CMU 15-418/618 Exam 2 Practice Problems

Miscellaneous Questions

A. You are working on parallelizing a matrix-vector multiplication, and try creating a result vector for each thread (p). Your code then has each thread sum a subset (n/p) of rows (n) out of each result vector. However, you find that the performance is unchanged as you increase the number of threads. What algorithmic flaw is present in this approach?

B. Recall that the semantics of `cilk_spawn` are that the spawned function is executed in a logically asynchronous thread of control that *may or may not* run in parallel with execution of the caller. Also recall that the *implementation of Cilk work scheduling* always runs the child (spawned) function first.

Consider the following two implementations of a parallel for loop in Cilk.

```
/* implementation 1 */
for (int i=0 i<N; i++)
   cilk_spawn myfunction(i);

/* implementation 2 */
void genwork(int start, int end) {

  while (start < end - GRANULARITY) {
    int mid = (end + start) / 2;
    cilk_spawn genwork(start, mid);
    start = mid;
  }

  for (int i=start; i<end; i++)
    myfunction(i);
}

genwork(0, N);  // launch all work
```

Give one reason why implementation 2 might give better performance than implementation 1 on a parallel machine. (Your answer should be precise and refer to how the two programs would be scheduled by the Cilk runtime.)

C. Graph algorithms often have low arthimetic intensity. Desribe a cause and a possible mitigation.

D. You are designing a heterogeneous multi-core processor to perform real-time "celebrity detection" on a future camera. The camera will continuously process low-resolution live video and snap a high-resolution picture whenever it identifies a subject in a database of 100 celebrities. Pseudocode for its behavior is below:

```
void process_video_frame(Image input_frame)
{
  Image face_image = detect_face(input_frame);
  for (int i=0; i<100; i++)
     if (match_face(face_image, database_face[i]))
        take_high_res_photo();
}
```

In order to not miss the shot, the camera MUST call `take_high_res_photo` within 500 ms of the start of the original call to `process_video_frame`! To keep things simple:

- Assume the code loops through all 100 database images regardless of whether a match is found (e.g., we want to find all matches).
- The system has plenty of bandwidth for any number of cores.

Two types of cores are available to use in your chip. One is a fixed-function unit that accelerates `detect_face`, the other is a general-purpose processor. The cost (in chip resources) of the cores and their performance (in ms) executing important functions in the pseudocode are given below:

Operation	Core Type	
	C1 (fixed-function)	C2 (Programmable)
Resource Cost	1	1
Perf (ms): `detect_face`	100	400
Perf (ms): `match_face`	N/A	20

Your team has just built a multi-core processor that contains a large number of cores of type C2. It achieves 5.9× speedup on the camera workload discussed above. Amdahl's Law says that the maximum speedup of the camera pipeline in this problem should be $2400/400 = 6\times$, so your team is happy. They are shocked when your boss demands a speedup of 10×. Your team is on the verge of quitting due "unreasonable demands". How do you argue to them that the goal is reasonable one if they consider all the possibilities in the above table? (Hint: What assumption are they making in their Amdahl's Law calculations, and why does it not hold?)

Load Linked / Store Conditional

A common set of instructions that enable atomic execution is load linked-store conditional (LL-SC). The idea is that when a processor loads from an address using a `load_linked` operation, the corresponding `store_conditional` to that address will succeed only if no other writes to that address from another processor have intervened. Note that unlike `test_and_set` or `compare_and_swap`, which are single atomic operations, load linked and store conditional are different operations and the processor may execute other instructions in between these two operations. Pseudocode for these instructions is given below.

```
int load_linked(int* addr) {
  return *addr;
}

bool store_conditional(int* addr, int new_val) {
  if (data in addr has not been changed since the corresponding load_linked) {
    *addr = new_val;
    return true;
  } else
     return false;
}

Example usage:

int x;
load_linked(&x);
y = f(x);  // do stuff with x here
store_conditional(&x, y);
```

A. (5 pts) Implement a spin lock using LL and SC primitives:

```
    void Lock(int* l) {









    }

    void Unlock(int* l) {



    }
```

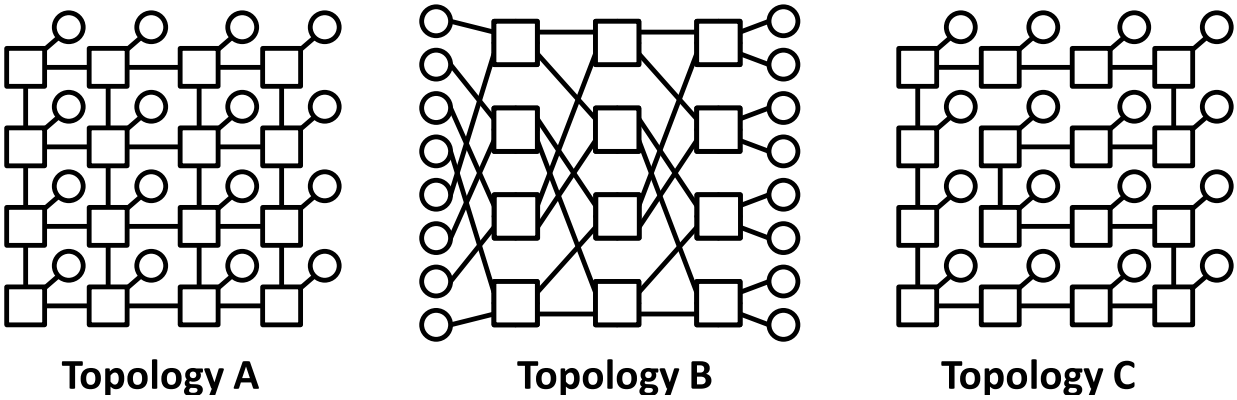
B. (5 pts) Given that you now have a good understanding of a basic implementation of invalidation-based cache coherence, describe how you would extend the behavior of MESI caches to implement the load-linked and store conditional instructions.

C. (4 pts) Even though the following implementation is lock free, it does not mean it is free of contention. In a system with P processors, imagine a situation where all P processors are contending to pop from the stack. Describe a potential performance problem with the current implementation and describe one potential solution strategy. (A simple descriptive answer is fine.)

```
// CAS function prototype: update address with new_value if its contents
// match expected_value.  Return value of addr (at start of operation).
Node* compare_and_swap(Node** addr, Node* expected_value, Node* new_value);

struct Node {
  Node* next;
  int value;
};

struct Stack {
  Node* top;
};

void init(Stack* s) {
  s->top = NULL;
}

void push(Stack* s, Node* n) {
  while (1) {
    Node* old_top = s->top;
    n->next = old_top;
    if (compare_and_swap(&s->top, old_top, n) == old_top)
      return;
  }
}

Node* pop(Stack* s) {
  while (1) {
    Node* top = s->top;
    if (top == NULL)
      return NULL;
    Node* new_top = top->next;
    if (compare_and_swap(&s->top, top, new_top) == top)
      return top;
  }
}
```

Interconnection Networks

A. (8 pts) The figure below shows four common network topologies (circles and squares represent network endpoints and routers respectively).



Identify each topology and fill in the table below. Express the bisection bandwidth in Gbit/s, assuming each link is 1 Gbit/s. Express the cost and latency in terms of the number of network nodes N, using Big O notation, e.g., O(logN)).

	Topology A	**Topology B**	**Topology C**
Topology Type/Name			
Direct or Indirect			
Blocking or Non-Blocking			
Bisection Bandwidth			
Cost			
Latency			