C14

Recursive tasks

Divide&Impera

```
begin solve(problem)
        if problem small enough
                return solveBaseCase(problem)
        else
                split(problem, subproblem1, subproblem2)
                solution1=solve(subproblem1)
                solution2=solve(subproblem2)
                return merge(solution1,solution2)
end solve
```

OpenMP

OpenMP: working with recursive tasks

```
int suma(int x[], int a, int b) {
if (a == b) return  x[a];
int sd = 0, ss = 0, s = 0;
#pragma omp parallel num_threads(5)
{
    #pragma omp single
    {
        #pragma omp task shared(sd)
        {
                sd = suma(x, (a + b) / 2+1, b);
        }
        #pragma omp task shared (ss)
        {
                ss = suma(x, a, (a + b) / 2 );
        }
        #pragma omp taskwait
        s = ss + sd;
        }
    }
return s;
}
```

Fibonacci

```
int fib(int n)
{
if(n == 0 || n == 1)
return n;
int result, F_1, F_2;
#pragma omp parallel
{
        #pragma omp single
        {
                #pragma omp task shared(F_1)
                        F_1 = fib(n-1);
                #pragma omp task shared(F_2)
                        F_2 = fib(n-2);
                #pragma omp taskwait
                        res = F_1 + F_2;
        }
}
return res;
}
```

C++ Futures

Recursive Array Sum (Sequential version)

```
1. int sum = computeSum(X, 0, X.length-1); // main
2. static int computeSum(int[] X, int lo, int hi) {
3.              if ( lo > hi ) return 0;
4.               else if ( lo == hi ) return X[lo];
5.               else {
6.                              int mid = (lo+hi)/2;
7.                               int sum1 =
8.                                              computeSum(X, lo, mid);
9.                              int sum2 =
10.                                             computeSum(X, mid+1, hi);
11.
12.                             return sum1 + sum2;
13.             }
14. } // computeSum
```

Recursive Array Sum using Future Tasks (Two futures per method call)

```
int sum = computeSum(X, 0, X.length-1); // main
 static int computeSum(int[] X, int lo, int hi) {
                if ( lo > hi ) return 0;
               else if ( lo == hi ) return X[lo];
                 else {
                        int mid = (lo+hi)/2;
                       future<int> sum1 =  std::async(       std::launch::async,
                              []()->int{
                                             return  computeSum(X, lo, mid);
                             };

                             future<int> sum2 = std::async(  std::launch::async,  //   std::launch::deferred,
                              []()->int{
                                              return computeSum(X, mid+1, hi);
                                             };
                             // Parent now waits for the container values
                             return sum1.get() + sum2.get();
}
} // computeSum
```

Computation Graph Extensions for Future Tasks

- Since a get() is a blocking operation,

May require splitting a statement into sub-statements e.g.,

– 12: int sum = sum1.get() + sum2.get();

can be split into three sub-statements

– 12a: int temp1 = sum1.get();

– 12b: int temp2 = sum2.get();

– 12c: int sum = temp1 + temp2;

Java

ForkJoinPool

- https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html

- A ForkJoinPool differs from other kinds of ExecutorService mainly by virtue of employing **work-stealing**: all threads in the pool attempt to find and execute tasks submitted to the pool and/or created by other active tasks (eventually blocking waiting for work if none exist). This enables efficient processing when most tasks spawn other subtasks (as do most ForkJoinTasks), as well as when many small tasks are submitted to the pool from external clients. Especially when setting asyncMode to true in constructors, ForkJoinPools may also be appropriate for use with event-style tasks that are never joined.

- A static **commonPool()** is available and appropriate for most applications. The common pool is used by any ForkJoinTask that is not explicitly submitted to a specified pool. Using the common pool normally reduces resource usage (its threads are slowly reclaimed during periods of non-use, and reinstated upon subsequent use).

ForkJoinTask -> RecursiveTask
-> RecursiveAction

ForkJoinTask

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinTask.html

- A ForkJoinTask is a lightweight form of Future.
- Abstract base class for tasks that run within a ForkJoinPool.
- A "main" ForkJoinTask begins execution when it is explicitly submitted to a ForkJoinPool, or, if not already engaged in a ForkJoin computation, commenced in the ForkJoinPool.commonPool() via
  - fork(),
  - invoke(), or
  - related methods.

Once started, it will usually in turn start other subtasks.

Sum: RecursiveTask

public abstract class RecursiveTask<V> extends ForkJoinTask<V>

```
class Sum extends RecursiveTask<Double> {
static final int SEQUENTIAL_THRESHOLD = 1<<3;
   static final int MAX = 1<<10;
   static   BinaryOperator<Double> adder =
                              (n1, n2) -> (  n1*n1+n2*n2    );
   int low;
   int high;
   double[] array;
   Sum(double[] arr, int lo, int hi) {
      array = arr;       low   = lo;       high  = hi;
   }
static Double sumParallel(double[] array) {
        return ForkJoinPool.commonPool().invoke(new
                Sum(array,0,array.length));
    }
 public static double sumSeq(double[] arr) {
         long len = arr.length;
         double ans = 0;
         for (int i = 0; i < len; i++) {  ans = adder.apply(ans ,  arr[i]); }
          return ans;
}
```

```
protected Double compute() {
        if(high - low <=
SEQUENTIAL_THRESHOLD) {
          double sum = 0;
           for(int i=low; i < high; ++i)
              sum = adder.apply(sum, array[i]);
           return sum;
         } else {
           int mid = ( low + high ) / 2;
           Sum left  = new Sum(array, low, mid);
           Sum right = new Sum(array, mid, high);
           left.fork(); //right.fork()
           double rightAns = right.compute();
// right.join()
           double leftAns  = left.join();
           return adder.apply(leftAns ,rightAns);
         }
     }
```

Sort: RecursionAction

public abstract class RecursiveAction extends ForkJoinTask<Void>
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RecursiveAction.html

```
static class SortTask extends RecursiveAction {
  final long[] array; final int lo, hi;
  SortTask(long[] array, int lo, int hi) {
    this.array = array; this.lo = lo; this.hi = hi;
  }
  SortTask(long[] array) { this(array, 0, array.length); }
  protected void compute() {
    if (hi - lo < THRESHOLD)
      sortSequentially(lo, hi);
    else {
      int mid = (lo + hi) >>> 1;
      invokeAll(new SortTask(array, lo, mid),
                new SortTask(array, mid, hi));
      merge(lo, mid, hi);
    }
  }
```

```
  static final int THRESHOLD = 1000;
    void sortSequentially(int lo, int hi) {
      Arrays.sort(array, lo, hi);
    }
    void merge(int lo, int mid, int hi) {
      long[] buf = Arrays.copyOfRange(array, lo, mid);
      for (int i = 0, j = lo, k = mid; i < buf.length; j++)
        array[j] = (k == hi || buf[i] < array[k]) ?
          buf[i++] : array[k++];
    }
  }
```