Intro to OpenMP

Bryce Mazurowski

October 23, 2023

NCSA Advanced Parallel Computing Cohort Fall 2023
Due date: October 25, 2023

1 What is loop scheduling? Describe at least two different schedules and discuss what type of workloads can benefit from each.

Loop scheduling describes different strategies to partition a given dataset used in a data parallelization framework. There are three main loop scheduling strategies: static, dynamic, and guided.

- Static: full dataset is cut into equal chunks based on the number of threads available to OpenMP. This is the default behavior and good for situations where workload is relatively constant for any index
- Dynamic: dataset is cut into chunks of size R, in principle this should lead to more chunks than there are threads available. Each thread is assigned a chunk, and when it finishes the task on a given chunk it a new one is assigned to it. This strategy can be useful when the workload of a task depends on the index of the loop.
- Guided: dataset is cut into equal portions and tasks are started. Once a thread finishes, the remaining data is redistributed equally again among all threads. This continues until the redistribution pieces are a minimum size R. This strategy is useful for cases where the workload increases with index.

2 What is a reduction clause? Why are they useful? Give a simple example of a situation where using a reduction clause would be appropriate.

Reduction clauses are instructions on how to combine data members in a parallel section. They give each thread a private copy of a variable that it can operate and and tell the program how each thread's pieces fit together in the end. These greatly simplify implementation and can avoid omp barrier that will slow down code.

An example where a reduction clause is appropriate would be computing the maximum element of a large vector. Using a reduction clause on a private variable max would allow each chunk to compute its own max and then perform a final comparison once all threads are complete to get one final value out.

3 Write/find a simple serial program with a race condition or a nested loop (ask ChatGPT if you can't find one).

1. Parallelize the loop using OpenMP directives without explicitly declaring data contexts
2. Check to see if your code is doing what it was supposed to do (it should not if you do have a race condition)
3. Use data contexts (shared and private) to resolve the race condition and check to see if you get the expected results
4. Discuss your results

This code has nested loops and race conditions. The code loops over a vector of ints. In each iteration of the top loop, a second loop occurs over a set number of points. In the second loop, a value is assigned to a dataSet for each iteration. On inner loop exit, the data structure is inserted into a std::map as a pair of outer loop int and data structure. This is to mimic a specific problem I have faced in numerical integration of nonlinear solid mechanics problems.

The nested loop requires a private variable or a race condition will occur.

The map insert will intermittently crash the code in parallel because one thread will try to insert in the map when another is resorting it on insert. This is a tricky bug that a reduction clause should fix.

Insert your serial code here

```
// Bryce Mazurowski (2023)
// brycepm2@gmail.com

#include <iostream>
#include <vector>
#include <map>
#include <math.h>

using std::cout;
using std::endl;
using std::map;
using std::vector;

struct stateVar {
  // default constructor
  stateVar()=default;
  // default destructor
  ~stateVar()=default;
  // default copy constructor
  stateVar(stateVar& svIn)=default;

  // vector to store info for each intPt
  std::vector<double> vals;
};



// how to print data structure
std::ostream& operator<<(std::ostream& os, const stateVar& svObj) {
  cout << "stateVar: " << endl;
  std::vector<double> valsOut = svObj.vals;
  std::vector<double>::iterator iter_val = valsOut.begin();
  for (; iter_val != valsOut.end(); ++iter_val) {
    cout << (*iter_val) << ' ';
  }
  return os;
}

// map for each element
typedef map<int, stateVar*> stateVMap;

/** @brief loop over integration points insert number
 * into data structure for each int point
 */
void integrateLoop(stateVar* p_elemStateVar);
```

```
int main() {
  // set loop limits
  const int loopEnd = 1 << 4;

  // instantiate stateVarMap
  stateVMap svMap;
  for (int iElem = 0; iElem < loopEnd; ++iElem) {
    // create stateVar data structure for each elem
    stateVar* p_elemStateVar = new stateVar();
    // run integration loop
    integrateLoop(p_elemStateVar);
    // insert data structure into map
    svMap.insert({iElem, p_elemStateVar});
  }

  // loop over map and print each point
  // ALSO delete pointers within
  stateVMap::iterator iter_svMap = svMap.begin();
  for (; iter_svMap != svMap.end(); ++iter_svMap) {
    cout << "Key: " << iter_svMap->first << endl;
    cout << "struct: " << (*iter_svMap->second) << endl;
    delete iter_svMap->second;
  }
}


void integrateLoop(stateVar* p_elemStateVar) {
  // number of integration points
  const int nPts = 8;
  for (int iPt = 0; iPt < nPts; ++iPt) {
    double x = double(iPt);
    // make value some function
    const double value = std::pow(x,3.0) + 5*x*x + 13;
    // add value to data structure
    p_elemStateVar->vals.push_back(value);
  }
}
```

Insert your parallelized code without explicit data contexts here

```
// Bryce Mazurowski (2023)
// brycepm2@gmail.com

#include <iostream>
#include <vector>
#include <map>
#include <math.h>

#include <omp.h>

using std::cout;
using std::endl;
using std::map;
```

```
using std::vector;

struct stateVar {
  // default constructor
  stateVar()=default;
  // default destructor
  ~stateVar()=default;
  // default copy constructor
  stateVar(stateVar& svIn)=default;

  // vector to store info for each intPt
  std::vector<double> vals;
};


// how to print data structure
std::ostream& operator<<(std::ostream& os, const stateVar& svObj) {
  cout << "stateVar: " << endl;
  std::vector<double> valsOut = svObj.vals;
  std::vector<double>::iterator iter_val = valsOut.begin();
  for (; iter_val != valsOut.end(); ++iter_val) {
    cout << (*iter_val) << ' ';
  }
  return os;
}

// map for each element
typedef map<int, stateVar*> stateVMap;

/** @brief loop over integration points insert number
 * into data structure for each int point
 */
void integrateLoop(stateVar* p_elemStateVar);

int main() {
  // set loop limits
  const int loopEnd = 1 << 4;

  // instantiate stateVarMap
  stateVMap svMap;
  double sTime = omp_get_wtime();
  #pragma omp parallel
  {
    #pragma omp for
    for (int iElem = 0; iElem < loopEnd; ++iElem) {
      // create stateVar data structure for each elem
      stateVar* p_elemStateVar = new stateVar();
      // run integration loop
      integrateLoop(p_elemStateVar);
      // insert data structure into map
      svMap.insert({iElem, p_elemStateVar});
    }
  }
```

```
  double eTime = omp_get_wtime();

  cout << "Total time (s): " << (eTime - sTime) << endl;

  // loop over map and print each point
  stateVMap::iterator iter_svMap = svMap.begin();
  for (; iter_svMap != svMap.end(); ++iter_svMap) {
    cout << "Key: " << iter_svMap->first << endl;
    cout << "struct: " << (*iter_svMap->second) << endl;
    delete iter_svMap->second;
  }
  return 0;
}


void integrateLoop(stateVar* p_elemStateVar) {
  // number of integration points
  const int nPts = 18;
  for (int iPt = 0; iPt < nPts; ++iPt) {
    double x = double(iPt);
    // make value some function
    const double value = std::pow(x,3.0) + 5*x*x + 13;
    // add value to data structure
    p_elemStateVar->vals.push_back(value);
  }
}
```

Insert your final parallelized code here (race condition resolved)

```
 // Bryce Mazurowski (2023)
// brycepm2@gmail.com

#include <iostream>
#include <vector>
#include <map>
#include <math.h>

#include <omp.h>

using std::cout;
using std::endl;
using std::map;
using std::vector;

struct stateVar {
  // default constructor
  stateVar()=default;
  // default destructor
  ~stateVar()=default;
  // default copy constructor
  stateVar(stateVar& svIn)=default;

  // get size of vector for printing
  int getNumPts() { return vals.size(); }
```

```
  // check values within vector
  bool checkVals() {
    for (int iPt = 0; iPt < vals.size(); ++iPt) {
      double test = std::pow(iPt,3.0) + 5*iPt*iPt + 13;
      if (vals[iPt] != test) {
        return false;
      }
    }
    return true;
  }

  // vector to store info for each intPt
  std::vector<double> vals;
};


// how to print data structure
std::ostream& operator<<(std::ostream& os, const stateVar& svObj) {
  cout << "stateVar: " << endl;
  std::vector<double> valsOut = svObj.vals;
  std::vector<double>::iterator iter_val = valsOut.begin();
  for (; iter_val != valsOut.end(); ++iter_val) {
    cout << (*iter_val) << ' ';
  }
  return os;
}

// map for each element
typedef map<int, stateVar*> stateVMap;

/** @brief loop over integration points insert number
 * into data structure for each int point
 */
void integrateLoop(const int iElem, stateVar* p_elemStateVar);

/** @brief merge maps together for omp reduction
 */
void mergeMap(stateVMap& svMapOut, stateVMap& svMapIn) {
  svMapOut.merge(svMapIn);
}

// custom reduction for omp code
#pragma omp declare reduction(                          \
                              mergeMap :                \
                              stateVMap :                   \
                              mergeMap(omp_out, omp_in) \
                              )                         \
initializer (omp_priv=omp_orig)


int main() {
  // set loop limits
```

```
  const int loopEnd = 1 << 20;

  // instantiate stateVarMap
  stateVMap svMap;
  double sTime = omp_get_wtime();
#pragma omp parallel
  {
#pragma omp for reduction(mergeMap:svMap)
    for (int iElem = 0; iElem < loopEnd; ++iElem) {
      // create stateVar data structure for each elem
      stateVar* p_elemStateVar = new stateVar();
      // run integration loop
      integrateLoop(iElem, p_elemStateVar);
      // insert data structure into map
      svMap.insert({iElem, p_elemStateVar});
    }
  }
  double eTime = omp_get_wtime();

  cout << "Total time (s): " << (eTime - sTime) << endl;

  // loop over map and print each point
  // and delete created pointers
  stateVMap::iterator iter_svMap = svMap.begin();
  cout << "map length: " << svMap.size()
       << " correct is " << loopEnd << endl;
  for (; iter_svMap != svMap.end(); ++iter_svMap) {
    if (iter_svMap->second->getNumPts() != 18 &&
        iter_svMap->second->checkVals()) {
      cout << "Key: " << iter_svMap->first
           << " structLength: " << iter_svMap->second->getNumPts()
           << endl;
    }
    delete iter_svMap->second;
  }

  return 0;
}


void integrateLoop(const int iElem, stateVar* elemStateVar) {
  // number of integration points
  const int nPts = 18;
  for (int iPt = 0; iPt < nPts; ++iPt) {
    double x = double(iPt);
    // make value some function
    const double value = std::pow(x,3.0) + 5*x*x + 13;
    // add value to data structure
    elemStateVar->vals.push_back(value);
  }
}
```

Discuss your results here

This code encounters 2 error categories:

1. If the nested iterator `iPt` is not declared private, the code can skip points to add to the map. In practice this does not show up, since the iterator is declared within the parallel region and OpenMP automatically makes anything in there private.

2. If the map `svMap` is not private, each thread shares it. Sometimes this results in a segmentation fault because the code is trying to insert into the map while another may be reSorting it. Sometimes this results in a element not being added to the map, so `map.size()` ends up incorrect.

To fix issue (2) I can't just make svMap private or it will always return a map with zero size.

```
pragma omp for private(svMap)
Total time (s): 0.000262022
map length: 0 correct is 16
```

I can do lastprivate, but that is still incorrect. That just takes the value from the last thread to finish, not the complete thing.

```
pragma omp for lastprivate(svMap)
Total time (s): 0.000140905
map length: 2 correct is 16
```

What I need is a reduction clause that will merge all of the maps.

```
pragma omp for reduction(mapMerge:svMap)
Total time (s): 0.0644159
map length: 32768 correct is 32768
```

This was quite interesting. Implementing the reduction clause for the map was a cool experience and this is a real problem I hit in scientific computing. I had a better solution for that particular problem, but was curious if OpenMP alone could do it without critical sections. It seems that it indeed can.

I did not strictly use a data context, but reduction clauses imply private variables for each thread. The reduction clause is just a special case of a private variable.

Threads	Run1	Run2	Run3	Run4	Avg	Speedup
1	3.96231	3.97014	3.98751	3.96854	3.972125	1.
2	2.33374	2.32085	2.31983	2.33406	2.32712	1.7068845
4	1.50738	1.49919	1.49846	1.48885	1.49847	2.6507871

Looking at the speedup, I do not get very good parallel improvement for the case of 1048576 entries. This could be a results of bloat from the maps. These things need to keep themselves sorted and I am now created multiple entities and merge them.

Another option, which is what we ended up doing in the research code, is to create a vector of pointers to maps. The vector can be shared, because each thread will only operate on one element. Then each thread creates its own map and operates on it. This is likely a more efficient solution than the map business within.