Fundementals of Parallel Programming
OpenMP Section

HPC 101 - 2024 Summer

Lin Xi, Li Chenxiao @ ZJUSCT

July 8, 2024

Outline

1. Introduction

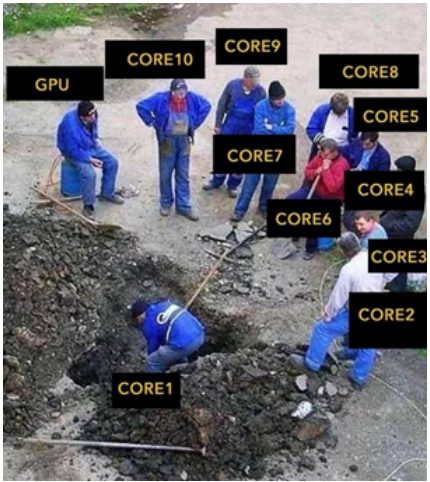
2. OpenMP directives and constructs

3. Shared Data and Data Hazards

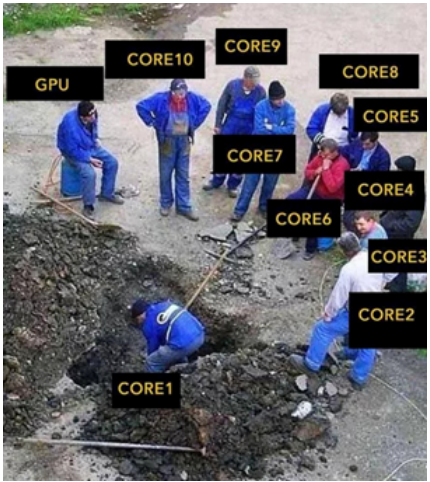
4. Miscellaneous

Introduction

🤔 Parallel Programming

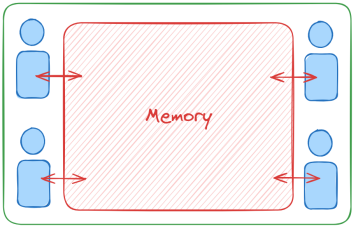


🤔 Parallel Programming



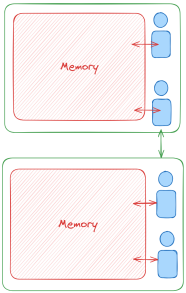
Shared Memory Parallel Model

UMA



Uniform memory access

NUMA



Non-uniform memory access

In real world

OpenMP

OpenMP (Open Multi-Processing) is an API that supports multi-platform shared-memory multi-processing programming in **C, C++, and Fortran**.
It provides a set of compiler directives, library routines, and environment variables that allow developers to specify parallel regions, tasks, and other parallelism constructs.

💡 **OpenMP provides us an easy way to transform serial programs into parallel.**

Example 1: Hello OpenMP

```c
#include <stdio.h>
#include <omp.h>
int main() {
  printf("Welcome to OpenMP!\n");
  #pragma omp parallel
  {
    int ID = omp_get_thread_num();
    printf("hello(%d)", ID);
    printf("world(%d)\n", ID);
  }
  printf("Bye!");
  return 0;
}
```

```
export OMP_NUM_THREADS=4
```

Output:

```
lcx@M602:~/openmp-examples$ ./1_hello_openmp
Welcome to OpenMP program!
hello (2)hello (1)hello (3)world (2)
world (3)
world (1)
hello (0)world (0)
Bye!
lcx@M602:~/openmp-examples$ ./1_hello_openmp
Welcome to OpenMP program!
hello (0)world (0)
hello (2)world (2)
hello (3)world (3)
hello (1)world (1)
Bye!
```

```
$ gcc -o hello_omp hello_omp.c -fopenmp # <-- Compiler Option
```

Example 1: Hello OpenMP

```c
#include <stdio.h>
#include <omp.h>
int main() {
  printf("Welcome to OpenMP!\n");
  #pragma omp parallel
  {
    int ID = omp_get_thread_num();
    printf("hello(%d)", ID);
    printf("world(%d)\n", ID);
  }
  printf("Bye!");
  return 0;
}
```

Differences:

- Import OpenMP Header
- Preprocessing directive
- Parallel Region

Example 1: Hello OpenMP

```c
#include <stdio.h>
#include <omp.h>
int main() {
  printf("Welcome to OpenMP!\n");
  #pragma omp parallel
  {
    int ID = omp_get_thread_num();
    printf("hello(%d)", ID);
    printf("world(%d)\n", ID);
  }
  printf("Bye!");
  return 0;
}
```

Differences:

- Import OpenMP Header
- Preprocessing directive
  - Will cover commonly used directives
- Parallel Region

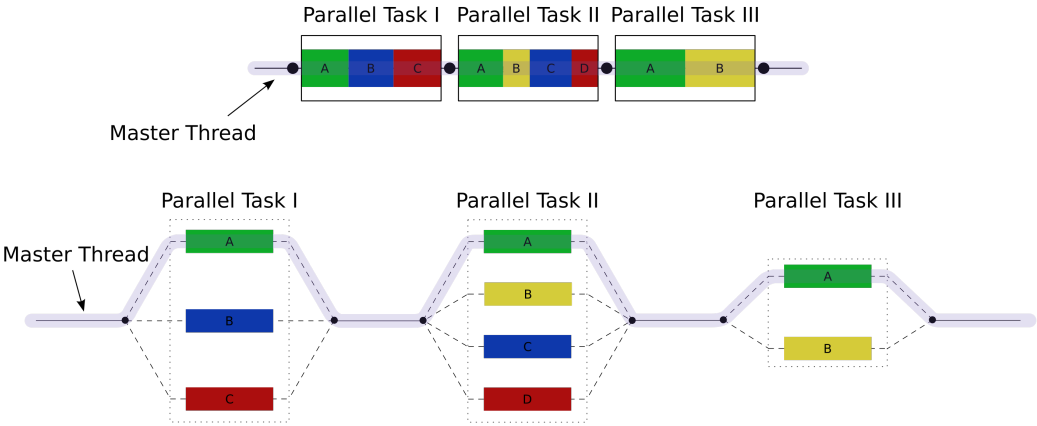
Example 1: Hello OpenMP

```c
#include <stdio.h>
#include <omp.h>
int main() {
  printf("Welcome to OpenMP!\n");
  #pragma omp parallel
  {
    int ID = omp_get_thread_num();
    printf("hello(%d)", ID);
    printf("world(%d)\n", ID);
  }
  printf("Bye!");
  return 0;
}
```

Differences:

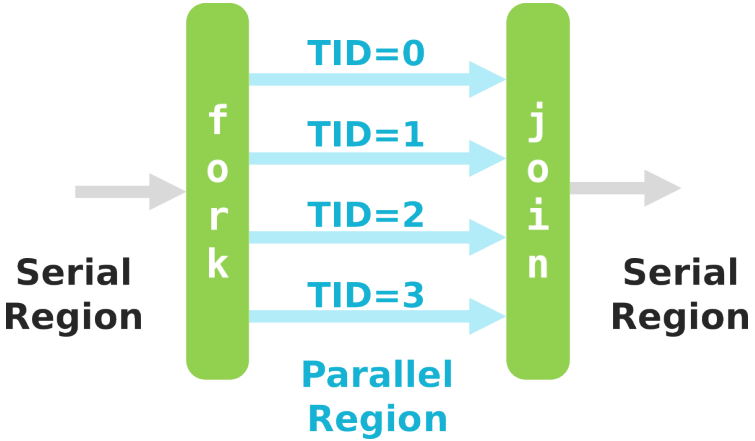
- Import OpenMP Header
- Preprocessing directive
  - Will cover commonly used directives
- **Parallel Region**
  - Relates to the **fork-join** model

Fork-Join Model



Fork-Join Model

Thread ID: `omp_get_thread_num()`



OpenMP directives and constructs

Overview for This Section

- How is work executed?
  - *parallel* directive
- How is work distributed to threads?
  - *single* directive
  - *section* directive
  - *for* directive
    - *for* loop schedule
    - Nested *for* loop
- In next section...
  - directive: *critical*, *atomic*, *barrier*
  - clause: *private*, *shared*, *reduction*

OpenMP Directives

A legal OpenMP Directive must has the following format (C/C++):

Pragma	Directive	[clause[[,]clause] ...]
#pragma omp	parallel, atomic, critical, ...	0 to many

- 🌰 **For example:**

```
#pragma omp parallel for collapse(2) private(tmp_v, d, v)
```

- Case sensitive
- Affects the block (single statement or wrapped by **{}**) after this directive
- 🤪 Here's an official **Cheat Sheet**

Constructs

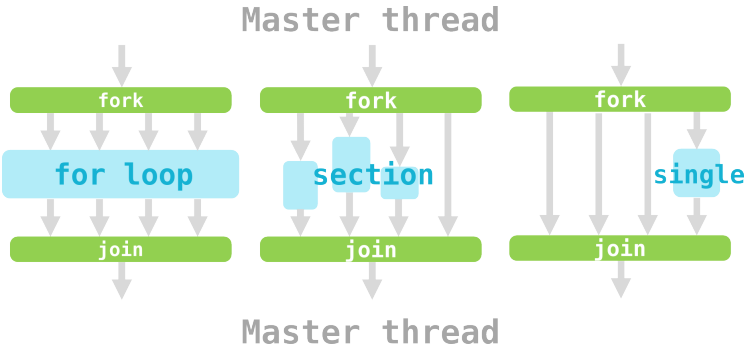
🤔 What is the difference between **construct** and **directive**?

🤓 An OpenMP construct is a formation for which the directive is executable.[1]

```
#pragma omp parallel    // <--\--- Directive
{                       //     |
    printf("Do sth.");  //     | Construct
}                       // ---/
```

[1] https://www.openmp.org/spec-html/5.2/openmpse14.html

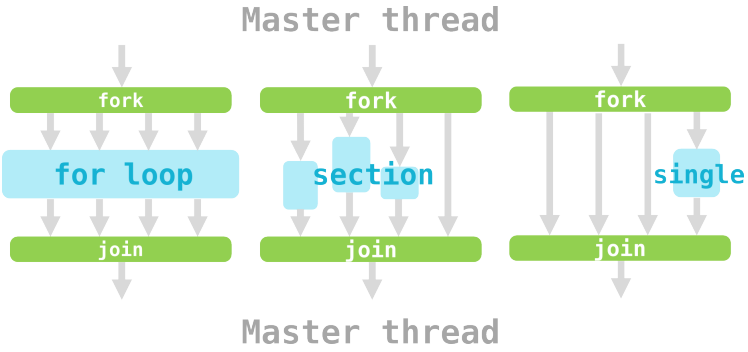
Work-distribution constructs



Work-distribution constructs:

- single
- section
- for

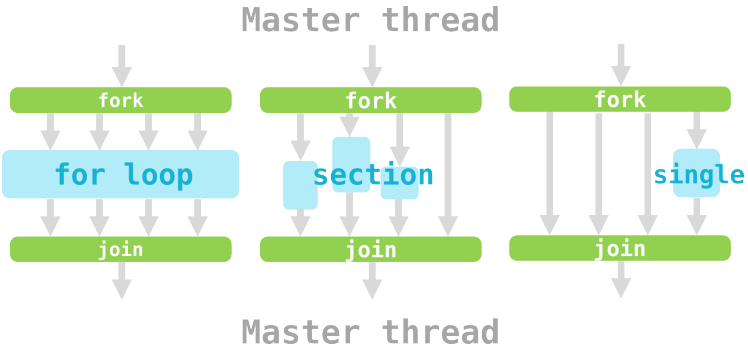
Work-distribution constructs



Work-distribution constructs:

- single
- **section**
- for

Work-distribution constructs



Work-distribution constructs:

- single
- section
- for

parallel Directive

```c
#include <stdio.h>
#include <omp.h>
int main() {
  printf("Welcome to OpenMP!\n");
  #pragma omp parallel
  {
    int ID = omp_get_thread_num();
    printf("hello(%d)", ID);
    printf("world(%d)\n", ID);
  }
  printf("Bye!");
  return 0;
}
```

Example 2: *parallel for* Directive

```
// Addition of two vectors
for (int i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
}
```

```
// Addition of two vectors
#pragma omp parallel for
for (int i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
}
```

```
lcx@M602:~/openmp-examples$ echo $OMP_NUM_THREADS
4
lcx@M602:~/openmp-examples$ ./2_vector_addition
Serial: 1290.71 us
Parallel: 419.164 us
Speed Up: 3.07926x
```

Example 2: *parallel for* Directive

```
// Addition of two vectors
for (int i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
}
```

```
// Addition of two vectors
#pragma omp parallel for
for (int i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
}
```

```
lcx@M602:~/openmp-examples$ echo $OMP_NUM_THREADS
4
lcx@M602:~/openmp-examples$ ./2_vector_addition
Serial: 1290.71 us
Parallel: 419.164 us
Speed Up: 3.07926x
```

🤯 Not 4x speed up

Example 2: *parallel for* Directive

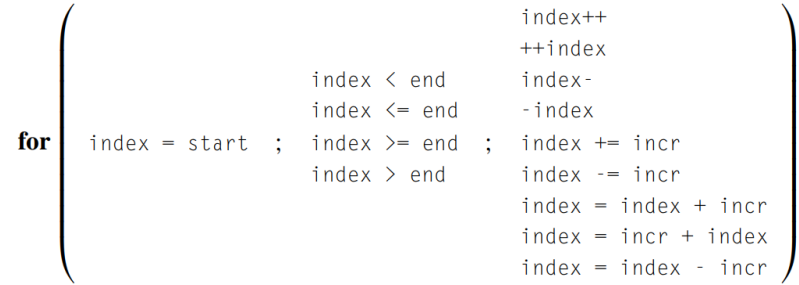
```
// Addition of two vectors
for (int i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
}
```

```
// Addition of two vectors
#pragma omp parallel for
for (int i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
}
```

🤗 **Overhead**: any combination of excess or indirect computation time, memory, bandwidth, or other resources that are required to perform a specific task.

Combined Constructs and Directives (Cont.)

Valid *for* loops that can be parallelized by OpenMP:



Loop Schedule

🤔 Think about this...

```
#pragma omp parallel for
for (int i = 0; i < N; i++) {
    c[i] = f(i); // What if f is not O(1), eg. O(N)
}
```

Loop Schedule

🤔 Think about this...

```
#pragma omp parallel for
for (int i = 0; i < N; i++) {
    c[i] = f(i); // What if f is not O(1), eg. O(N)
}
```

Workload is unbalanced!

Loop Schedule

🤔 Think about this...

```
#pragma omp parallel for
for (int i = 0; i < N; i++) {
    c[i] = f(i); // What if f is not O(1), eg. O(N)
}
```

Workload is unbalanced!

- *schedule* clause specifies how iterations of associatedloops are divided into contiguous non-empty subsets, aka. **chunks**.

Loop Schedule

🤔 Think about this...

```
#pragma omp parallel for
for (int i = 0; i < N; i++) {
    c[i] = f(i); // What if f is not O(1), eg. O(N)
}
```

Workload is unbalanced!

- *schedule* clause specifies how iterations of associatedloops are divided into contiguous non-empty subsets, aka. **chunks**.
- Type: Static, Dynamic, Guided, Runtime, Auto

Loop Schedule - Static

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < N; i++) {
    c[i] = f(i);
}
```

Is this a good choice?

- 👍 Pros: Less overhead in scheduling
- 👎 Cons: Workload may be unbalanced

Loop Schedule - Dynamic

```
#pragma omp parallel for schedule(dynamic, 2)
for (int i = 0; i < N; i++) {
    c[i] = f(i);
}
```

👍 Pros: More flexible scheduling

👎 Cons: More overhead in scheduling

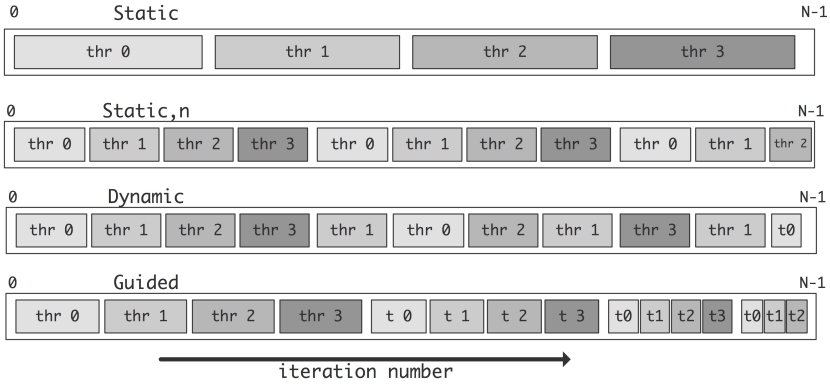
Loop Schedule - Guided

```
#pragma omp parallel for schedule(guided, 2)
for (int i = 0; i < N; i++) {
    c[i] = f(i);
}
```

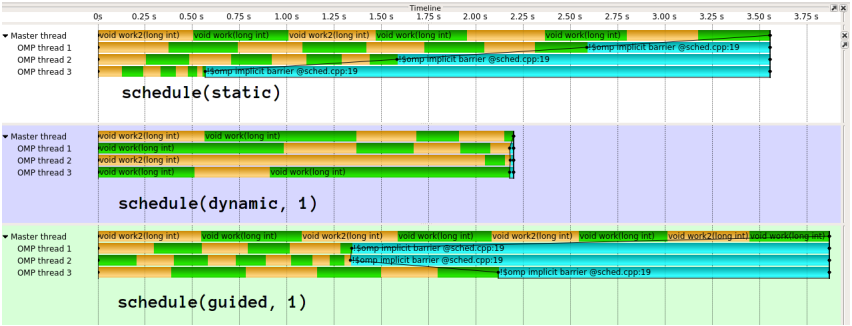
- Compare to *static*: *May* yield a better workload balance under certain circumstances
- Compare to *dynamic*: Less overhead to dispatch tasks
- In practice: Just try if it works better

Example 3: Let's try out!

Source code at: `3_schedule.c`

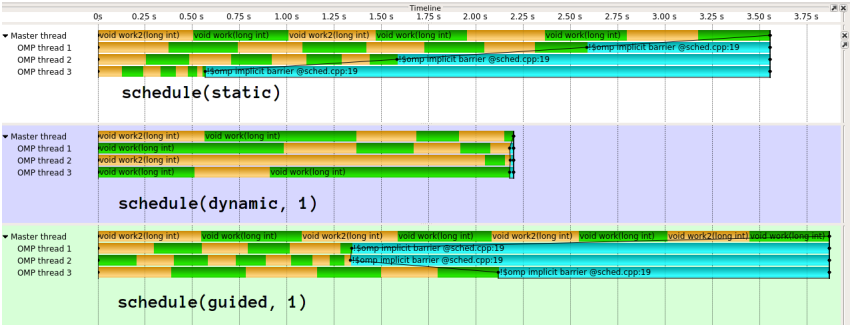


Loop Schedule



Static, Dynamic, Guided, Runtime, Auto

Loop Schedule



Find optimal schedule? 😖 NP-complete!

Nested *for* Loop

In some cases, we will encounter nested *for* loops. For example, matrix addition:

```
// Matrix Element-wise Addition
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        c[i][j] = a[i][j] + b[i][j];
    }
}
```

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        c[i][j] = a[i][j] + b[i][j];
    }
}
```

Nested *for* Loop

In some cases, we will encounter nested *for* loops. For example, matrix addition:

```
// Matrix Element-wise Addition
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        c[i][j] = a[i][j] + b[i][j];
    }
}
```

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        c[i][j] = a[i][j] + b[i][j];
    }
}
```

It's not always a good idea to parallelize nested loops.

Nested *for* Loop

In some cases, we will encounter nested *for* loops. For example, matrix addition:

```
// Matrix Element-wise Addition
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        c[i][j] = a[i][j] + b[i][j];
    }
}
```

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        c[i][j] = a[i][j] + b[i][j];
    }
}
```

It's not always a good idea to parallelize nested loops.

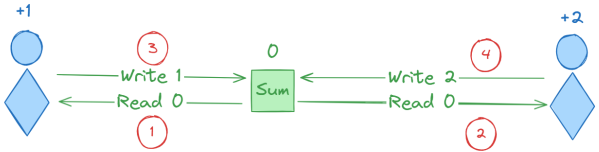
Think about locality and data dependency before you use *collapse* clause.

Shared Data and Data Hazards

Example: Data Hazards in Summation

```c
#include <stdio.h>
#include <omp.h>
int main() {
    int a[100];
    int sum = 0;
    // initialize
    for (int i = 0; i < 100; i++) a[i] = i + 1;
    // Sum up from 1 to 100
#pragma omp parallel for
    for (int i = 0; i <= 99; i++) {
        sum += a[i];
    }
    printf("Sum = %d\n", sum);
}
```

How Data Hazards Happen?



Thread 1	Thread 2
Read sum	
	Read sum
Write sum	
	Write sum

This case is especially important, you will encounter this again and again in future study, eg. Computer Organization and Database System.

Scope and Data Hazard

- Shared & private data in default
- Explicit scopes definition
  - *private*
  - *shared*
  - *firstprivate*
  - *lastprivate*
- Data hazards happen when operating shared data

```
    int sum = 0;
    // Sum up from 1 to 100
#pragma omp parallel for
    for (int i = 0; i <= 99; i++) {
        sum += a[i];
    }
```

Resolve Data Hazard

- Critical Section
- Atomic Operations
- Reduction

Example: Solution with Critical Section

- Only one thread can enter critical section at the same time.
- A critical section can contain multiple statements.

```
#pragma omp parallel for
    for (int i = 0; i <= 99; i++) {
#pragma omp critical
        { sum += a[i]; }
    }
    printf("Sum = %d\n", sum);
```

Example: Solution with Critical Section

- Only one thread can enter critical section at the same time.
- A critical section can contain multiple statements.
- 🤯 Isn't this serial again?

```
#pragma omp parallel for
    for (int i = 0; i <= 99; i++) {
#pragma omp critical
        { sum += a[i]; }
    }
    printf("Sum = %d\n", sum);
```

Example: Solution with Atomic Operation

- Atomic operation cannot be separated.
- Only can be applied to one operation
- Limited set of operators supported

```
#pragma omp parallel for
    for (int i = 0; i <= 99; i++) {
#pragma omp atomic
        sum += a[i];
    }
    printf("Sum = %d\n", sum);
```

Example: Solution with Atomic Operation

- Atomic operation cannot be separated.
- Only can be applied to one operation
- Limited set of operators supported

C/C++ statement:

if *atomic clause* **is...**	*statement*:
read	*v* = *x*;
write	*x* = *expr*;
update	*x*++; *x*--; ++*x*; --*x*; *x binop* = *expr*; *x* = *x binop expr*; *x* = *expr binop x*;
compare is present	*cond-expr-stmt:* *x* = *expr ordop x* ? *expr* : *x*; *x* = *x ordop expr* ? *expr* : *x*; *x* = *x* == *e* ? *d* : *x*; *cond-update-stmt:* **if**(*expr ordop x*) { *x* = *expr*; } **if**(*x ordop expr*) { *x* = *expr*; } **if**(*x* == *e*) { *x* = *d*; }
capture is present	*v* = *expr-stmt* { *v* = *x*; *expr-stmt* } { *expr-stmt v* = *x*; } (*expr-stmt*: *write-expr-stmt*, *update-expr-stmt*, or *cond-expr-stmt*.)
both **compare** and **capture** are present	{ *v* = *x*; *cond-update-stmt* } { *cond-update-stmt v* = *x*; } **if**(*x* == *e*) { *x* = *d*; } **else** { *v* = *x*; } { *r* = *x* == *e*; **if**(*r*) { *x* = *d*; } } { *r* = *x* == *e*; **if**(*r*) { *x* = *d*; } **else** { *v* = *x*; } }

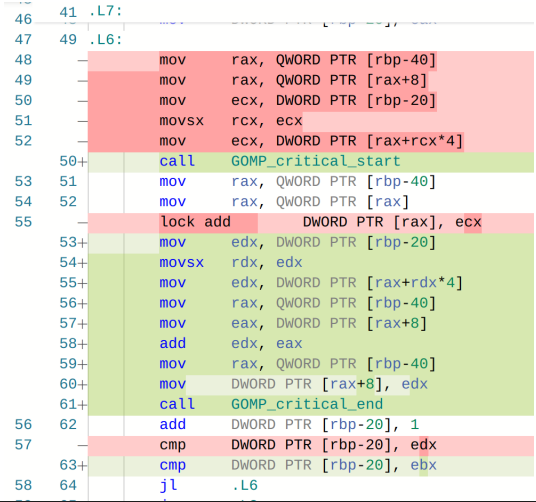
Example: Solution with Reduction

- Create temporary private variables for each thread
- Reduce these private variables in the end
- Limited set of operators supported

```
#pragma omp parallel for reduction(+:sum)
    for (int i = 0; i <= 99; i++) {
        sum += a[i];
    }
    printf("Sum = %d\n", sum);
```

Comparison

- **Critical Region**: Based on locking
- **Atomic Operation**: Based on hardware atomic operations
- **Reduction**: only synchronize in the end



Another Example: Naïve GEMM

```
//General Matrix Multiplication (GEMM)
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        for (int k = 0; k < N; k++) {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
```

Miscellaneous

Threads Synchronize

- ⛔ **Barrier**: Wait until all thread reach here
  - Implicit barrier in parallel region
  - *nowait* clause
- 🔒 **Locking**: wait until obtain the lock
  - Often apply to data structures

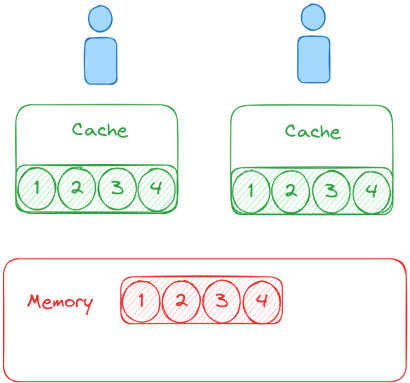
Nested Parallel Region

- Disabled by default.
- Use *omp_set_nested* to enable.
- Consider refactor the code.

```
int f(int n, int* a, int* b, int* c) {
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        c[i] = a[i] + b[i];
    }
}
int main(){
    ...
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        f(n[i], a[i], b[i], c[i]);
    }
}
```

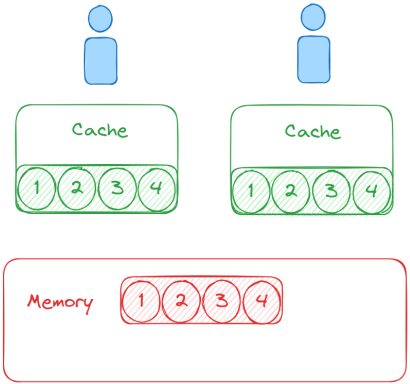
False Sharing

- Supposed to be private, but actually perform like sharing
- Seems working independently, but actually sharing the same cache line
- Does harm to parallel performance



False Sharing

- Supposed to be private, but actually perform like sharing
- Seems working independently, but actually sharing the same cache line
- Does harm to parallel performance



Be cache-friendly and avoid false sharing: Make use of space locality

Summary: How to Optimize a program with OpenMP

1. **Where to parallelize**: Profiling
2. **Whether to parallelize**: Analyze data dependency
3. **How to parallelize**: Analysis and Skills
   - Sub-task Distribution
   - Scheduling Strategy
   - Cache and Locality
   - Hardware Environment
   - Somtimes: transform recursion to iteration
4. **Get Down to Work**: Testing

Tips

1. Ensure correctness while parallelizing
2. Be aware of overhead
3. Check more details in official documents
    - For example, OpenMP on GPU.

References

- An introduction to parallel programming - Peter S. Pacheco, Matthew Malensek
- SC17 - Loop Schedule for OMP

Thank You

Let's have a break!