Introduction to Parallel Computing

a.y. 2025 – 2026

Dr. Brahmaiah Gandham

Lecture 10

Shared Memory programming with OpenMP

Outline

- **Data Sharing Clauses**
- **Work Sharing Directives**
- **Synchronization Directives**

Data Sharing Clauses

- `private`
  - each thread gets a private copy of variable, no longer *"storage-associated"* with original variable
  - private copy is not initialized
  - default for variables declared inside parallel region
  - often better to declare variables inside parallel region, reduces amount of code (also minimizes *"vertical distance"* in source code)

```c
int main() {
    int x = 10;

    #pragma omp parallel private(x)
    {
        int tid = omp_get_thread_num();
        x = 10;  // must initialize, each thread has its own copy
        x = x + tid;
        printf("Thread %d: x = %d\n", tid, x);
    }

    printf("After parallel region: x = %d\n", x);
    return 0;
}
```

Data Sharing Clauses

- **shared**
  - each thread references the same, global copy
  - data races if access is not synchronized
  - default for variables declared outside parallel region and global variables, often used for read-only access

```c
int main() {
    int x = 10;

    #pragma omp parallel shared(x)
    {
        int tid = omp_get_thread_num();
        x = x + tid;  // all threads modify the same variable
        printf("Thread %d: x = %d\n", tid, x);
    }

    printf("After parallel region: x = %d\n", x);
    return 0;
}
```

Data Sharing Clauses

- **`default`**
  - can be set to `shared`, or none for C/C++
  - none helpful for detecting missing variables in clauses (compiler will complain!)

```c
int main() {
    int x = 10;

    // Must explicitly specify every variable
    #pragma omp parallel default(none) private(x)
    {
        int tid = omp_get_thread_num();
        x = 10;  // each thread has its own copy
        x = x + tid;
        printf("Thread %d: x = %d\n", tid, x);
    }

    printf("After parallel region: x = %d\n", x);
    return 0;
}
```

Data Sharing Clauses cont’d

- `firstprivate`
  - like `private`, but private copies are initialized with value of copy outside of parallel region

```c
int x = 10;

#pragma omp parallel firstprivate(x)
{
    x += omp_get_thread_num();  // each thread starts with x=10
    printf("Thread %d: x = %d\n", omp_get_thread_num(), x);
}

printf("After parallel: x = %d\n", x); // still 10
```

Data Sharing Clauses cont'd

UNITRENTO

- `lastprivate`
  - like `private`, but outside copy is set to the private copy of the final iteration (`for` loops) or last section (`sections`), **NOT** the iteration/section that was chronologically executed last

```c
int x = 0;

#pragma omp parallel for lastprivate(x)
for (int i = 0; i < 4; i++) {
    x = i * 10;
    printf("Thread %d: x = %d\n", omp_get_thread_num(), x);
}

printf("After parallel for: x = %d\n", x);
```

Data Sharing Clauses cont'd

- `threadprivate`
  - like private, but will persist across parallel regions
  - master thread variable is storage-associated with original variable (not the case for `private`!)

```c
static int counter = 0;
#pragma omp threadprivate(counter)

#pragma omp parallel
{
    counter = omp_get_thread_num();
    printf("Region 1 - Thread %d: counter = %d\n", omp_get_thread_num(), counter);
}

#pragma omp parallel
{
    printf("Region 2 - Thread %d: counter = %d\n", omp_get_thread_num(), counter);
}
```

Reduction Clause

- performs reduction to a single variable in parallel or loop context
  - arithmetic ops: +, -, *, max, min
  - logical ops: &, &&, |, ||, ^
  - careful with associativity of floating-point operations!

- user-defined reductions are possible (version 4.0)
  - need to be declared with `#pragma omp declare reduction`

```
#pragma omp parallel
{
  #pragma omp for reduction(+:x)
  for(int  i = 0; i < 10; ++i) {
    x += i;
  }
}

// or

#pragma omp parallel reduction(-:x)
x -= omp_get_thread_num();
```

Work Sharing Directives

- distribute execution of following code region among existing threads of the team
- must be enclosed in parallel region, cannot be directly nested
- do not launch new threads but assign work to existing threads
- no barrier on entry
- implicit barrier on exit
  - **unless nowait clause specified!**

- `for`
- `sections`
- `single`
- `task`
- `simd`

`sections` Directive

- sections may be executed concurrently, each by an arbitrary thread of the team
- matches MIMD programming patterns
- easily leads to load imbalance if individual sections not equally work-intensive
  - also, maximum degree of parallelism limited by number of sections

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    { ... }
    #pragma omp section
    { ... }
    #pragma omp section
    { ... }
  }
}
```

`single` Directive

- code region will only be executed by a single, arbitrary thread
  - useful for interacting with libraries, that are not multi-threading-aware
- implicit barrier at the end for all threads in the team
- also available as `master` variant
  - like single, but for master thread
  - no implicit barrier at the end

```
#pragma omp parallel
{
  #pragma omp single
  {
    ...
  }
}
```

Synchronization Directives

- constructs discussed so far are fairly high-level in terms of synchronization
  - e.g. unable to enforce specific order or simply wait for all threads
- OpenMP also offers more fine-grained synchronization directives

- `barrier`
- `critical`
- `atomic`
- `ordered`
- `flush`

`barrier` Directive

- explicit barrier requested by user

- threads are not allowed to continue until all have reached the barrier

- Implicit barrier at the end of `for`, `sections`, `single`, `task`, `simd` unless `nowait` specified

```
#pragma omp parallel
{
   ...
   #pragma omp barrier
   ...
}
```

`critical` Directive

code region executed by all threads but only one at a time

can be named to allow finer-grained mutual exclusion

- only critical regions with the same name enforce mutual exclusion
- all regions without a name have the same name

```
#pragma omp parallel
{
  #pragma omp critical(foo)
  { ... }
  #pragma omp critical(foo)
  { ... }
  #pragma omp critical(bar)
  { ... }
  #pragma omp critical
  { ... }
  #pragma omp critical
  { ... }
}
```

`atomic` Directive

- same as `critical`, but restricted to a single memory location and certain operations
- restriction allows mapping to fast hardware mechanisms
  - optional specialization clauses `read`, `write`, `update`, and `capture`
- keeps code hardware- and compiler-independent compared to using intrinsics
  - but may just be a wrapper for `critical` e.g. when lacking hardware support

```
int count = 0;
#pragma omp parallel
{
    #pragma omp atomic
    count++;
}
```

ordered Directive + Clause

- enforces critical region with sequential execution order
- required for strong sequential equivalence
  - but at high performance cost
- only efficient if code outside the ordered region is expensive enough
  - remember guidelines about strong / weak sequential equivalence
  - check whether your numerical method really needs strong sequential equivalence

```
double total = 0.0;
#pragma omp parallel for ordered
for(int i = 0; i < N; ++i) {
    double part = f(i);
    #pragma omp ordered
    total += part;
}
```

`flush` Directive

problem: writes by one thread are not immediately visible to another in the same parallel region without synchronization

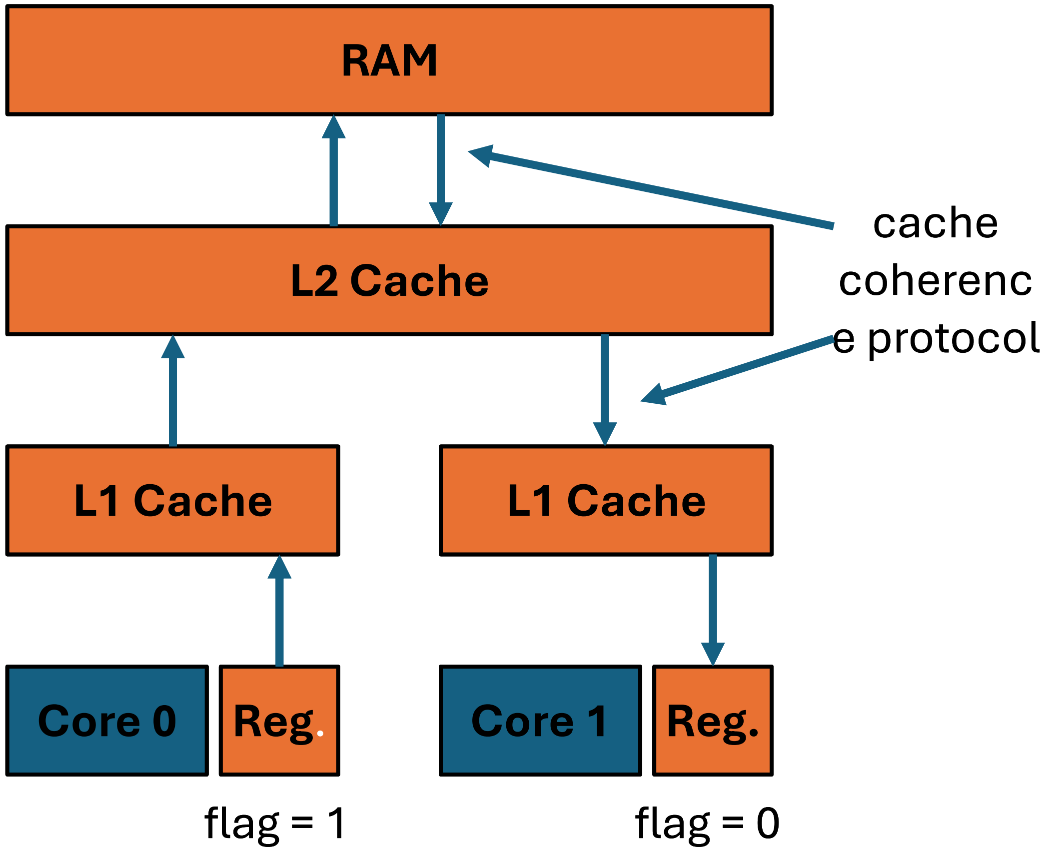
Example:

the second section could hang in while the loop

- But why?

```
#pragma omp parallel sections
{
    #pragma omp section
    { // producer section
        // produce some data
        flag = 1;
    }
    #pragma omp section
    { // consumer section
        while (flag == 0) { }
        // use data
    }
}
```

Detour: Register Spilling (or Lack Thereof)




```
#pragma omp parallel sections
{
    #pragma omp section
    { // producer section
        // produce some data
        flag = 1;
    }
    #pragma omp section
    { // consumer section
        while (flag == 0) { }
        // use data
    }
}
```

`flush` Directive cont'd

```c
#pragma omp parallel sections
{
  #pragma omp section
  { // producer section
    // produce and flush data
    #pragma omp flush
    #pragma omp atomic write
    flag = 1;
    #pragma omp flush(flag)
  }
```

```c
  #pragma omp section
  { // consumer section
    while (1) {
      #pragma omp flush(flag)
      #pragma omp atomic read
      int temp_flag = flag;
      if(temp_flag == 1) break;
    }
    // use data
  }
} // end sections
```

`flush` Directive cont'd

- `flush` provides a consistent view of the data across threads

- it is implied at
  - `barrier`
  - entry and exit of `critical`
  - exit of `parallel`, `for`, `sections`, `single`
  - set/unset of locks

- if otherwise required, use `flush` directive explicitly but with care (performance impact!)

Summary

- main characteristics
  - incremental parallelization

- programming, execution and memory models
  - based on threads and shared data access
  - mainly relies on pragmas as programmer interface

- directives
  - parallelism, work sharing, data sharing, synchronization

- reference material
  - "Parallel Programming for Science and Engineering" by Victor Eijkhout, https://web.corral.tacc.utexas.edu/CompEdu/pdf/pcse/EijkhoutParallelProgramming.pdf