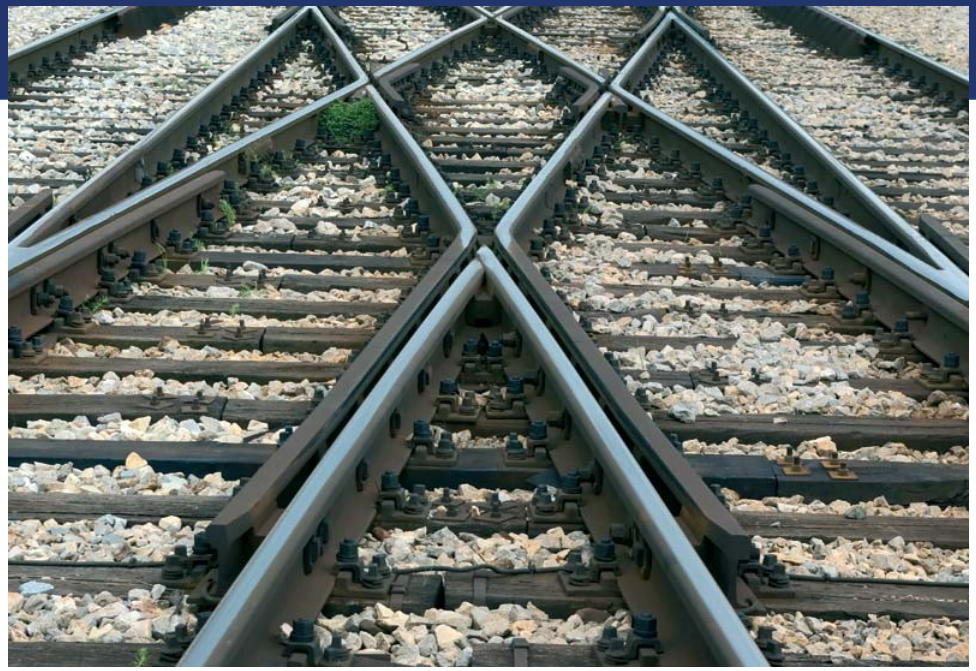
Parallel programming

OpenMP part 2



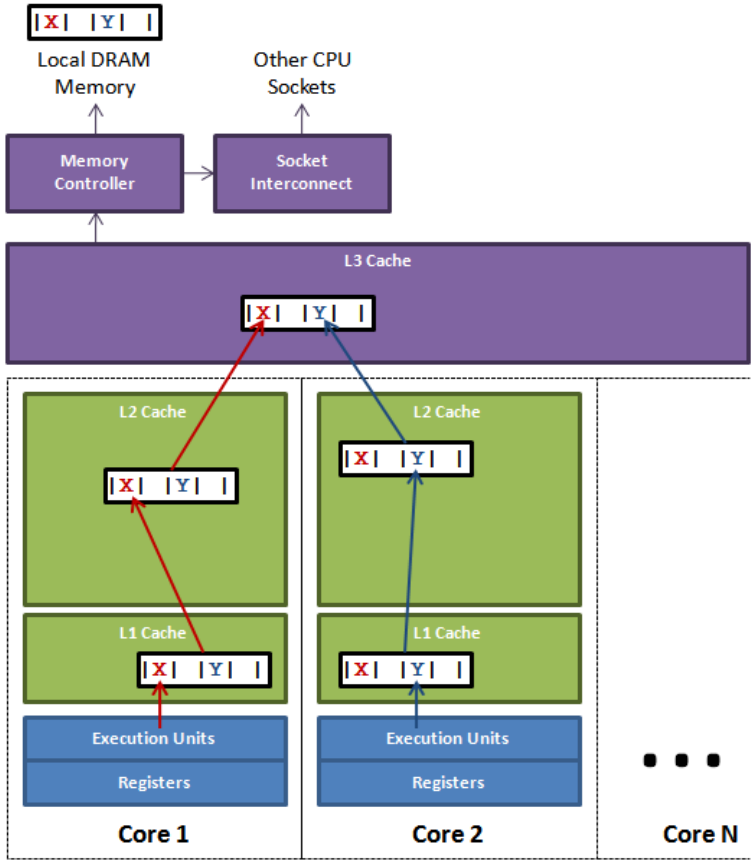
FAKULTA
ELEKTROTECHNICKÁ
ČVUT V PRAZE

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 → modifications from one thread are propagated to another thread by cache line invalidation



source

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)

Atomic

- OpenMP critical section 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

Example

- Write parallel vector normalization using parallel for and atomic operation
  - Implement method `normalizationParallelForAndAtomic` in `VectorNormalization.cpp`

To remind normalization:

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

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

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.

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


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

Scheduling: guided

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

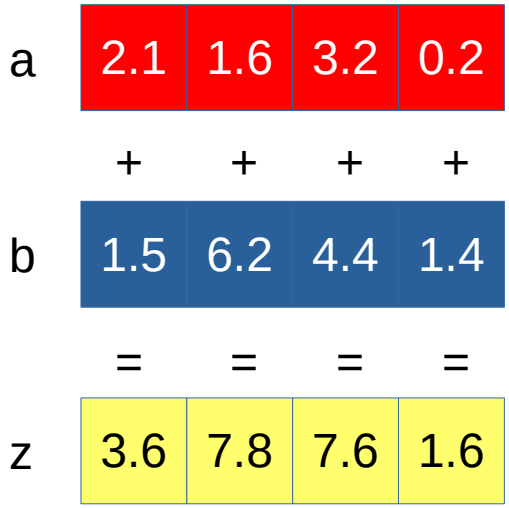
- The size of chunk is proportional to the number of unassigned iterations divided by number of threads. Specifies minimum size of chunk.

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

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

Last thread can process smaller chunk than minChunkSize

Vectorization

- Performs the **S**ame **I**nstruction on **M**ultiple **D**ata - SIMD
- Special vector instructions and registers




```
#include <immintrin.h>

__m256d a;
__m256d b;
__m256d z = _mm256_add_pd(a, b);
```

Vectorization using intrinsic functions

- History
  - MMX (1997): 64b registers (e.g., 2x32b ints), only integers
  - SSE (1998): 128b registers, supports floats
    - SSE2 (2000): supports doubles
  - AVX (2011): 256b registers

Vectorization in OpenMP

- Use **simd** directive

```
#pragma omp simd
for (int i = 0; i < size; i++) {
    w[i] = u[i] * v[i];
}
```

Loop is split into chunks that fit into SIMD register. No parallelization.

Preferably, use constants instead of function calls for iteration bounds in SIMD loops.

```
#pragma omp parallel for simd
for (int i = 0; i < size; i++) {
    w[i] = u[i] * v[i];
}
```

Distribute iterations across team of threads. Iteration of one thread are then split into chunks that fit into SIMD register.

HELP! Code is not vectorized

- Sometimes, compilers need some guidance for successful vectorization
  - Number of loop iterations known before loop execution
  - No ***break***
  - ***if*** statement only as masked assignment
  - No function calls (basic math are allowed)
  - Avoid loop dependencies (e.g., Read-After-Write is not vectorizable)
  - Unit strides are recommended (informally: iterate on the rightmost index for multidimensional arrays)
  - More information
    - https://www.cac.cornell.edu/education/training/StampedeJan2017/Stampede2-VectorizationOnKNL.pdf
    - https://easyperf.net/blog/2017/11/10/Tips_for_writing_vectorizable_code
- For g++
  - Why vectorization did not occur? Use flag **-fopt-info-vec-missed**
  - Check whether vectorization occured

```
>> g++ -fopenmp -fopt-info-vec -O3 VectorNormalization.cpp
VectorNormalization.cpp:125:26: note: loop vectorized
VectorNormalization.cpp:131:12: note: loop vectorized
```

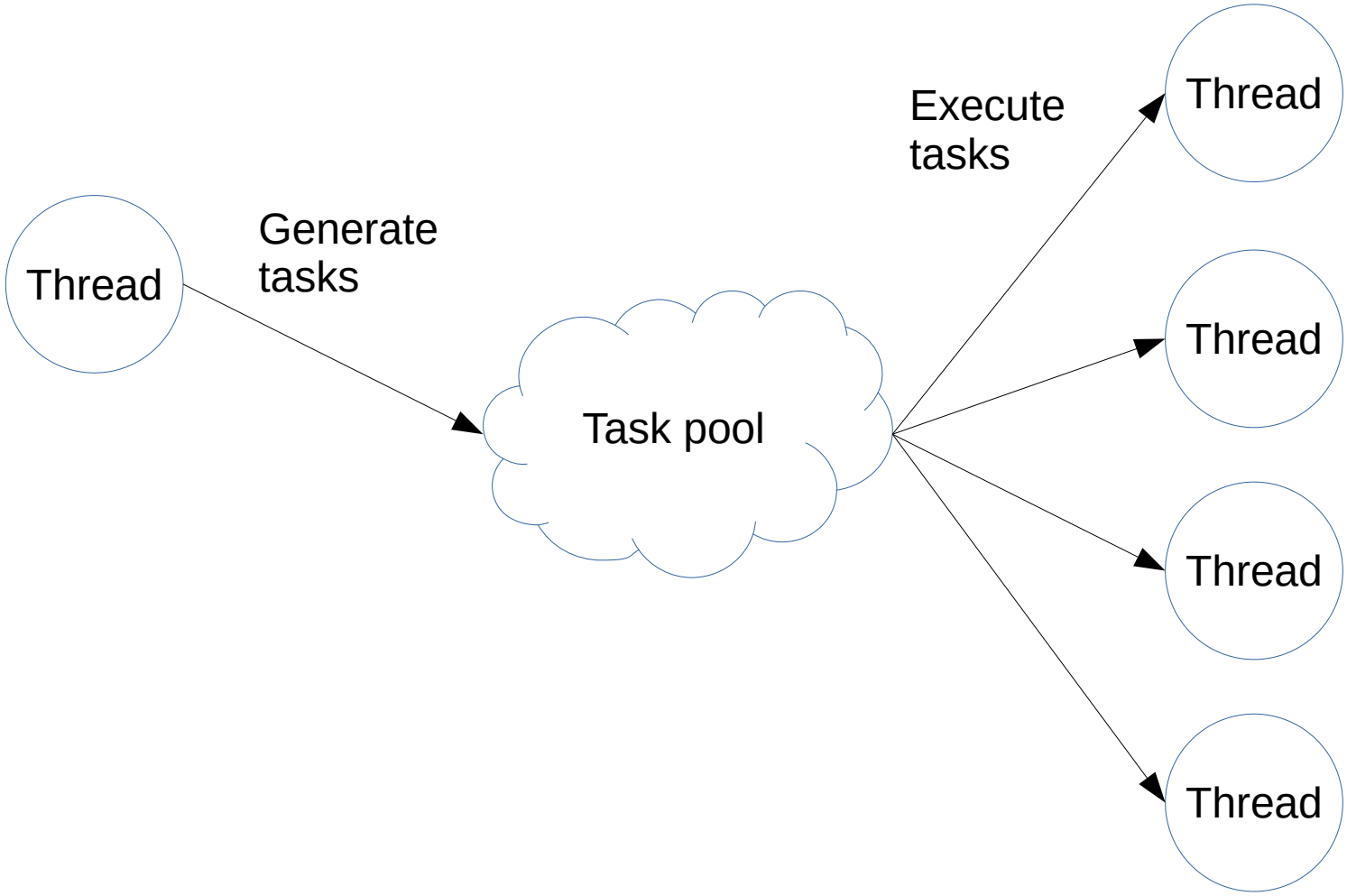
Example

- Try vector normalization using OpenMP SIMD
  - Implement method `normalizationParallelSimd` in `VectorNormalization.cpp`

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

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
```

Iterations processed by all threads in team

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

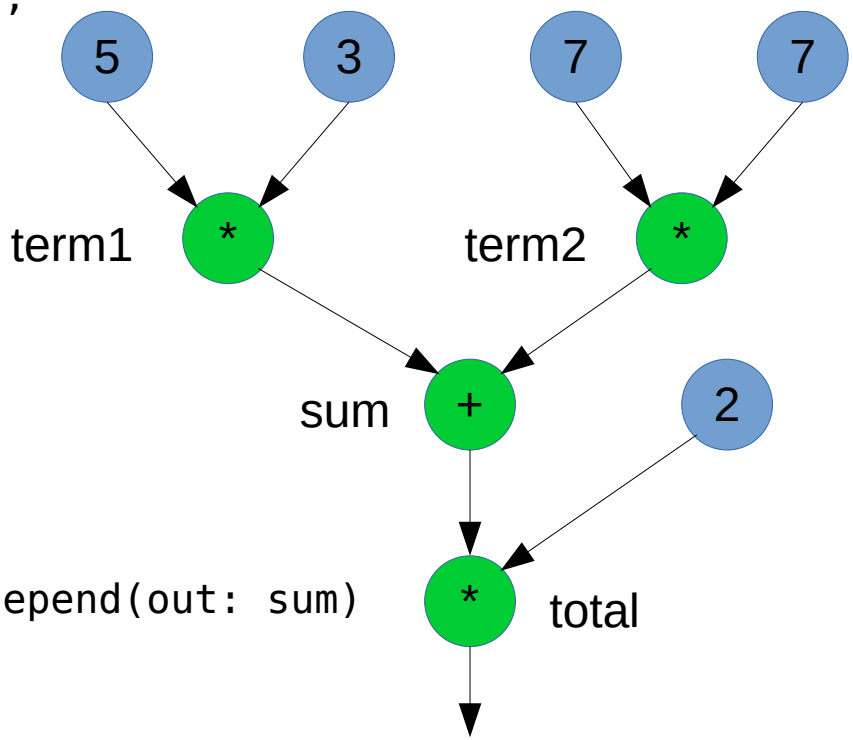
Waits until the completion of all enclosed and descendant tasks

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`

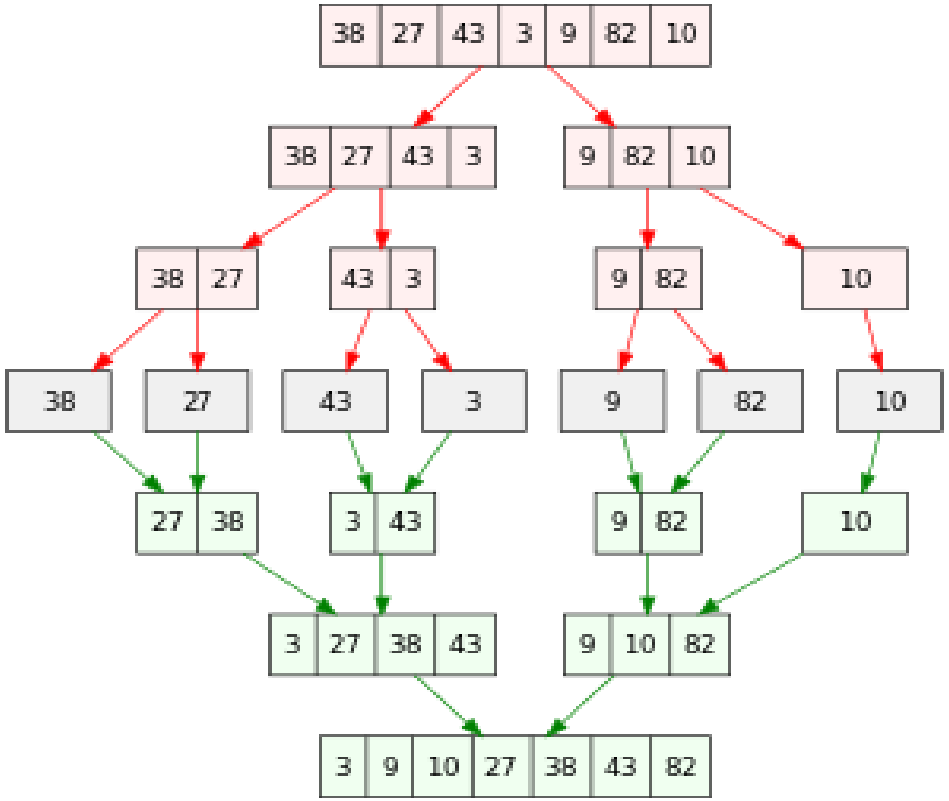


image source

References

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