Omp Threads

- omp_get_num_threads() omp_get_max_threads() omp_in_parallel()
- omp_get_nested() omp get thread num()

Pragma omp

1. pragma omp parallel
    a. pragma omp parallel for
        i. #pragma omp barrier
    b. omp parallel reduction(+: acc)
        i. acc should be global
    c. #pragma omp parallel for reduction(+: acc)
        i. static scheduling
    d. #pragma omp parallel for reduction(+: acc) schedule(dynamic)
        i. dynamic scheduling
        ii. for uneven for loop
    e. #pragma omp parallel for reduction(max:max) reduction(min:min)
    f. #pragma omp parallel for reduction(merge : result)
2. pragma omp tasks
    a. pragma omp taskwait
    b. #pragma omp task shared(x)
3. pragma omp critical { … }
4. pragma omp atomic
    a. ++, --, +, -, *, /, &, ^, |, <<, >>
5. pragma omp cancellation point parallel
    a. if() { … pragma omp cancel parallel }

```
const unsigned numOfThreads = omp_get_num_threads();
const unsigned subInterval = step_count / numOfThreads;
const unsigned threadNum = omp_get_thread_num();

unsigned start = threadNum * subInterval;
unsigned end = (threadNum+1) * subInterval;
double thread_accum = 0;

if (omp_get_thread_num() == omp_get_num_threads()-1) { end = (unsigned) step_count; }

for (unsigned i=start; i<end; i++) {  // do something (thread_accum += …)    }
#pragma omp critical { acc += thread_accum; }
```

Uzamknutie

```
omp_lock_t writelock;

omp_init_lock(&writelock);
omp_set_lock(&writelock);
omp_unset_lock(&writelock);
```

- CV
    - global
        - std::mutex m; / std::unique_lock<std::mutex> lock(m);
        - std::condition_variable cv;
    - 1st thread

```
std::unique_lock<std::mutex> lock(m);
// something
```

        - cv.notify_all(); / cv.notify_one();
    - 2nd thread

```
while(running) {
    std::unique_lock<std::mutex> lock(m);
    cv.wait(lock, [&] { return last_value != value; });
    // do something
}
```

- compare and swap

- atomická operácia s pamäťou na objekte
  - std::atomic<T> X(init_value);
- bool X.compare_exchange_strong(T& expected, T desired)
  if (X == expected) { X = desired; return true; }
  else { expected = X; return false; }

Cancellation point

```
std::atomic<unsigned long> i(1);          std::atomic<unsigned long> solution(1);

#pragma omp parallel
while (true) {
   #pragma omp cancellation point parallel
  if(collatz_value >= criteria) {
    solution = current_i;
     #pragma omp cancel parallel
  }
}
```

Tasks

```
unsigned long long fibonacci_parallel(unsigned int n) {
  unsigned long long result = 0;
  #pragma omp parallel
  #pragma omp single
      { result = fibonacci_parallel_worker(n, n-3); };
  return result;
}

unsigned long long fibonacci_parallel_worker(unsigned int n, unsigned int cutoff) {
  if (n <= cutoff) return fibonacci_sequential(n);
  unsigned long long x, y;

  #pragma omp task shared(x)                #pragma omp task shared(y)
  x = fibonacci_parallel_worker(n-1, cutoff);    y = fibonacci_parallel_worker(n-2, cutoff);

  #pragma omp taskwait
  x += y;
  return x;
}
```

Vectorization

```
float b[], float c[], float a[N];                    for (unsigned int i = 0; i < N; i += 8) {
__m256 B = _mm256_loadu_ps(&b[i]);                   __m256 A = _mm256_mul_ps(B, B);
A = _mm256_fmadd_ps(B, C, A);                        A = _mm256_add_ps(A, _mm256_div_ps(C, B));
A = _mm256_add_ps(A, _mm256_div_ps(B, _mm256_add_ps(B, C)));
A = _mm256_sqrt_ps(A);                          _mm256_storeu_ps(&a[i], A);
```

Tips

- firstprivate
- shared
- atomic variable: std::atomic<unsigned long> min(std::numeric_limits<unsigned long>::max());
- std::abs(acc)
- vector<int> g1;

```
#pragma omp parallel reduction(max:maxFalseDepth)
#pragma omp single
avl_parallel_worker(args)
    also, atomickou promennou, pak updatovat pres compare_exchange_strong
    cut off jsem mel nastaveny na 7
```

```
auto begin = steady_clock::now();
auto end = steady_clock::now();
printf("%s        %7ldms\n", test_name.c_str(), duration_cast<milliseconds>(end - begin).count());
```