Introduction to Parallel Computing with OpenMP

Furkan Enderer

Université de Montréal

frknndrr@gmail.com

June 28, 2016

Overview

Introduction

- A set of compiler directives and library routines for parallel application programmers.
- Greatly simplifies writing Multi-Threaded programs.
- Most of the constructs in OpenMP are compiler directives that start with #*pragma* such that #pragma omp construct [clause [clause]].
- Example: #pragma omp parallel num_threads(4).
- Function prototypes and types in the file : #include <omp.h>.

Configuring Visual Studio for OpenMp

1. Open the project's Property Pages dialog box.
2. Expand the Configuration Properties node.
3. Expand the C/C++ node.
4. Select the Language property page.
5. Modify the OpenMP Support property.

Resource available online : https://computing.llnl.gov/tutorials/openMP/
Resource is very precise and to the point. All the clauses, constructs that we will not discuss today can be found in the documentaation.

OpenMP Constructs

- Parallel Construct
- Loop construct
- Sections Construct
- Single Construct
- Task Construct

Examples: See helloworld.c for parallel construct. See constructs.c for loop, sections and single constructs.

How do Threads Interact?

- OpenMP is a multi-threading shared address model in which threads communicate by sharing variables.
- Unintended sharing of data causes race conditions: when the programs outcome changes as the threads are scheduled differently.
- To control race conditions: use synchronization to protect data conflicts.
- Synchronization is expensive thus, manipulate how the data is accessed to minimize the need for synchronization.

Creating Threads

- Unless the number of threads is specified, compiler uses the max number of threads specified.
- Not safe to set the number of threads more than available amount.

- **omp_get_num_procs()** to obtain number of processors.
- **omp_get_max_threads()** to obtain the number of threads available.
- **omp_set_num_threads(int)** to set the number of threads used throughout the program.
- **omp_get_thread_num()** can be called at any point in the program to obtain the thread ID executing that specific chunk of code.

Synchronization

Synchronization is used to impose order constraints and to protect access to shared data.

1. High Level Synchronization
   - critical
   - atomical
   - barrier
   - ordered
2. Low Level Synchronization
   - flush
   - locks (simple and nested)

Synchronization: Critical

- One of the most commonly used constructs in OpenMP.
- Mutual exclusion: Only one thread at a time can enter a critical region.
- If a thread is currently executing inside a CRITICAL region and another thread reaches that CRITICAL region and attempts to execute it, it will be blocked until the first thread exits that CRITICAL region.
- Critical region can contain anything, meaning that it can be formed of several functions or it can simply be an operation on several variables.

Example: see synchronization1.c

Synchronization: Atomical

- Atomic provides mutual exclusion but only applies to the update of a memory location.
- The statement under the directive can only be a single C assignment statement such that: x++, ++x, x– or –x.
- No other statement is allowed.
- Can also be tackled by "reduction" clause.

Example: see synchronization2.c

Synchronization: Barrier

- The BARRIER directive synchronizes all threads in the team.
- When a BARRIER directive is reached, a thread will wait at that point until all other threads have reached that barrier. All threads then resume executing in parallel the code that follows the barrier.

Example: see synchronization3.c

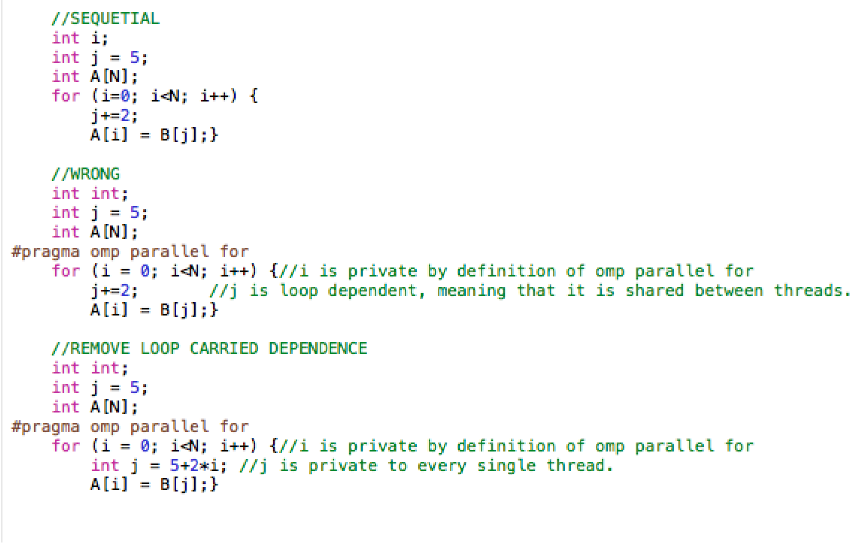
The SPMD Pattern

The most common approach for parallel algorithms is the Single Program Multiple Data pattern.
Each thread runs a single program, but using the thread ID, they operate on different data (Multiple Data) or take slightly different paths through the code.
In OpenMP this means:

- A parallel region near the top of the code.
- Pick up the thread ID and number of threads.
- Use the thread ID and number of threads to split up loops and select different data blocks to work on.

A better approach: Use the loop construct to share the workload in between threads.
Example: pisequential.c, piparallel1.c, piparallel2.c, piparallel3.c.

Working with Loops

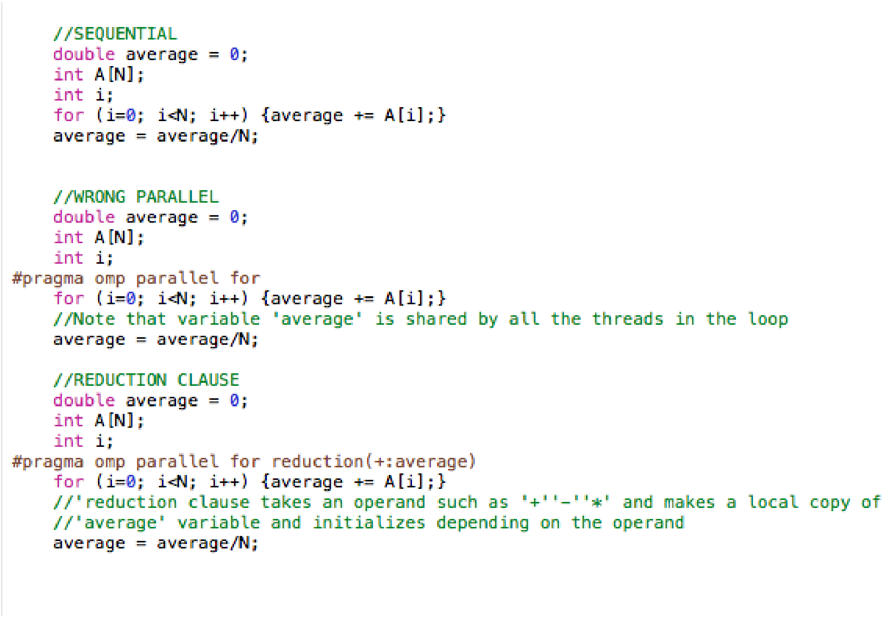


```
    //SEQUETIAL
    int i;
    int j = 5;
    int A[N];
    for (i=0; i<N; i++) {
        j+=2;
        A[i] = B[j];}

    //WRONG
    int int;
    int j = 5;
    int A[N];
#pragma omp parallel for
    for (i = 0; i<N; i++) {//i is private by definition of omp parallel for
        j+=2;       //j is loop dependent, meaning that it is shared between threads.
        A[i] = B[j];}

    //REMOVE LOOP CARRIED DEPENDENCE
    int int;
    int j = 5;
    int A[N];
#pragma omp parallel for
    for (i = 0; i<N; i++) {//i is private by definition of omp parallel for
        int j = 5+2*i; //j is private to every single thread.
        A[i] = B[j];}
```

Reduction



```
    //SEQUENTIAL
    double average = 0;
    int A[N];
    int i;
    for (i=0; i<N; i++) {average += A[i];}
    average = average/N;


    //WRONG PARALLEL
    double average = 0;
    int A[N];
    int i;
#pragma omp parallel for
    for (i=0; i<N; i++) {average += A[i];}
    //Note that variable 'average' is shared by all the threads in the loop
    average = average/N;

    //REDUCTION CLAUSE
    double average = 0;
    int A[N];
    int i;
#pragma omp parallel for reduction(+:average)
    for (i=0; i<N; i++) {average += A[i];}
    //'reduction clause takes an operand such as '+''-''*' and makes a local copy of
    //'average' variable and initializes depending on the operand
    average = average/N;
```

Private vs. Shared Variables

Any variable declared outside the parallel region is shared between the threads inside the parallel region.

Example:Any variable declared outside can be manipulated by 'private' and 'shared' clauses

```
int x,y;
#pragma omp parallel for private(x,y) shared(a,b)
for(i=0; i<N; i++){
read(a,b);
work(x,y);
#pragma omp critical{merge(x,y); }
}
```

- Variable *i* is private by default.
- *a* and *b* are shared because threads only read from it.
- Since *x* and *y* are independent of the loop, they must be private and then they must be merged with a critical in the end.

Private vs. Shared Variables

A better and a safer way is to declare private variables inside the foor loop. Especially if we are using complicated data structures that we update many times inside the loop.

Example

```
#pragma omp parallel for
for(i=0; i<N; i++){
int x,y;
read(a,b);
work(x,y);
#pragma omp critical{merge(x,y); }}
```

Private vs. Shared Variables

A better and a safer way is to declare private variables inside the foor loop. Especially if we are using complicated data structures that we update many times inside the loop.

Example

```
#pragma omp parallel for
for(i=0; i<N; i++){
int x,y;
read(a,b);
work(x,y);
#pragma omp critical{merge(x,y); }}
```

How can we get rid of 'critical'?

Private vs. Shared Variables

A better and a safer way is to declare private variables inside the foor loop. Especially if we are using complicated data structures that we update many times inside the loop.

Example

```
#pragma omp parallel for
for(i=0; i<N; i++){
int x,y;
read(a,b);
work(x,y);
#pragma omp critical{merge(x,y); }}
```

How can we get rid of 'critical'? Answer: Very code specific. If it's a simple (+,-,*) operand we can use reduction.

Private vs. Shared Variables

A better and a safer way is to declare private variables inside the foor loop. Especially if we are using complicated data structures that we update many times inside the loop.

Example

```
#pragma omp parallel for
for(i=0; i<N; i++){
int x,y;
read(a,b);
work(x,y);
#pragma omp critical{merge(x,y); }}
```

How can we get rid of 'critical'? Answer: Very code specific. If it's a simple (+,-,*) operand we can use reduction. However, if *x* and *y* are data structures (arrays, matrices etc.), we must use either critical or we use one of the techniques used in piparallel1.c and piparallel2.c

Computing π with a Dart Board

- Throw darts at the square.
- Chance of falling into circle is proportional to areas.
- $A_c = r^2 * \pi$.
- $A_s = 4 * r^2$.
- $P = A_c / A_s = \pi / 4$.
- Algorithm:
  1. Randomly choose points in a 2-dimensional space.
  2. Count the fraction that falls in the circle.
  3. Estimate π.
- Example: pirandomnumber.

Matrix Multiplication

Example: See matrixmultiplyseq.c and matrixmultiply.c

How Can We Profit from Parallelization in our Algorithms?

Branch-and-Bound:

1. Parallelize the tree search (CPLEX already does this)
2. Instead of processing one node at a time, process as much as you can by taking advantage of your computer structure.
3. Critical value here is the upper bound, cause it might be updated by several several threads at a time.

Column Generation:

1. Most Column Generation algorithms have decomposable subproblems.
2. Use this fact to deploy a parallel algorithm.
3. Solve as many subproblems as possible instead of solving one at a time.
4. Be careful with shared information (such as dual values coming from the master LP).

How Can We Profit from Parallelization in our Algorithms?

Benders Decomposition:

1. Similar to CG, most Benders subproblems are decomposable.
2. Solve as many subproblems as possible by assigning each thread a different subproblem.

Cutting Plane Algorithms:

1. At a certain iteration of a Cutting Plane algorithm, there exists not one but many valid inequalities that are violated by the current solution.
2. Use this fact to derive several VIs instead of generating one at a time.
3. Also, in many algorithms we have not one but several different separation algorithms (to derive VIs) and one can parallelize the algorithm such that certain threads will only execute certain seperation procedures.

And even Heuristics...

Neighborhood based Heuristics:

1. Single initial solution multiple neighborhoods - Parallelization in the search.
2. Multiple initial solutions single neighborhood - Parallelization in the initial solution scheme.
3. Multiple initial solutions multiple neighborhoods - Parallelization in both.

Operators used in population based Heuristics:

1. Selection
2. Crossover
3. Mutation
4. Fitness-evaluation

Any Questions?