Recitation 3: OpenMP Programming

15-418 Parallel Computer Architecture and Programming
CMU 15-418/15-618, Fall 2023

Overview of OpenMP

- OpenMP (*Open Multi-Processing*) is an API for shared-memory parallel programming in C, C++, and Fortran.
- It is based on **pragmas** or **directives** which augment the source code and change how a compiler processes the source code.
- `#pragma omp directive-name [clause[ [,] clause] ... ]`

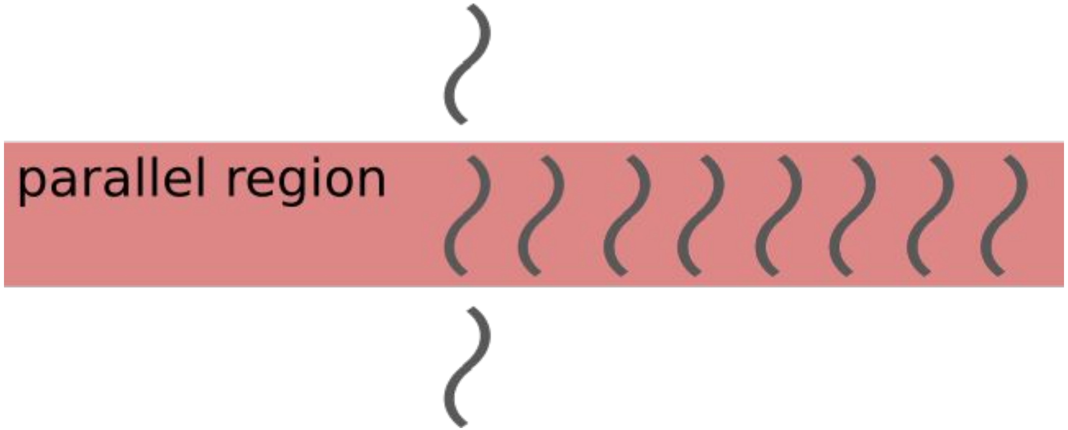
Parallel construct

```
#pragma omp parallel [clause[ [,] clause] ... ]
    structured-block
```

- creates a **team** of OpenMP threads that executes the `structured-block` as a **parallel region**

Parallel construct

```
#pragma omp parallel ...
{
     statement1;
     statement2;
     ...
 }
```



Parallel Construct: Example

```c
int main() {
    #pragma omp parallel
    printf("Hello world!\n");
     return 0;
 }
```

```
$ gcc -o my_program my_program.c -Wall -fopenmp
$ ./my_program
Hello world!
Hello world!
...
Hello world!
```

Parallel construct

- Behavior of a parallel construct can be modified with several **clauses**:

- E.g. `#pragma omp parallel if (a > b) num_threads(3)`

...

Data sharing rules in parallel construct

- Transition from sequential region to parallel region

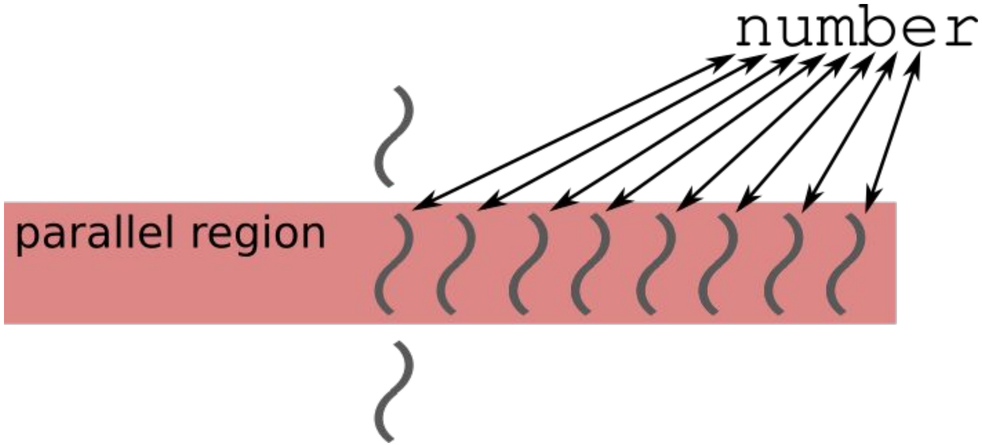
- Have to make sure that the related data accesses do not cause any conflicts

Data sharing rules: example

```
int main() {
    int number = 1;
    #pragma omp parallel
    printf("%d.\n", number++);
    return 0;
}
```

Data sharing rules: example

```
int main() {
    int number = 1;
    #pragma omp parallel
    printf("%d.\n", number++);
    return 0;
}
```



Data sharing rule

All variables are either **private** or **shared:**

1. All variables declared outside parallel region are shared.
2. All variables declared inside a parallel region are private.
3. Loop counters are private (in parallel loops).

Data sharing example

```
int a = 5;                    // shared/private?
int main() {
    int b = 44;               // ??
    #pragma omp parallel
    {
        int c = 3;            // ??
    }
}
```

Data sharing example

```c
int a = 5;                      // shared

int main() {

    int b = 44;                 // shared

    #pragma omp parallel

    {

        int c = 3;              // private

    }

}
```

Creating private variables

- Use private clause to turn a variable that has been declared outside a parallel region into a private variable:

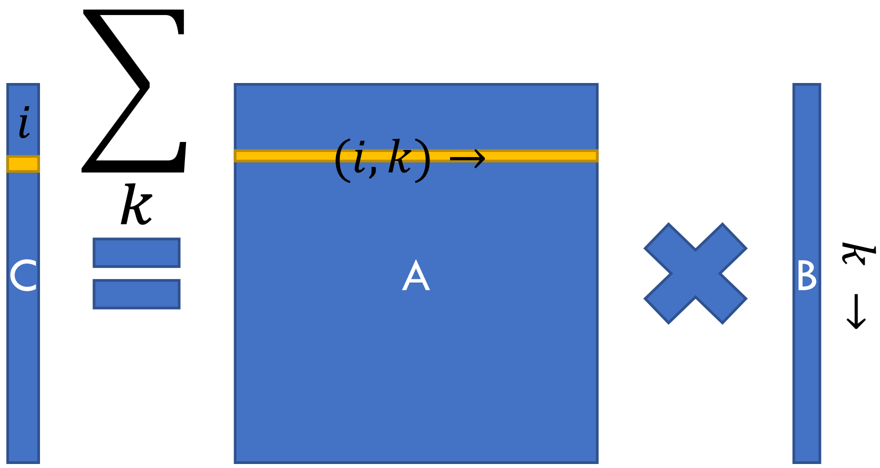
```
int main() {
    int number = 1;
    #pragma omp parallel private(number)
    printf("%d.\n", number++);
    return 0;
}
```

Private vs firstprivate

- The **private** variables do NOT inherit the value of the original variable
- If we want this to happen, then we must use the **firstprivate** clause

```
int main() {
    int number = 1;
    #pragma omp parallel\
firstprivate(number)
    printf("%d.\n", number++);
    return 0;
 }
```

Today: Matrix-vector multiplication



- $(n{\times}n){\times}(n{\times}1) \Rightarrow (n{\times}1)$ output vector
- Output = dot-products of rows from A and the vector B

Matrix-vector multiplication

- Simple C++ implementation:

```
/* Find element based on row-major ordering */
#define RM(r, c, width) ((r) * (width) + (c))

void matrixVectorProduct(int N, float *matA, float *vecB, float *vecC) {
    for (int i = 0; i < N; i++)
        float sum = 0.0;
        for (int k = 0; k < N; k++)
            sum += matA[RM(i,k,N)] * vecB[k];
        vecC[i] = sum;
    }
}
```

Matrix-vector multiplication

- Our code is slightly refactored:

```
typedef float data_t;

typedef unsigned index_t;

float rvp_dense_seq(dense_t *m, vec_t *x, index_t r) {
    index_t nrow = m->nrow;
    index_t rstart = r*nrow;
    data_t val = 0.0;
    for (index_t c = 0; c < nrow; c++)
        val += x->value[c] * m->value[rstart+c];
    return val;
}

void mvp_dense_seq(dense_t *m, vec_t *x, vec_t *y, rvp_dense_t rp_fun) {
    index_t nrow = m->nrow;
    for (index_t r = 0; r < nrow; r++) {
        y->value[r] = rp_fun(m, x, r);
    }
}
```

Row dot product (the inner loop over k in original code)

The outer loop over rows (over i in original code)

Parallel Outer Loop

```
void mvp_dense_mps(dense_t *m, vec_t *x, vec_t *y, rvp_dense_t rp_fun) {
    index_t nrow = m->nrow;

    #pragma omp parallel for schedule(static)

    for (index_t r = 0; r < nrow; r++) {
        y->value[r] = rp_fun(m, x, r);
    }
}
```

- Recruit multiple threads
- Have each do subrange of row indices

Understanding Parallel Outer Loop

```
void mvp_dense_mps_impl(dense_t *m, vec_t *x, vec_t *y, rvp_dense_t rp_fun)
{
    index_t nrow = m->nrow;
    #pragma omp parallel
    {
        // Following code executed by each thread
        index_t t = omp_get_thread_num();
        index_t tcount = omp_get_num_threads();
        index_t delta = (nrow+tcount-1)/tcount;
        index_t rstart = t * delta;
        index_t rend = (t+1) * delta;
        if (rend > nrow) rend = nrow;
        for (index_t r = rstart; r < rend; r++) {
            y->value[r] = rp_fun(m, x, r);
        }
    }
}
```

Activate tcount threads

Partition range into blocks of size delta

Assign separate block to each thread

- Each thread t does its range of rows

Parallel Inner Loop

```
data_t rvp_dense_mpr(dense_t *m, vec_t *x, index_t r) {
    index_t nrow = m->nrow;
    index_t rstart = r*nrow;
    data_t val = 0.0;

    #pragma omp parallel for reduction(+:val)

    for (index_t c = 0; c < nrow; c++) {
        data_t mval = m->value[rstart+c];
        data_t xval = x->value[c];
        val += mval * xval;
    }
    return val;
}
```

Partition range into blocks of size delta

Each thread accumulates its subrange of values

Combine values across threads

- Recruit multiple threads
- Accumulate separate copies of val and combine

Benchmarking dense mat-vec

- Matrix: 256 x 256 (65,536 entries)
  - Sequential: 2.48 GF
  - Parallel Rows: 15.43 GF (6.22 X)
  - Parallel Columns: 4.90 GF (1.98 X)
    - Tasks are too fine-grained

Task construct

```
#pragma omp task [clause[ [,] clause] ... ] new-line
     structured-block
```

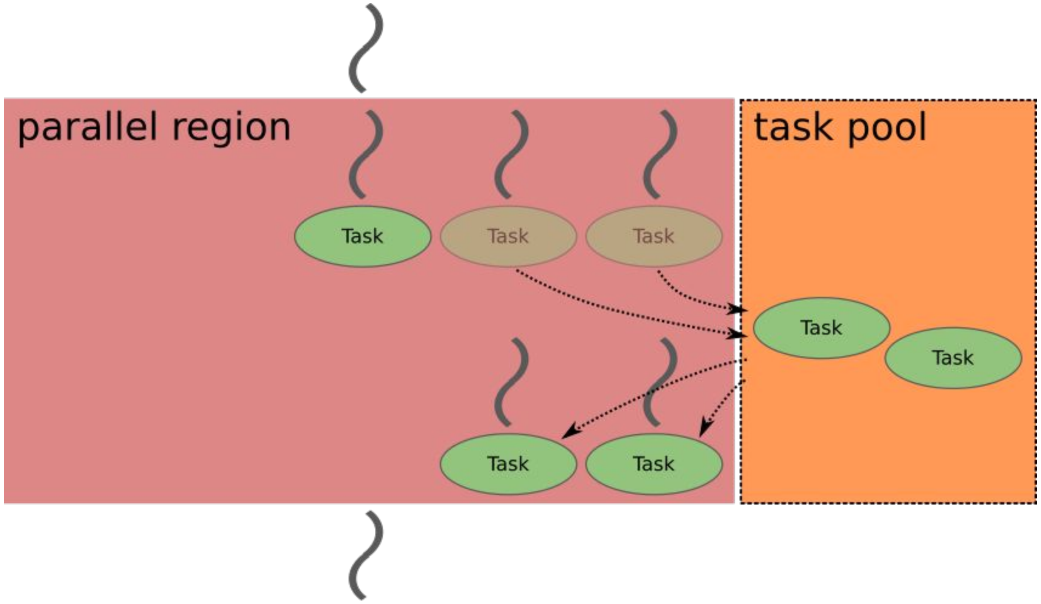
The thread that encounters the **task** construct creates an explicit task from the structured block.

The encountering thread

- may execute the task immediately or
- **defer** its execution to one of the other threads in the team.
- Task is generally used within a parallel region

Task construct example

```c
int main() {
    #pragma omp parallel
    {
        #pragma omp task
        printf("Hello world!\n");
     }
     return 0;
}
```

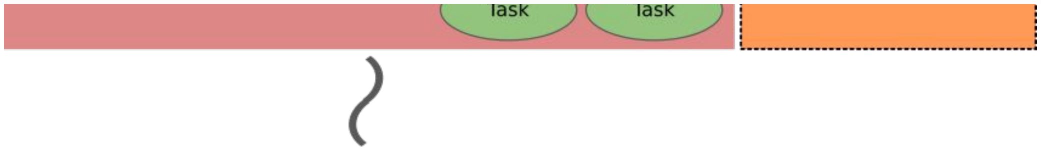


Task construct example

```c
int main() {
    #pragma omp parallel
```

```
$ gcc -o my_program my_program.c -Wall -fopenmp
$ ./my_program
Hello world!
Hello world!
...
Hello world!
```

```c
    return 0;
}
```



Data sharing rules in task constructs

- variables declared outside the parallel construct are **shared** by default

- If we move the variable number inside the parallel construct, then the variable becomes **firstprivate** by default (i.e. unique copy + inherit value)

Data sharing rules in task constructs: example

```c
int main() {
    int number = 1;
    #pragma omp parallel
    {
        #pragma omp task
        {
            printf("I think the number is %d.\n", number);
            number++;
        }
    }
    return 0;
}
```

Data sharing rules in task constructs: example

```
int main() {
    int number = 1;
    #pragma omp parallel
    {
        #pragma omp task
        {
            printf("I think the number is %d.\n", number);
            number++;
        }
    }
    return 0;
}
```

Data sharing rules in task constructs: example

```c
int main() {

    int number = 1;
```

```
$ gcc -o my_program my_program.c -Wall -fopenmp
$ ./my_program
I think the number is 1.
I think the number is 2.
I think the number is 2.
I think the number is 3.
...
```

```c
                number++;

            }

        }

    return 0;

}
```

Data sharing rules in task constructs: example

```c
int main() {
    #pragma omp parallel
    {
        int number = 1;
        #pragma omp task
        {
            printf("I think the number is %d.\n", number);
            number++;
        }
    }
    return 0;
}
```

Data sharing rules in task constructs: example

```
int main() {
    #pragma omp parallel
```

```
$ gcc -o my_program my_program.c -Wall -fopenmp
$ ./my_program
I think the number is 1.
I think the number is 1.
I think the number is 1.
I think the number is 1.
...
```

```
            number++;
        }
    }
    return 0;
}
```

Single Construct

- In earlier examples, each thread in the **team** created a **task**
- What if only one thread should generate the task?
  - Single construct

```
#pragma omp single
```

Single construct: example

```c
int main() {
    #pragma omp parallel
    #pragma omp single nowait
    {
        #pragma omp task
        printf("Hello world!\n");
    }
    return 0;
}
```

```
$ gcc -o my_program my_program.c -Wall -fopenmp
$ ./my_program
Hello world!
```

Child tasks and Taskwait construct

- Child tasks: task generated within a task
- Tasks don't have an execution order, use **taskwait** to enforce execution order
- Wait on the completion of **child tasks** of the generating task
- `#pragma omp taskwait [clause[ [,] clause] ... ]`

Child tasks and Taskwait construct: example

```
int main() {
    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp task
        {
            #pragma omp task
            printf("Hello.\n");

            #pragma omp taskwait

            printf("Hi.\n");
        }

        printf("Hej.\n");
    }

    printf("Goodbye.\n");

    return 0;
}
```

Child tasks and Taskwait construct: example

```c
int main() {
    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp task
        {
            #pragma omp task
            printf("Hello.\n");

            #pragma omp taskwait

            printf("Hi.\n");
        }

        printf("Hej.\n");
    }

    printf("Goodbye.\n");

    return 0;
}
```

```
$ gcc -o my_program my_program.c -Wall -fopenmp
$ ./my_program
Hej.
Hello.
Hi.
Goodbye.
```

Example: implementing fibonacci with tasks

```c
int fib(int n)
{
    if (n < 2)
        return n;

    int i, j;
    i = fib(n-1);
    j = fib(n-2);

    return i + j;
}
```

```c
int main() {
    printf("fib(10) = %d\n", fib(10));

    return 0;
}
```

```c
int fib(int n)
{
    if (n < 2)
        return n;

    int i, j;

    #pragma omp task  default(none) shared(i) firstprivate(n)
    i = fib(n-1);

    #pragma omp task  default(none) shared(j) firstprivate(n)
    j = fib(n-2);

    #pragma omp taskwait

    return i + j;
}
```

```c
int main() {
    #pragma omp parallel
    #pragma omp single
    printf("fib(10) = %d\n", fib(10));

    return 0;
}
```

Common Mistake #1

```
void mvp_dense_mps_impl(dense_t *m, vec_t *x, vec_t *y, rvp_dense_t rp_fun)
{
    index_t nrow = m->nrow;
    index_t t, tcount, delta, rstart, rend;
    #pragma omp parallel
    {
        // Following code executed by each thread
        t = omp_get_thread_num();
        tcount = omp_get_num_threads();
        delta = (nrow+tcount-1)/tcount;
        rstart = t * delta;
        rend = (t+1) * delta;
        if (rend > nrow) rend = nrow;
        for (index_t r = rstart; r < rend; r++) {
            y->value[r] = rp_fun(m, x, r);
        }
    }
}
```

Variables declared outside scope of omp parallel are global to all threads

- Variables outside of parallel are global
- Either wrong answers or poor performance

Common Mistake #2

```
data_t rvp_dense_mpr(dense_t *m, vec_t *x, index_t r) {
    index_t nrow = m->nrow;
    index_t idx = r*nrow;
    data_t val = 0.0;

    #pragma omp parallel for reduction(+:val)

    for (index_t c = 0; c < nrow; c++) {
        data_t mval = m->value[idx++];
        data_t xval = x->value[c];
        val += mval * xval;
    }
    return val;
}
```

Sequential version stepped through matrix values sequentially

But, that's not true for parallel version

- Low-level optimization can often introduce sequential dependency

Common Mistake #3

```
void full_mvp_csr_allocate(csr_t *m, vec_t *x, vec_t *y) {
    index_t nrow = m->nrow;
    index_t nnz = m->nnz;
    // Allocate new scratch vectors
    vec_t *scratch_vector[MAXTHREAD];
    #pragma omp parallel
    {
        index_t t = omp_get_thread_num();
        index_t tcount = omp_get_num_threads();
        scratch_vector[t] = new_vector(nrow);
    . . .
```

Scratch vectors allocated every time multiplication performed

- Allocate all data structures beforehand
  - Typical computation uses them repeatedly

Resources

OpenMP official documentation:

https://www.openmp.org/resources/refguides/

OpenMP Tutorial:

https://hpc2n.github.io/Task-based-parallelism/branch/master/openmp-basics1/