OpenMP6
Scheduling

- Static
- Dynamic
- Guided
  - GroupSize = \( \min(\text{remain}, \max(\text{chunk}, \text{ceiling}(\text{remain}/p))) \)
  - Remain is the remaining iterations
  - Chunk is the chunk size
- Auto
- Runtime
Figure 4.7: Examples of resulting iteration partitioning and assignment based on the scheduling scheme for a 1000-iteration for loop. We end up with 334 groups for static, 250 groups for dynamic, and 15 groups for guided scheduling for the given chunk_size parameters. Only for the static scheme do we have *apriori* knowledge of the iteration group assignment to threads.
Data Dependencies

• Flow dependency
  – RAW
• Antidependency
  – WAR
• Output dependency
  – WAW
Flow Dependency

```plaintext
x = 10;    // S1
y = 2 * x + 5;   // S2
```
Antidependency

```plaintext
y = x + 3; // S1
x ++ ;    // S2
```
Output Dependency

\[
x = 10; \quad // \quad S1
\]
\[
x = x + c \ ; \quad // \quad S2
\]
Flow Dependency

```c
double v = start;
double sum = 0;
for (int i = 0; i < N; i++)
{
    sum = sum + f(v);  // S1
    v = v + step;      // S2
}
```
Eliminate Flow Dependency

double v = start;
double sum = 0;

#pragma omp parallel for reduction(+ : sum)
for (int i = 0; i < N; i++)
{
    v = i * step + start;
    sum = sum + f(v);
}
Flow Dependence

Another technique involves the rearrangement of the loop body statements. Example with:

```c
for (int i = 1; i < N; i++)
{
    y[i] = f(x[i-1]); // S1
    x[i] = x[i] + c[i]; // S2
}
```
Eliminate Flow Dependency
Loop Skewing

\[
y[1] = f(x[0]);
\]

\[
\text{for (int } i = 1; i < N - 1; i++)
\]
\[
\{
    x[i] = x[i] + c[i];
    y[i + 1] = f(x[i]);
\}
\]

\[
x[N - 1] = x[N - 1] + c[N - 1];
\]
Flow Dependencies: Refactoring

● Refactoring refers to rewriting the loop(s) so that parallelism can be exposed.

The ISDG for the following example:

```java
for (int i = 1; i < N; i++)
    for (int j = 1; j < M; j++)
        data[i][j] = data[i - 1][j] + data[i][j - 1] + data[i - 1][j - 1]; // S1
```
Figure 4.5: Iteration space dependency graph for Listing 4.14 for N=4 and M=7.
Figure 4.6: The iteration space dependency graph of Figure 4.5, with highlighted groups of iterations that can be executed concurrently. The groups have to be executed in sequence.
Practice Problem

• A bounded stack is a stack that has an upper limit on the number of values that can be stored in the stack at any time. Create a .h file and a .c file that implements a thread safe bounded stack. The implementation should include a data structure to hold stack contents and functions to create a stack, destroy a stack, push and pop
Bounded Stack
Bounded Stack

struct BoundedStack {
    int maxSize;
    int top;
    void ** stack;
    omp_lock_t lock;
};

struct BoundedStack *CreateBoundedStack(int size);

void DestroyBoundedStack(struct BoundedStack *s);

void push(struct BoundedStack *s, void * d);

void *pop(struct BoundedStack *s);
Bounded Stack

```c
struct BoundedStack *CreateBoundedStack(int size) {
    struct BoundedStack *s = (struct BoundedStack *) malloc(sizeof(struct BoundedStack));
    s->maxSize = size;
    s->top = 0;
    s->stack = (void **) malloc(size*sizeof(void *));
    omp_init_lock(&s->lock);
    return s;
}
```
void push(struct BoundedStack *s, void * d){
    omp_set_lock(&s->lock);
    while (s->top == s->maxSize) {
        omp_unset_lock(&s->lock);
        usleep(100);
        omp_set_lock(&s->lock);
    }
    s->stack[s->top] = d;
    (s->top)++;
    omp_unset_lock(&s->lock);
}
Bounded Stack

void *pop(struct BoundedStack *s) {
    void * d;
    omp_set_lock(&s->lock);
    if (s->top == 0) d = NULL;
    else {
        s->top--;
        d = s->stack[s->top];
    }
    omp_unset_lock(&s->lock);
    return d;
}