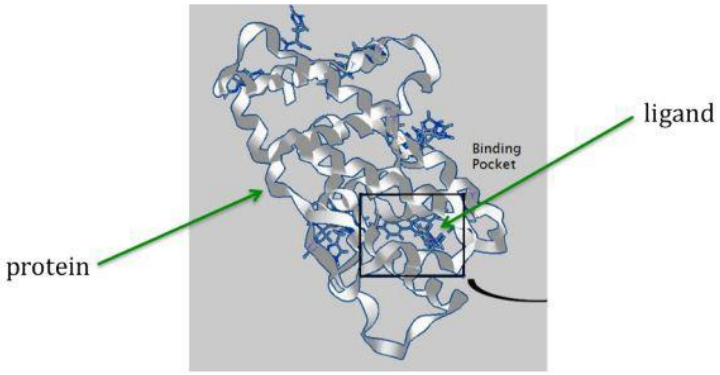
Drug Design and DNA in Parallel:

These instructions assume that you have a Raspberry Pi up and running. After starting up your Raspberry Pi, you should have a GUI interface to Raspbian, the Linux variant operating system for these little machines. For this tutorial, you will primarily use the terminal window and type commands into it. You will also use an editor (more on that below).

1) Drug Design:

An important problem in the biological sciences is that of drug design. The goal is to find small molecules, called *ligands*, that are good candidates for use as drugs.



At a high level, the problem is simple to state: a protein associated with the disease of interest is identified, and its three- dimensional structure is found either experimentally or through a molecular modeling computation. A collection of ligands is tested against the protein: for example, for every orientation of the ligand relative to the protein, computation is done to test whether the ligand binds with the protein in useful ways (such as tying up a biologically active region on the protein). A score is set based on these binding properties, and the best scores are flagged, identifying ligands that would make good drug candidates.

1.1 Algorithmic Strategy

We will apply a *map-reduce* strategy to this problem, which can be implemented using a *master-worker* design pattern. Our map-reduce strategy uses three stages of processing.

1. First, we will generate many ligands to be tested agains a given protein, using a function `Generate_tasks()`. This function produces many `[ligand, protein]` pairs (in this case, all with the same protein) for the next stage.
2. Next, we will apply a Map() function to each ligand and the given protein, which will compute the binding score for that `[ligand, protein]` pair. This Map() function will produce a pair `[score, ligand]` since we want to know the highest-scoring ligands.
3. Finally, we identify the ligands with the highest scores, using a function `Reduce()` applied to the `[score, ligand]` pairs.

These functions could be implemented sequentially, or they can be called by multiple processes or threads to perform the drug-design computation in parallel: one process, called the *master*, can fill a task queue with pairs obtained from Generate_tasks(). Many *worker* processes can pull tasks off the task queue and apply the function Map() to them. The master can then collect results from the workers and apply Reduce() to determine the highest-scoring ligand(s).

Note that if the Reduce() function is expensive to apply, or if the stream of [score, ligand] pairs produced by calls to Map() becomes too large, the Reduce() stage may be parallelized as well.

1.2 Simplified Problem Definition

Working with actual ligand and protein data is beyond the scope of this example, so we will represent the computation by a simpler string-based comparison.

Specifically, we simplify the computation as follows:

- Proteins and ligands will be represented as (randomly-generated) character strings.
- The docking-problem computation will be represented by comparing a ligand string L to a protein string P. The score for a pair [L, P] will be the maximum number of matching characters among all possibilities when L is compared to P, moving from left to right, allowing possible insertions and deletions. For example, if L is the string "cxtbcrv" and P is the string "lcacxtqvivg," then the score is 4, arising from this comparison of L to a segment of P:

```
lcacxet  qvivg
cx tbcrv
```

This is not the only comparison of that ligand to that protein that yields four matching characters. Another one is

```
lcacxetqvivg
c  x trbcv
```

However, there is no comparison that matches five characters while moving from left to right, so the score is 4.

2) A Sequential Solution

Let's first examine a traditional sequential (also called serial) solution that uses no parallelism, of the simplified version of this algorithm. As with many parallel implementations of algorithms, this serial version can form the basis for a parallel solution. We will later see several parallel solutions that have been updated from the following serial version.

2.1 Implementation (This applies to dd_omp.cpp and dd_threads.cpp too)

At iCollege in Assignment5 folder you will find three implementations of a program that attempts to locate small molecules called ligands that are good candidates for use as drugs. The example programs provides C++ implementation of our simplified drug design problem. The three parallel implementations:

- dd_serial.cpp:
  No OpenMP to solve the problem.
- dd_omp.cpp:
  Uses OpenMP to solve the problem.
- dd_threads.cpp:
  Uses C++11 threads (instead of OpenMP threads) to solve the problem.

Note: The program optionally accepts up to three command-line arguments:

1. maximum length of the (randomly generated) ligand strings
2. number of ligands generated
3. protein string to which ligands will be compared

2.2 Compilation:

To compile sequential solution

1. For sequential (also called serial) solution, the individual files to download from iCollege are:

   dd_serial.cpp
   Makefile
2. Start the terminal window by choosing its icon in the menu bar at the top (it looks like a black square box).
3. Create a director and name it sequential
4. Copy the sequential solution *cpp* and *make* files into sequential directory in your PI.
5. Cd into this directory
6. **type make**

Or, A straightforward compile can be used for this sequential example:

g++ -o dd_serial dd_serial.cpp

The Code:

In this implementation, the class MR encapsulates the map-reduce steps Generate_tasks(), Map(), and Reduce() as private methods (member functions of the class), and a public method run() invokes those steps according to a map-reduce algorithmic strategy (see previous Introduction section for detailed explanation). We have highlighted calls to the methods representing map-reduce steps in the following code segment from MR::run().

```
Generate_tasks(tasks);
// assert -- tasks is non-empty

while (!tasks.empty()) {
  Map(tasks.front(),  pairs);
  tasks.pop();
}

do_sort(pairs);
int next = 0;  // index of first unprocessed pair in pairs[]
while (next < pairs.size()) {
  string values;
  values = "";
  int key = pairs[next].key;
  next = Reduce(key, pairs, next,
  values); Pair p(key, values);
  results.push_back(p);
 }
```

Comments

- We use the STL containers queue<> and vector<> to hold the results from each of the map-reduce steps: namely, the task queue of ligands to process, the list key-value pairs produced by the Map() phase, and the list of resulting key-value pairs produced by calls to Reduce(). We define those container variables as data members in the class MR:

```
queue<string> tasks;
vector<Pair> pairs,
results;
```

- Here, Pair is a struct representing key-value pairs with the desired types:

```
struct Pair {
  int key;
  string  val;
  Pair(int k, const string &v) {key=k; val=v;}
};
```

- In the example code, Generate_tasks() merely produces *nligands* strings of random lower-case letters, each having a random length between 0 and *max_ligand*. The program stores those strings in a task queue named tasks.
- For each ligand in the task queue, the Map() function computes the match score from comparing a string representing that ligand to a global string representing a target protein, using the simplified match-scoring algorithm described above. Map() then yields a key-value pair consisting of that score and that ligand, respectively.
- The key-value pairs produced by all calls to Map() are sorted by key in order to group pairs with the same score. Then Reduce() is called once for each of those groups in order to yield a vector of Pairs consisting of a score *s* together with a list of all ligands whose best score was *s*.

Note: Map-reduce frameworks such as the open-source Hadoop commonly use sorting to group values for a given key, as does our program. This has the additional benefit of producing sorted results from the reduce stage. Also, the staged processes of performing all Map() calls before sorting and of performing all Reduce() calls after the completion of sorting are also common among map-reduce frameworks.

- The methods Generate_tasks(), Map(), and Reduce() may seem like unnecessary complication for this problem since they abstract so little code. Indeed, we could certainly rewrite the program more simply and briefly without them. We chose this expression for several reasons:
  - We can compare code segments from MR::run() directly with corresponding segments in upcoming parallel implementations to focus on the parallelization changes and hide the common code in method calls.
  - The methods Generate_tasks(), Map(), and Reduce() make it obvious where to insert more realistic task generation, docking algorithm, etc., and where to change our map-reduce code examples for problems other than drug design.
  - We use these three method names in descriptions of the map-reduce pattern elsewhere.
- We have not attempted to implement the fault tolerance and scalability features of a production map-reduce framework such as Hadoop.

3) OpenMP Solution

Here, we implement our drug design simulation in parallel using OpenMP, an API that provides compiler directives, library routines, and environment variables that allow shared-memory multithreading in C/C++. A master thread will fork off a specified number of worker threads and assign parts of a task to them

3.1 Implementation

The implementation dd_omp.cpp parallelizes the Map() loop using OpenMP and uses a thread-safe container from TBB, a C++ template library designed to help avoid some of the difficulties associated with multithreading.

Since we expect the docking algorithm (here represented by computing a match score for comparing a ligand string to a protein string) to require the bulk of compute time, we will parallelize the Map() stage in our sequential algorithm. The loop to be parallelized is shown below, from the full sequential implementation, dd_serial.cpp, discussed in the previous section.

```
while (!tasks.empty()) {
Map(tasks.front(), pairs); tasks.pop();
}
```

We will now parallelize this mapping loop by converting it to a for loop, then applying OpenMP's parallel for feature - there is no parallel while. For easier use with a for loop, we will replace the tasks queue with a vector (of the same name) and iterate on index values for that vector.

This causes a potential concurrency problem, though, because multiple OpenMP threads will now each be calling Map(), and those multiple calls by parallel threads may overlap. There is no potential for error from the first argument ligand of Map(), since Map() requires simply read-only access for that argument. However, multiple calls of Map() in different threads might interfere with each other when changing the writable second argument pairs of Map(), leading to a data race condition. The STL containers are not thread safe, meaning that they provide no protection against such interference, and errors may result.

Therefore, we will use TBB's thread-safe concurrent_vector container for pairs, leading to the following code segments in our OpenMP implementation.

```
vector<string> tasks; tbb::concurrent_vector<Pair> pairs; vector<Pair> results;

Generate_tasks(tasks);
// assert -- tasks is non-empty

#pragma omp parallel for  num_threads(nthreads)
for (int t = 0;   t <  tasks.size();          t++) { Map(tasks[t], pairs);
}
```

Since the main thread (i.e., the thread that executes run()) is the only thread that performs the stages that call Generate_tasks(), to_sort(), and Reduce(), it is safe for the vectors tasks or results to remain implemented as (non-thread safe) STL containers. See the implementation (dd_omp.cpp) for complete details.

Notes:

- Most of the changes between the sequential version and this OpenMP version arise from the change in type for the data member MR::pairs to a thread-safe data type; a few changes have to do with managing the number of threads to use nthreads. All of the parallel computation is specified by the one-line #pragma directive shown above - without it, the computation would proceed sequentially.
- This OpenMP implementation has four (optional) command-line arguments. The third argument specifies the number of OpenMP threads to use (note that this differs from the third argument in the sequential version). In dd_omp.cpp, the command-line arguments have these effects:
    1. maximum length of a (randomly generated) ligand string
    2. number of ligands generated
    3. number of OpenMP threads to request
    4. protein string to which ligands will be compared

3.2 Compilation:

To compile sequential solution

1. For the OpenMP solution, the individual files to download from iCollege are:

    dd_omp.cpp.cpp
    Makefile
2. Start the terminal window by choosing its icon in the menu bar at the top (it looks like a black square box).
3. Create a director and name it openmp1
4. Copy the OpenMP solution *cpp* and *make* files into openmp directory in your PI.
5. Cd into this directory
6. **type make**

Or, a straightforward compile can be used for this example:

g++ -o dd_omp dd_omp.cpp

4) C++11 Threads Solution

In the OpenMP implementation, the OpenMP runtime system implicitly creates and manages threads for us. The dd_threads.cpp implementation parallelizes the computationally expensive Map() stage by using the new C++11 standard threads instead of OpenMP. This requires us to explicitly create and manage our own threads, using a master- worker parallel programming pattern driven by tasks, and a task queue produced by Generate_tasks().

We will examine the C++11 threads implementation by comparing it to the sequential implementation. You may want to have each of them open in an editor as you read along.

The main routine for map-reduce computing in both implementations is MR::run(), and this routine is identical in the two except for the "map" stage and for the threads version handling an extra argument nthreads. In the serial implementation, the "map" stage simply removes elements from the task queue and calls Map() for each such element, via the following code.

```
while (!tasks.empty()) {
        Map(tasks.front(),
        pairs); tasks.pop();
}
```

However, the threads implementation of the "map" stage creates an array pool of threads to perform the calls to Map(), then waits for those threads to complete their work by calling the join() method for each thread:

```
thread *pool = new
thread[nthreads]; for (int i
= 0;            i < nthreads;
                i++) pool[i] =
thread(&MR::do_Maps,
this);

for (int i = 0;             i
< nthreads;  i++)
pool[i].join();
```

In this snippet from the threads implementation, we define the function MR::do_Maps() for performing calls to Map():

```
void MR::do_Maps(void) {
        string lig;
        tasks.pop(lig)
        ;
        while (lig != SENTINEL) {
                Map(lig, pairs);
                tasks.pop(lig);
        }
        tasks.push(SENTINEL);       // restore end marker for
        another thread
}
```

This method do_Maps() serves as the "main program" for each thread, and that method repeatedly pops a new ligand string `lig` from the task queue, and calls Map() with `lig` until it encounters the end marker SENTINEL.

Since multiple threads may access the shared task queue `tasks` at the same time, that task queue must be thread-safe, so we defined it using a TBB container:

```
tbb::concurrent_bounded_queue<string>  tasks;
```

We chose `tbb::concurrent_bounded_queue` instead of `tbb::concurrent_queue` because the bounded type offers a blocking `pop()` method, which will cause a thread to wait until work becomes available in the queue; also, we do not anticipate a need for a task queue of unbounded size. Blocking on the task queue isn't actually necessary for our simplified application, because all the tasks are generated before any of the threads begin operating on those tasks. However, this blocking strategy supports a *dynamic* task queue, in which new tasks can be added to the queue while other tasks are being processed, a requirement that often arises in other applications.

Notes

- The SENTINEL task value indicates that no more tasks remain. Each thread consumes one SENTINEL from the task queue so it can know when to exit, and adds one SENTINEL to the task queue just before that thread exits, which then enables another thread to finish.

- As with the OpenMP version, the threads implementation uses a thread-safe vector (`tbb::concurrent_vector<Pair> pairs;`) for storing the key-value pairs produced by calls to `Map()`, since multiple threads might access that shared vector at the same time.

4.1 Compilation:

To compile sequential solution

1. For the C++11 standard threads solution, the individual files to download from iCollege are:

   dd_threads.cpp
   Makefile

2. Start the terminal window by choosing its icon in the menu bar at the top (it looks like a black square box).
3. Create a director and name it cplusthreads1
4. Copy the C++11 standard threads solution *cpp* and *make* files into cplusthreads1 directory in your PI.
5. Cd into this directory
6. **type make**

Or, a straightforward compile can be used for this example:

```
g++ -o dd_threads  dd_threads.cpp
```

5) Measure Run-Time

To measure the running time of each implementation, use time –p. For example (assuming you run make already to create exe file for drugdesign-static):

time –p ./ dd_serial

time –p ./ dd_omp 1

time –p ./ dd_threads 1

Run and Fill out the table below (use real for time):

Implementation	Time (s)
dd_serial	
dd_omp	
dd_threads	

Run and Fill out the table below (use real for time):

time –p ./ dd_omp 2

time –p ./ dd_threads 2

time –p ./ dd_omp 3

time –p ./ dd_threads 3

time –p ./ dd_omp 4

time –p ./ dd_threads 4

Implementation	Time (s) 2 Threads	Time (s) 3 Threads	Time (s) 4 Threads
dd_omp			
dd_threads			

2.3 Discussion Questions

1. Which approach is the fastest?

2. Determine the number of lines in each file (use wc -l). How does the C++11 implementation compare to the OpenMP implementations?

3. Increase the number of threads to 5 threads. What is the run time for each?

4. Increase the maximum ligand length to 7, and rerun each program. What is the run time for each?