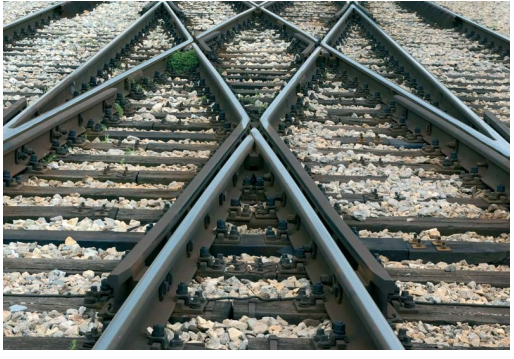
Parallel programming

OpenMP

Industrial Informatics Department
Czech Institute of Informatics Robotics and Cybernetics
Czech Technical University in Prague



Outline

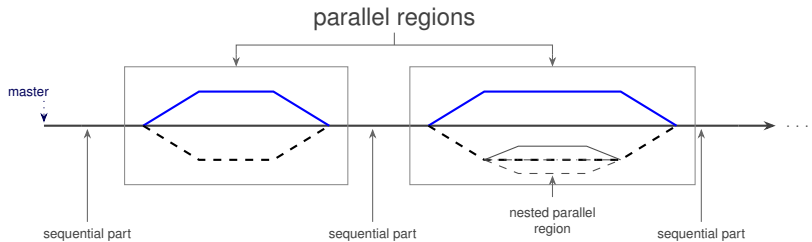
1. OpenMP basics: parallel regions, sections, synchronization
2. False sharing
3. Work sharing: parallel for and its clauses
4. Atomic operations
5. Scheduling of work
6. Tasks

Introduction to OpenMP

- OpenMP (Open Multi-Processing) provides constructs for parallel programming in C++, C, and Fortran on Linux, MacOS, and Windows.
- A sequential code is transformed to a parallel one by adding compiler pragmas, so if a compiler does not support OpenMP, the pragmas are skipped and the output is a sequential program.
  - **OpenMP manual: 1.3 Execution model:** *more detailed (first paragraph)*
- OpenMP is widely used in software like Blender, fftw, OpenBLAS (MATLAB, Python libraries – e.g. NumPy or Scipy, R), and eigen to accelerate computations.
- It is easy to use!

Execution model

- OpenMP program starts with a single thread only (*master thread*).
- It is executed sequentially until it reaches a **parallel region** defined by OpenMP pragma.
- At the entry of parallel region, new **team of threads is created**. Each thread executes concurrently with the others sharing the work.



Using OpenMP

- Include header file

```
#include <omp.h>
```

- Cmake (multi-platform)

```
find_package(OpenMP)
if (OPENMP_FOUND)
    set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} ${OpenMP_C_FLAGS}")
    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${OpenMP_CXX_FLAGS}")
    set(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} ${OpenMP_EXE_LINKER_FLAGS}")
endif()
```

- gcc

```
g++ -fopenmp ...
```

Hello world! In OpenMP...

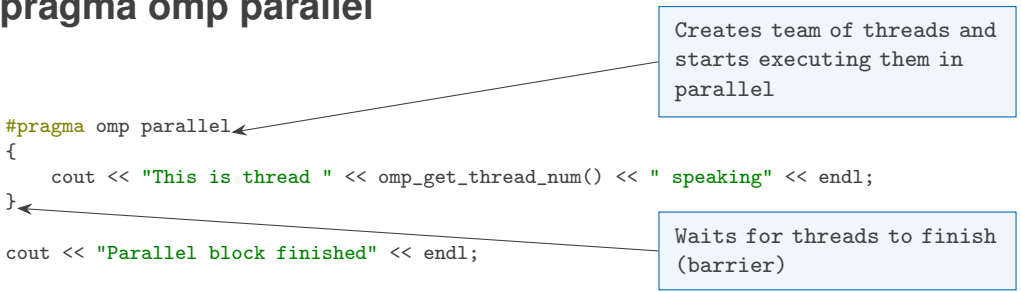
`lab_codes/src/HelloWorld.cpp`

- Demonstrated task:
  1. Vector multiplied by scalar value
  2. Sum of vector components

Runtime Library Routines

- `omp_get_num_procs()`
  Returns the number of processors that are available to the program
- `omp_get_num_threads()`
  Returns the number of threads that are currently in the team executing the parallel region from which it is called
- `omp_get_thread_num()`
  Returns the calling thread index within the current team
- ...

#pragma omp parallel



Output:

```
This is thread 0 speaking
This is thread 3 speaking
This is thread 2 speaking
This is thread 1 speaking
Parallel block finished
```

#pragma omp parallel

```
#pragma omp parallel num_threads(8)
{
    cout << "This is thread " << omp_get_thread_num() << " speaking" << endl;
}
```

Creates team of 8 threads

Output:

```
This is thread 0 speaking
This is thread 3 speaking
This is thread 6 speaking
This is thread 1 speaking
This is thread 2 speaking
This is thread 7 speaking
This is thread 4 speaking
This is thread 5 speaking
```

#pragma omp single

```
#pragma omp parallel
{
    cout << "This is thread " << omp_get_thread_num() << " speaking" << endl;

    #pragma omp single
    {
        cout << "The single part was done by thread " << omp_get_thread_num() << endl;
    }
}
```

Block performed by single thread

Output:

```
This is thread 3 speaking
The single part was done by thread 3
This is thread 1 speaking
This is thread 2 speaking
This is thread 0 speaking
```

Later used to let **one** thread generate tasks for the whole team (slide 31).

#pragma omp sections

Each section is performed by only one thread

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            cout << "section 1, first: " << omp_get_thread_num() << endl;
            cout << "section 1, second: " << omp_get_thread_num() << endl;
        }
        #pragma omp section
        {
            cout << "section 2, first: " << omp_get_thread_num() << endl;
            cout << "section 2, second: " << omp_get_thread_num() << endl;
        }
    }
}
```

Output:

```
section 2, first:  0
```
section 2, second: 0
section 1, first: 1
section 1, second: 1

Waits for threads to finish (barrier). Can be changed by `#pragma omp sections nowait`

#pragma omp critical

Critical region, performed by all threads but not at once (mutual exclusion)

```
int sum;

#pragma omp parallel
{
    #pragma omp critical
    {
        cout << "Thread " << omp_get_thread_num() << " in critical region" << endl;
        sum += omp_get_thread_num();
    }
}

cout << sum << endl;
```

Output:

```
Thread 1 in critical region
Thread 0 in critical region
Thread 3 in critical region
Thread 2 in critical region
6
```

#pragma omp barrier

Threads in team wait on the barrier

```
#pragma omp parallel
{
    cout << "Before barrier thread " << omp_get_thread_num() << endl;
    #pragma omp barrier
    cout << "After barrier thread " << omp_get_thread_num() << endl;
}
```

Output:

```
Before barrier thread 0
Before barrier thread 3
Before barrier thread 1
Before barrier thread 2
After barrier thread 1
After barrier thread 2
After barrier thread 0
After barrier thread 3
```

Example

- Write a function for computing vector normalization. Split the vector into two halves, each one is processed a separate section.
  - You may use skeleton `lab_codes/src/VectorNormalization.cpp`

To remind normalization:

$$\|\vec{a}\| = \sqrt{a_1^2 + a_2^2 + \cdots + a_n^2};\ \vec{a} \in \mathbb{R}^n$$

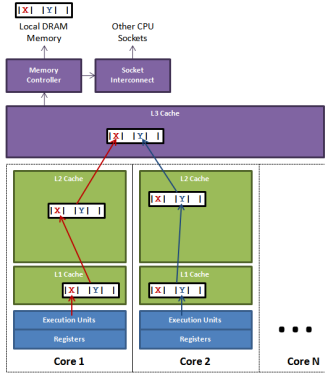
$$\vec{n} = \frac{\vec{a}}{\|\vec{a}\|} = \left(\frac{a_1}{\|\vec{a}\|}; \frac{a_2}{\|\vec{a}\|}; \cdots ; \frac{a_n}{\|\vec{a}\|} \right)$$

Try the example at first

Run example FalseSharing.cpp

What? False sharing

- Cache related problem: each of the two threads modifies own variable that resides in the same cache line $\rightarrow$ modifications from one thread are propagated to another thread by cache line invalidation



source

Fixing false sharing

- Accumulate into a **thread private** variable and write the shared one only once

withFalseSharing

```
double x = 0.0, y = 0.0;

#pragma omp section
for (int i = 0; i < n; i++)
  x += sin(i)*cos(i);

#pragma omp section
for (int i = 0; i < n; i++)
  y += sin(i)*cos(i);
```

x and y sit next to each other, so they share one cache line that is invalidated on every single update.

noFalseSharing

```
double x = 0.0, y = 0.0;

#pragma omp section
{
  double local = 0.0;
  for (int i = 0; i < n; i++)
    local += sin(i)*cos(i);
  x = local;
}
```

The hot accumulator is private, the shared variable is written once per section.

- The `reduction` clause (slide 21) does exactly this for you

False sharing detection

- Can be detected using Intel VTune, only Intel processors
  - Microarchitecture Exploration → Summary → L3 Bound → Contested Accesses
  - (driver for event based sampling must be installed and on some platforms, HyperThreading has to be disabled)



Parallel summing of matrix rows

```
#pragma omp parallel
{
    #pragma omp for
    for (int i = 0; i < matrix.size(); i++) {
        for (int j = 0; j < matrix[i].size(); j++) {
            rowSums[i] += matrix[i][j];
        }
    }
}
```

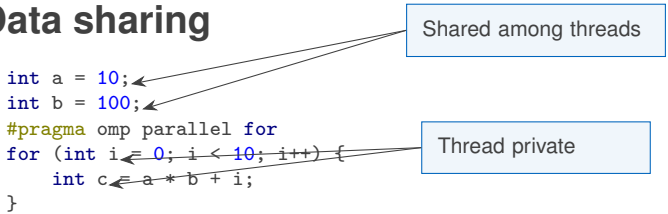
Each iteration of for loop performed by a thread (in parallel) from the team

A shorter code...

```
#pragma omp parallel for
for (int i = 0; i < matrix.size(); i++) {
    for (int j = 0; j < matrix[i].size(); j++) {
        rowSums[i] += matrix[i][j];
    }
}
```

Question: what happens if you write the pragma on the inner loop?

Data sharing

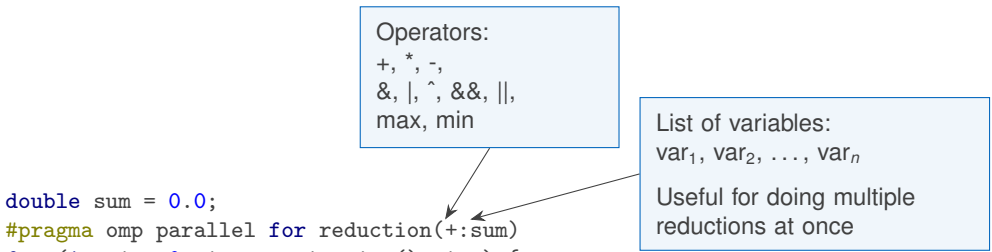


- The sharing can be stated explicitly as a clause
  - `#pragma` omp parallel for private(a, b)
    - Variables a and b are private to each thread (without global initialization)
  - `#pragma` omp parallel for firstprivate(a, b)
    - Variables a and b are private to each thread (with global initialization)
  - `#pragma` omp parallel for shared(a, b)
    - Variables a and b are shared among threads
- The default policy can be set to
  - `#pragma` omp parallel for default(shared)
    - By default, all the variables outside of the parallel section are shared
  - `#pragma` omp parallel for default(none)
    - The programmer must explicitly state the sharing policy of the variables

Reduction

- Parallel aggregation of an expression, e.g., a sum

```
sum = rowSums[0] + rowSums[1] + rowSums[2] + ... + rowSums[matrix.size() - 1];
```



Atomic

- OpenMP critical section (slide 12) is intended for general code to avoid data races
- Atomic operations are more efficient if the critical section can be transformed to atomic hardware instructions

```
int sum = 0;
int currentSum;
#pragma omp atomic read
currentSum = sum;

#pragma omp atomic write
sum = 0;

#pragma omp atomic update
sum += 10;

#pragma omp atomic capture
{
    currentSum = sum;
    sum += 10;
}
```

Updates the value of a variable while capturing the original or final value of the variable atomically

Accumulating into one variable

Synchronize on every iteration

```
#pragma omp parallel for
for (int i = 0; i < n; i++) {
  #pragma omp critical
  sum += u[i] * u[i];
}
```

`...CriticalGlobal`, `...AtomicGlobal`

One lock or atomic per element, and all threads keep touching the same cache line.

Combine once per thread

```
#pragma omp parallel
{
  double local = 0.0;
  #pragma omp for
  for (int i = 0; i < n; i++)
    local += u[i] * u[i];

  #pragma omp atomic update
  sum += local;
}
```

`...CriticalLocal`, `...AtomicLocal`

One synchronization per thread.

. . . or let OpenMP write the second version for you:

```
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < n; i++)
  sum += u[i] * u[i];
```

Example

- Parallel vector normalization in `VectorNormalization.cpp`; implement as many variants as you can and **compare the measured times**
  1. `normalizationParallelForAndReduction` – start here
  2. `normalizationParallelForAndCriticalLocal`
  3. `normalizationParallelForAndAtomicGlobal` and `normalizationParallelForAndAtomicLocal`
- Which variant wins, and why? Compare with `normalizationParallelForAndCriticalGlobal`, which is already implemented

To remind normalization:

$$\|\vec{a}\| = \sqrt{a_1^2 + a_2^2 + \cdots + a_n^2};\ \vec{a} \in \mathbb{R}^n$$

$$\vec{n} = \frac{\vec{a}}{\|\vec{a}\|} = \left(\frac{a_1}{\|\vec{a}\|}; \frac{a_2}{\|\vec{a}\|}; \cdots ; \frac{a_n}{\|\vec{a}\|} \right)$$

Scheduling of work

- Distribution of work among threads in parallel for
  - `#pragma omp parallel for schedule(...)`
    - policy: static, dynamic, guided
- Why? Different scenarios require different policies for efficient execution

Scheduling: static

```
#pragma omp parallel for schedule(static, chunkSize) num_threads(4)
```

- Divides the iterations into `chunkSize` and distributes them in circular order among threads.

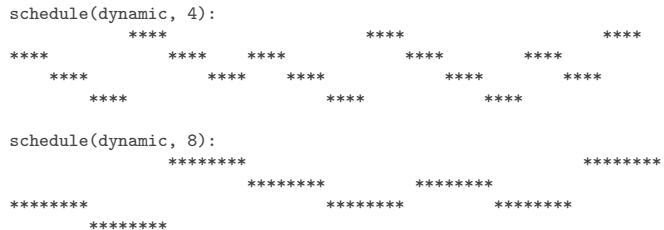
```
schedule(static, 4):
****            ****            ****            ****
    ****            ****            ****            ****
        ****            ****            ****            ****
            ****            ****            ****            ****

schedule(static, 8):
********                        ********
        ********                        ********
                ********                        ********
                        ********                        ********
```

Scheduling: dynamic

```
#pragma omp parallel for schedule(dynamic, chunkSize) num_threads(4)
```

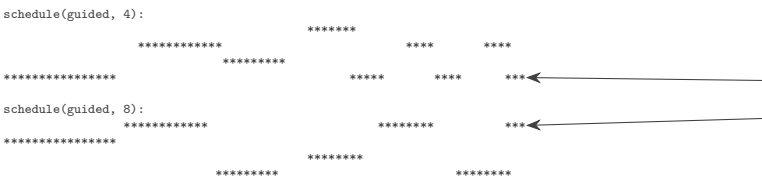
- Divides the iterations into `chunkSize`. Threads execute the chunks as they finish.



Scheduling: guided

```
#pragma omp parallel for schedule(guided, minChunkSize) num_threads(4)
```

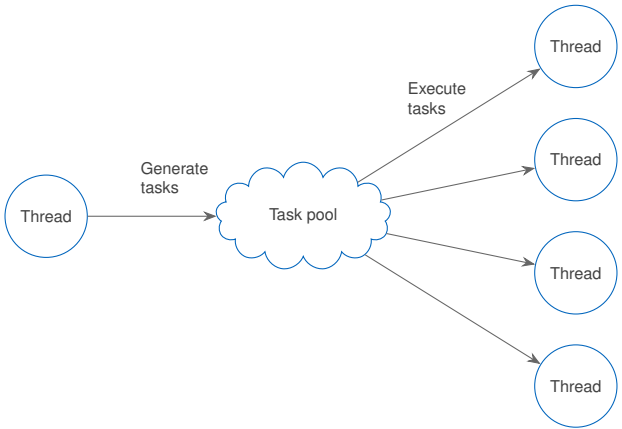
- The size of chunk is proportional to the number of unassigned iterations divided by number of threads. Specifies minimum size of chunk.



Tasks

- A task is block of code to be run in parallel
- When a thread encounters a task construct, it may either run it immediately or defer its execution
- Deferred tasks are added to a task pool, which is processed by all threads in the team

Task pool


Thread
Generate
tasks
Task pool
Execute
tasks
Thread
Thread
Thread
Thread

Task pool

```
#pragma omp parallel
{
    #pragma omp single
    {
        for (int i = 0; i < 10;i++) {
            #pragma omp task
            {
                cout << "Iteration " << i
                     << " processed by thread " << omp_get_thread_num() << endl;
            }
        }
    }
}
```

One thread generates tasks

Generation of task

Iterations processed by all threads in team

Output:

```
Iteration 0 processed by thread 1
Iteration 2 processed by thread 3
Iteration 3 processed by thread 1
Iteration 1 processed by thread 0
Iteration 5 processed by thread 1
Iteration 4 processed by thread 3
Iteration 7 processed by thread 1
Iteration 8 processed by thread 3
Iteration 9 processed by thread 1
Iteration 6 processed by thread 0
```

Tasks vs Sections

- Sections are for static number of parallel regions
- Previous example cannot be reimplemented with sections, code will not compile

```
error:  work-sharing region may not be closely nested inside of work-sharing
```

Tasks synchronization

```
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        cout << "I'm a lonely task outside of task group :(" << endl;
        #pragma omp task
        cout << "I'm a lonely task outside of task group :(" << endl;

        #pragma omp taskwait
        #pragma omp taskgroup
        {
            #pragma omp task
            cout << "I'm a happy task inside of task group :)" << endl;
            #pragma omp task
            cout << "I'm a happy task inside of task group :)" << endl;
        }
    }
}
```

Waits on all tasks generated by the current task from the beginning

Waits until the completion of all enclosed and descendant tasks

Output:

```
I'm a lonely task outside of task group :(
I'm a lonely task outside of task group :(
I'm a happy task inside of task group :)
I'm a happy task inside of task group :)
```

Tasks with dependencies

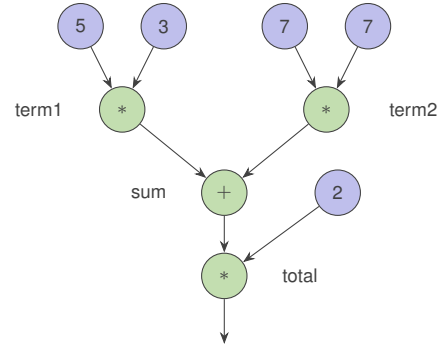
```
cout<< "Evaluating expression: 2*(5*3+7*7)" << endl;
int term1 = 0, term2 = 0, sum = 0, total = 0;
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task depend(out: term1)
        term1 = 5*3;

        #pragma omp task depend(out: term2)
        term2 = 7*7;

        #pragma omp task depend(in: term1, term2) depend(out: sum)
        sum = term1+term2;

        #pragma omp task depend(in: sum)
        total = 2*sum;

        #pragma omp taskwait
        cout << "Final value of the expression: "<< total <<endl;
    }
}
```



Without taskwait the total could be 0!

Output:

```
Evaluating expression:  2*(5*3+7*7)
Final value of the expression:  128
```

Data scoping for tasks

- Similar as for sections, with one notable exception:
  - Variables of orphaned tasks (not directly in parallel section) are **firstprivate by default!**

```
void task(vector<double> &u) {
    #pragma omp task
    cout << "In orphaned task: " << &u << endl;
}

...

cout << "In main thread: " << &u << endl;
#pragma omp parallel num_threads(1)
{
    #pragma omp task
    cout << "In explicit task: " << &u << endl;
}

#pragma omp parallel num_threads(1)
{
    #pragma omp single
    task(u);
}
```

Can be corrected by using shared(u)

Output:

```
In main thread:  0x7fff3fd79650
In explicit task:  0x7fff3fd79650
In orphaned task:  0xc14b50
```

Example

- Implement parallel merge sort using tasks
  - Use the provided skeleton `MergeSort.cpp`

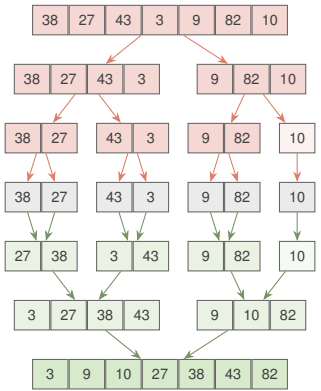


image source

References

- http://jakascorner.com/blog/2016/06/omp-for-scheduling.html
- https://en.wikipedia.org/wiki/Merge_sort