When Should You Use OpenMP?

Goal

- Here, we are going to look at the higher level concepts associated with programming for shared memory
- Look at how to know when to use it
- Refresh: What is shared memory?

Introduction

- Shared memory parallelism provides a way of giving all processors access to the same memory while running in parallel
- A way to program for shared memory is OpenMP
- Here, we will be doing some in person exercises to better understand why we might want to use shared memory parallelism and OpenMP in our applications

Why Shared Memory Parallelism over Other Types?

- There are other types of parallelism (distributed memory parallel, hybrid shared/distributed memory parallel)
- Works well for problems that have enough computation and complexity that they would benefit from parallelism, but might not have enough complexity to maximize on multiple nodes (distributed memory) or all of the information being used needs to be shared among all processes and the communication across nodes would cause too much overhead (everything is shared in memory on a single node, hence shared memory parallelism)

Exercises

Exercise #1: Test Shared Memory Paradigm

- We need 4 volunteers
  - Person 1: Your number is 4
  - Person 2: Your number is 10
  - Person 3: Your number is 12
  - Person 4: Your number is 28
- We have 7 pieces of paper, each with a number on it. Each person’s goal is to use at least two numbers (or more) from the bank of pieces of paper to add up to the number you’ve been assigned

Exercise #1, cont.

- Let’s do this in iterations and see what problems arise
  - Iteration 1: Find numbers from the pile of numbers that add up to whatever your assigned number was

Exercise #1, cont.

- Let’s do this in iterations and see what problems arise
  - Iteration 2: The numbers are now scattered across the room. Before you can tell us what numbers you are going to be using to add up to your number, you must walk to each number and see what’s available. Then come back to your spot and write them on your piece of paper

Exercise #1, cont.

- Let’s do this in iterations and see what problems arise
  - Iteration 3: Now, the numbers are still scattered across the room, but this time, if you need a certain number for your task, you must take the number from its place, bring it to your spot, write it down, and take it back to its original location
  - If you need a certain number and it is not at its original spot because someone is using it, you must wait until someone brings it back before you can proceed

Exercise #1, cont.

- Let’s do this in iterations and see what problems arise
  - Iteration 4: This time, the numbers are still scattered across the room, and if you need a certain number for your task, you must take the number from its place, bring it to your spot, write it down, and take it back to its original location
  - If you need a certain number and it is not at its original spot because someone else is using it, you can choose to either wait for it or move on to a different number and come back. It is up to you what order you want to do the numbers in and whether you wait or move on to different numbers

Exercise #1, cont.

- Let’s do this in iterations and see what problems arise
  - Iteration 5: Now, the numbers are still scattered around the room, but this time you only have access to the numbers as follows:
    - Person 1: 1 and 2
    - Person 2: 5 and 4
    - Person 3: 6 and 7 and 3
    - Person 4: 1, 3, 5, and 7
  - Notice that the numbers that Person 4 was given are copies of the other numbers that other people have. Can they do their computations? What if we rearrange the numbers? What configurations allow people to accomplish their task? Why does/doesn’t this model work?

Exercise #1, cont.

- What did and didn't work about this exercise?
  - Remember, when everyone had access to the same information and were each allowed to use this information, this is a shared memory model because everyone had access to the same information. While they sometimes had to wait for the information to be updated and brought back to its location, they still had access to it
  - Once we restricted access to some data for each person, the problem didn't work anymore. Why is that? How could we possibly fix that?

Exercise #1, cont.

- Because everyone needed access to the full set of data that was available, the shared memory model made sense for this type of problem
- What type of problems wouldn’t work with a shared memory model? Can you think of types of problems that would work with a shared memory model?

Exercise #2: Parallelize Instructions

- In Exercise #1, 4 different math problems (additions) were done simultaneously. Each person that did a math problem can be thought of as a “thread”
- Goal of this exercise: do 2 of the tasks given in the following slides in parallel
- What steps should be done in parallel? These can’t be chosen randomly. Doing some steps in parallel don’t make sense or can’t be done

Exercise #2, cont.

ORIGINAL DATA SET: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

1. Initialize all of our necessary variables, arrays, etc.
    1. We will have an array called odds[] and an array called evens[]
2. Need to divide the original dataset into two smaller datasets□one is all of the odd numbers, the other all the evens. The final goal is to have:
    1. odds[] = 1, 3, 5, 7, 9, 11, 13, 15, 17, 19
    2. evens[] = 2, 4, 6, 8, 10, 12, 14, 16, 18, 20
3. Initialize 4 threads
4. Divide the dataset into 4 parts and pass the smaller datasets to each thread
    1. Thread 1: 1, 2, 3, 4, 5
    2. Thread 2: 6, 7, 8, 9, 10
    3. Thread 3: 11, 12, 13, 14, 15
    4. Thread 4: 16, 17, 18, 19, 20
5. Split the data that each thread is assigned into odds and evens and write the data into the odds[] and evens[] array

Exercise #2, cont.

ORIGINAL DATA SET: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

6. Add the contents of each of the arrays. The results will be single numbers
    1. odds[] addition result will be saved to the variable odds_result
    2. evens[] addition result will be saved to the variable evens_result
7. There are several ways to divide and conquer coming up with the results for odds_result and evens_result. What are some ways this can be done?
    1. How many steps do each of the different ways you came up with require? What way do you think is best? Why?
8. Write results to odds_result and evens_result
9. We now want to multiply the numbers saved to odds_result and evens_result. Write the result to a variable called multiplied
10. Print the single value result to the user

How many of these steps can be done in parallel?

Exercise #2, cont.

- We see here there are multiple ways to do some of the steps involved in the problem. This brings up an important concept: **Design of your Program**. Programming in parallel requires a different way of thinking and additional work to redesign the problem so it can be done in parallel

Exercise #3: What Gets Done First?

- The order in which things are done when using the shared memory model is important
- When using shared memory, keep in mind all threads doing computations in parallel have access to all of the data

Exercise #3, cont.

- Suppose we have our two arrays again (odds[] and evens[]) and they each contain the following numbers:
  - odds[] = 1, 3, 5, 7, 9, 11, 13, 15, 17, 19
  - evens[] = 2, 4, 6, 8, 10, 12, 14, 16, 18, 20
- Let’s use 2 threads. Both threads have access to both arrays

Exercise #3, cont.

- Each thread will be assigned to do computation with a different array, but they will both still have access to both arrays. Each thread will have 2 tasks
  - Thread 1: Add contents of odds[] and write the result to odds_result. Next, swap out the 5th element of the odds[] array with the 5th element of the evens[] array
  - Thread 2: Add contents of evens[] and write the result to evens_result. Next, swap the 5th element of the evens[] array with the 5th element of the odds[] array

Exercise #3, cont.

- If we did this by hand, our expected result would be:
  - odds[] = 1, 3, 5, 7, 10, 11, 13, 15, 17, 19
  - evens[] = 2, 4, 6, 8, 9, 12, 14, 16, 18, 20
- Depending on the order the instructions get executed, the results you get could be incorrect. What if Thread 1 finishes adding all of the contents of odds[] and switches the 5th element before Thread 2 finishes adding the contents of evens[]?

Important Note

- What if the rest of the program’s execution is determined by getting the correct results from previous computations? What if the result is incorrect after being run in parallel?
- Can you think of examples where it is crucial that instructions get executed in the correct order?
  - In serial, this isn’t a problem, but in parallel we must keep execution order in mind!