Computer Science 605.611

Problem Set 12

1. (5) A multiprocessor system contains 10 identical processors each of which accesses a separate memory so that tasks executing on different processors do not interfere with each other. A program must complete 268 identical independent tasks each of which requires 4 million cycles to execute on a single processor. None of the tasks can be further subdivided. To employ as much parallelism as possible, what integer number of tasks should be assigned to each processor so that the total number of cycles required to complete all 268 tasks is minimized? Explain your answer.

This is a relatively simple thing to determine. If we have 268 identical tasks and 10 processors we can do 268/10 = 26.8 tasks per processor. However, a task cannot be split into .2 and .8. Thus, we cand find that if we were to split it up by 26 we would have 8 tasks left over. Thus 8 processor should be assigned 27 tasks to complete and 2 processor should be assigned 26 tasks to complete.

2. (5) A uni-processor system contains a 1 GB local memory and a larger 2.5 GB remote memory (1 GB = 2^30 bytes). The local memory has a 4-cycle access time and each access to the remote memory requires 200 cycles. Hence the system is a non-uniform memory access (NUMA) system. The system clock rate is 2 GHz (2 billion cycles per second). Program1 runs on this system and makes a total of N memory accesses. Seventy-five percent of Program1’s accesses are to the local memory and all of its remaining accesses are to the remote memory. Based just on the memory accesses, what speedup is provided for the system by reducing the number of cycles required for each remote memory access from 200 cycles each down to 120 cycles each? Round your answer to two decimal places. Speedup = \_\_1.61\_\_.

3. Program2 consists of a purely sequential part and a parallel part. The sequential part must execute first and requires 20 seconds on a single-core processor. The parallel part executes after the sequential part is complete. The parallel part contains 4 independent tasks. The four indivisible tasks take 12, 10, 9 and 7 seconds, respectively, to run on the single core.

a) (5) What is the total number of seconds required to execute the complete program on the single-core processor? Total execution time = \_\_\_58 seconds\_\_.

We know that the sequential part takes 20 seconds. The parallel part can be determined by just adding up the other tasks to get 38 seconds. Adding these together gives us the execution time.

b) (5) How many seconds are consumed by Program2 if it runs on a 4-core version of the processor? On the 4-core system, after the sequential part has executed on a single core, each of the four independent tasks executes in parallel on a different core. Base your answer on the execution times given in the problem and express your answer as an integer.

The sequential part will still take 20 seconds. Since each task can be run in parallel the total time for parallel would be 12 seconds (longest task). Thus we would get **38 seconds**.

4. Program3 is made up of three independent parts A, B, and C. The three parts can execute in parallel on separate processors within a 4-processor SMP (symmetric multiprocessor system). All processors run at the same 2 GHz clock rate. None of the independent parts can be further subdivided, each part must execute on a single processor. Part A requires 79 billion cycles to complete, part B requires 51 billion cycles to complete and part C requires 62 billion cycles to complete.

a) (3) How many seconds does it take to execute all 3 parts of Program3 if only one of the processors is used to execute the program and the other 3 processors remain idle?

To determine this we can add up each section. 79 + 51 + 62 = 192 billion cycles to complete every part. Since 2Ghz is 2 billion instructions a second we can find that 192/2 = **96 seconds**.

b) (3) What is the minimum number of seconds required to execute all 3 parts of Program3 if only two of the processors are used to execute the program and the other 2 processors remain idle?

To determine this, we need to find the shortest time based on how we split up the parts. We can determine the best way by simply putting the two largest parts in the parallel execution time and then starting the final part with the remaining time. So Part A and C are thrown in first, once C is complete you can then add part B into the second processor. Thus C + B = minimum time, so 25.5 + 31 = **56.5 seconds**.

c) (3) Compared to executing the entire program on just one processor, what speedup is achieved by using all 4 processors to execute Program3?

If we can utilize all 4 processors the time it would take would be the longest part, so time = 39.5 seconds. Then if we do 96/39.5 = **2.43**

5. The code below runs on a non-pipelined single-cycle processor and implements a function (max) that accepts input arguments in registers $a0 and $a1. Since this is not a pipelined system, it does not employ delayed branching. Register $a0 contains a count of the number of elements in an integer array. Register $a1 contains the address of the first element within the array (i.e., the starting address of the array). The function scans the array and checks each element to determine the largest element. The function uses register $v0 to return the largest array element. Assume that, by chance, the array elements are already arranged in increasing order so that each element is less than the next element.

max:

lw $v0, 0($a1) ; use the first array element as the current max

loop:

addiu $a0, $a0, -1 ; decrement count of remaining elements

bgtz $a0, check ; check next element if there is one

jr $ra ; return if all elements have been checked

check:

lw $t0, 4($a1) ; get the next element as a candidate maximum

slt $t1, $v0, $t0 ; is current max < candidate?

beq $t1, $0, skip ; no, then skip to next element

or $v0, $0, $t0 ; else use candidate as new current max

skip:

addiu $a1, $a1, 4 ; point to next array element

j loop ; repeat for any remaining elements

a) (5) Running on a single-core system, how many clock cycles are required by this code to obtain the largest element in an array of 150 elements?

To determine this we need to look at how many clock cycles does the loop take. Since we never have an instance where the next element is less than the previous element we never go to the skip clause. So we can determine there would be 6 clock cycles taken inside the loop (one for each instruction), so we would have 149 iterations \* 6 = 897 + 3 (initial element). **So 897 clock cycles** are required.

b) (5) Assume that a separate copy of the code runs on each of the cores in a 4-core system. The 150-element array is partitioned into separate subsets, which can be processed in parallel by the four cores in a minimum number of cycles. Each core returns the largest element in its subset. The results returned are partitioned again into separate subsets which are then processed in parallel. This continues until the largest element in the entire array is determined. Using the code shown above, what is the minimum number of cycles required to obtain the largest element in the 150-element array on the 4-core system?

This is a little more tricky than the previous answer. To do this we need to think about how everything is split up. I will assume that the subsets of the data are 37, 37, 37, 39. Since it still takes 6 cycles to complete the code we would look at the longest value, 39. 3 + (39 -1) \* 6 = 231 clock cycles. Then we need to add 18 extra cycles due to the final comparisons, **so 249 cycles**.

c) (5) Suppose that each core in an 8-core system executes a separate copy of this code to determine the largest element in an array by applying the code to equal subsets of the original array. Compared to a single-core system, what speedup does the 8-core system provide when it is used to obtain the maximum element in the array if the number of elements is 100,000? Again, the elements happen to be in ascending order. Round your answer to two decimal places. Speedup = \_8.64\_\_.

6. A dual processor SMP system includes an L1 data cache for each processor and employs the MESI protocol to maintain cache consistency. Data can be transferred between each cache and its associated processor or with the shared memory. However, there are no data transfers between the two caches. Each cache is a 2-way set associative copy-back cache that contains a total of 8192 cache lines. All cache lines are 256 bytes in size. The ways within each empty set are filled from lowest to highest (way0, way1, etc.). A write-allocate policy is used for each cache. One software process, P1, runs on the first processor in parallel with another software process, P2, that runs on the other processor.

a) (3) Both caches are initially empty, so each cache line starts out in the invalid (I) state.

P1 makes the first memory access by writing the value 80 to a variable X at address 0x400804C0. Which set is affected in P1’s cache and what is the MESI state for the affected line after the write? Which set in P2’s cache is affected when P1 writes to X? Explain your answer.

To determine which set is affected, we convert 0x400804C0 to binary and look at the 12 bits above the lowest 8 bits. This gives us the **set 2052** is affected in P1’s cache. The MESI state for the affected line is M since it was modified. And the effect on P2’s cache is that its not directly affected, however the MESI protocol would ensure coherence between the two caches.

b) (3) After P1 writes X, P2 then writes the value 156 into a variable Y at memory address 0x400804F8. What action should be taken for P1’s cache in response to the write by P2, and what is the MESI state for any affected line in P1’s cache after P2 writes to Y? Explain your answer.

Since P2 writes the value 156 into Y at that memory address, P1 should change the MESI state to I since it may no longer be valid. Any affected line should be changed to I.

c) (3) What set is affected in P2’s cache when P2 writes the value 156 into Y and what is the MESI state for the affected line in P2’s cache after the value 156 is written into Y? Explain your answer.

This hex address actually links to the same set of 2052. Since the value in that location is being changed, the MESI state would be changed to M.

7. a) (5) A certain program consists of a serial (i.e., sequential) part and a parallel part. On a single core, the serial part requires 286 ms to execute and the parallel part requires 894 ms to execute. Assume that the parallel part must be completed after the serial part and the parallel part can be evenly divided among 8 processor cores.

The sub module 12B: mod12\_1.2 Parallel Performance explains Amdahl’s law as it relates to multi-processor systems. What speedup ratio does Amdahl’s law predict for this program using the 8 cores compared to using a single core? Show how you obtained your answer.

Lets start by looking at the data we already have. The serial part takes 286 ms and the parallel part takes 894 ms. This means that the total time is 1180 ms. We then need to calculate parallel time/total time = ~0.758. Amdahl’s law is 1/(1-total time) + (total time/num of cores). So 1/((1-0.758) + (0.758/8)) = 1/(0.242 + 0.09475) = 1/0.33675 = **2.97 speedup.**

b) (5) Problem 5 d) on Module 12 Example Set 4 defines and illustrates Gustafson’s law. What speedup ratio does Gustafson’s law predict for this program using the 8 cores compared to using a single core? Show how you obtained your answer.

We know that speedup is (1-f) + (f \* N). Based on the previous answer we know that f = 0.758. Thus (1-0.758) + (0.758 \* 8) = **6.306 speedup**

8. a) (5) A certain program executes in 320 seconds on a single-core system. A proposal is made to switch to a 16-core system. If the program can be evenly divided among the 16 cores, the potential speedup of 16 would provide an execution time of only 20 seconds for the program. However, the actual observed running time for the program on the 16-core system is 120 seconds. Some fraction (f\_parallel) of the original execution time on the single-core system is due to a parallel part of the program that can be evenly divided among the 16 cores. The remaining part of the program must be executed on one core. Compute the numeric value for the fraction f\_parallel and show your computation.

To determine this we have a few things to determine. Tparallel = f\_parallel \* T and Tserial = (1-f\_parallel) \* T. for 16 cores we would need to find T’ = (f\_parallel\*T/16) + (1 – f\_parallel) \* 320 = 120. This simplified down leaves us with **f\_parallel = 0.667**

b) (5) Suppose that it is desired to achieve a speedup of 80 for some other computation by switching from a single-core system to a 100-core system. Let f\_sequential be the fraction of the total time that corresponds to the sequential part of this new computation. What is the maximum numeric value for f\_sequential? Round your answer to 6 decimal places.

Maximum value for f\_sequential = \_\_0.002525\_\_

So we want a speedup of 80 and have 100 cores. Speedup = 1/((f\_seq) + (1-f\_seq)/N) = 80 = 1/((f\_seq) + (1-f\_seq)/100) = 0.002525

9. A program executes on a multiprocessor system and computes a vector product containing one hundred 32-bit integer elements P[i]. Each element in the vector product is computed as P[i]=A[i]\*B[i] where A and B are both vectors that contain a hundred 32-bit integers. It takes 1 clock cycle to compute each element in the array P. After the vector product is computed, the program then computes a matrix product by multiplying every 32-bit integer element in a matrix by a 32-bit integer constant. The matrix contains 1000 rows and 1000 columns. Multiplying a single matrix element by the constant takes one clock cycle.

a) (5) What is the total number of clock cycles required by the single processor to compute both products: the vector product followed by the matrix product?

To determine the number of clock cycles required by a single processor we will find the vector product and matrix product. The vector product is simple, each calc P[i]=A[i]\*B[i] takes 1 cycles, and there are 100 elements thus 100 clock cycles. To do the matrix product we simply multiply rows and columns number, giving us 1,000,000 elements, each taking 1 clock cycle. So for the single processor it will take **1,000,100 clock cycles**.

b) (5) The same program is executed again, this time with a total of 200 identical processors. There are no memory conflicts caused by the processors sharing memory and there are no other dependencies, so no stalls are necessary. If the work load is divided evenly among the 200 processors, what is the total number of clock cycles required to compute the vector product followed by the matrix product on the 200-processor system?

We will follow the same method as the previous answer. However, we can split the answer between 200 processor. For Vector product there are more elements than processors thus it should only take 1 clock cycle. To determine matrix product we can take the 1,000,000 and divide by 200 leaving us with 5000 clock cycles. So for 200 processors it would only take **5001 clock cycles**.

10. (5) A multiprocessor system contains 32 nodes. Each node has a processor and a local memory. All processors have a 0.5 ns cycle time. When executing a certain application, the multiprocessor system achieves an effective or average CPI of one instruction per cycle when there is no communication required between nodes. Each instruction that requires communication with a different node incurs a penalty of 400 extra nano-seconds.

Suppose that 0.4% of the instructions that are executed will communicate with other nodes. None of the remaining instructions communicate with other nodes. What speedup is provided by somehow eliminating all communication between nodes?

We know that cycle time is 0.5ns, CPI without comms is 1 instruction per cycle, comm overhead is 400ns, and 0.4% of instructions require comms. To find the speedup we need to calculate the execution time with comms and execution time without comms. We can easily find execution time without comms by taking 1billion instructions \* 0.5ns = 500mil ns. To find the execution time with comms we need to determine that there are 4 million instructions that take 400ns (0.4% of instructions). So non comm instructions take 498 mil ns, instructions that communicate take 1.602 billion. Thus, we have a speedup of 0.238, or 4.2 times faster.

11. (5) Explain the difference between weak scaling and strong scaling when increasing the number of processors used for some problem.

The difference between weak scaling and strong scaling when increasing the number of processors used for some problem is like this. Weak scaling refers to when the problem size and number of processors scale up proportionately. Strong scaling on the other hand is when the problem is a fixed size as the number of processors increases.

12. (7) A uniprocessor system accesses memory over a 32-bit bus. The processor has a write-back (as opposed to write-through) direct mapped data cache that operates in look-through mode and employs a write-allocate policy. The data cache contains 65536 lines, each of which is 512 bytes in size. Detecting a cache miss, or performing a cache read, or performing a cache write for the data cache takes 2 CPU clock cycles each. Loading a memory block into the data cache or writing a data cache line back to memory takes 400 CPU clock cycles. Recall that for a cache miss, the data item that is needed from the memory block is transferred to the CPU in parallel with loading the memory block containing the data item into cache.

The code shown below reads and updates each of the 33554432 four-byte elements in an integer array. The memory address of the array is 0x10084000.

Text

Description automatically generated

Each cache line can hold 512/4 = 128 array elements. The entire array will occupy 33554432/128 = 262144 cache lines. Hence, processing the entire array will require filling the cache 262144/65536 = 4 times. After the first cache fill, lines will have to be replaced by new memory blocks. This requires writing the old line back to memory before loading the new memory block into the cache line.

To process the array in less time, it is decided to execute the code on a 4-core system in which each core has a separate data cache identical to the data cache for the uniprocessor. Each of the 4 cores processes a separate contiguous subset of elements by executing a separate copy of the above code sequence. Each subset corresponds to 1/4th of the array. A speedup factor of 4 for the 4-core system compared to the uniprocessor is defined as “linear” speedup.

Assume that all data caches are initially empty. Based on the time required to read and update the array elements, what is the actual speedup provided by the 4-core system? Ignore the time required to fetch and execute the instructions and base your answer just on the times required for the memory accesses, cache accesses, cache line fills and cache line replacements. The final cache flush when the program ends is handled by the operating system (so the time required for the final cache flush should also be ignored).

First we calculate the total cycles of the uniprocessor system:

For each line we can get 2 (miss) + 400 (load) + 2 (read) + 2 (write) + 400 (write-back) = 806 cycles. Since there are no write backs in the first fill, 2 (miss) + 400 (load) + 2 (read) + 2 (write) = 406 cycles. So 65,536 lines × 406 cycles = 26,617,856 cycles for first fill. And 3 fills × 65,536 lines × 806 cycles = 158,605,312 cycles. So the total number of cycles is 26,617,856 (first fill) + 158,605,312 (subsequent fills) = 185,223,168 cycles

Then we need to do a similar process but for the 4 core system:

Since this is split between 4 cores, 65,536 lines × 406 cycles = 26,617,856 cycles each core/total.

Thus, 185,223,168 cycles / 26,617,856 cycles = **6.96** = speedup.