Synchronization

- multiple threads reading a shared memory location is not problematic
- when writing to a single memory item we need synchronization to avoid data races
- OpenMP provides mainly two means to isolate read/write accesses to variables
  critical and atomic directives

Example

```
int N = 1000000000;
std::vector<int> prime_numbers;
#pragma omp parallel for schedule(dynamic)
for (int i = 0; i < N; ++i) {
  if (is_prime(i))
    prime_numbers.push_back(i);
}
```

- multiple threads alter internal state of std::vector concurrently
- the STL in general, and std::vector in particular are not thread-safe
- we need to make sure only one thread at a time adds an element to the vector

Synchronization: critical-directive

- a region of code that must be executed by only one thread at a time
- `#pragma omp critical [(name)]`
- critical sections with the same name are treated as the same protected section
- when no name is given, critical sections belong to the global name
- give your critical sections meaningful names according to their semantic

Example

```
int N = 1000000
std::vector<int> prime_numbers;
#pragma omp parallel for schedule(dynamic)
for (int i = 0; i < N; ++i)
  if (is_prime(i))
    #pragma omp critical (prime_insert)
    prime_numbers.push_back(i);
```

What does critical do internally?

```
std::mutex prime_insert;
int N = 1000000000;
std::vector<int> prime_numbers;
#pragma omp parallel for schedule(dynamic)
for (int i = 0; i < N; ++i) {
  if (is_prime(i)) {
    std::lock_guard<std::mutex> prime_insert_lock(prime_insert);
    prime_numbers.push_back(i);
  }
}
```

- all major OpenMP implementations use locks to implement the critical-directive
- locks are expensive: use wisely and sparingly
- can destroy performance in tight loops on frequently updated data-structures

Synchronization: atomic-directive

- often a cheaper alternative to expensive locks
- `#pragma omp atomic`
- can be applied only to certain binary operations +, +, *, /, shift- and logic operators
- mostly used to increment/decrement a variable
- may not be available on all architectures
- requires special hardware support
- basic building block for lock-free programming

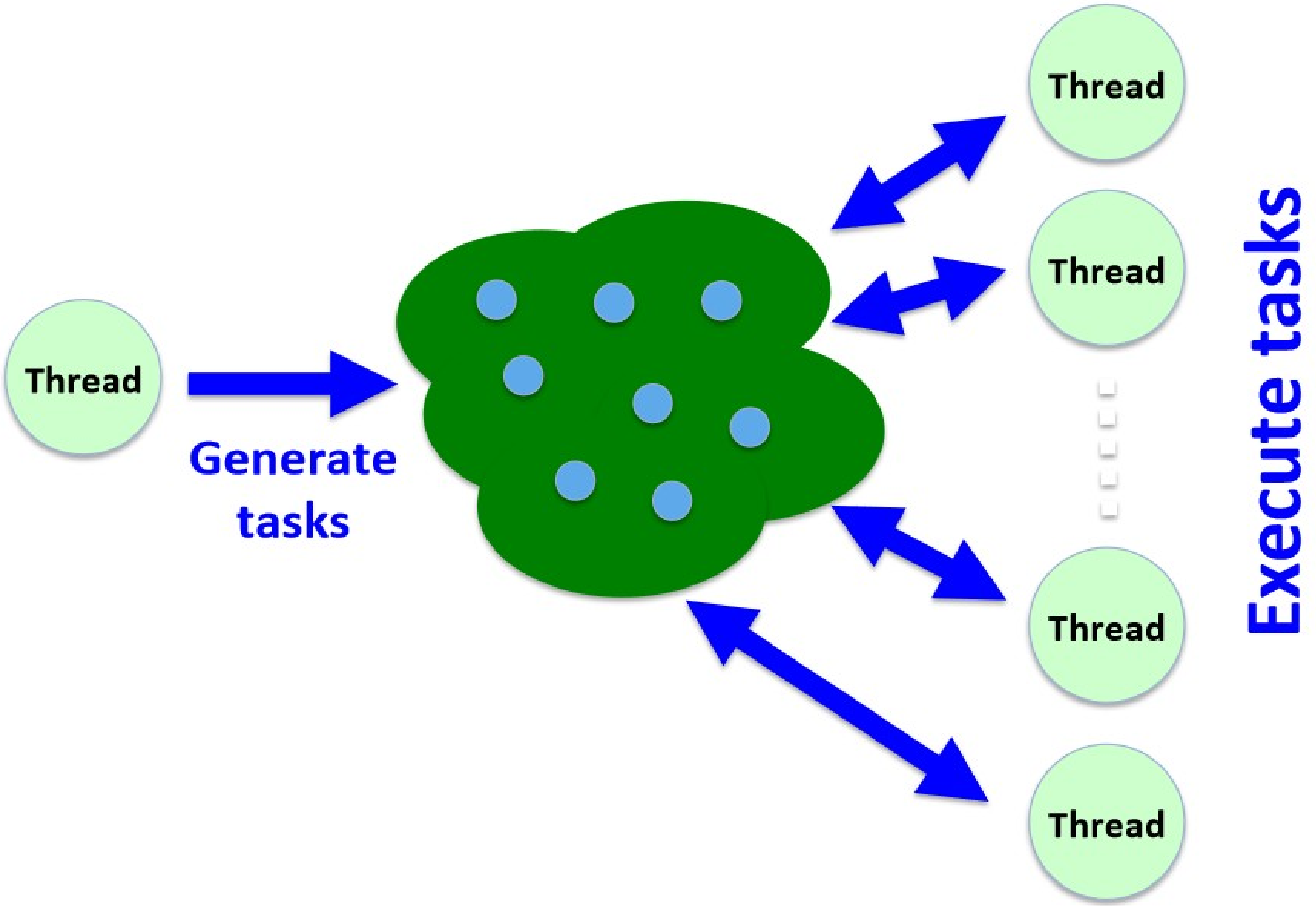
Example

```c
int N = 1000000000;
int num_primes = 0;
#pragma omp parallel for schedule(dynamic)
for (int i = 0; i < N; ++i) {
  if (is_prime(i))
    #pragma omp atomic
    ++num_primes;
}
```

OpenMP Tasking

- up until OpenMP 2.5 directives were directed towards more regular program structure
  - loop iterations known at runtime (fixed!)
  - only a finite number of parallel regions (nesting + finite thread number)
- not suitable for linked lists or recursive algorithms (possible, but ugly at best)
- OpenMP 3.0 Tasks: good for irregular problems
- tasks are lightweight encapsulations of work

Tasking concept



OpenMP task directive

- `#pragma omp task [`**`clauses`**`]`
- known clauses: **default, private, firstprivate, shared, if**
- new: **final, untied, mergeable**
- task is bound to the innermost enclosing parallel region and its thread team
- the tasks code is the following structured block together with a data-environment according to the usual data sharing rules
- tasks can be executed immediately or deferred

task clauses

- if(expr) : if expression evalutes to false, no task is generated, the current thread executed the code immediately -> performance optimization
- final(expr): similar to if, but all child tasks inherit the final property (e.g. recursive algorithms reached certain depth)
- untied: if executing thread is suspended (for whatever reason), another thread may “steal” the task and continue
- mergeable: not really beneficial in practice: allow the runtime to merge the tasks data environment with its calling environment

Example

```
#pragma omp parallel
{
  #pragma omp single
  while (my_pointer) {
  #pragma omp task firstprivate(my_pointer)
  (void) do_independent_work(my_pointer);  // the task's code
  my_pointer = my_pointer->next;
  }  // implicit end of single
} // end of parallel region
```

When do task get executed?

- depending on state: immediately or deferred
- immediate
  if-clause evaluate to false or final-clause evaluate to true
- deferred tasks are executed at barriers or taskwait constructs
  - `#pragma omp barrier` and all implicit barriers (end of parallel region, for loop sharing, ...)
  - `#pragma omp taskwait`

taskyield directive

- performance hint to the OpenMP runtime
- suspend the current task to allow the executing thread to do other (useful) work

```c
#include <omp.h>
void something_useful();
void something_critical();
void foo(omp_lock_t * lock, int n) {
  for(int i = 0; i < n; i++)
  #pragma omp task
  {
    something_useful();
    while( !omp_test_lock(lock) ) {
      #pragma omp taskyield
    }
    something_critical();
    omp_unset_lock(lock);
  }
}
```

OpenMP environment

- Variables that control the runtime behavior of an application
- OMP_NUM_THREADS
  sets the maximum default number of threads in a team
  `OMP_NUM_THREADS = 2 ./program`
  no recompilation needed
- OMP_PROC_BIND
  tell the runtime to pin threads to specific cores
  no context switch and hot caches
  `OMP_PROC_BIND=true ./program`
- OMP_NESTED
  activates nested parallelism, which is deactivated by default
  `OMP_NESTED = true ./program`