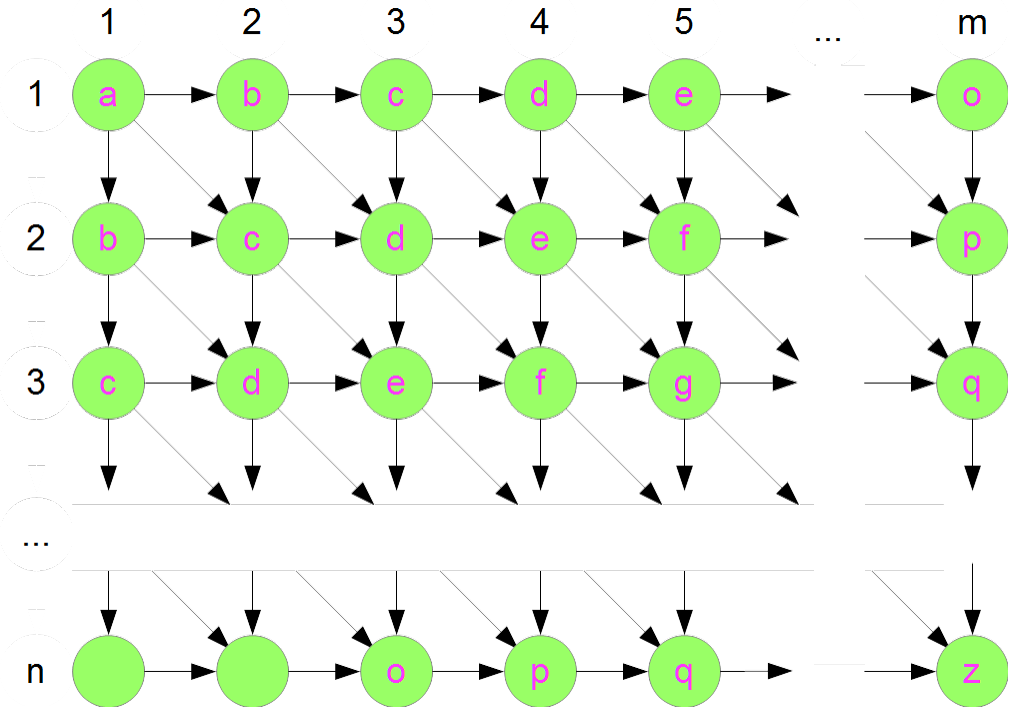
```
for (int i = 1; i < n; i++)
    #pragma omp parallel for
    for (int j = 1; j < m; j++)
        a[i][j] = a[i - 1][j] + a[i - 1][j - 1];
```

Example 2 - Independent Elements along the Diagonals

Consider the statements listed below. The value for each element in row i is the sum of one adjacect value in row i and two adjacent values in row i-1.

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

The figure below shows the ISDG for this nested iteration. There are no edges between the nodes of the same diagonal from the lower left to the upper right. That is, the nodes identified by the same letter are independent of one another and can be executed in parallel. The set of parallel executions represents a wave passing through all of the nodes in the graph. The size of each group increases at the beginning, is constant for the interior iterations and decreases at the end.



Iteration Space Dependency Graph - Example 2

The OpenMP directive for parallelizing the inner iteration uses the **`parallel for`** work sharing construct. The **`default`** clause specifies no sharing and the **`shared`** clause lists the variables that are exceptions to the default.

```
// assume n < m
for (int w = 1; w <= n + m - 3; w++) {
    int width = w;
    if (w + 1 >= n)
        width = n - 1;
    if (w + 1 >= m)
        width = (n - 1) - (w - m) - 1;

    #pragma omp parallel for default(none) shared(a, w, width, n, m)
    for (int k = 0; k < width; k++) {
        int i = w - k;
        int j = k + 1;
        if (w > n - 1) {
            i = n - 1 - k;
            j = w - (n - 1) + k + 1;
        }
        a[i][j] = a[i - 1][j] + a[i][j - 1] + a[i - 1][j - 1];
    }
}
```

EXERCISES

- Barlas, G. (2015). Multicore and GPU Programming - An Integrated Approach. Morgan-Kaufmann. 978-0-12-417137-4. pp.179-192.
- Jordan, H. F., Alaghband, G. (2003). Fundamentals of Parallel Processing. Prentice-Hall. 978-0-13-901158-7. pp. 269-273.