**Parallel Programming Report**

**Foundation**

**Define the following: Task, Pipelining, Shared Memory, Communications, Synchronization. (in your own words):**

Task: A program or set of instructions that does a certain task in a program like way. When you split up a program with parallelism, the program is broken into separate tasks.

Pipelining: A way to try to keep as many parts of a single processing unit busy. When some instruction is being used by some parts of the processor, but not others, another instruction that requires the idle parts will start to process and use those parts.

Shared Memory: When all processors, and the parallel tasks on them access the same memory and can read/write it.

Communications: The way in which the parallel tasks exchange information and data to come to the conclusion of the program

Synchronization: When there are parallel tasks, some tasks finish before others but the next step may require all tasks to be done first. Therefore some tasks need to wait and when all are done they are “synchronized” to go onto the next step.

**Classify parallel computers based on Flynn's taxonomy. Briefly describe every one of them**

Single Instruction, Single Data(SISD): A computer that is serial(goes step by step without parallelism). There is only one instruction and data stream used at once. This is the way the oldest computers ran.

Single Instruction, Multiple Data(SIMD): Every processing unit does the same instruction, but can act on different data. This is good for tasks where it will be very repetitive. Graphical processing units often make use of this

Multiple Instruction, Single Data(MISD): One data stream will be acted upon by different types of instructions. Very few examples of this, but one might be trying to decipher cryptography. The cryptic message would be the data, and the instructions would be different deciphering techniques.

Multiple Instruction, Multiple Data(MIMD): The processing units can have different types of instructions operating on different data streams. In modern day computing, this is the most popular.

**What are the Parallel Programming Models**

Shared Memory(without threads), Threads, Distributed Memory/Message Passing, Data Parallel, Hybrid, Single program multiple Data(SPMD), Multiple Program Multiple Data(MPMD)

**List and briefly describe the types of Parallel Computer Memory Architectures. What type is used by OpenMP and why**

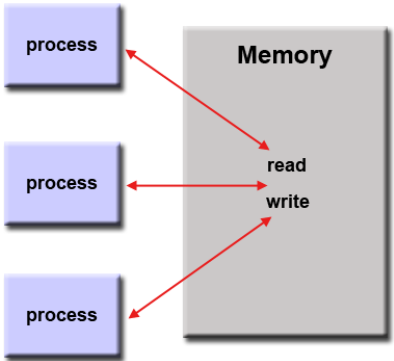
Uniform Memory Access(UMA): Identical processors all having equal access to memory. Commonly represented by Symmetric Multiprocessor (SMP)

Non uniform Memory access(NUMA): Linking of multiple SMPs. Not all processors have equal access time to all memories.

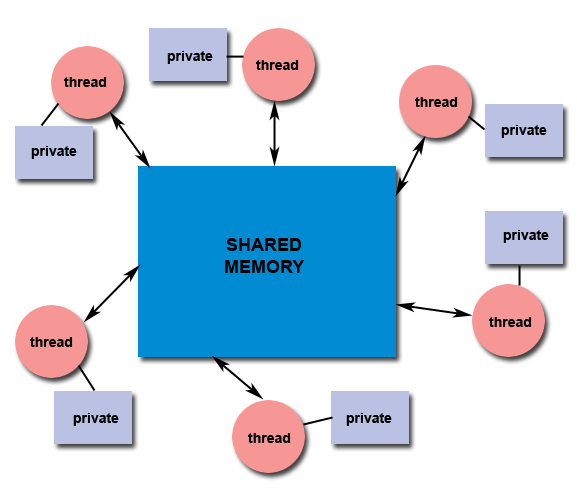
Openmp is using UMA. The program is broken into threads put onto multiple processors. The processors should all have equal access to the memory in order to synchronize well to finish the program. UMA has equal access and access times between the processors.

**Compare Shared Memory Model with Threads Model? (in your own words and show pictures**

Shared Memory(without threads): Processes all use the same address space. It is a simple parallel programming model, but it becomes difficult to manage the data since each process has little restriction.



Shared Memory(with threads): The main program has a memory space it uses, but this program is broken in to smaller threads. These threads each have local data they can use while still being able to access the more global data of the main process. This gets rid of the problem of processes messing with each other and disrupting the program like in shared memory without threads.



**What is Parallel Programming**

Having multiple processes going on at the same time, sometimes related processes and sometimes unrelated processes.

**What is system on chip(SoC)? Does Raspberry PI use system on SoC?**

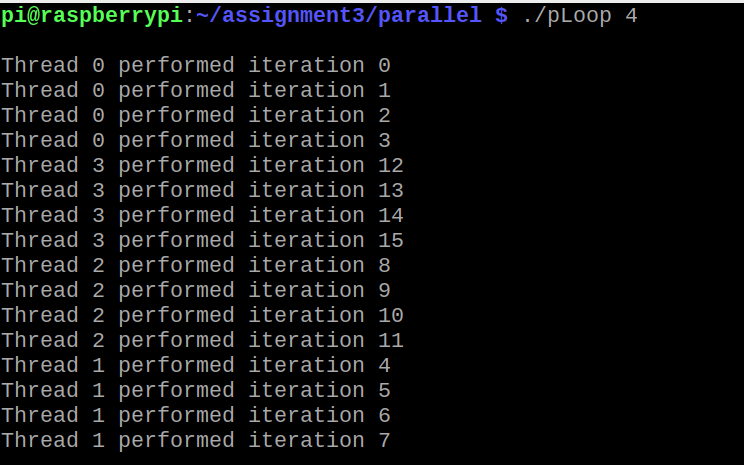
A chip that combines the cpu, the gpu, memory, a usb controller, power management circuits, and wireless radios all into one. This is opposed to the CPU which is normally more separated from everything else on most computers. The Raspberry PI uses a Broadcom System On Chip(SOC)

**Explain what the advantages are of having a System on a Chip rather than separate CPU, GPU and RAM components.**

The SOC conserves a lot of space and uses less power. This would allow for entire computers that are normally large to become easy to transport. Since there are less physical separate chips, the computer will be cheaper as well.

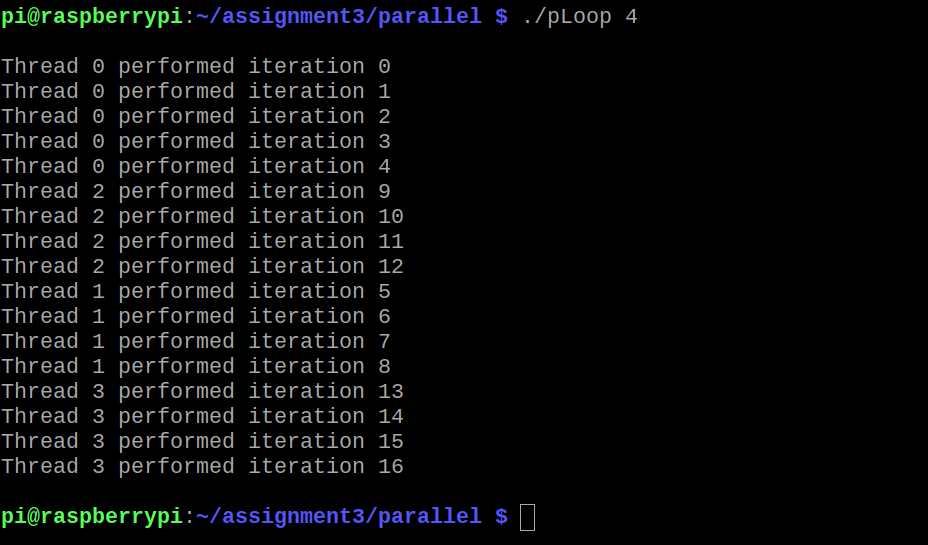
**Parallel Programming Basics**

**Questions and explanations related to the first program: parallelLoopEqualChunks.c**



Here the iterations of the for loop were split between the threads, as opposed to each thread doing 16 iterations like it would without the for statement(in #pragma omp parallel for). The behavior is what is expected based on the diagram we were supposed to compare to in the instructions.

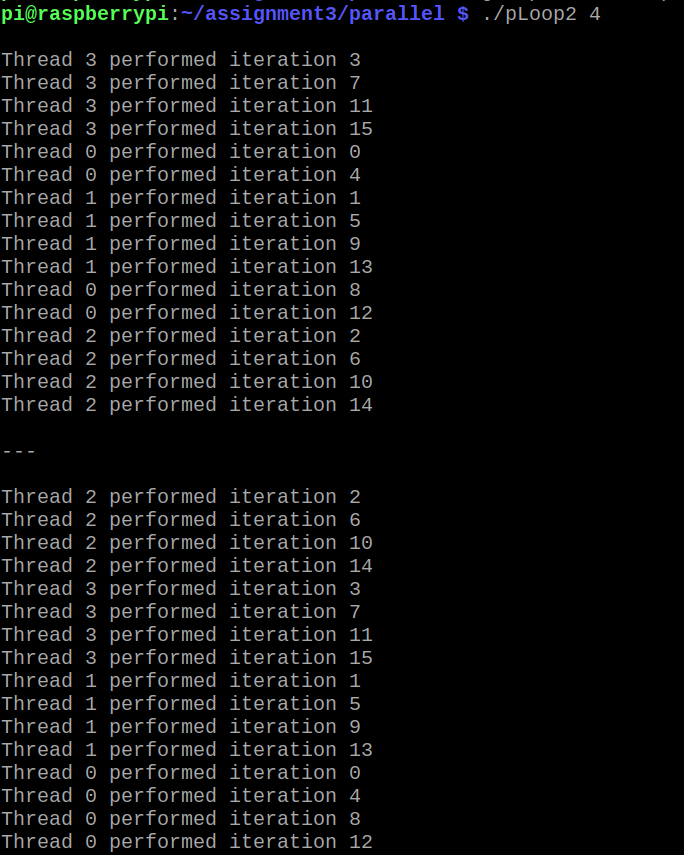
**Change number of iterations to something not evenly divisible by the number of threads:**

****

I changed the number of iterations to 17 which is not divisible by 4. This time 0 got the additional iteration. I tried with other number such as 18 and 19. The pattern seemed to be this:

Any multiple of 4 would result in every thread getting an equal number of iterations. When it was not a multiple of 4, first find (Number of iterations)%(number of threads). Assuming threads = 4, then this will result in either 0,1,2, or 3. 1 means 1 iteration past a multiple of 4, and this is given to thread 0. 2 means two past, so thread 0 gets an iteration and thread 1 gets the other iteration. 3 means thread 0,1, and 2 will get an iteration.

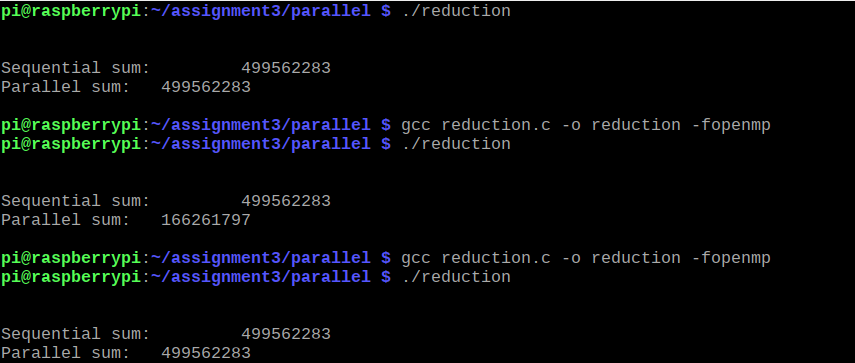
**Questions and explanations related to the second program: parallelLoopEqualChunksOf1.c**



Here is the output when you also uncomment the commented code that does not make use of the schedule(static, 1) clause. Like the comment message at the top of the file says, both methods give the same result.

The difference between this program and the previous program is which threads get which iterations. In the previous program, each thread got consecutive iterations. In this program the iterations were given one by one sequentially to the next thread up(and back to thread 0 when the last iteration was given to thread 3). The schedule(static, 1) means that the chunks are given one at a time in size 1 iteration in the way described above. If schedule(static,2) were written, the chunk size would be 2, so thread 0 would get iterations 0,1, thread 1 would get 2,3, thread 2 would get 4,5 and so on. The number could be changed to 4 as well which would actually give the same output as the first program.

**Questions and explanations related to the third program: reduction.c**

****

The program initialized an array of size 1,000,000 with 1,000,000 random numbers through the initialize function. Then the sequentialSum and parallelSum functions are called.

The first run of the code is when the entire #pragma line is commented out in the parallelSum function. This means that the parallelSum function does the same thing as the sequentialSum function.

The next run is when the #pragma line is uncommented up to the reduction clause in the parallelSum function. The output shows different numbers for each, meaning the parallel sum is produced an incorrect number. The reason this happened is because all the threads are trying to access the sum variable. Some threads may try to access sum at the same time creating what is called a “data race”. This causes unforeseen effects and thus we get a wrong number from the parallelSum function.

The next run is when the reduction clause is uncommented too. This solves the issue talked about above. Now each thread gets its own local copy of sum, getting rid of the problem of multiple threads trying to access a memory location at the same time. After each thread has gone through its partition of the array and summed them, all the sums are brought together and stored in the sum variable. As we can see now the parallelSum function produces the same number as the sequentialSum function.