Algorithm Engineering

Ordered Clause, Collapse Clause, Reductions, Barriers, and Storage Attributes

Mark Blacher

Friedrich Schiller University Jena

Winter Semester of 2022/23

The ordered Clause

Different threads execute concurrently until they encounter the **ordered region**, which is then **executed sequentially** in the **same order as** it would get executed in a **serial loop**.

ordered.cpp

```cpp
int main() { // find smallest solution with the ordered clause
  constexpr int biggest_possible_number = 10000;
  atomic<bool> solution_found(false); // if true than we found the solution
  int final_solution = INT32_MAX;
  const double start = omp_get_wtime();
#pragma omp parallel for ordered schedule(dynamic) // start parallel region
  for (int i = 0; i < biggest_possible_number; ++i) {
    if (solution_found) // we found the solution, just continue iterating
      continue;
    if (is_solution(i)) {
#pragma omp ordered // ordered region
      if (!solution_found) { // ordered execution of if statement
        solution_found = true; // no race condition
        final_solution = i;
      }
    }
  } // end parallel region
  cout << "The solution is: " << final_solution << endl;
  cout << omp_get_wtime() - start << " seconds" << endl;
}
```

Cancellation Points

cancellation_points.cpp

```cpp
// actually, we can stop a parallel for loop with cancellation points, but it's a little complicated
int main(int argc, char **argv) { // since OpenMP 4.0 we have cancellation points
  if (!omp_get_cancellation()) { // if no cancellations enabled, enable it
    cout << "Enabling cancellation and rerunning program\n" << endl;
    const char* enable_cancellation = "OMP_CANCELLATION=true";
    // const_cast can be used to pass const data to a function that doesn't receive const
    putenv(const_cast<char *>(enable_cancellation));  // set cancellation environment variable
    execv(argv[0], argv); // rerun program, because can't enable cancellations in the program itself
  } // execv replaces the current process image with a new process image
  constexpr int biggest_possible_number = 10000;
  atomic<int> final_solution(INT32_MAX);
  const double start = omp_get_wtime();

#pragma omp parallel // start parallel region
  {
#pragma omp for schedule(dynamic)
    for (int i = 0; i < biggest_possible_number; ++i) {
      if (is_solution(i)) { // find some solution, not necessary the smallest
        final_solution = i;
#pragma omp cancel for // signal cancellation, because we found a solution
      }
#pragma omp cancellation point for // check for cancellations signalled from other threads
      // cancel for loop if cancellations signalled
    }
  } // end parallel region

  // check if we've found a solution at all is omitted, you can add the check
  cout << "The solution is: " << final_solution << endl;
  cout << omp_get_wtime() - start << " seconds" << endl;
}
```

Nested For Loops

We can **parallelize nested for loops** with the `collapse` clause.

collapse.cpp

```cpp
int main() {
  const int N = 10;
  const int M = 12;

  /**
   * will form a single loop of length N * M = 120
   * and then parallelize that
   *
   * can be useful for balancing the work
   */
#pragma omp parallel for collapse(2)
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
      // do useful work with i and j
    }
  }
}
```

Collapse Clause Example

grid_search.cpp

```cpp
// mock k-nearest neighbors classification
void knn(int k, const string &weight, const string &metric) {
  // larger k needs more runtime
  for (volatile int i = 0; i < 10000000 * k; ++i) {}
  // computing accuracy of classification is omitted here
#pragma omp critical // output which thread did what
  cout << "k: " << k << ", weight: " << weight << ", metric: " << metric
       << ", computed with thread: " << omp_get_thread_num() << endl;
}

int main() {
  vector<int> ks{1, 3, 5, 7, 9, 11};
  vector<string> weights{"uniform", "distance"};
  vector<string> metrics{"euclidean", "manhattan"};
  const double start = omp_get_wtime();
  // parallel grid search for tuning the hyperparameters in knn
#pragma omp parallel for collapse(3) schedule(dynamic)
  for (uint64_t i = 0; i < ks.size(); ++i)
    for (uint64_t j = 0; j < weights.size(); ++j)
      for (uint64_t k = 0; k < metrics.size(); ++k)
        knn(ks[i], weights[j], metrics[k]);

  cout << omp_get_wtime() - start << " seconds" << endl;
}
```

Reduction

Reduction is an **associative** and **commutative operation**. It is used in parallel programming to **reduce many values** into a **single result**.

In OpenMP the reduction clause looks like this:
`reduction(op:list)`

op can be `+`, `*`, `-`, `min`, `max`, `&`, `|`, `^`, `&&` and `||`

list contains the variables, separated by commas, that are to be reduced

Reduction Example

reduction.cpp

```cpp
int main() {
  int n = 100000000;
  // create a vector of size n, all values 1.0
  // long double is "usually" a 128 bit float data type
  // L indicates that 1.0 is a long double literal
  vector<long double> vec(n, 1.0L);
  long double sum = 0.0L;

  const double start = omp_get_wtime();

#pragma omp parallel for reduction(+ : sum)
  for (int i = 0; i < n; ++i) {
    sum += vec[i]; // sum up all values of vec
  }

  cout << fixed << "sum: " << sum << endl;
  cout << omp_get_wtime() - start << " seconds" << endl;
}
```

How Reductions Work Internally

- A local copy of each ***`list`*** variable is made and initialized depending on the ***`op`*** (0 for **`+`**)
- Updates occur on the local copy
- Local copies are reduced into a single value and combined with the original global value

Synchronization With **Barriers**

Barrier

A barrier means that **any thread must stop** at **this point** and **cannot proceed until all other threads reach this barrier**.

barrier.cpp

```cpp
int main() {
#pragma omp parallel
  {
    stringstream info;
    info << "Hello from thread " << omp_get_thread_num() << endl;
    cout << info.str();
#pragma omp barrier // move on as soon as all threads printed Hello ...
    info.str(""); // "clear" stringstream variable
    info << "Goodbye from thread " << omp_get_thread_num() << endl;
    cout << info.str();
  }
}
```

Implicit Barriers in OpenMP

The `for`, `sections`, `single` and `parallel` constructs in OpenMP have an **implicit barrier**.

Implicit barrier means that **there is a barrier without you explicitly positioning it there**.

In the constructs `for`, `sections` and `single` you can remove the implicit barrier with the `nowait` clause.

You **can't remove the implicit barrier** in the `parallel` construct.

Sections

sections.cpp

```cpp
#include <iostream>

using namespace std;

int main() {
#pragma omp parallel
  {
#pragma omp sections // add nowait to remove the implicit barrier
    { // Each section is executed once by one of the threads in the team
#pragma omp section // some thread executes this block of code
      { cout << "Hello from section 1\n"; }
#pragma omp section // some thread executes this block of code
      { cout << "Hello from section 2\n"; }
#pragma omp section // some thread executes this block of code
      { cout << "Hello from section 3\n"; }
    } // after the sections construct is an implicit barrier
    cout << "Hello after the sections\n"; // this line is executed by
                                          // every thread in the team
  }
}
```

Single

The **single construct** specifies that the **associated structured block** is **executed by only one of the threads** in the team (not necessarily the master thread).

single.cpp

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

using namespace std;

int main() {
#pragma omp parallel
  {
#pragma omp single // only one thread executes the code in the single
    {
      cout << "from single: " << omp_get_thread_num() << "\n";
    } // implicit barrier after single
    cout << "after single\n";
  };
}
```

Useful **Runtime Library Routines**

These are the **routines you already know**:

- `omp_set_num_threads(int)` *// setting desired number of threads in the parallel region*
- `omp_get_num_threads()` *// number of threads*
- `omp_get_thread_num()` *// thread id*
- `omp_get_wtime()` *// wall-clock time in seconds*

Here are **some more routines**:

- `omp_get_num_procs()` *// number of logical cores*
- `omp_get_max_threads()` *// maximum number of threads in a parallel region*
- `omp_in_parallel()` *// true if called inside a parallel region*

Library Routines Demo

library_routines.cpp

```cpp
int main() {
  cout << "num_threads: " << omp_get_num_threads() << "\n"; // prints 1
  // but, if we call omp_get_num_threads in a parallel region
  // then we get the number of threads that are executed inside the parallel region
#pragma omp parallel
#pragma omp single
  cout << "num_threads: " << omp_get_num_threads() << "\n\n"; // e.g 4

  // maximum number of threads in a parallel region (e.g. 4)
  cout << "max_threads: " << omp_get_max_threads() << "\n"; // prints 4
  omp_set_num_threads(2); // change amount of threads in a parallel region
  cout << "max_threads: " << omp_get_max_threads() << "\n\n"; // now it's 2

  // prints 4, if my machine has 4 logical cores
  cout << "num_procs: " << omp_get_num_procs() << "\n\n";
  // logical cores are the number of physical cores times the number
  // of threads (called "hyperthreads") that can run on each core

  cout << "in_parallel: " << omp_in_parallel() << "\n"; // prints 0 ("false")
  // if we call omp_in_parallel in a parallel region then we get 1 ("true")
#pragma omp parallel
#pragma omp single
  cout << "in_parallel: " << omp_in_parallel() << "\n";
}
```

Storage Attributes

Storage attributes determine how **variables defined outside the parallel region** are **handled** ("stored") **inside the parallel region**. *`// simplified explanation but enough for our purposes`*

Useful storage attributes:

- **`shared`** *`// this is default, the variable is shared between threads`*
- **`private`** *`// create uninitialized copy of the variable for each thread`*
- **`firstprivate`** *`// create initialized one-to-one copy of the variable for each thread`*

Storage Attributes Demo

private_vs_firstprivate.cpp

```cpp
int main() {
  int x = 10;
  omp_set_num_threads(4);

  cout << "private(x): ";
#pragma omp parallel private(x)
  { // x is an uninitialized local copy in each thread
    x += omp_get_thread_num(); // (probably) 0 + thread_num
#pragma omp critical
    cout << x << " "; // (probably) something like: 3 2 1 0
  }

  cout << "\nfirstprivate(x): ";
#pragma omp parallel firstprivate(x)
  { // x is an initialized local one-to-one copy in each thread
    x += omp_get_thread_num(); // 10 + thread_num
#pragma omp critical
    cout << x << " "; // something like: 10 12 11 13
  }

  cout << "\n'global' x: " << x << endl; // x = 10 (global x didn't change)
}
```

Storage Attributes Remarks

- Pragmas like this one:

  `#pragma omp parallel default(none) shared(a, b) firstprivate(x, my_vector)`

  are common
- `default(none)` is good for debugging, since you must consciously specify the storage attribute for each 'global' variable used within the parallel region
- Be careful if you use `private(my_vector)`, each thread inside the parallel region creates a vector object of size=0

segmentation_fault.cpp

```cpp
int main() {
  vector<int> my_vector{1, 2, 3};
#pragma omp parallel default(none) shared(cout) private(my_vector)
  {
    // program crashes since my_vector is uninitialized (size = 0)
    cout << my_vector[0]; // segmentation fault
  }
}
```

Coding Warmup

1. **Parallelize** the **serial** π **program** from the first lecture by **adding** only **one extra line** of code.
2. Write a **parallel program** for computing π using **only standard C**++ (no OpenMP).
3. The program on page 20 generates and saves an image of the mandelbrot set. **Reduce** the **runtime** of the program **without degrading** the **image quality**.

Compile the programs with at least the following **flags**:

```
-Ofast -std=c++11 -march=native -fopenmp
```

OpenMP vs C++ Threads

pi_openmp.cpp

```cpp
#pragma omp parallel for reduction(+:sum)
  for (int i = 0; i < num_steps; i++) {
    double x = (i + 0.5) * width; // midpoint
    sum = sum + (1.0 / (1.0 + x * x)); // add new height of a rectangle
  }
```

pi_cpp_threads.cpp

```cpp
// thunk is called from different threads
void thunk(int num_threads, int thread_id, int num_steps, double width,
           double &sum, mutex &m) {
  double sum_local = 0.0; // for summing up heights locally
  for (int i = thread_id; i < num_steps; i += num_threads) {
    double x = (i + 0.5) * width; // midpoint
    sum_local = sum_local + (1.0 / (1.0 + x * x)); // add new height
  }
  lock_guard<mutex> lg(m); // avoids race condition on sum
  sum += sum_local; // update global sum
}
```

It's good to know how you could parallelize without OpenMP.

Generating an Image

mandelbrot.cpp

```cpp
int main() { // generate mandelbrot pgm (portable graymap)
  const string image_name = "mandelbrot.pgm";
  remove(image_name.c_str()); // remove file from disk
  const double start = omp_get_wtime();
  ofstream image(image_name); // file output stream
  if (image.is_open()) {
    image << "P2\n" << width << " " << height << " 255\n"; // pgm header
    for (int i = 0; i < height; i++) {
      for (int j = 0; j < width; j++) {
        image << compute_pixel(j, i) << "\n"; // write pixel value
      }
    }
    image.close(); // close file output stream
  } else {
    cout << "Could not open the file!";
  }
  cout << omp_get_wtime() - start << " seconds" << endl;
}
```

Possible Solution for the Warmup Image Exercise

mandelbrot_tuned.cpp

```cpp
int main() { // generate mandelbrot pgm (portable graymap)
  const string image_name = "mandelbrot.pgm";
  remove(image_name.c_str()); // remove file from disk
  const double start = omp_get_wtime();
  vector<string> look_up{256}; // look-up table for strings
  for (int i = 0; i < 256; ++i) { look_up[i] = to_string(i) + "\n"; } // fill look-up table
  ofstream image(image_name); // file output stream
  if (image.is_open()) {
    image << "P2\n" << width << " " << height << " 255\n"; // write pgm header
#pragma omp parallel // start parallel region
    {
      string buffer; // each thread has it's own string buffer
      buffer.reserve(width * 4); // reserve enough space to avoid reallocation
#pragma omp for schedule(dynamic) ordered // we could also use schedule(static, 1)
      for (int i = 0; i < height; i++) {
        buffer.clear(); // clear content of buffer
        for (int j = 0; j < width; j++) {
          buffer += look_up[compute_pixel(j, i)]; // fill buffer
        }
#pragma omp ordered // write rows of image in sequential order
        image << buffer; // write string buffer to file
      }
    } // end parallel region
    image.close(); // close file output stream
  } else {
    cout << "Could not open the file!";
  }
  cout << omp_get_wtime() - start << " seconds" << endl;
}
```

Exam Assignments

- How does the **ordered clause** in OpenMP work in conjunction with a parallel for loop?
- What is the **collapse clause** in OpenMP good for?
- Explain how **reductions** work internally in OpenMP.
- What is the purpose of a **barrier** in parallel computing?
- Explain the differences between the library routines **`omp_get_num_threads()`**, **`omp_get_num_procs()`** and **`omp_get_max_threads()`**.
- Clarify how the storage attributes **private** and **firstprivate** differ from each other.
- Do the **coding warmup** on **slide 18**. Write in **pseudo code** how the **computation of** π can be **parallelized** with **simple threads**.