Shared Memory Concepts

Blue Waters Petascale Institute 2018

Week 1, Tuesday, May 22, 9:30am-10:00am

Lead Instructor: Colleen Heinemann, NCSA/UIUC

Goals

- Understand what Shared Memory is
- Understand why we would want to use Shared Memory and what it is used for
- Begin to think in a parallel mindset for a shared memory architecture

Introduction

Today will be spent discussing the concept of Shared Memory and how to program using a Shared Memory model known as OpenMP. Before we can dive in and begin programming for such an architecture, we need to understand not only what is meant by Shared Memory, but also what is happening behind the scenes when programming specifically for Shared Memory. Once we conceptually understand what is happening, it is much easier to program for such a model.

Exercise

1. We need 4 volunteers
    a. `Person 1`: Your number is `4`
    b. `Person 2`: Your number is `10`
    c. `Person 3`: Your number is `12`
    d. `Person 4`: Your number is `28`
2. 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 a different number that you've been assigned
3. Let's do this in iterations and see what problems arise
    a. `Iteration 1`

        Find numbers from the list that add up to whatever your assigned number was

    b. `Iteration 2`

        The numbers are now scattered across the room. Before you can tell us the numbers are that you are using, you must walk over to each number. Then come back up to your spot and write them down on your piece of paper

    c. `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 then 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 must wait for it until someone brings it back before you can use it

d. `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 then 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 until someone brings it back or you can move on to a different number and come back to it. If all of the numbers left that you need are taken, you must wait for them to come back. It is up to you what order you do them in and whether you wait or move on to different numbers

e. `Iteration 5`

Now, the numbers are still scattered around the room, but this time they are grouped into 4 different groups of numbers. Each of you now only have access to one set of 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?

Let's think about this model and what did and didn't work about it. When everyone had access to the same information and were each allowed to use the information, this was a shared memory model. Everyone had access to the same information so that they can each use it. While there were times that someone had to wait for information, they still had access to it.

Once we restricted access to some data for each person, then the problem didn't work anymore. Why is that?

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 other types of problems that would work with a shared memory model?

Thinking How to Program in Shared Memory Parallel

Before we can actually write code to program in parallel using shared memory, we need to think about how we would not only write this code, but how this code will work as well. In the example we saw earlier, the idea was to do 4 different additions simultaneously. We can think of each person that was doing a math problem as a "thread", which we will discuss in detail later.

Let's think about the following workflow. Our final goal is to have done 2 of the following different tasks in parallel. Can you figure out which of the following steps could be done in parallel? Keep in mind that we are using the shared memory model, where everything has access to all of our data. We also can't just pick and choose randomly which steps get done in parallel because doing some things in parallel does not make sense, so we need to choose carefully.

Ignore any programming syntax that may make the problem more difficult or that you feel would change the problem. We are just thinking conceptually at the moment. We will worry about programming later.

`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.
    a. We will have an array called `odds[]` and an array called `evens[]`
2. Now we need to divide the original dataset into two smaller datasets→ one will be all of the odd numbers and the other will be all of the even numbers. The final goal is to get the following:
    a. `odds[]` = 1, 3, 5, 7, 9, 11, 13, 15, 17, 19
    b. `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, 2, 3, and 4
    a. Thread 1: 1, 2, 3, 4, 5
    b. Thread 2: 6, 7, 8, 9, 10
    c. Thread 3: 11, 12, 13, 14, 15
    d. Thread 4: 16, 17, 18, 19, 20
5. Split the data into odds and evens and write the data into the `odds[]` and `evens[]` arrays. The result is:
    a. `odds[]` = 1, 3, 5, 7, 9, 11, 13, 15, 17, 19
    b. `evens[]` = 2, 4, 6, 8, 10, 12, 14, 16, 18, 20
6. The next step is to add the contents of each of the arrays. The results will be single numbers.
    a. `odds[]` addition result will be saved to the variable `odds_result`
    b. `evens[]` addition result will be saved to the variable `evens_result`

7. There are two ways to divide and conquer coming up with results. One would require two threads and the other would require all 4 threads that we have allocated. What are these two ways?
    a. Way 1:
    b. Way 2:

    How many steps do each of the different ways require? Which way do you think is better? Why?
8. Write results to `odds_result` and `evens_result` respectively
9. We now want to multiply the numbers saved to `odds_result` and `evens_result`. We will write the result to a variable called `multiplied`
10. Print the single value result to the user

We saw here that there are multiple ways to do some of the steps involved in this problem. This brings to light a very important concept: **Design of Your Program**. Programming in parallel requires a different way of thinking and this problem required additional work to redesign it so that it would work in parallel

What Gets Done First?

One last concept that we will look at is the idea of the order that things are done in when using the shared memory model. Keep in mind that when using shared memory, all of the threads doing computations in parallel have access to all of the data.

Let's think about this example:

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

In this case, let's use 2 threads. Because we are using shared memory, both threads have access to both arrays. Let's use the following instructions:

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

a. Thread 1: Add up all of the 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 (the 5th element of the `odds[]` array should equal whatever the 5th element of the `evens[]` array is)
b. Thread 2: Add up all of the 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 (the 5th element of the `evens[]` array should equal whatever the 5th element of the `odds[]` array is)

If we were to do this by hand, our expected result at the very end of this would be:

`odds[]` = 1, 3, 5, 7, 10, 11, 13, 15, 17, 19

`evens[]` = 2, 4, 6, 8, 9, 12, 14, 16, 18, 20

However, depending on the order that instructions get executed, the results that you get could be incorrect. What if Thread 1 finishes adding all of the contents of odds[] and switches the 5th element before evens[] finishes adding all of the contents of evens[] and switches the 5th element? Then the 5th element would be the same in both of the arrays. What if one of the threads finishes all of its additions and swaps the 5th element before the second thread even gets around to adding the 5th element in its respective array? Then the result in its array's result variable would be incorrect as well.

What if the rest of the program's execution is determined by getting the correct results from previous computations that were run in parallel? This shows that we have to be very careful about what order instructions get executed. Can you think of examples other than the one that we just talked about where it is crucial that instructions get executed in the correct order?