HPCSE I

OpenMP

The OpenMP standard

- An **Open** specification for **M**ulti **P**rocessing, http://openmp.org
  - simple threading on shared memory platforms
  - portable and standardized across many platforms and compilers
  - supports C/C++ and Fortran
  - lean and ease, easy to use, augment code with compiler directives
- OpenMP is easy to use but is **not**
  - checking for data dependencies, conflicts, race conditions, or deadlocks
  - giving you the best optimized code
  - implemented in the same way on all compilers

OpenMP function calls

- OpenMP provides semi-automatic parallelization through **compiler directives**, **environment variables** and **function calls**
- Functions defined in header <omp.h>

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

int main()
{
  std::cout << "I am thread " << omp_get_thread_num()
            << " of " << omp_get_num_threads() << " threads." << std::endl;
}
```

- Compile and link using the -fopenmp option on gcc:

```
g++ -fopenmp openmp1.cpp
```

OpenMP directives

- Directives control multi-threading, in C/C++ through pragmas

`#pragma omp` *`directive`*

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

int main()
{
  #pragma omp parallel
  {
    // now we execute this block in multiple threads
    std::cout << "I am thread " << omp_get_thread_num()
              << " of " << omp_get_num_threads() << " threads." << std::endl;
  }

}
```

Synchronization – critical sections

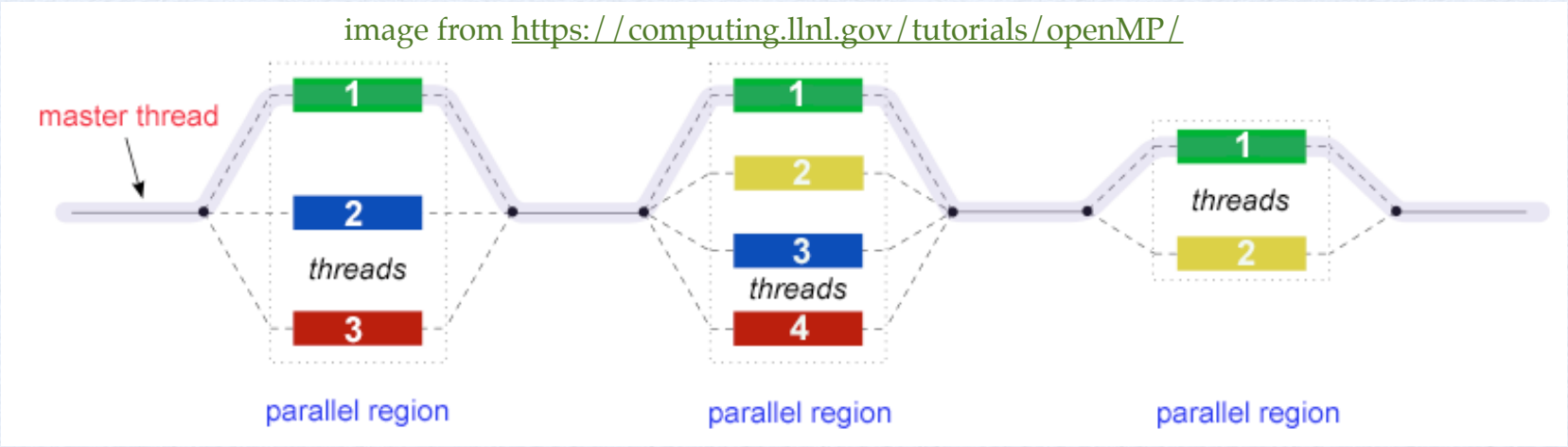
- The output was messed up. Let's use a "critical" directive to have only one thread run that statement at any given time

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

int main()
{
  #pragma omp parallel
  {
    // now we execute this block in multiple threads
    #pragma omp critical (output)
    std::cout << "I am thread " << omp_get_thread_num()
              << " of " << omp_get_num_threads() << " threads." << std::endl;
  }

}
```

Parallel regions

- The **parallel** directive indicates a parallel region
  - executes the following block in multiple threads
  - the threads join at the end of the block

image from https://computing.llnl.gov/tutorials/openMP/



Environment variables

- Environment variables can also be used to control the execution

```
powerbook-troyer-2:openmp troyer$ ./a.out
I am thread 2 of 8 threads.
I am thread 1 of 8 threads.
I am thread 3 of 8 threads.
I am thread 6 of 8 threads.
I am thread 5 of 8 threads.
I am thread 4 of 8 threads.
I am thread 0 of 8 threads.
I am thread 7 of 8 threads.
powerbook-troyer-2:openmp troyer$ export OMP_NUM_THREADS=4
powerbook-troyer-2:openmp troyer$ ./a.out
I am thread 0 of 4 threads.
I am thread 2 of 4 threads.
I am thread 1 of 4 threads.
I am thread 3 of 4 threads.
```

Simpson integration (1)

- Simpson integration done by OpenMP

```
int main()
{
  double a;            // lower bound of integration
  double b;            // upper bound of integration
  unsigned int nsteps; // number of subintervals for integration

  // read the parameters
  std::cin >> a >> b >> nsteps;

  double result = 0.;

  #pragma omp parallel
  {
    int i = omp_get_thread_num();
    int n = omp_get_num_threads();
    double delta = (b-a)/n;
    // integrate just one part in each thread
    double r = simpson(func,a+i*delta,a+(i+1)*delta,nsteps/n);
    result += r;
  }

  std::cout << result << std::endl;

  return 0;
}
```

Problem: race condition

Simpson integration (2)

- Use a critical section directive

```
int main()
{
  double a;             // lower bound of integration
  double b;             // upper bound of integration
  unsigned int nsteps; // number of subintervals for integration

  // read the parameters
  std::cin >> a >> b >> nsteps;

  double result = 0.;

  #pragma omp parallel
  {
    int i = omp_get_thread_num();
    int n = omp_get_num_threads();
    double delta = (b-a)/n;
    // integrate just one part in each thread
    double r = simpson(func,a+i*delta,a+(i+1)*delta,nsteps/n);

    #pragma omp critical (simpsonresult)
    result += r;
  }

  std::cout << result << std::endl;

  return 0;
}
```

No two threads will simultaneously be in a critical section with the same name (simpsonresult)

might be expensive

Simpson integration (3)

- Use an atomic directive

```
int main()
{
  double a;            // lower bound of integration
  double b;            // upper bound of integration
  unsigned int nsteps; // number of subintervals for integration

  // read the parameters
  std::cin >> a >> b >> nsteps;

  double result = 0.;

  #pragma omp parallel
  {
    int i = omp_get_thread_num();
    int n = omp_get_num_threads();
    double delta = (b-a)/n;
    // integrate just one part in each thread
    double r = simpson(func,a+i*delta,a+(i+1)*delta,nsteps/n);

    #pragma omp atomic
    result += r;
  }

  std::cout << result << std::endl;

  return 0;
}
```

The following update (+=) will be atomic

better, but not the best

Synchronization directives

`#pragma omp master`	The block is performed only by the **master** thread
`#pragma omp single`	The block is performed only by one single thread
`#pragma omp critical [ (name) ]`	If a thread is already in the (named) section, all other threads entering it will wait.
`#pragma omp barrier`	All threads wait until every thread has called the barrier
`#pragma omp atomic`	The following update operation is atomic
`#pragma omp threadprivate (list)`	Declares the listed variables to be thread-private, *i.e.* every thread gets its own copy
`#pragma omp flush (list)`	Makes sure that all (or all listed) variables are written back to memory. This ensures that all threads then have the same value of these variables

Clauses for omp parallel

- The parallel directive takes optional clauses

`if (scalar_expression)`	Only parallelize if the expression is true. Can be used to stop parallelization if the work is too little
`private (list)`	The specified variables are thread-private
`shared (list)`	The specified variables are shared among all threads
`default (shared | none)`	Unspecified variables are shared or not
`copyin (list)`	Initialize private variables from the master thread
`firstprivate (list)`	A combination of **`private`** and **`copyin`**
`reduction (operator: list)`	Perform a reduction on the thread-local variables and assign it to the master thread
`num_threads (integer-expression)`	Set the number of threads

- An example:

```
#pragma omp parallel private(i) shared (n) if (n>10)
{
  ...
}
```

Simpson integration (4)

- Use a reduction clause

```
int main()
{
  double a;             // lower bound of integration
  double b;             // upper bound of integration
  unsigned int nsteps; // number of subintervals for integration

  // read the parameters
  std::cin >> a >> b >> nsteps;

  double result;

  #pragma omp parallel reduction(+:result)
  {
    int i = omp_get_thread_num();
    int n = omp_get_num_threads();
    double delta = (b-a)/n;
    // integrate just one part in each thread
    result = simpson(func,a+i*delta,a+(i+1)*delta,nsteps/n);
  }

  std::cout << result << std::endl;

  return 0;
}
```

automatically sums up result
no need to initialize it first

the easiest and fastest

- Allowed reduction operations are +, -, *, &, ^, |, &&, ||,min,max

Sections

- using parallel sections

```
int main()
{
  double a;            // lower bound of integration
  double b;            // upper bound of integration
  unsigned int nsteps; // number of subintervals for integration

  // read the parameters
  std::cin >> a >> b >> nsteps;

  double result=0.;

  #pragma omp parallel shared(result)
  {
    #pragma omp sections reduction(+:result)
    {
      #pragma omp section
      {
        result = simpson(func,a,a+0.5*(b-a),nsteps/2);
      }
      #pragma omp section
      {
        result = simpson(func,a+0.5*(b-a),b,nsteps/2);
      }
    }
  }

  std::cout << result << std::endl;

  return 0;
}
```

each section gets assigned to a different thread

The MASTER directive

- Only the master thread (number 0) will execute the code.

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

int main() {

  #pragma omp parallel
  #pragma omp master
  std::cout << "Only thread " << omp_get_thread_num()
            << " of " << omp_get_num_threads() << " is printing.\n";
  return 0;
}
```

Calculating π through a series

- recall the calculation of π through a series

```
#include <vector>
#include <iomanip>
#include <iostream>

int main()
{
  unsigned long const nterms = 100000000;

  long double sum=0.;

  for (std::size_t t = 0; t < nterms; ++t)
    sum += (1.0 - 2* (t % 2)) / (2*t + 1);

  std::cout << "pi=" << std::setprecision(18) << 4.*sum << std::endl;
  return 0;
}
```

Multhreading the series

- let us parallelize a for loop, and be smart now using reductions

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

int main()
{
  unsigned long const nterms = 100000000;
  long double sum=0.;

  #pragma omp parallel reduction(+:sum)
  {
    int i = omp_get_thread_num();
    int nthreads = omp_get_num_threads();

    long double const step = (nterms+0.5l) /  nthreads;
    int j = (i+1) * step;
    for (std::size_t t = i * step; t < j; ++t)
      sum += (1.0 - 2* (t % 2)) / (2*t + 1);
  }

  std::cout << "pi=" << std::setprecision(18) << 4.*sum << std::endl;
  return 0;
}
```

- easier than manual threading but still cumbersome

Using the for directive

- let the compiler split the for loop

```
#include <iomanip>
#include <iostream>

int main()
{
  unsigned long const nterms = 100000000;

  long double sum=0.;

  #pragma omp parallel shared(sum)
  {
    #pragma omp for reduction(+:sum)
    for (std::size_t t = 0; t < nterms; ++t)
      sum += (1.0 - 2* (t % 2)) / (2*t + 1);
  }

  std::cout << "pi=" << std::setprecision(18) << 4.*sum << std::endl;
  return 0;
}
```

- that's already easier

Using the parallel for directive

- we can merge the parallel and for directives

```
#include <iomanip>
#include <iostream>

int main()
{
  unsigned long const nterms = 100000000;

  long double sum=0.;

  #pragma omp parallel for reduction(+:sum)
  for (std::size_t t = 0; t < nterms; ++t)
    sum += (1.0 - 2* (t % 2)) / (2*t + 1);

  std::cout << "pi=" << std::setprecision(18) << 4.*sum << std::endl;
  return 0;
}
```

- We only added a **single line** to the serial code!!!
- The for directive only works if
  - the loop control variable is an integer, pointer, or C++ random access iterator
  - the loop condition is a simple binary comparison (<, <=, !=, > or >=) with a constant
  - the increment is x++, ++x, x--, ==x, x+= *inc*, x-= *inc*, x=x + *inc*, or x= x - *inc* with a constant increment *inc*

Additional clauses for for directive

- The for directive takes additional optional clauses

`nowait`	There is no implicit barrier at the end of the for. Useful, e.g. if there are two for loops in a parallel section.
`ordered`	The same ordering as in the serial code can be enforced
`collapse (n)`	collapse n nested loops into one and parallelize it
`schedule (type [,chunk])`	specify the schedule for loop parallelization (see below)

- The *schedule* clause specifies how the iterations get divided onto threads. The type can be

STATIC	Loop iterations are divided into fixed chunks and assigned statically
DYNAMIC	Loop iterations are divided into fixed chunks and assigned dynamically whenever a thread finished with a chunk.
GUIDED	Like dynamic but with decreasing chunk sizes. The chunk parameter defines the minimum block size
RUNTIME	decide at runtime depending on the OMP_SCHEDULE environment variable
AUTO	decided by compiler and/or runtime system

NOWAIT and COLLAPSE examples

- NOWAIT useful for multiple independent loops

```
#pragma omp parallel
{
  #pragma omp for nowait
  for (int i=1; i<n; i++)
    b[i] = (a[i] + a[i-1]) / 2.0;

  #pragma omp for nowait
  for (int i=0; i<m; i++)
    y[i] = sqrt(z[i]);
}
```

- COLLAPSE useful for multiple nested loops

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

ORDERED examples

- ORDERED can enforce sequential ordering in loops

```
#include <iostream>

int main()
{
  #pragma omp parallel for ordered
  for (int i=0; i < 100; ++i) {
    // do some (fake) work
    int j=i;
    #pragma omp ordered
    std::cout << "Hello from the " << j << "-th iteration\n";
  }
}
```

- Attention: this enforces very strict synchronization and can be slow

OpenMP environment variables

- The behavior of the code can be controlled at runtime using environment variables

OMP_NUM_THREADS	The maximum number of threads to be used in parallelization of one parallel region
OMP_DYNAMIC	Set to TRUE or FALSE to enable or disable dynamic adjustment of the number of threads
OMP_PROC_BIND	Supported since OpenMP 3.0. Set to TRUE to bind threads to processors and disable migration to other processors. Important on NUMA architectures
OMP_NESTED	Supported since OpenMP 3.0. Set to TRUE or FALSE to enable or disable nested parallelization. Nested parallelism occurs when a function containing a parallel region is called from another parallel region.
OMP_STACKSIZE	Supported since OpenMP 3.0. Controls the stack size of non-Master threads. Be careful: if the stack gets exhausted a program may segfault or just continue running with corrupted data.
OMP_MAX_ACTIVE_LEVELS	Supported since OpenMP 3.0. Limits the number of nested levels.
OMP_THREAD_LIMIT	Supported since OpenMP 3.0. Limits the total number of threads.
OMP_WAIT_POLICY	If set to ACTIVE waiting threads spin actively, if set to PASSIVE they sleep

OpenMP runtime library

`void omp_set_num_threads(int n)`	Sets the number of threads to be used.
`int omp_get_num_threads()`	Gets the number of currently running threads.
`int omp_get_max_threads()`	Gets the maximum number of threads that can be used for one parallel region.
`int omp_get_thread_num()`	Get the id of the calling thread
`int omp_get_thread_limit ()`	Gets the maximum number of threads used for nested parallel region.
`int omp_get_num_procs()`	Gets the number of processors available
`int omp_in_parallel()`	Returns true if called from within a parallel region
`void omp_set_dynamic(int n)`	Sets dynamic adjustment of threads with the given number as maximum. This overrides the environment variable.
`int omp_get_dynamic()`	Returns true if dynamic scheduling is enables
`double omp_get_wtime()`	A portable wallclock timing routine, returns time in seconds. The time is **not synchronized** across threads to be fast.
`double omp_get_wtick()`	Returns the number of seconds between successive clock ticks

Functions for nested parallelism

`void omp_set_nested(int l)`	Enables nested parallelism if called with true as argument
`int omp_get_nested ()`	Returns true if nested parallelism is available
`void omp_set_max_active_levels (int l)`	Sets the maximum number of nested parallel regions allowed. If parallel regions are nested deeper, the additional ones will be ***inactive*** and only be run by one thread.
`int omp_get_max_active_levels()`	Gets the maximum number of nested parallel regions allowed
`int omp_get_level()`	Returns the number parallel regions enclosing the call
`int omp_get_active_level()`	Returns the number of active parallel regions (run by more than one thread) enclosing the call
`int omp_get_ancestor_thread_num(int l)`	Returns the thread number of the ancestor of the current thread in a higher level. Returns -1 if the level is invalid.
`int omp_get_team_size(int level)`	Returns the size of the thread team to which the ancestor at the given level belongs. Returns -1 if the level is invalid.

Functions for loop schedules

- These functions are used to set runtime loop schedules

`void omp_set_schedule(omp_sched_t kind, int modifier)`	sets the schedule
`void omp_get_schedule(omp_sched_t * kind, int * modifier )`	gets the schedule

- The schedules are defined by an enum and must support at least (can be extended by vendor-specific extensions)

```
typedef enum omp_sched_t {
      omp_sched_static = 1,
      omp_sched_dynamic = 2,
      omp_sched_guided = 3,
      omp_sched_auto = 4,
} omp_sched_t;
```

- The modifier is the chunk size for those schedules which take a chunk size

Locking functions

- OpenMP also contains locks that need to be manually created, locked, unlocked, and destroyed. The nested lock allows recursive locking by the same thread.

`void omp_init_lock(omp_lock_t *lock)` `void omp_init_nest_lock(omp_nest_lock_t *lock)`	Initializes a lock variable
`void omp_destroy_lock(omp_lock_t *lock)` `void omp_destroy_nest_lock(omp_nest_lock_t *lock)`	Destroys a lock variable
`void omp_set_lock(omp_lock_t *lock)` `void omp_set_nest__lock(omp_nest_lock_t *lock)`	Sets (locks) the lock. The thread suspends until the lock can be obtained.
`void omp_unset_lock(omp_lock_t *lock)` `void omp_unset_nest_lock(omp_nest_lock_t *lock)`	Releases the lock.
`int omp_test_lock(omp_lock_t *lock)`	Tests whether a lock is available. If yes the lock is set and true is returned, otherwise false is returned.

- This is ugly, thus let's create C++ objects using RAII - we can then never forget to initialize, destroy, or unlock.

Requirements for a C++11 mutex

- std::lock_guard<Mutex> template requires a mutex m of type Mutex to model the BasicLockable concept:

Expression	Requires	Effects
m.lock()		Blocks until a lock can be obtained for the current execution agent. If an exception is thrown, no lock is obtained.
m.unlock()	The current execution agent should hold the lock m.	Releases the lock held by the execution agent. Throws no exceptions.

- We can implement an OpenMP mutex and use it with std::lock_guard

A C++11 conforming OpenMP mutex

- Use the RAII pattern to intialize and destroy in constructor and destructor
- Implement the required lock and unlock functions

```
#include <omp.h>

class omp_mutex
{
public:
  omp_mutex() { omp_init_lock(&_mutex); }
  ~omp_mutex() { omp_destroy_lock(&_mutex); }
  void lock() { omp_set_lock(&_mutex); }
  void unlock() { omp_unset_lock(&_mutex); }
private:
  omp_lock_t _mutex;
};


class omp_recursive_mutex
{
public:
  omp_recursive_mutex() { omp_init_nest_lock(&_mutex); }
  ~omp_recursive_mutex() { omp_destroy_nest_lock(&_mutex); }
  void lock() { omp_set_nest_lock(&_mutex); }
  void unlock() { omp_unset_nest_lock(&_mutex); }
private:
  omp_nest_lock_t _mutex;
};
```

Using our mutex

- Use a lock instead of a directive to avoid garbling of output

```
#include <iostream>
#include <mutex>
#include "omp_mutex.hpp"

int main()
{

  omp_mutex m;

  #pragma omp parallel for
  for (int i=0; i < 100; ++i) {
    {
      std::lock_guard<omp_mutex> lock(m);
      std::cout << "Hello from the " << i << "-th iteration\n";
    }
  }
}
```

Locks and synchronization

Always remember:

Synchronization is expensive, thus whenever possible change data access to avoid the need for locks, barriers, or other synchronization.