Problem set 1, Part 2

1 Theory

1 Caching

---

1. Explain the difference between a fully associative cache, a direct mapped cache and an n-way set associative cache.

Fully associative cache: A system in which a new line can be placed at any location on the cache.

Direct mapped cache: A system in which each cache line has a unique location in the cache to which it will be assigned.

n-way associative cache: Intermediate schemes in which each cache line can be placed in one of different locations in cache.

---

2. According to Pacheco, programmers only have indirect control over caching. Why is this important? Give an example.

As programmers only have indirect control over the cache it is important to remember how the cache may access data and in which order.

For example if a programmer is working on a two dimensional matrix looping through every element. The most intuitive method looping through the elements may for a programmer be access every column in row 0 and then row 1 and so on. But in the cache the memory will be stored as one large one dimensional vector and the performance of the looping program will depend on the programmers implementation related to the cache construction.

3. Give a brief description of the two main approaches to ensuring cache coherence.

Snooping cache coherence: In a snooping system, all caches on the bus monitor the bus to determine if they have a copy of the block of data that is requested on the bus. Every cache has a copy of the sharing status of every block of physical memory it has.

Directory-based cache coherence: A protocol which uses a data structure called directory. The directory stores the status of each cache line. Typically, this data structure is distributed; each core/memory pair might be responsible for storing the part of the structure that specifies the status of the cache lines in its local memory. When a variable is updated, the directory is consulted, and the cache controllers of the cores that have that variable's cache line in their caches are invalidated.

2. Gustafson's law

Consider the following code:

```
for(int i = 0; i < problem_size*f; i++){
   function_a();
}
for(int i = 0; i < problem_size; i++){
   function_b();
}
```

Where function a() is inherently serial, and function b() can be parallelized fully without overhead. The serial execution times of function a() and function b() are the same.

---

1. The value of f can be 0 or 1. In which case does the program fit the assumptions of Amdahl's law and in which case the assumptions of Gustafson's law?

If f = 0 the computation size of the serial part will be fixed if the problem size increases. Which means for increased problem size the serial computational part will decrease compared to parallel part of the program. This follows the assumption of Gustafson's law.

If f = 1 both the serial part and the parallel part of the program will increase computation time for increased problem size. This will lead to the enhancement of adding processors

in the program will be limited by the computation time of the serial part. This follows the assumption of Amdahl's law.

---

2. Calculate the speedup of the program using N = 2,4, and 8 processors and problem size=5.

- With f = 0, using Gustafson's law

ts = 1 , tp = problem_size

Processors	**Speedup**
2	(ts + tp * N) / (ts + tp) = 1.83
4	= 3.50
8	= 6.83

- With f = 0, using Amdahl's law

ts = 1, tp = problem_size

Processors	**Speedup**
2	(ts + tp)/(ts + tp/N) = 1.71
4	(1+100)/(1+100/4) = 2.67
8	(1+100)/(1+100/8) = 3.69

- With f = 1, using Gustafson's law.

Processors	**Speedup**
2	= 1.5
4	= 2.5
8	= 4.5

- With f = 1, using Amdahl's law.

Processors	Speedup
2	= 1.33
4	= 1.60
8	= 1.78

---

3. Explain why/why not you get the same results using Gustafson's and Amdahl's laws.

The results will not be the same for Amdahl's law and Gustafson's law. The reason for that is that Gustafson's law assumes the amount of work to be done will increase as the number of processor increases. Amdahl's law assumes the amount of work to be done is constant no matter how much parallelisation is available.

2 MPI Programming

1 Mandatory

---

2. Time your program, with P = {2,4,8} and scale the size of your problem by S = {1,2,3}. That is, XSIZE = S*2560, YSIZE = S*2048 and MAXITER = S*255.

Timing performed on personal computer:

Serial timing with S= {1,2,3} = {2.41 s, 61.7 s, 60.76 s}

	S = 1	**S = 2**	**S = 3**
P = 2	2.45 sec	18.65 sec	61.8 sec
P = 4	1.72 sec	13.12 sec	43.8 sec
P = 8	0.98 sec	7.0 sec	23.8 sec