Sample Mid-Term Exam 1

CS 4960-01, Fall 2008

Actual exam scheduled for October 10

Name: ____

Instructions: You have fifty minutes to complete this open-book, open-note, closed-computer exam. Please write all answers in the provided space plus the back of the exam.

1) Suppose that you have the following program:

```
int go(int q, int s)
{
  int i, j, fv = 0;
  for (j = 0; j < s; j++) {
    for (i = 0; i < q; i++) {
      if (j > 0)
        fv = f(fv);
      else
        fv += f(i);
    }
  }
  return fv;
}
```

You know nothing about `f`, except that it's purely functional (i.e., it does not change the state of any global variable). You're not allowed to change them or look at its implementation.

When you run `go(`n`, 1)`, the computation takes n seconds. When you run `go(`n`, `m`)`, the computation takes $n \cdot m$ seconds.

You're allowed to change `go` to make it work with P processors. Sketch an implementation on Peril-L that takes advantage of parallelism, and estimate speedup for `go(`n`, `m`)` where $n >> P$.

Explain your time estimate in terms of your implementation sketch. Assume that the overhead for parallelism is negligible, which is realistic compared to the time apparently required for a call to `f`. You can also assume that simple operations, such as assigning to an array slot or local variable, are essentially free (again, compared to the expensive calls to `f`).

2) Suppose that you're writing a program to modify a square 2-D array of numbers. You'll modify it using P threads, where P is much less than the width of the array. Each number in the input array is an integer between 0 and 2^{32}, but the distribution of numbers is not specified.

Here's what your program should do:

1. Each number should be replaced with the average of itself with the four surrounding numbers.
2. Then, imagine a line through $row = column$ diagonal of the array; each number should be averaged with its "opposite" across the diagonal. For example, the number at row 5 column 7 should be averaged with the number at row 7 column 5, and diagonal elements are left alone.
3. Finally, each number n should be replaced with an average from sampling n random values across the whole array.

Describe how to organize threads and data for this problem to maximize parallelism. The data distribution does not have to be static or fixed for given run.

3) Define the types `data_t`, `tally_t`, and `result_t` and implement the following functions to plug into a reduce framework:

```
tally_t init();
tally_t accum(tally_t, int, data_t*);
tally_t combine(tally_t, tally_t);
result_t reduceGen(tally_t);
```

The functions should implement the following computation: given an array of floating-point numbers, determine the maximum absolute difference between any number in the array and the average of all numbers in the array.

You don't need to write the "client" half of the code. Assume that someone else will use your choice of `data_t` to set up suitable inputs.

4) The following code in the book (page 159) doesn't work as advertised:

```
EatJuicyFruit()
{
  pthread_mutex_lock(&lock);
  while (apples==0||oranges==0) {
    pthread_cond_wait(&more_apples, &lock);
    pthread_cond_wait(&more_oranges, &lock);
  }
  /* Critical Section: Eat both an apple and an orange */
  pthread_mutex_unlock(&lock);
}
```

Imagine that thread 1 is waiting. Then, a while later, thread 2 picks an orange and makes it available to eat; it increments `oranges` and signals `more_oranges`. Then, still later, thread 3 picks an apple and makes it available to eat, incrementing `apple` and signalling `more_apples`. Thread 1 will still be stuck, instead of eating the apple and orange. Why? Explain the problem and suggest a way to fix the above code.

Answers

1) Calling `f` apparently takes constant time, independent of its argument. Moreover, after the first iteration of the outer loop, each call to `f` depends on the previous one, so the only opportunity for parallelism is in the first iteration. In the first iteration, all the calls to `f` are independent. We should expect to make `go(`n`, `m`)` take about $n/P + n \cdot (m-1)$ seconds.

```
int go(int q, int s)
{
  int i, j, fv = 0;

  if (s > 0) {
    int _tfv_[q];
    /* run one iteration in parallel */
    forall (k in [0 .. P-1]) {
      int l, start, end;
      start = k * (_q_/P);
      end = (k + 1) * (_q_/P); /* assume that q/P is an integer */
      for (l = start; l < end; l++) {
        _tfv_[l] = f(l);
      }
    }
    /* sum the results of the first iteration */
    for (i = 0; i < q; i++)
      fv += tfv[i];
    }
  }

  /* remaining iterations are inherently sequential */
  for (j = 1; j < s; j++) {
    for (i = 0; i < q; i++) {
      fv = f(fv);
    }
  }

  return fv;
}
```

2) The data organization should change with each phase of the computation:

1. Use a block allocation, which minimizes non-local data access when surrounding values are needed.
2. Give each thread some data on both sized of the diagonal, where the region for half of the data for each thread is the "mirror image" across the dialog of the region of the other half, so that the average computations and updates are completely local.
3. There's no particularly good choice in this case, but a work queue is probably a good idea, since some regions of the array may have larger numbers than other regions, and the amount of work to perform for each array element is proportional to the size of the number.

3)

```
typedef double data_t;
typedef struct { double minv, maxv, total; int count; } tally_t;
typedef double result_t;
```

```
tally_t init() {
  tally_t t = { 0, 0, 0, 0 };
  return t;
}

tally_t accum(tally_t t, int i, data_t* array)
{
  if (t.count == 0) {
    t.minv = t.maxv = array[i];
  } else {
    t.minv = min(t.minv, array[i]);
    t.maxv = max(t.maxv, array[i]);
  }
  t.total += array[i];
  t.count++;
  return t;
}

tally_t combine(tally_t t1, tally_t t2)
{
  if (t1.count == 0) return t2;
  if (t2.count == 0) return t1;
  t1.minv = min(t1.minv, t2.minv);
  t1.maxv = min(t1.maxv, t2.maxv);
  t1.total += t2.total;
  t1.count += t2.count;
  return t1;
}

result_t reduceGen(tally_t t)
{
  double avg = t.total / t.count;
  return max(avg - t.minv,  t.maxv - avg);
}
```

4) The problem is that a `more_oranges` signal might be missed while waiting on `more_apples`, but the loop is committed to waiting on both. As noted in the book, it doesn't work to separately loop on `apples` and `oranges`. However, within the combined loop, we should should always check `apples` before waiting on `more_apples`, and we should check `oranges` before waiting on `more_oranges`, like this:

```
while (apples==0||oranges==0) {
  if (apples == 0)
    pthread_cond_wait(&more_apples, &lock);
  if (oranges == 0)
    pthread_cond_wait(&more_oranges, &lock);
}
```