README for week 7 in IN3200/IN4200

Exercise 1) OpenMP

a)

```
for (i=0; i < (int) sqrt(x); i++) {
    a[i] = 2.3 * x;
    if (i < 10) b[i] = a[i];
}
```

Some obfuscating declarations here, but this is very simple.

```
#pragma omp parallel for
for (i=0; i < (int) sqrt(x); i++) {
    a[i] = 2.3 * x;
    if (i < 10) b[i] = a[i];
}
```

b)

```
flag = 0;
for (i = 0; (i<n) & (!flag); i++) {
    a[i] = 2.3 * i;
    if (a[i] < b[i]) flag = 1;
}
```

Here the loop terminates the first time `a[i] < b[i]`. If we parallelize this loop, the values in `a` might be different when we exit the loop vs the serial case.

When using openMP this will not compile due to the condition in the for loop. But even if we could, the results would change depending on how the iterations are scheduled.

c)

```
for (i = 0; i < n; i++)
    a[i] = foo(i);
```

If we assume no black magic going on in the function `foo()`. Then it's rather trivial to parallelize.

```
#pragma omp parallel for
for (i = 0; i < n; i++)
    a[i] = foo(i);
```

d)

```
for (i = 0; i < n; i++) {
    a[i] = foo(i);
    if (a[i] < b[i]) a[i] = b[i];
}
```

There is still no dependence between the different entries in `a`.

```
#pragma omp parallel for
for (i = 0; i < n; i++) {
    a[i] = foo(i);
    if (a[i] < b[i]) a[i] = b[i];
}
```

e)

```
for (i = 0; i < n; i++) {
    a[i] = foo(i);
    if (a[i] < b[i]) break;
}
```

Here the contents of `a` after exiting the loop depends on the order we iterate over `a`. If we parallelize we get a race condition. Not suitable for parallelization.

f)

```
dotp = 0;
for (i = 0; i < n; i++)
    dotp += a[i] * b[i];
```

In this case each iteration of the loop loads values that does not depend on the iteration before it. But all the iterations write to the same variable. This is what we call a reduction operation. To parallelize this we tell openMP that `dotp` is a reduction variable, and that we want the sum total.

```
dotp = 0;
#pragma omp parallel for reduction(+ : dotp)
for (i = 0; i < n; i++)
    dotp += a[i] * b[i];
```

When the loop is exited and the threads synchronized, `dotp` will have the correct value. There are a few reduction operations, see section 2.14.3.6 in the openMP API.

g)

```
for (i = k; i < 2*k; i++)
    a[i] = a[i] + a[i-k];
```

Here `a[i]` depends on a value earlier in the array. Often this means that there is limited potential for parallelization, but the index `i-k` is outside the area we are writing to for every iteration (we start with `i=k`). So there is no race condition.

```
#pragma omp parallel for
for (i = k; i < 2*k; i++)
    a[i] = a[i] + a[i-k];
```

h)

```
for (i = k; i < n; i++)
    a[i] = b * a[i-k];
```

Here we can not be certain that we are writing to elements not changed inside the loop body. There is a potential for a race condition, and without knowing the values of `k` and `n` we can not be certain.

If $n \leq 2k$ we are in the same situation as in **g)**, and we can parallelize. It is possible to parallelize this even when $n > 2k$, but it takes some effort. See section 6.3 in the book if you are interested.

Exercise 2) Dot Product

Inner product is the same as a dot product and the code for this is given in **ex. 1, f)**. We only have to change the type of openMP scheduler we are using. There are several types: - static - dynamic - guided - auto - runtime

You can read about what the different methods do in the openMP API, in section 2.7.1.

Adding the `schedule` directive to our pragma. `C dotp = 0; #pragma omp parallel for reduction(+ : dotp) schedule(schedule_type, chunk_size) for (i = 0; i < n; i++) dotp += a[i] * b[i];`

The openMP library comes with a timer function that is very easy to use, and is useful now that we are moving to programs with multiple threads executing concurrently.

```
double start = omp_get_wtime();
// Do some parallel work here
double end = omp_get_wtime();
double total = end - start;
```

This function returns the current time in seconds since a reference time.

Exercise 3) Matrix-Vector Multiplication

```
void dense_mat_vec(int m, int n, double *x, double *A, double *y)
{
    int i, j;
    for (i=0; i<m; i++){
        double tmp = 0.;
        for (j=0; j<n; j++)
            tmp += A[i*n+j] * y[j];
        x[i] = tmp;
    }
}
```

One thing to note before we start. In this function `A` is not a pointer to pointer. We are instead using `i*n + j` as the index to access the elements in the right order.

a) Serial Version

The different versions of the matrix-vector multiplication functions are put in its own header file, `matvec.h`, this week. Review the source code for a solution. Notice that there is an a index function defined near the top that is used when assigning values to `A`.

```
#define idx(i,j) (i*n + j)
```

Wherever the preprocessor finds an expression `idx(i,j)`, it is replaced with `(i*n + j)` before compilation.

b) OpenMP Parallel Version

To create a parallel version of `dense_mat_vec()` we just add a `#pragma` right before the outer loop.

```
void dense_mat_vec_omp(int m, int n, double *x, double *A, double *y)
{
    int i, j;
    #pragma omp parallel for private(j)
    for (i=0; i<m; i++){
        double tmp = 0.;
        for (j=0; j<n; j++)
            tmp += A[i*n+j] * y[j];
        x[i] = tmp;
    }
}
```

The outer loop is divided among the threads, so we do not need to declare `i` private. `j` is defined before the loop but each thread should work with its own private one. `tmp` is declared inside the parallel region and is by default private.

c) Constraints

For an arbitrary function this task is impossible, but in this specific case we can find a way to decompose the problem. Last week we worked on block decomposition. We can use the same technique here to divide the matrix A, and vector x, in to parts that each thread can call `dense_mat_vec()` on.

```
#pragma omp parallel
{
  int thread_id = omp_get_thread_num();
  int num_threads = omp_get_num_threads();

  // Using integer division to divide up the rows means that the last
  // remainder = m%num_threads, gets an extra row.
  int start_m = (thread_id*m)/num_threads;
  int stop_m = ((thread_id+1) * m)/num_threads;

  // calls the serial dense_mat_vec function
  dense_mat_vec(stop_m-start_m, n, &x[start_m], &A[start_m*n], y);
}
```

Notice that we are only decomposing `A` and `x`. We want each element in `x` to be written to by only one thread to minimize false sharing. False sharing is when multiple threads try to write to the same cache line. When an element in a cache line is updated, the line is marked as modified, forcing the other threads to reload the line before their own write operations can be completed. See section 4.2.1 in the book. This takes memory bandwidth, and there is also a latency incurred every time.