CENG3420 Homework 4

NO need to submit

Solutions

Q1 (25%) Given an original code as follows:

```
          ADD R5,R0,R0
Again:    BEQ R5,R6,End
          ADD R10,R5,R1
          LW R11,0(R10)
          LW R10,4(R10)
          SUB R10,R11,R10
          ADD R11,R5,R2
          SW R10,0(R11)
          ADDI R5,R5,8
          BEQ R0,R0,Again
End:
```

1. What is the speedup of going from a 1-issue processor to a 2-issue processor from Figure 4.69 in textbook? In other words, no forwarding unit contained in the 2-issue processor. Assume that 1,000,000 iterations of the loop are executed, that the processor has perfect branch predictions, and that a 2-issue processor from Figure 4.69 in textbook can fetch any two instructions in the same cycle.
2. Rearrange the code to achieve better performance on a 2-issue statically scheduled processor.

A1 1. 1-issue:
1.11 (10 cycles per 9 instructions). There is 1 stall cycle in each iteration due to a data hazard between the second LW and the next instruction (SUB).
2-issue: 1.06 (19 cycles per 18 instructions). Neither of the two LW instructions can execute in parallel with another instruction, and SUB stalls because it depends on the second LW. The SW instruction executes in parallel with ADDI in even-numbered iterations.
Speedup: 1.05

2. The only way to execute 2 instructions fully in parallel is for a load/store to execute together with another instruction. To achieve this, around each load/store instruction we will try to put non-load/store instructions that have no dependences with the load/store.

```
ADD R5,R0,R0
Again:  ADD R10,R5,R1
BEQ R5,R6,End
LW R11,0(R10)
ADD R12,R5,R2
LW R10,4(R10)
```

```
ADDI R5,R5,8
SUB R10,R11,R10
SW R10,0(R12)
BEQ R0,R0,Again
End:
```

(Note that we are now computing a+i before we check whether we should continue the loop. This is OK because we are allowed to "trash" R10. If we exit the loop one extra instruction is executed, but if we stay in the loop we allow both of the memory instructions to execute in parallel with other instructions)

Q2 (**20%**) In the design of a multi-core processor, there are fixed on chip cache resources. We assume maximum of n cores can be designed with those resources. Let k be the real designed core number ($r = \frac{n}{k}$ is integer.) Define a speed up factor $s(r)$ as sequential performance gain by using the resources equivalent to r cores to form a single core, and obviously $s(1) = 1$. Given f the fraction of software that is parallelizable across multiple cores, prove the speed up of the multi-core processor in terms of f, r, n, and $s(r)$ is

$$S(f, r, n) = \frac{1}{\frac{1-f}{s(r)} + \frac{f \times r}{n \times s(r)}} \tag{1}$$

A2

$$\begin{aligned} S(f, r, n) &= s(r) \times \frac{1}{(1-f) + \frac{f}{k}} \\ &= s(r) \times \frac{1}{(1-f) + \frac{f \times r}{n}} \\ &= \frac{1}{\frac{1-f}{s(r)} + \frac{f \times r}{n \times s(r)}}. \end{aligned} \tag{2}$$

Q3 (25%) Consider the following recursive mergesort algorithm (another classic divide and conquer algorithm). Mergesort was first described by John Von Neumann in 1945. The basic idea is to divide an unsorted list x of m elements into two sublists of about half the size of the original list. Repeat this operation on each sublist, and continue until we have lists of size 1 in length. Then starting with sublists of length 1, "merge" the two sublists into a single sorted list.

```
Mergesort(m)
        var list left, right, result
        if length(m) <= 1
                return m
        else
            var middle = length(m) / 2
            for each x in m up to middle
                add x to left
            for each x in m after middle
                add x to right
            left = Mergesort(left)
```

```
            right = Mergesort(right)
            result = Merge(left, right)
            return result
}
```

The merge step is carried out by the following pesudocode:

```
Merge(left, right)
         var list result
        while length(left) > 0 and length(right) > 0
              if first(left) <= first(right)
                  append first(left) to result
                  left = rest(left)
              else
                  append first(right) to result
                  right = rest(right)
        if length(left) > 0
            append rest(left) to result
        if length(right) > 0
            append rest(right) to result
        return result
}
```

1. Assume that you have Y cores on a multi-core processor to run MergeSort. Assuming that Y is much smaller than length(m), express the speedup factor you might expect to obtain for values of Y and length(m).
2. Next, assume that Y is equal to length(m). How would this affect your conclusions your previous answer? If you were tasked with obtaining the best speedup factor possible (i.e., strong scaling), explain how you might change this code to obtain it.

A3

1. This problem is again a divide and conquer problem, but utilizes recursion to produce a very compact piece of code. In part A the student is asked to compute the speedup when the number of cores is small. We when forming the lists, we spawn a thread for the computation of left in the MergeSort code, and spawn a thread for the computation of the right. If we consider this recursively, for m initial elements in the array, we can utilize $1 + 2 + 4 + 8 + 16 + ... + \log_2(m)$ processors to obtain speedup.
2. In this question, $\log_2(m)$ is the largest value of Y for which we can obtain any speedup without restructuring. But if we had m cores, we could perform sorting using a very different algorithm. For instance, if we have greater than $m/2$ cores, we can compare all pairs of data elements, swap the elements if the left element is greater than the right element, and then repeat this step m times. So this is one possible answer for the question. It is known as parallel comparison sort. Various comparison sort algorithms include odd-even sort and cocktail sort.

Q4 (**10%**) Consider the following portions of two different programs running at the same time on four processors in a share memory multiprocessor (SMP). Assume that before this code is run, both x and y are 0.

Core1: `x = 3;`

Core2: `y = 3;`

Core3: `w = x * y + 4;`

Core4: `z = x - y;`

Core5: `r = w + z;`

1. What are all the possible resulting values of w, x, y, z, and r? For each possible outcome, explain how we might arrive at those values.

A4 1. As shown in the following table:

Table 1: One correct column for 1 point

x	3	3	3	3	3	3
y	3	3	3	3	3	3
w	4	4	4	13	13	13
z	0	-3	3	0	-3	3
r	4	1	7	13	10	16

Q5 (**20%**) Here are two different I/O systems intended for use in transaction proceeding:

- System A can support 15,000 I/O operations per second and use the processor with MIPS rate of 50.
- System B can support 1,000 I/O operations per second and use the processor with MIPS rate of 500.

Assume that each transaction requires 5 I/O operations and each I/O operation requires 10,000 instructions. Ignoring response time, what is the maximum transactions per second for each system.

A5 Each transaction requires 10,000 × 5 = 50,000 instructions.

- For System A:
  CPU limit: 50M / 50K =1000 trans/second;
  I/O limit: 15,000 / 5=3000 trans/second;
  Therefore, max 1000 trans/second.
- For System B:
  CPU limit: 500M / 50K =10000 trans/second;
  I/O limit: 1,000 / 5=200 trans/second;
  Therefore, max 200 trans/second.